CHAPTER 3

Machines with Memory

As we saw in Chapter 1, every finite computational task can be realized by a combinational circuit. While this is an important concept, it is not very practical; we cannot afford to design a special circuit for each computational task. Instead we generally perform computational tasks with machines having memory. In a strong sense to be explored in this chapter, the memory of such machines allows them to reuse their equivalent circuits to realize functions of high circuit complexity.

In this chapter we examine the deterministic and nondeterministic finite-state machine (FSM), the random-access machine (RAM), and the Turing machine. The finite-state machine moves from state to state while reading input and producing output. The RAM has a central processing unit (CPU) and a random-access memory with the property that each memory word can be accessed in one unit of time. Its CPU executes instructions, reading and writing data from and to the memory. The Turing machine has a control unit that is a finite-state machine and a tape unit with a head that moves from one tape cell to a neighboring one in each unit of time. The control unit reads from, writes to, and moves the head of the tape unit.

We demonstrate through simulation that the RAM and the Turing machine are universal in the sense that every finite-state machine can be simulated by the RAM and that it and the Turing machine can simulate each other. Since they are equally powerful, either can be used as a reference model of computation.

We also simulate with circuits computations performed by the FSM, RAM, and Turing machine. These circuit simulations establish two important results. First, they show that all computations are constrained by the available resources, such as space and time. For example, if a function f is computed in T steps by the RAM with storage capacity S (in bits), then S and T must satisfy the inequality $C_\Omega(f) = O(ST)$, where $C_\Omega(f)$ is the size of the smallest circuit for f over the complete basis Ω. Any attempt to compute f on the RAM using space S and time T whose product is too small will fail. Second, an $O(\log ST)$-space, $O(ST)$-time program exists to write the descriptions of circuits simulating the above machines. This fact leads to the identification in this chapter of the first examples of **P**-complete and **NP**-complete problems.

3.1 Finite-State Machines

The finite-state machine (FSM) has a set of states, one of which is its initial state. At each unit of time an FSM is given a letter from its input alphabet. This causes the machine to move from its current state to a potentially new state. While in a state, the FSM produces a letter from its output alphabet. Such a machine computes the function defined by the mapping from its initial state and strings of input letters to strings of output letters. FSMs can also be used to accept strings, as discussed in Chapter 4. Some states are called final states. A string is recognized (or accepted) by an FSM if the last state entered by the machine on that input string is a final state. The language recognized (or accepted) by an FSM is the set of strings accepted by it. We now give a formal definition of an FSM.

DEFINITION 3.1.1 *A* **finite-state machine (FSM)** *M is a seven-tuple $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$, where Σ is the* **input alphabet***, Ψ is the* **output alphabet***, Q is the finite* **set of states***, $\delta : Q \times \Sigma \mapsto Q$ is the* **next-state function***, $\lambda : Q \mapsto \Psi$ is the* **output function***, s is the* **initial state** *(which may be fixed or variable), and F is the set of* **final states** *($F \subseteq Q$). If the FSM is given input letter a when in state q, it enters state $\delta(q, a)$. While in state q it produces the output letter $\lambda(q)$.*

The FSM M **accepts the string** $\boldsymbol{w} \in \Sigma^*$ *if the last state entered by M on the input string $\boldsymbol{w}$ starting in state s is in the set F. M* **recognizes** *(or* **accepts***)* **the language** *L consisting of the set of such strings.*

When the initial state of the FSM M is not fixed, for each integer T M maps the initial state s and its T external inputs $\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_T$ onto its T external outputs $\boldsymbol{y}_1, \boldsymbol{y}_2, \dots, \boldsymbol{y}_T$ and the final state $q^{(T)}$. We say that in T steps the FSM M **computes** *the function $f_M^{(T)} : Q \times \Sigma^T \mapsto Q \times \Psi^T$. It is assumed that the sets Σ, Ψ, and Q are encoded in binary so that $f_M^{(T)}$ is a binary function.*

The next-state and output functions of an FSM, δ and λ, can be represented as in Fig. 3.1. We visualize these functions taking a state value from a memory and an input value from an external input and producing next-state and output values. Next-state values are stored in the memory and output values are released to the external world. From this representation an actual machine (a sequential circuit) can be constructed (see Section 3.3). Once circuits are constructed for δ and λ, we need only add memory units and a clock to construct a sequential circuit that emulates an FSM.

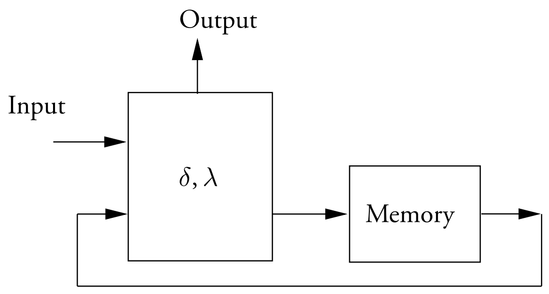


Figure 3.1 The finite-state machine model.

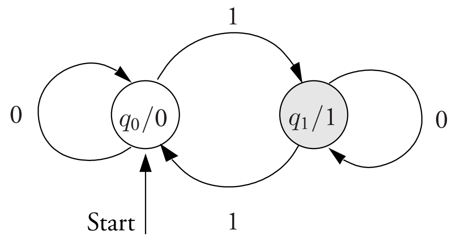


Figure 3.2 A finite-state machine computing the EXCLUSIVE OR of its inputs.

An example of an FSM is shown in Fig. 3.2. Its input and output alphabets and state sets are $\Sigma = \{0, 1\}$, $\Psi = \{0, 1\}$, and $Q = \{q_0, q_1\}$, respectively. Its next-state and output functions, δ and λ, are given below.

q	σ	$\delta(q, \sigma)$
q_0	0	q_0
q_0	1	q_1
q_1	0	q_1
q_1	1	q_0

q	$\lambda(q)$
q_0	0
q_1	1

The FSM has initial state q_0 and final state q_1. As a convenience we explicitly identify final states by shading, although in practice they can be associated with states producing a particular output letter.

Each state has a label q_j/v_j, where q_j is the name of the state and v_j is the output produced while in this state. The initial state has an arrow labeled with the word "start" pointing to it. Clearly, the set of strings accepted by this FSM are those containing an odd number of instances of 1. Thus it computes the EXCLUSIVE OR function on an arbitrary number of inputs.

While it is conventional to think of the finite-state machine as a severely restricted computational model, it is actually a very powerful one. The random-access machine (RAM) described in Section 3.4 is an FSM when the number of memory locations that it contains is bounded, as is always so in practice. When a program is first placed in the memory of the RAM, the program sets the initial state of the RAM. The RAM, which may or may not read external inputs or produce external outputs, generally will leave its result in its memory; that is, the result of the computation often determines the final state of the random-access machine.

The FSM defined above is called a **Moore machine** because it was defined by E.F. Moore [222] in 1956. An alternative FSM, the **Mealy machine** (defined by Mealy [214] in 1955), has an output function $\lambda^* : Q \times \Sigma \mapsto \Psi$ that generates an output on each transition from one state to another. This output is determined by both the state in which the machine resides before the state transition and the input letter causing the transition. It can be shown that the two machine models are equivalent (see Problem 3.6): any computation one can do, the other can do also.

3.1.1 Functions Computed by FSMs

We now examine the ways in which an FSM might compute a function. Since our goal is to understand the power and limits of computation, we must be careful not to assume that an FSM can have hidden access to an external computing device. All computing devices must be explicit. It follows that we allow FSMs only to compute functions that receive inputs and produce outputs at data-independent times.

To understand the function computed by an FSM M, observe that in initial state $q^{(0)} = s$ and receiving input letter w_1, M enters state $q^{(1)} = \delta(q^{(0)}, w_1)$ and produces output $y_1 = \lambda(q^{(1)})$. If M then receives input w_2, it enters state $q^{(2)} = \delta(q^{(1)}, w_2)$ and produces output $y_2 = \lambda(q^{(2)})$. Repeated applications of the functions δ and λ on successive states with successive inputs, as suggested by Fig. 3.3, generate the outputs $y_1, y_2, \dots, y_T$ and the final state $q^{(T)}$. The function $f_M^{(T)} : Q \times \Sigma^T \mapsto Q \times \Psi^T$ given in Definition 3.1.1 defines this mapping from an initial state and inputs to the final state and outputs:

$$f_M^{(T)}\left(q^{(0)}, w_1, w_2, \dots, w_T\right) = \left(q^{(T)}, y_1, y_2, \dots, y_T\right)$$

This simulation of a machine with memory by a circuit illustrates a fundamental point about computation, namely, that the role of memory is to hold intermediate results on which the logical circuitry of the machine can operate in successive cycles.

When an FSM M is used in a T-step computation, it usually does not compute the most general function $f_M^{(T)}$ that it can. Instead, some restrictions are generally placed on the possible initial states, on the values of the external inputs provided to M, and on the components of the final state and output letters used in the computation. Consider three examples of the specialization of an FSM to a particular task. In the first, let the FSM model be that shown in Fig. 3.2 and let it be used to form the EXCLUSIVE OR of n variables. In this case, we supply n bits to the FSM but ignore all but the last output value it produces. In the second example, let the FSM be a programmable machine in which a program is loaded into its memory before the start of a computation, thereby setting its initial state. The program ignores all external inputs and produces no output, leaving the value of the function in memory. In the third example, again let the FSM be programmable, but let the program that resides initially residing in its memory be a "boot program" that treats its inputs as program statements. (Thus, the FSM has a fixed initial state.) The boot program forms a program by loading these statements into successive memory locations. It then jumps to the first location in this program.

In each of these examples, the function f that is actually computed by M in T steps is a subfunction of the function $f_M^{(T)}$ because f is obtained by either restricting the values of

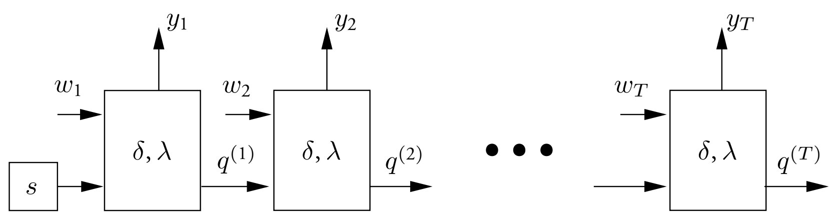


Figure 3.3 A circuit computing the same function, $f_M^{(T)}$, as a finite-state machine M in T steps.

the initial state and inputs to M or deleting outputs or both. We assume that every function computed by M in T steps is a subfunction f of the function $f_M^{(T)}$.

The simple construction of Fig. 3.3 is the first step in deriving a space-time product inequality for the random-access machine in Section 3.5 and in establishing a connection between Turing time and circuit complexity in Section 3.9.2. It is also involved in the definition of the **P**-complete and **NP**-complete problems in Section 3.9.4.

3.1.2 Computational Inequalities for the FSM

In this book we model each computational task by a function that, we assume without loss of generality, is binary. We also assume that the function $f_M^{(T)} : Q \times \Sigma^T \mapsto Q \times \Psi^T$ computed in T steps by an FSM M is binary. In particular, we assume that the next-state and output functions, δ and λ, are also binary; that is, we assume that their input, state, and output alphabets are encoded in binary. We now derive some consequences of the fact that a computation by an FSM can be simulated by a circuit.

The size $C_\Omega\left(f_M^{(T)}\right)$ of the smallest circuit to compute the function $f_M^{(T)}$ is no larger than the size of the circuit shown in Fig. 3.3. But this circuit has size $T \cdot C_\Omega(\delta, \lambda)$, where $C_\Omega(\delta, \lambda)$ is the size of the smallest circuit to compute the functions δ and λ. The depth of the shallowest circuit for $f_M^{(T)}$ is no more than $T \cdot D_\Omega(\delta, \lambda)$ because the longest path through the circuit of Fig. 3.3 has this length.

Let f be the function computed by M in T steps. Since it is a subfunction of $f_M^{(T)}$, it follows from Lemma 2.4.1 that the size of the smallest circuit for f is no larger than the size of the circuit for $f_M^{(T)}$. Similarly, the depth of f, $D_\Omega(f)$, is no more than that of $f_M^{(T)}$. Combining the observations of this paragraph with those of the preceding paragraph yields the following computational inequalities. A **computational inequality** is an inequality relating parameters of computation, such as time and the circuit size and depth of the next-state and output function, to the size or depth of the smallest circuit for the function being computed.

THEOREM 3.1.1 *Let $f_M^{(T)}$ be the function computed by the FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ in T steps, where δ and λ are the binary next-state and output functions of M. The circuit size and depth over the basis Ω of any function f computed by M in T steps satisfy the following inequalities:*

$$\begin{array}{lclcl} C_\Omega(f) & \leq & C_\Omega\left(f_M^{(T)}\right) & \leq & TC_\Omega(\delta, \lambda) \\ D_\Omega(f) & \leq & D_\Omega\left(f_M^{(T)}\right) & \leq & TD_\Omega(\delta, \lambda) \end{array}$$

The circuit size $C_\Omega(\delta, \lambda)$ and depth $D_\Omega(\delta, \lambda)$ of the next-state and output functions of an FSM M are measures of its complexity, that is, of how useful they are in computing functions. The above theorem, which says nothing about the actual technologies used to realize M, relates these two measures of the complexity of M to the complexities of the function f being computed. This is a theorem about computational complexity, not technology.

These inequalities stipulate constraints that must hold between the time T and the circuit size and depth of the machine M if it is used to compute the function f in T steps. Let the product $TC_\Omega(\delta, \lambda)$ be defined as the **equivalent number of logic operations performed by** M. The first inequality of the above theorem can be interpreted as saying that the number of equivalent logic operations performed by an FSM to compute a function f must be at least the minimum number of gates necessary to compute f with a circuit. A similar interpretation can be given to the second inequality involving circuit depth.

The first inequality of Theorem 3.1.1 and the interpretation given to $T \cdot C_\Omega(\delta, \lambda)$ justify the following definitions of computational work and power. Here power is interpreted as the time rate at which work is done. These measures correlate nicely with our intuition that machines that contain more equivalent computing elements are more powerful.

DEFINITION 3.1.2 *The* **computational work** *done by an FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ is $TC_\Omega(\delta, \lambda)$, the number of equivalent logical operations performed by M, which is the product of T, the number of steps executed by M, and $C_\Omega(\delta, \lambda)$, the size complexity of its next-state and output functions. The* **power** *of an FSM M is $C_\Omega(\delta, \lambda)$, the number of logical operations performed by M per step.*

Theorem 3.1.1 is also a form of **impossibility theorem**: it is impossible to compute functions f for which $TC_\Omega(\delta, \lambda)$ and $TD_\Omega(\delta, \lambda)$ are respectively less than the size and depth complexity of f. It may be possible to compute a function on some points of its domain with smaller values of these parameters, but not on all points. The halting problem, another example of an impossibility theorem, is presented in Section 5.8.2. However, it deals with the computation of functions over infinite domains.

The inequalities of Theorem 3.1.1 also place upper limits on the size and depth complexities of functions that can be computed in a bounded number of steps by an FSM, regardless of how the FSM performs the computation.

Note that there is no guarantee that the upper bounds stated in Theorem 3.1.1 are at all close to the lower bounds. It is always possible to compute a function inefficiently, that is, with resources that are greater than the minimal resources necessary.

3.1.3 Circuits Are Universal for Bounded FSM Computations

We now ask whether the classes of functions computed by circuits and by FSMs executing a bounded number of steps are different. We show that they are the same. Many different functions can be computed from the function $f_M^{(T)}$ by specializing inputs and/or deleting outputs.

THEOREM 3.1.2 *Every subfunction of the function $f_M^{(n)}$ computable by an FSM on n inputs is computable by a Boolean circuit and vice versa.*

Proof A Boolean function on n inputs, f, may be computed by an FSM with $2^{n+1} - 1$ states by branching from the current state to one of two different states on inputs 0 and 1 until all n inputs have been read; it then produces the output that would be produced by f on these n inputs. A fifteen-state version of this machine that computes the EXCLUSIVE OR on three inputs as a subfunction is shown in Fig. 3.4.

The proof in the other direction is also straightforward, as described above and represented schematically in Fig. 3.3. Given a binary representation of the input, output, and state symbols of an FSM, their associated next-state and output functions are binary functions. They can be realized by circuits, as can $f_M^{(n)}(s, \boldsymbol{w}) = (q^{(n)}, \boldsymbol{y})$, the function computed by the FSM on n inputs, as suggested by Fig. 3.3. Finally, the subfunction f is obtained by fixing the appropriate inputs, assigning variable names to the remaining inputs, and deleting the appropriate outputs. ■

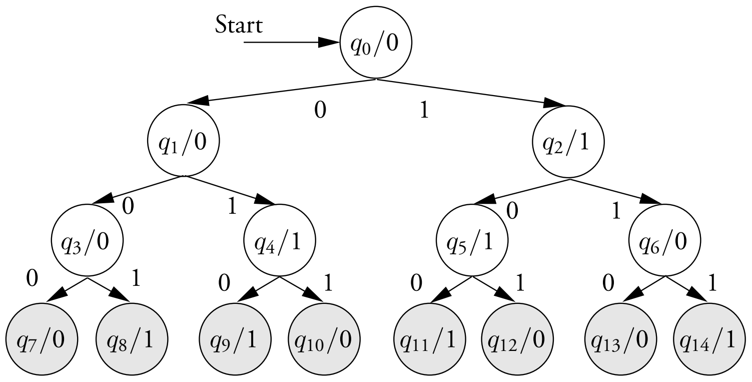


Figure 3.4 A fifteen-state FSM that computes the EXCLUSIVE OR of three inputs as a subfunction of $f_M^{(3)}$ obtained by deleting all outputs except the third.

3.1.4 Interconnections of Finite-State Machines

Later in this chapter we examine a family of FSMs characterized by a computational unit connected to storage devices of increasing size. The random-access machine that has a CPU of small complexity and a random-access memory of large but indeterminate size is of this type. The Turing machine having a fixed control unit that moves a tape head over a potentially infinite tape is another example.

This idea is captured by the **interconnection of synchronous FSMs**. Synchronous FSMs read inputs, advance from state to state, and produce outputs in synchronism. We allow two or more synchronous FSMs to be interconnected so that some outputs from one FSM are supplied as inputs of another, as illustrated in Fig. 3.5. Below we generalize Theorem 3.1.1 to a pair of synchronous FSMs. We model random-access machines and Turing machines in this fashion when each uses a finite amount of storage.

THEOREM 3.1.3 *Let $f^{(T)}_{M_1 \times M_2}$ be a function computed in T steps by a pair of interconnected synchronous FSMs, $M_1 = (\Sigma_1, \Psi_1, Q_1, \delta_1, \lambda_1, s_1, F_1)$ and $M_2 = (\Sigma_2, \Psi_2, Q_2, \delta_2, \lambda_2, s_2, F_2)$.*

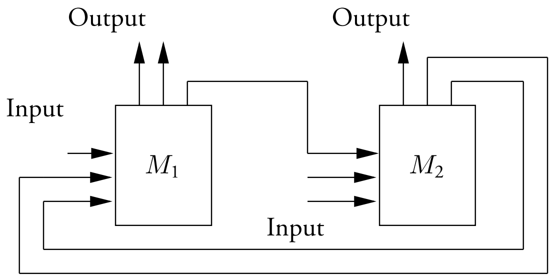


Figure 3.5 The interconnection of two finite-state machines in which one of the three outputs of M_1 is supplied as an input to M_2 and two of the three outputs of M_2 are supplied to M_1 as inputs.

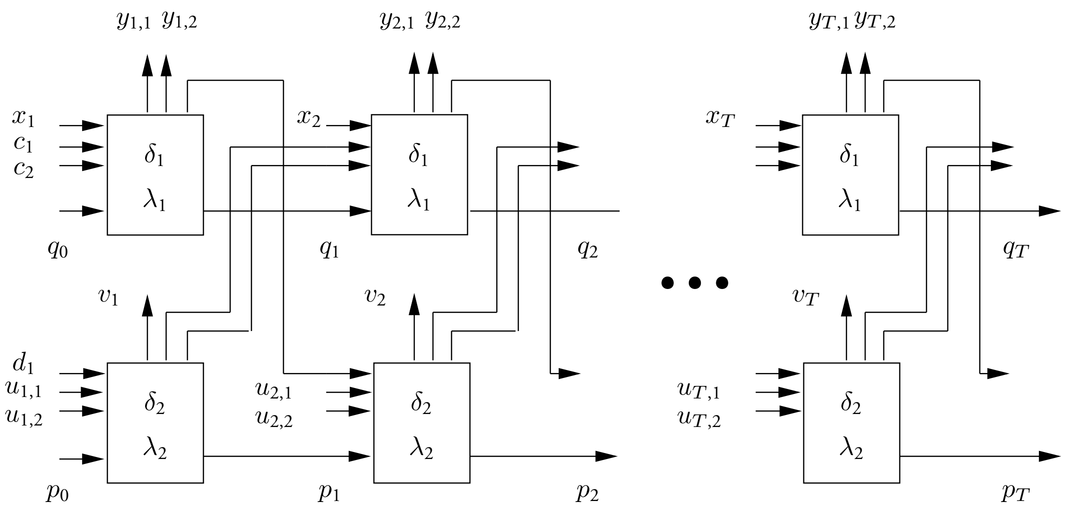


Figure 3.6 A circuit simulating T steps of the two synchronous interconnected FSMs shown in Fig. 3.5. The top row of circuits simulates a T-step computation by M_1 and the bottom row simulates a T-step computation by M_2. One of the three outputs of M_1 is supplied as an input to M_2 and two of the three outputs of M_2 are supplied to M_1 as inputs. The states of M_1 on the initial and T successive steps are $q_0, q_1, \dots, q_T$. Those of M_2 are $p_0, p_1, \dots, p_T$.

Let $C_\Omega(\delta, \lambda)$ and $D_\Omega(\delta, \lambda)$ be the size and depth of encodings of the next-state and output functions. Then, the circuit size and depth over the basis Ω of any function f computed by the pair $M_1 \times M_2$ in T steps (that is, a subfunction of $f^{(T)}_{M_1 \times M_2}$) satisfy the following inequalities:

$$C_\Omega(f) \le T[C_\Omega(\delta_1, \lambda_1) + C_\Omega(\delta_2, \lambda_2)]$$
$$D_\Omega(f) \le T[\max(D_\Omega(\delta_1, \lambda_1), D_\Omega(\delta_2, \lambda_2))]$$

Proof The construction that leads to this result is suggested by Fig. 3.6. We unwind both FSMs and connect the appropriate outputs from one to the other to produce a circuit that computes $f^{(T)}_{M_1 \times M_2}$. Observe that the number of gates in the simulated circuit is T times the sum of the number of gates, whereas the depth is T times the depth of the deeper circuit. ■

3.1.5 Nondeterministic Finite-State Machines

The finite-state machine model described above is called a **deterministic FSM** (DFSM) because, given a current state and an input, the next state of the FSM is uniquely determined. A potentially more general FSM model is the **nondeterministic** FSM (NFSM) characterized by the possibility that several next states can be reached from the current state for some given input letter.

One might ask if such a model has any use, especially since to the untrained eye a nondeterministic machine would appear to be a dysfunctional deterministic one. The value of an NFSM is that it may recognize languages with fewer states and in less time than needed by a DFSM. The concept of nondeterminism will be extended later to the Turing machine, where it is used to classify languages in terms of the time and space they need for recognition. For example, it will be used to identify the class **NP** of languages that are recognized by nondeterministic Turing machines in a number of steps that is polynomial in the length of their inputs. (See Section 3.9.6.) Many important combinatorial problems, such as the traveling salesperson problem, fall into this class.

The formal definition of the NFSM is given in Section 4.1, where the next-state function $\delta : Q \times \Sigma \mapsto Q$ of the FSM is replaced by a next-state function $\delta : Q \times \Sigma \mapsto 2^Q$. Such functions assign to each state q and input letter a a subset $\delta(q, a)$ of the set Q of states of the NFSM (2^Q, the power set, is the set of all subsets of Q. It is introduced in Section 1.2.1.) Since the value of $\delta(q, a)$ can be the empty set, there may be no successor to the state q on input a. Also, since $\delta(q, a)$ when viewed as a set can contain more than one element, a state q can have edges labeled a to several other states. Since a DFSM has a single successor to each state on every input, a DFSM is an NFSM in which $\delta(q, a)$ is a singleton set.

While a DFSM M accepts a string $\boldsymbol{w}$ if $\boldsymbol{w}$ causes M to move from the initial state to a final state in F, an NFSM accepts $\boldsymbol{w}$ if there is some set of next-state choices for $\boldsymbol{w}$ that causes M to move from the initial state to a final state in F.

An NFSM can be viewed as a purely deterministic finite-state machine that has two inputs, as suggested in Fig. 3.7. The first, the **standard input**, a, accepts the user's data. The second, the **choice input**, c, is used to choose a successor state when there is more than one. The information provided via the choice input is not under the control of the user supplying data via the standard input. As a consequence, the machine is nondeterministic from the point of view of the user but fully deterministic to an outside observer. It is assumed that the **choice agent** supplies the choice input and, with full knowledge of the input to be provided by the user, chooses state transitions that, if possible, lead to acceptance of the user input. On the other hand, the choice agent cannot force the machine to accept inputs for which it is not designed.

In an NFSM it is not required that a state q have a successor for each value of the standard and choice inputs. This possibility is captured by allowing $\delta(q, a, c)$ to have no value, denoted by $\delta(q, a, c) = \perp$.

Figure 3.8 shows an NFSM that recognizes strings over $\mathcal{B}^*$ that end in 00101. In this figure parentheses surround the choice input when its value is needed to decide the next state. In this machine the choice input is set to 1 when the choice agent knows that the user is about to supply the suffix 00101.

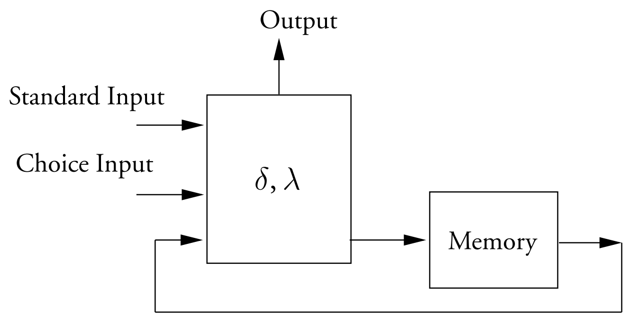


Figure 3.7 A nondeterministic finite-state machine modeled as a deterministic one that has a second choice input whose value disambiguates the value of the next state.

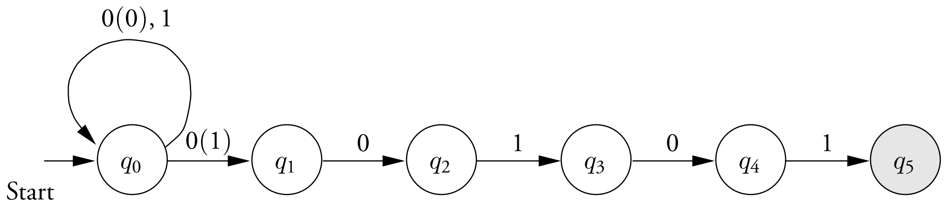


Figure 3.8 A nondeterministic FSM that accepts binary strings ending in 00101. Choice inputs are shown in parentheses for those user inputs for which the value of choice inputs can disambiguate next-state moves.

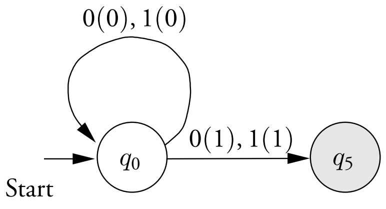


Figure 3.9 An example of an NFSM whose choice agent (its values are in parentheses) accepts not only strings in a language L, but all strings.

Although we use the anthropomorphic phrase "choice agent," it is important to note that this choice agent cannot freely decide which strings to accept and which not. Instead, it must when possible make choices leading to acceptance. Consider, for example, the machine in Fig. 3.9. It would appear that its choice agent can accept strings in an arbitrary language L. In fact, the language that it accepts contains all strings.

Given a string $\boldsymbol{w}$ in the language L accepted by an NFSM, a choice string that leads to its acceptance is said to be a **succinct certificate** for its membership in L.

It is important to note that the nondeterministic finite-state machine is not a model of reality, but is used instead primarily to classify languages. In Section 4.1 we explore the language-recognition capability of the deterministic and nondeterministic finite-state machines and show that they are the same. However, the situation is not so clear with regard to Turing machines that have access to unlimited storage capacity. In this case, we do not know whether or not the set of languages accepted in polynomial time on deterministic Turing machines (the class **P**) is the same set of languages that is accepted in polynomial time by nondeterministic Turing machines (the class **NP**).

3.2 Simulating FSMs with Shallow Circuits*

In Section 3.1 we demonstrated that every T-step FSM computation can be simulated by a circuit whose size and depth are both $O(T)$. In this section we show that every T-step finite-state machine computation can be simulated by a circuit whose size and depth are $O(T)$ and $O(\log T)$, respectively. While this seems a serious improvement in the depth bound, the coefficients hidden in the big-O notation for both bounds depend on the number of states of the FSM and can be very large. Nevertheless, for simple problems, such as binary addition, the

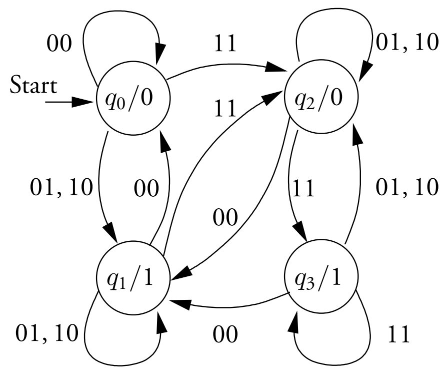


Figure 3.10 A finite-state machine that adds two binary numbers. Their two least significant bits are supplied first followed by those of increasing significance. The output bits represent the sum of the two numbers.

results of this section can be useful. We illustrate this here for binary addition by exhibiting small and shallow circuits for the adder FSM of Fig. 3.10. The circuit simulation for this FSM produces the carry-lookahead adder circuit of Section 2.7. In this section we use matrix multiplication, which is covered in Chapter 6.

The new method is based on the representation of the function $f_M^{(T)} : Q \times \Sigma^T \mapsto Q \times \Psi^T$ computed in T steps by an FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ in terms of the set of **state-to-state mappings** $S = \{h : Q \mapsto Q\}$ where S contains the mappings $\{\Delta_x : Q \mapsto Q \,|\, x \in \Sigma\}$ and Δ_x is defined below.

$$\Delta_x(q) = \delta(q, x) \tag{3.1}$$

That is, $\Delta_x(q)$ is the state to which state q is carried by the input letter x.

The FSM shown in Fig. 3.10 adds two binary numbers sequentially by simulating a ripple adder. (See Section 2.7.) Its input alphabet is $\mathcal{B}^2$, that is, the set of pairs of 0's and 1's. Its output alphabet is $\mathcal{B}$ and its state set is $Q = \{q_0, q_1, q_2, q_3\}$. (A sequential circuit for this machine is designed in Section 3.3.) It has the state-to-state mappings shown in Fig. 3.11.

Let $\odot : S^2 \mapsto S$ be the operator defined on the set S of state-to-state mappings where for arbitrary $h_1, h_2 \in S$ and state $q \in Q$ the operator $\odot$ is defined as follows:

$$(h_1 \odot h_2)(q) = h_2(h_1(q)) \tag{3.2}$$

q	$\Delta_{0,0}(q)$
q_0	q_0
q_1	q_0
q_2	q_1
q_3	q_1

q	$\Delta_{0,1}(q)$
q_0	q_1
q_1	q_1
q_2	q_2
q_3	q_2

q	$\Delta_{1,0}(q)$
q_0	q_1
q_1	q_1
q_2	q_2
q_3	q_2

q	$\Delta_{1,1}(q)$
q_0	q_2
q_1	q_2
q_2	q_3
q_3	q_3

Figure 3.11 The state-to-state mappings associated with the FSM of Fig. 3.10.

The state-to-state mappings in S will be obtained by composing the mappings $\{\Delta_x : Q \mapsto Q \,|\, x \in \Sigma\}$ using this operator.

Below we show that the operator $\odot$ is **associative**, that is, $\odot$ satisfies the property $(h_1 \odot h_2) \odot h_3 = h_1 \odot (h_2 \odot h_3)$. This means that for each $q \in Q$, $((h_1 \odot h_2) \odot h_3)(q) = (h_1 \odot (h_2 \odot h_3))(q) = h_3(h_2(h_1(q)))$. Applying the definition of $\odot$ in Equation (3.2), we have the following for each $q \in Q$:

$$\begin{aligned} ((h_1 \odot h_2) \odot h_3)(q) &= h_3((h_1 \odot h_2)(q)) \\ &= h_3(h_2(h_1(q))) \\ &= (h_2 \odot h_3)(h_1(q)) \\ &= (h_1 \odot (h_2 \odot h_3))(q) \end{aligned} \tag{3.3}$$

Thus, $\odot$ is associative and $(S, \odot)$ is a semigroup. (See Section 2.6.) It follows that a prefix computation can be done on a sequence of state-to-state mappings.

We now use this observation to construct a shallow circuit for the function $f_M^{(T)}$. Let $\boldsymbol{w} = (w_1, w_2, \dots, w_T)$ be a sequence of T inputs to M where w_j is supplied on the jth step. Let $q^{(j)}$ be the state of M after receiving the jth input. From the definition of $\odot$ it follows that $q^{(j)}$ has the following value where s is the initial state of M:

$$q^{(j)} = (\Delta_{w_1} \odot \Delta_{w_2} \odot \cdots \odot \Delta_{w_j})(s)$$

The value of $f_M^{(T)}$ on initial state s and T inputs can be represented in terms of $\boldsymbol{q} = (q^{(1)}, \dots, q^{(T)})$ as follows:

$$f_M^{(T)}(s, \boldsymbol{w}) = \left(q^{(n)}, \lambda(q^{(1)}), \lambda(q^{(2)}), \dots, \lambda(q^{(T)})\right)$$

Let $\boldsymbol{\Lambda^{(T)}}$ be the following sequence of state-to-state mappings:

$$\boldsymbol{\Lambda^{(T)}} = (\Delta_{w_1}, \Delta_{w_2}, \dots, \Delta_{w_T})$$

It follows that $\boldsymbol{q}$ can be obtained by computing the state-to-state mappings $\Delta_{w_1} \odot \Delta_{w_2} \odot \cdots \odot \Delta_{w_j}$, $1 \le j \le T$, and applying them to the initial state s. Because $\odot$ is associative, these T state-to-state mappings are produced by the prefix operator $\mathcal{P}_\odot^{(T)}$ on the sequence $\boldsymbol{\Lambda^{(T)}}$ (see Theorem 2.6.1):

$$\mathcal{P}_\odot^{(T)}(\boldsymbol{\Lambda^{(T)}}) = (\Delta_{w_1}, (\Delta_{w_1} \odot \Delta_{w_2}), \dots, (\Delta_{w_1} \odot \Delta_{w_2} \odot \dots \odot \Delta_{w_T}))$$

Restating Theorem 2.6.1 for this problem, we have the following result.

THEOREM 3.2.1 *For $T = 2^k$, k an integer, the T state-to-state mappings defined by the T inputs to an FSM M can be computed by a circuit over the basis $\Omega = \{\odot\}$ whose size and depth satisfy the following bounds:*

$$\begin{aligned} C_\Omega\left(\mathcal{P}_\odot^{(T)}\right) &\le 2T - \log_2 T - 2 \\ D_\Omega\left(\mathcal{P}_\odot^{(T)}\right) &\le 2 \log_2 T \end{aligned}$$

The construction of a shallow Boolean circuit for $f_M^{(T)}$ is reduced to a five-step problem: 1) for each input letter x design a circuit whose input and output are representations of states and which defines the state-to-state mapping Δ_x for input letter x; 2) construct a circuit for the associative operator $\odot$ that accepts the representations of two state-to-state mappings Δ_y and Δ_z and produces a representation for the state-to-state mapping $\Delta_y \odot \Delta_z$; 3) use the circuit for $\odot$ in a parallel prefix circuit to produce the T state-to-state mappings; 4) construct a circuit that combines the representation of the initial state s with that of the state-to-state mapping $\Delta_{w_1} \odot \Delta_{w_2} \odot \cdots \odot \Delta_{w_j}$ to obtain a representation for the successor state $\Delta_{w_1} \odot \Delta_{w_2} \odot \cdots \odot \Delta_{w_j}(s)$; and 5) construct a circuit for λ that computes an output from the representation of a state.

We now describe a generic, though not necessarily efficient, implementation of these steps.

Let $Q = \{q_0, q_1, \dots, q_{|Q|-1}\}$ be the states of M. The state-to-state mapping Δ_x for the FSM M needed for the first step can be represented by a $|Q| \times |Q|$ Boolean matrix $N(x) = \{n_{ij}(x)\}$ in which the entry in row i and column j, $n_{ij}(x)$, satisfies

$$n_{i,j}(x) = \begin{cases} 1 & \text{if } M \text{ moves from state } q_i \text{ to state } q_j \text{ on input } x \\ 0 & \text{otherwise} \end{cases}$$

Consider again the FSM shown in Fig. 3.10. The matrices associated with its four pairs of inputs $x \in \{(0,0), (0,1), (1,0), (1,1)\}$ are shown below, where $N((0,1)) = N((1,0))$:

$$N((0,0)) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad N((0,1)) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$N((1,1)) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From these matrices the generic matrix $N((u,v))$ parameterized by the values of the inputs (a pair (u,v) in this example) is produced from the following Boolean functions: $t = \overline{u} \wedge \overline{v}$, the **carry-terminate function**, $p = u \oplus v$, the **carry-propagate function**, and $g = u \wedge v$, the **carry-generate function**.

$$N((u,v)) = \begin{bmatrix} t & p & g & 0 \\ t & p & g & 0 \\ 0 & t & p & g \\ 0 & t & p & g \end{bmatrix}$$

Let $\sigma(i) = (0, 0, \dots, 0, 1, 0, \dots 0)$ be the unit $|Q|$-vector that has value 1 in the ith position and zeros elsewhere. Let $\sigma(i)N(x)$ denote Boolean vector-matrix multiplication in which addition is OR and multiplication is AND. Then, for each i, $\sigma(i)N(x) = (n_{i,1}, n_{i,2}, \dots, n_{i,|Q|})$ is the unit vector denoting the state that M enters when it is in state q_i and receives input x.

Let $N(x, y) = N(x) \times N(y)$ be the Boolean matrix-matrix multiplication of matrices $N(x)$ and $N(y)$ in which addition is OR and multiplication is AND. Then, for each x and y the entry in row i and column j of $N(x) \times N(y)$, namely $n_{i,j}^{(2)}(x, y)$, satisfies the following identity:

$$n_{i,j}^{(2)}(x, y) = \bigvee_{q_t \in Q} n_{i,t}(x) \cdot n_{t,j}(y)$$

That is, $n_{i,j}^{(2)}(x, y) = 1$ if there is a state $q_t \in Q$ such that in state q_i, M is given input x, moves to state q_t, and then moves to state q_j on input y. Thus, the composition operator $\odot$ can be realized through the multiplication of Boolean matrices. It is straightforward to show that matrix multiplication is associative. (See Problem 3.10.)

Since matrix multiplication is associative, a prefix computation using matrix multiplication as a composition operator for each prefix $\boldsymbol{x}^{(j)} = (x_1, x_2, \dots, x_j)$ of the input string $\boldsymbol{x}$ generates a matrix $N(\boldsymbol{x}^{(j)}) = N(x_1) \times N(x_2) \times \cdots \times N(x_j)$ defining the state-to-state mapping associated with $\boldsymbol{x}^{(j)}$ for each value of $1 \leq j \leq n$.

The fourth step, the application of a sequence of state-to-state mappings to the initial state $s = q_r$, represented by the $|Q|$-vector $\sigma(r)$, is obtained through the vector-matrix multiplication $\sigma(r)N(\boldsymbol{x}^{(j)})$ for $1 \leq j \leq n$.

The fifth step involves the computation of the output word from the current state. Let the column $|Q|$-vector $\boldsymbol{\lambda}$ contain in the tth position the output of the FSM M when in state q_t. Then, the output produced by the FSM after the jth input is the product $\sigma(r)N(\boldsymbol{x}^{(j)})\boldsymbol{\lambda}$. This result is summarized below.

THEOREM 3.2.2 *Let the finite-state machine $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ with $|Q|$ states compute a subfunction f of $f_M^{(T)}$ in T steps. Then f has the following size and depth bounds over the standard basis Ω_0 for some $\kappa \geq 1$:*

$$C_{\Omega_0}(f) = O\left(M_{\text{matrix}}(|Q|, \kappa)T\right)$$
$$D_{\Omega_0}(f) = O\left((\kappa \log |Q|)(\log T)\right)$$

Here $M_{\text{matrix}}(n, \kappa)$ is the size of a circuit to multiply two $n \times n$ matrices with a circuit of depth $\kappa \log n$. These bounds can be achieved simultaneously.

Proof The circuits realizing the Boolean functions $\{n_{i,j}(x) \mid 1 \leq i, j \leq |Q|\}$, x an input, each have a size determined by the size of the input alphabet Σ, which is constant. The number of operations required to multiply two Boolean matrices with a circuit of depth $\kappa \log |Q|$, $\kappa \geq 1$, is $M_{\text{matrix}}(|Q|, \kappa)$. (See Section 6.3. Note that $M_{\text{matrix}}(|Q|, \kappa) \leq |Q|^3$.) Finally, the prefix circuit uses $O(T)$ copies of the matrix multiplication circuit and has a depth of $O(\log T)$ copies of the matrix multiplication circuit along the longest path. (See Section 2.6.) ■

When an FSM has a large number of states but its next-state function is relatively simple, that is, it has a size that is at worst a polynomial in $\log |Q|$, the above size bound will be much larger than the size bound given in Theorem 3.1.1 because $M_{\text{matrix}}(n, \kappa)$ grows exponentially in $\log |Q|$. The depth bound grows linearly with $\log |Q|$ whereas the depth of the next-state function on which the depth bound of Theorem 3.1.1 depends will typically grow either linearly or as a small polynomial in $\log \log |Q|$ for an FSM with a relatively simple next-state function. Thus, the depth bound will be smaller than that of Theorem 3.1.1 for very large values of T, but for smaller values, the latter bound will dominate.

3.2.1 A Shallow Circuit Simulating Addition

Applying the above result to the adder FSM of Fig. 3.10, we produce a circuit that accepts T pairs of binary inputs and computes the sum as T-bit binary numbers. Since this FSM has four states, the theorem states that the circuit has size $O(T)$ and depth $O(\log T)$. The carry-lookahead adder of Section 2.7 has these characteristics.

We can actually produce the carry-lookahead circuit by a more careful design of the state-to-state mappings. We use the following encodings for states, where states are represented by pairs $\{(c, s)\}$.

State Encoding

q	c	s
q_0	0	0
q_1	0	1
q_2	1	0
q_3	1	1

Since the next-state mappings are the same for inputs $0, 1$, and $1, 0$, we encode an input pair (u, v) by (g, p), where $g = u \wedge v$ and $p = u \oplus v$ are the carry-generate and carry-propagate variables introduced in Section 2.7 and used above. With these encodings, the three different next-state mappings $\{\Delta_{0,0}, \Delta_{0,1}, \Delta_{1,1}\}$ defined in Fig. 3.11 can be encoded as shown in the table below. The entry at the intersection of row (c, s) and column (p, g) in this table is the value (c^*, s^*) of the generic next-state function $(c^*, s^*) = \Delta_{p,g}(c, s)$. (Here we abuse notation slightly to let $\Delta_{p,g}$ denote the state-to-state mapping associated with the pair (u, v) and represent the state q of M by the pair (c, s).)

	g	0		0		1	
	p	0		1		0	
c	s	c^*	s^*	c^*	s^*	c^*	s^*
0	0	0	0	0	1	1	0
0	1	0	0	0	1	1	0
1	0	0	1	1	0	1	1
1	1	0	1	1	0	1	1

Inspection of this table shows that we can write the following formulas for c^* and s^*:

$$c^* = (p \wedge c) \vee g, \quad s^* = p \oplus c$$

Consider two successive input pairs (u_1, v_1) and (u_2, v_2) and associated pairs (p_1, g_1) and (p_2, g_2). If the FSM of Fig. 3.10 is in state (c_0, s_0) and receives input (u_1, v_1), it enters the state $(c_1, s_1) = (p_1 \wedge c_0 \vee g_1, p_1 \oplus c_0)$. This new state can be obtained by combining p_1 and g_1 with c_0. Let (c_2, s_2) be the successor state when the mapping Δ_{p_2,g_2} is applied to (c_1, s_1). The effect of the operator $\odot$ on successive state-to-state mappings Δ_{p_1,g_1} and Δ_{p_2,g_2} is shown below, in which (3.2) is used:

$$\begin{aligned}(\Delta_{p_1,g_1} \odot \Delta_{p_2,g_2})(q) &= \Delta_{p_2,g_2}(\Delta_{p_1,g_1}((c_0, s_0))) \\ &= \Delta_{p_2,g_2}(p_1 \wedge c_0 \vee g_1, p_1 \oplus c_0)\end{aligned}$$

$$
\begin{aligned}
&= (p_2 \wedge (p_1 \wedge c_0 \vee g_1) \vee g_2, p_2 \oplus (p_1 \wedge c_0 \vee g_1)) \\
&= ((p_2 \wedge p_1) \wedge c_0 \vee (g_2 \vee p_2 \wedge g_1)), p_2 \oplus (p_1 \wedge c_0 \vee g_1)) \\
&= (c_2, s_2)
\end{aligned}
$$

It follows that c_2 can be computed from $p^* = p_2 \wedge p_1$ and $g^* = g_2 \vee p_2 \wedge g_1$ and c_0. The value of s_2 is obtained from p_2 and c_1. Thus the mapping $\Delta_{p_1,g_1} \odot \Delta_{p_2,g_2}$ is defined by p^* and g^*, quantities obtained by combining the pairs (p_1, g_1) and (p_2, g_2) using the same associative operator $\diamond$ defined for the carry-lookahead adder in Section 2.7.1.

To summarize, the state-to-state mappings corresponding to subsequences of an input string $((u_0, v_0), (u_1, v_1), \dots, (u_{n-2}, v_{n-2}), (u_{n-1}, v_{n-1}))$ can be computed by representing this string by the carry-propagate, carry-generate string $((p_0, g_0), (p_1, g_1), \dots, (p_{n-2}, g_{n-2}), (p_{n-1}, g_{n-1}))$, computing the prefix operation on this string using the operator $\diamond$, then computing c_i from c_0 and the carry-propagate and carry-generate functions for the ith stage and s_i from this carry-propagate function and c_{i-1}. This leads to the carry-lookahead adder circuit of Section 2.7.1.

3.3 Designing Sequential Circuits

Sequential circuits are concrete machines constructed of gates and binary memory devices. Given an FSM, a sequential machine can be constructed for it, as we show.

A **sequential circuit** is constructed from a logic circuit and a collection of clocked binary memory units, as suggested in Figs. 3.12(a) and 3.15. (Shown in Fig. 3.12(a) is a simple sequential circuit that computes the EXCLUSIVE OR of the initial value in memory and the external input to the sequential circuit.) Inputs to the logic circuit consist of outputs from the binary memory units as well as external inputs. The outputs of the logic circuit serve as inputs to the clocked binary memory units as well as external outputs.

A clocked binary memory unit is driven by a **clock**, a periodic signal that has value 1 (it is **high**) during short, uniformly spaced time intervals and is otherwise 0 (it is **low**), as suggested in Figs. 3.12(b). For correct operation it is assumed that the input to a memory unit does not change when the clock is high. Thus, the outputs of a logic circuit feeding the memory units cannot change during these intervals. This in turn requires that all changes in the inputs to

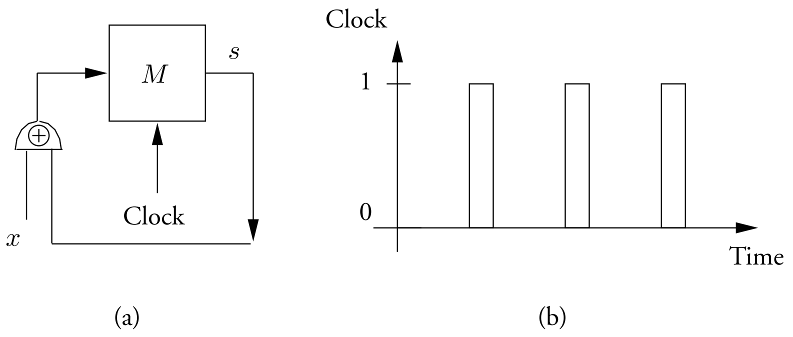


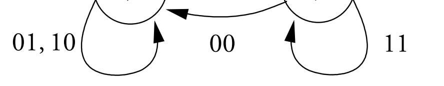
Figure 3.12 (a) A sequential circuit with one gate and one clocked memory unit computing the EXCLUSIVE OR of its inputs; (b) a periodic clock pattern.

this circuit be fully propagated to its outputs in the intervals when the clock is low. A circuit that operates this way is considered **safe**. Designers of sequential circuits calculate the time for signals to pass through a logic circuit and set the interval between clock pulses to insure that the operation of the sequential circuit is safe.

Sequential circuits are designed from finite-state machines (FSMs) in a series of steps. Consider an FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s)$ with input alphabet Σ, output alphabet Ψ, state set Q, next-state function $\delta : Q \times \Sigma \mapsto Q$, output function $\lambda : Q \mapsto \Psi$, and initial state s. (For this discussion we ignore the set of final states; they are important only when discussing language recognition.) We illustrate the design of a sequential machine using the FSM of Fig. 3.10, which is repeated in Fig. 3.13.

The first step in producing a sequential circuit from an FSM is to assign unique binary tuples to each input letter, output letter, and state (the state-assignment problem). This is illustrated for our FSM by the tables of Fig. 3.14 in which the identity encoding is used on inputs and outputs. This step can have a large impact on the size of the logic circuit produced. Second, tables for $\delta : B^4 \mapsto B^2$ and $\lambda : B^2 \mapsto B$, the next-state and output functions of the FSM, respectively, are produced from the description of the FSM, as shown in the same figure. Here c^* and s^* represent the successor to the state (c, s). Third, circuits are designed that realize the binary functions associated with c^* and s^*. Fourth and finally, these circuits are connected to clocked binary memory devices, as shown in Fig. 3.15, to produce a sequential circuit that realizes the FSM. We leave to the reader the task of demonstrating that these circuits compute the functions defined by the tables. (See Problem 3.11.)

Since gates and clocked memory devices can be constructed from semiconductor materials, a sequential circuit can be assembled from physical components by someone skilled in the use of this technology. We design sequential circuits in this book to obtain upper bounds on the size and depth of the next-state and output functions of a sequential machine so that we can derive computational inequalities.



$$\begin{tikzcd}
\arrow[r, "\text{Start}"] & q_0/0 \arrow[loop above, "00"] \arrow[rr, bend left, "11"] \arrow[d, bend right, "{01,10}"'] & & q_2/0 \arrow[loop above, "{01,10}"] \arrow[d, bend right, "11"'] \arrow[dll, bend left, "00"] \\
& q_1/1 \arrow[loop below, "{01,10}"] \arrow[u, bend right, "00"'] \arrow[urr, bend left, "11"] & & q_3/1 \arrow[loop below, "11"] \arrow[u, bend right, "{01,10}"'] \arrow[ll, bend left, "00"]
\end{tikzcd}$$

Figure 3.13 A finite-state machine that simulates the ripple adder of Fig. 2.14. It is in state q_r if the carry-and-sum pair (c_{j+1}, s_j) generated by the jth full adder of the ripple adder represents the integer r, $0 \le r \le 3$. The output produced is the sum bit.

Input Encoding

$\sigma \in \Sigma$		u	v
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Output Encoding

$\lambda(q) \in \Psi$	$\lambda(q)$
0	0
1	1

State Encoding

q	c	s
q_0	0	0
q_1	0	1
q_2	1	0
q_3	1	1

$\delta : B^4 \mapsto B^2$

c	s	u	v	c^*	s^*
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	1
1	1	0	0	0	1
0	0	0	1	0	1
0	1	0	1	0	1
1	0	0	1	1	0
1	1	0	1	1	0
0	0	1	0	0	1
0	1	1	0	0	1
1	0	1	0	1	0
1	1	1	0	1	0
0	0	1	1	1	0
0	1	1	1	1	0
1	0	1	1	1	1
1	1	1	1	1	1

$\lambda : B^2 \mapsto B$

c^*	s^*	s
0	0	0
0	1	1
1	0	0
1	1	1

Figure 3.14 Encodings for inputs, outputs, states, and the next-state and output functions of the FSM adder.

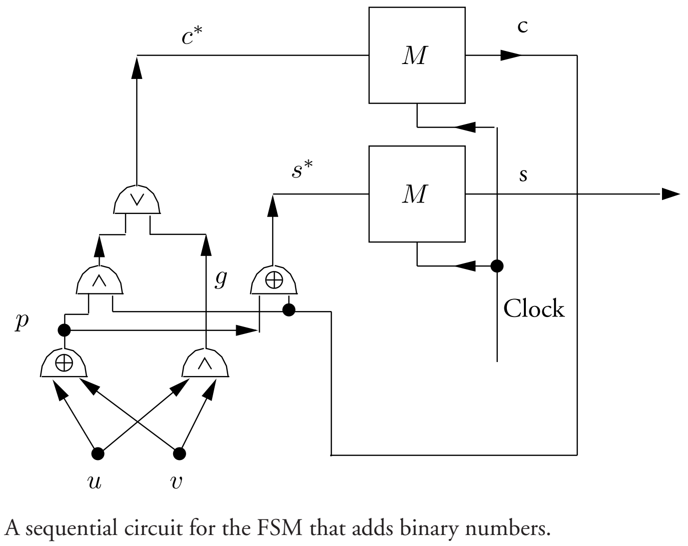


Figure 3.15 A sequential circuit for the FSM that adds binary numbers.

3.3.1 Binary Memory Devices

It is useful to fix ideas about memory units by designing one (a latch) from logic gates. We use two latchs to create a flip-flop, the standard binary storage device. A collection of clocked flip-flops is called a **register**. A clocked **latch** can be constructed from a few AND and NOT gates, as shown in Fig. 3.16(a). The NAND gates (they compute NOT of AND) labeled g_3 and g_4 form the heart of the latch. Consider the inputs to g_3 and g_4, the lines connected to the outputs of NAND gates g_1 and g_2. If one is set to 1 and the other reset to 0, after all signals settle down, ρ and ρ^* will assume complementary values (one will have value 1 and the other will have value 0), regardless of their previous values. The gate with input 1 will assume output 0 and vice versa.

Now if the outputs of g_1 and g_2 are both set to 1 and the values previously assumed by ρ and ρ^* are complementary, these values will be retained due to the feedback between g_3 and g_4, as the reader can verify. Since the outputs of g_1 and g_2 are both 1 when the clock input (CLK in Fig. 3.16) has value 0, the complementary outputs of g_3 and g_4 remain unchanged when the clock is low. Since the outputs of a latch provide inputs to the logic-circuit portion of a sequential circuit, it is important that the latch outputs remain constant when the clock is low.

When the clock input is 1, the outputs of g_1 and g_2 are $\overline{S}$ and $\overline{R}$, the Boolean complements of S and R. If S and R are complementary, as is true for this latch since $R = \overline{S}$, this device will store the value of S in ρ and its complement in ρ^*. Thus, if $S = 1$, the latch is **set** to 1, whereas if $R = 1$ (and $S = 0$) it is **reset** to 0. This type of device is called a **D-type latch**. For this reason we change the name of the external input to this memory device from S to D.

Because the output of the D-type latch shown in Fig. 3.16(a) changes when the clock pulse is high, it cannot be used as a stable input to a logic circuit that feeds this or another such flip-flop. Adding another stage like the first but having the complementary value for the clock pulse, as shown in Fig. 3.16(b), causes the output of the second stage to change only while the clock pulse is low. The output of the first stage does change when the clock pulse is high to record the new value of the state. This is called a **master-slave edge-triggered flip-flop.** Other types of flip-flop are described in texts on computer architecture.

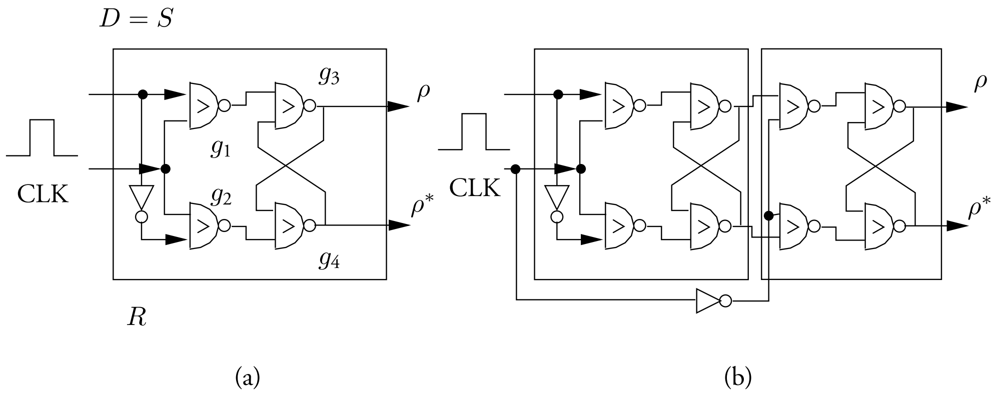


Figure 3.16 (a) Design of a D-type latch from NAND gates. (b) A master-slave edge-triggered D-type flip-flop.

3.4 Random-Access Machines

The **random-access machine** (RAM) models the essential features of the traditional serial computer. The RAM is modeled by two synchronous interconnected FSMs, a central processing unit (CPU) and a random-access memory. (See Fig. 3.17.) The CPU has a small number of storage locations called **registers** whereas the random-access memory has a large number. All operations performed by the CPU are performed on data stored in its registers. This is done for efficiency; no increase in functionality is obtained by allowing operations on data stored in memory locations as well.

3.4.1 The RAM Architecture

The CPU implements a **fetch-and-execute cycle** in which it alternately reads an instruction from a program stored in the random-access memory (the **stored-program concept**) and executes it. Instructions are read and executed from consecutive locations in the random-access memory unless a **jump instruction** is executed, in which case an instruction from a non-consecutive location is executed next.

A CPU typically has five basic kinds of instruction: a) arithmetic and logical instructions of the kind described in Sections 2.5.1, 2.7, 2.9, and 2.10, b) memory load and store instructions for moving data between memory locations and registers, c) jump instructions for breaking out of the current program sequence, d) input and output (I/O) instructions, and e) a halt instruction.

The basic random-access memory has an output word (out_wrd) and three input words, an address ($addr$), a data word (in_wrd), and a command (cmd). The command specifies one of three actions, a) read from a memory location, b) write to a memory location, or c) do nothing. Reading from address $addr$ deposits the value of the word at this location into out_wrd whereas writing to $addr$ replaces the word at this address with the value of in_wrd.

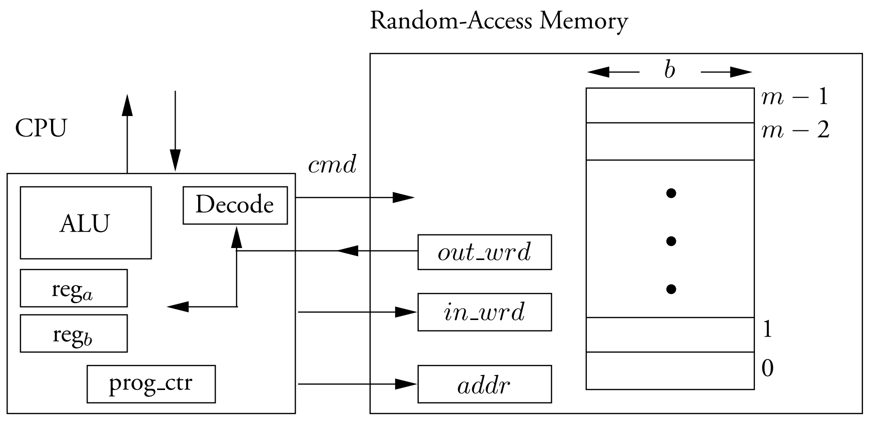


Figure 3.17 The random-access machine has a central processing unit (CPU) and a random-access memory unit.

This memory is called random-access because the time to access a word is the same for all words. The Turing machine introduced in Section 3.7 has a tape memory in which the time to access a word increases with its distance from the tape head.

The random-access memory in the model in Fig. 3.17 has $m = 2^\mu$ storage locations each containing a b-bit word, where μ and b are integers. Each word has a μ-bit address and the addresses are consecutive starting at zero. The combination of this memory and the CPU described above is the **bounded-memory RAM**. When no limit is placed on the number and size of memory words, this combination defines the **unbounded-memory RAM**. We use the term RAM for these two machines when context unambiguously determines which is intended.

DESIGN OF A SIMPLE CPU The design of a simple CPU is given in Section 3.10. (See Fig. 3.31.) This CPU has eight registers, a **program counter** (PC), **accumulator** (AC), **memory address register** (MAR), **memory data register** (MDR), **operation code (opcode) register** (OPC), **input register** (INR), **output register** (OUTR), and **halt register** (HALT). Each operation that requires two operands, such as addition or vector AND, uses AC and MDR as sources for the operands and places the result in AC. Each operation with one operand, such as the NOT of a vector, uses AC as both source and destination for the result. PC contains the address of the next instruction to be executed. Unless a jump instruction is executed, PC is incremented on the execution of each instruction. If a jump instruction is executed, the value of PC is changed. Jumps occur in our simple CPU if AC is zero.

To fetch the next instruction, the CPU copies PC to MAR and then commands the random-access memory to read the word at the address in MAR. This word appears in MDR. The portion of this word containing the identity of the opcode is transferred to OPC. The CPU then inspects the value of OPC and performs the small local operations to execute the instruction represented by it. For example, to perform an addition it commands the arithmetic/logical unit (ALU) to combine the contents of MDR and AC in an adder circuit and deposit the result in AC. If the instruction is a *load accumulator* instruction (LDA), the CPU treats the bits other than opcode bits as address bits and moves them to the MAR. It then commands the random-access memory to deposit the word at this address in MDR, after which it moves the contents of MDR to AC. In Section 3.4.3 we illustrate programming in an assembly language, the language of a machine enhanced by mnemonics and labels. We further illustrate assembly-language programming in Section 3.10.4 for the instruction set of the machine designed in Section 3.10.

3.4.2 The Bounded-Memory RAM as FSM

As this discussion illustrates, the CPU and the random-access memory are both finite-state machines. The CPU receives input from the random-access memory as well as from external sources. Its output is to the memory and the output port. Its state is determined by the contents of its registers. The random-access memory receives input from and produces output to the CPU. Its state is represented by an m-tuple $(\boldsymbol{w}_0, \boldsymbol{w}_1, \dots, \boldsymbol{w}_{m-1})$ of b-bit words, one per memory location, as well as by the values of in_wrd, out_word, and $addr$. We say that the random-access memory has a **storage capacity** of $S = mb$ bits. The RAM has input and output registers (not shown in Fig. 3.17) through which it reads external inputs and produces external outputs.

As the RAM example illustrates, some FSMs are programmable. In fact, a program stored in the RAM memory selects one of very many state sequences that the RAM may execute. The number of states of a RAM can be very large; just the random-access memory alone has more than 2^S states.

The programmability of the unbounded-memory RAM makes it universal for FSMs, as we show in Section 3.4.4. Before taking up this subject, we pause to introduce an assembly-language program for the unbounded-memory RAM. This model will play a role in Chapter 5.

3.4.3 Unbounded-Memory RAM Programs

We now introduce **assembly-language programs** to make concrete the use of the RAM. An assembly language contains one instruction for each machine-level instruction of a CPU. However, instead of bit patterns, it uses mnemonics for opcodes and labels as symbolic addresses. Labels are used in *jump* instructions.

Figure 3.18 shows a simple assembly language. It implements all the instructions of the CPU defined in Section 3.10 and vice versa if the CPU has a sufficiently long word length.

Our new assembly language treats all memory locations as equivalent and calls them registers. Thus, no distinction is made between the memory locations in the CPU and those in the random-access memory. Such a distinction is made on real machines for efficiency: it is much quicker to access registers internal to a CPU than memory locations in an external random-access memory.

Registers are used for data storage and contain integers. Register names are drawn from the set $\{\mathrm{R}_0, \mathrm{R}_1, \mathrm{R}_2, \ldots\}$. The **address of register** R_i is i. Thus, both the number of registers and their size are potentially unlimited. All registers are initialized with the value zero. Registers used as **input registers** to a program are initialized to input values. Results of a computation are placed in **output registers**. Such registers may also serve as input registers. Each instruction may be given a **label** drawn from the set $\{N_0, N_1, N_2, \ldots\}$. Labels are used by jump instructions, as explained below.

Instruction	*Meaning*
INC R_i	Increment the contents of R_i by 1.
DEC R_i	Decrement the contents of R_i by 1.
CLR R_i	Replace the contents of R_i with 0.
$\mathrm{R}_i \leftarrow \mathrm{R}_j$	Replace the contents of R_i with those of R_j.
JMP_+ N_i	Jump to closest instruction above current one with label N_i.
JMP_- N_i	Jump to closest instruction below current one with label N_i.
R_j JMP_+ N_i	If R_j contains 0, jump to closest instruction above current one with label N_i.
R_j JMP_- N_i	If R_j contains 0, jump to closest instruction below current one with label N_i.
CONTINUE	Continue to next instruction; halt if none.

Figure 3.18 The instructions in a simple assembly language.

The meaning of each instruction should be clear except possibly for the CONTINUE and JUMP. If the program reaches a CONTINUE statement other than the last CONTINUE, it executes the following instruction. If it reaches the last CONTINUE statement, the program halts.

The jump instructions R_j JMP_+ N_i, R_j JMP_- N_i, JMP_+ N_i, and JMP_- N_i cause a break in the program sequence. Instead of executing the next instruction in sequence, they cause jumps to instructions with labels N_i. In the first two cases these jumps occur only when the content of register R_j is zero. In the last two cases, these jumps occur unconditionally. The instructions with JMP_+ (JMP_-) cause a jump to the closest instruction with label N_i above (below) the current instruction. The use of the suffixes $+$ and $-$ permit the insertion of program fragments into an existing program without relabeling instructions.

A **RAM program** is a finite sequence of assembly language instructions terminated with CONTINUE. A valid program is one for which each jump is to an existing label. A **halting program** is one that halts.

TWO RAM PROGRAMS We illustrate this assembly language with the two simple programs shown in Fig. 3.19. The first adds two numbers and the second uses the first to square a number. The heading of each program explains its operation. Registers R_0 and R_1 contain the initial values on which the addition program operates. On each step it increments R_0 by 1 and decrements R_1 by 1 until R_1 is 0. Thus, on completion, the value of R_0 is its original value plus the value of R_1 and R_1 contains 0.

The squaring program uses the addition program. It makes three copies of the initial value x of R_0 and stores them in R_1, R_2, and R_3. It also clears R_0. R_2 will be used to reset R_1 to x after adding R_1 to R_0. R_3 is used as a counter and decremented x times, after which x is added to zero x times in R_0; that is, x^2 is computed.

	$R_0 \leftarrow R_0 + R_1$	*Comments*
N_0	R_1 JMP_- N_1	End if R_1 = 0
	INC R_0	Increment R_0
	DEC R_1	Decrement R_1
	JMP_+ N_0	Repeat
N_1	CONTINUE	

	$R_0 \leftarrow R_0^2$	*Comments*
	$R_2 \leftarrow R_0$	Copy R_0 (x) to R_2
	$R_3 \leftarrow R_0$	Copy R_0 (x) to R_3
	CLR R_0	Clear the contents of R_0
N_2	$R_1 \leftarrow R_2$	Copy R_2 (x) to R_1
N_0	R_1 JMP_- N_1	$R_0 \leftarrow R_0 + R_1$
	INC R_0	
	DEC R_1	
	JMP_+ N_0	
N_1	CONTINUE	
	DEC R_3	Decrement R_3
	R_3 JMP_- N_3	End when zero
	JMP_+ N_2	Add x to R_0
N_3	CONTINUE	

Figure 3.19 Two simple RAM programs. The first adds two integers stored initially in registers R_0 and R_1, leaving the result in R_0. The second uses the first to square the contents of R_0, leaving the result in R_0.

As indicated above, with large enough words each of the above assembly-language instructions can be realized with a few instructions from the instruction set of the CPU designed in Section 3.10. It is also true that each of these CPU instructions can be implemented by a fixed number of instructions in the above assembly language. That is, with sufficiently long memory words in the CPU and random-access memory, the two languages allow the same computations with about the same use of time and space.

However, the above assembly-language instructions are richer than is absolutely essential to perform all computations. In fact with just five assembly-language instructions, namely INC, DEC, CONTINUE, R_j JMP_+ N_i, and R_j JMP_- N_i, all the other instructions can be realized. (See Problem 3.21.)

3.4.4 Universality of the Unbounded-Memory RAM

The unbounded-memory RAM is universal in two senses. First, it can simulate any finite-state machine including another random-access machine, and second, it can execute any RAM program.

DEFINITION 3.4.1 *A machine M is* **universal** *for a class of machines $\mathcal{C}$ if every machine in $\mathcal{C}$ can be simulated by M. (A stronger definition requiring that M also be in $\mathcal{C}$ is used in Section 3.8.)*

We now show that the RAM is universal for the class $\mathcal{C}$ of finite-state machines. We show that in $O(T)$ steps and with constant storage capacity S the RAM can simulate T steps of any other FSM. Since any random-access machine that uses a bounded amount of memory can be described by a logic circuit such as the one defined in Section 3.10, it can also be simulated by the RAM.

THEOREM 3.4.1 *Every T-step FSM $M = (\Sigma, \Psi, Q, \delta, \lambda, s, F)$ computation can be simulated by a RAM in $O(T)$ steps with constant space. Thus, the RAM is universal for finite-state machines.*

Proof We sketch a proof. Since an FSM is characterized completely by its next-state and output functions, both of which are assumed to be encoded by binary functions, it suffices to write a fixed-length RAM program to perform a state transition, generate output, and record the FSM state in the RAM memory using the tabular descriptions of the next-state and output functions. This program is then run repeatedly. The amount of memory necessary for this simulation is finite and consists of the memory to store the program plus one state (requiring at least $\log_2 |Q|$ bits). While the amount of storage and time to record and compute these functions is constant, they can be exponential in $\log_2 |Q|$ because the next-state and output functions can be a complex binary function. (See Section 2.12.) Thus, the number of steps taken by the RAM per FSM state transition is constant. ■

The second notion of universality is captured by the idea that the RAM can execute RAM programs. We discuss two execution models for RAM programs. In the first, a RAM program is stored in a private memory of the RAM, say in the CPU. The RAM alternates between reading instructions from its private memory and executing them. In this case the registers described in Section 3.4.3 are locations in the random-access memory. The program counter either advances to the next instruction in its private memory or jumps to a new location as a result of a jump instruction.

In the second model (called by some [10] the **random-access stored program machine (RASP)**), a RAM program is stored in the random-access memory itself. A RAM program can be translated to a RASP program by replacing the names of RAM registers by the names of random-access memory locations not used for storing the RAM program. The execution of a RASP program directly parallels that of the RAM program; that is, the RASP alternates between reading instructions and executing them. Since we do not consider the distinction between RASP and RAM significant, we call them both the RAM.

3.5 Random-Access Memory Design

In this section we model the random-access memory described in Section 3.4 as an FSM $M_{\mathrm{RMEM}}(\mu, b)$ that has $m = 2^\mu$ b-bit data words, $\boldsymbol{w}_0, \boldsymbol{w}_1, \dots, \boldsymbol{w}_{m-1}$, as well as an input data word $\boldsymbol{d}$ (**in_wrd**), an input address $\boldsymbol{a}$ (**addr**), and an output data word $\boldsymbol{z}$ (**out_wrd**). (See Fig. 3.20.) The state of this FSM is the concatenation of the contents of the data, input and output words, input address, and the command word. We construct an efficient logic circuit for its next-state and transition function.

To simplify the design of the FSM M_{RMEM} we use the following encodings of the three input commands:

Name	s_1	s_0
no-op	0	0
read	0	1
write	1	0

An input to M_{RMEM} is a binary $(\mu + b + 2)$-bit binary tuple, two bits to represent a command, μ bits to specify an address, and b bits to specify a data word. The output function of M_{RMEM}, λ_{RMEM}, is a simple projection operator and is realized by a circuit without any gates. Applied to the state vector, it produces the output word.

We now describe a circuit for δ_{RMEM}, the next-state function of M_{RMEM}. Memory words remain unchanged if either **no-op** or **read** commands are executed. In these cases the value of the command bit s_1 is 0. One memory word changes if $s_1 = 1$, namely, the one whose

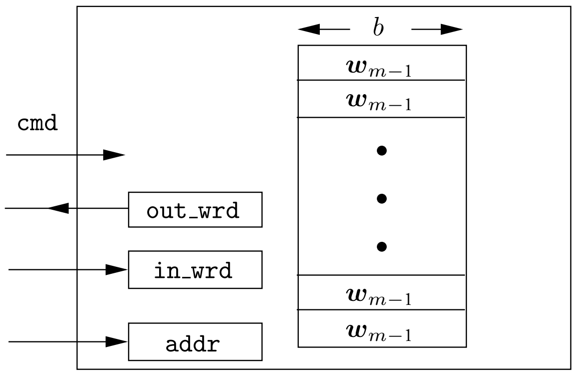


Figure 3.20 A random-access memory unit M_{RMEM} that holds m b-bit words. Its inputs consist of a command (`cmd`), an input word (`in_wrd`), and an address (`addr`). It has one output word (`out_wrd`).

address is $\boldsymbol{a}$. Thus, the memory words $\boldsymbol{w}_0, \boldsymbol{w}_1, \dots, \boldsymbol{w}_{m-1}$ change only when $s_1 = 1$. The word that changes is determined by the μ-bit address $\boldsymbol{a}$ supplied as part of the input. Let $a_{\mu-1}, \dots, a_1, a_0$ be the μ bits of $\boldsymbol{a}$. Let these bits be supplied as inputs to an μ-bit decoder function $f^{(\mu)}_{\text{decode}}$ (see Section 2.5.4). Let $y_{m-1}, \dots, y_1, y_0$ be the m outputs of a decoder circuit. Then, the Boolean function $c_i = s_1 y_i$ (shown in Fig. 3.21(a)) is 1 exactly when the input address $\boldsymbol{a}$ is the binary representation of the integer i and the FSM M_{RMEM} is commanded to **write** the word $\boldsymbol{d}$ at address $\boldsymbol{a}$.

Let $\boldsymbol{w}^*_0, \boldsymbol{w}^*_1, \dots, \boldsymbol{w}^*_{m-1}$ be the new values for the memory words. Let $w^*_{i,j}$ and $w_{i,j}$ be the jth components of $\boldsymbol{w}^*_i$ and $\boldsymbol{w}_i$, respectively. Then, for $0 \le i \le m-1$ and $0 \le j \le b-1$ we write $w^*_{i,j}$ in terms of $w_{i,j}$ and the jth component d_j of $\boldsymbol{d}$ as follows:

$$c_i = s_1 y_i$$
$$w^*_{i,j} = \overline{c}_i w_{i,j} \vee c_i d_j$$

Figures 3.21(a) and (b) show circuits described by these formulas. It follows that changes to memory words can be realized by a circuit containing $C_\Omega\left(f^{(\mu)}_{\text{decode}}\right)$ gates for the decoder, m gates to compute all the terms c_i, $0 \le i \le m-1$, and $4mb$ gates to compute $w^*_{i,j}$, $0 \le i \le m-1$, $0 \le j \le b-1$ (NOTs are counted). Combining this with Lemma 2.5.4, we have that

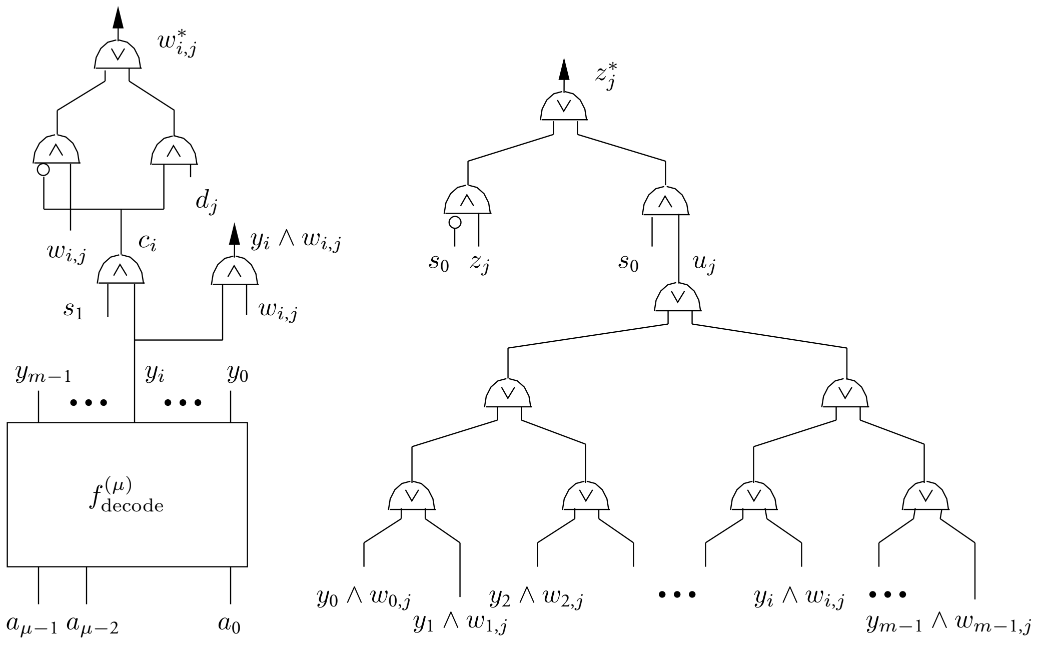


Figure 3.21 A circuit that realizes the next-state and output function of the random-access memory. The circuit in (a) computes the next values $\{w^*_{i,j}\}$ for components of memory words, whereas that in (b) computes components $\{z^*_i\}$ of the output word. The output $y_j \wedge w_{i,j}$ of (a) is an input to (b).

a circuit realizing this portion of the next-state function has at most $m(4b+2)+(2\mu-2)\sqrt{m}$ gates. The depth of this portion of the circuit is the depth of the decoder plus 4 because the longest path between an input and an output $\boldsymbol{w}_0^*, \boldsymbol{w}_1^*, \dots, \boldsymbol{w}_{m-1}^*$ is through the decoder and then through the gates that form $\overline{c}_i w_{i,j}$. This depth is at most $\lceil \log_2 \mu \rceil + 5$.

The circuit description is complete after we give a circuit to compute the output word $\boldsymbol{z}$. The value of $\boldsymbol{z}$ changes only when $s_0 = 1$, that is, when a **read** command is issued. The jth component of $\boldsymbol{z}$, namely z_j, is replaced by the value of $w_{i,j}$, where i is the address specified by the input $\boldsymbol{a}$. Thus, the new value of z_j, z_j^*, can be represented by the following formula (see the circuit of Fig. 3.21(b)):

$$z_j^* = \overline{s}_0 z_j \vee s_0 \left(\bigvee_{k=0}^{m-1} y_k w_{k,j} \right) \qquad \text{for } 0 \le j \le b-1$$

Here $\bigvee$ denotes the OR of the m terms $y_k w_{k,j}$, $m = 2^\mu$. It follows that for each value of j this portion of the circuit can be realized with m two-input AND gates and $m-1$ two-input OR gates (to form $\bigvee$) plus four additional operations. Thus, it is realized by an additional $(2m+3)b$ gates. The depth of this circuit is the depth of the decoder ($\lceil \log \mu \rceil + 1$) plus $\mu = \log_2 m$, the depth of a tree of m inputs to form $\bigvee$, plus three more levels. Thus, the depth of the circuit to produce the output word is $\mu + \lceil \log_2 \mu \rceil + 4$.

The size of the complete circuit for the next-state function is at most $m(6b+2) + (2\mu - 2)\sqrt{m} + 3b$. Its depth is at most $\mu + \lceil \log_2 \mu \rceil + 4$. We state these results as a lemma.

LEMMA 3.5.1 *The next-state and output functions of the FSM $M_{\text{RMEM}}(\mu, b)$, δ_{RMEM} and λ_{RMEM}, can be realized with the following size and depth bounds over the standard basis Ω_0, where $S = mb$ is its storage capacity in bits:*

$$\begin{aligned} C_{\Omega_0}(\delta_{\text{RMEM}}, \lambda_{\text{RMEM}}) &\le m(6b+2) + (2\mu-2)\sqrt{m} + 3b = O(S) \\ D_{\Omega_0}(\delta_{\text{RMEM}}, \lambda_{\text{RMEM}}) &\le \mu + \lceil \log_2 \mu \rceil + 4 \qquad = O(\log(S/b)) \end{aligned}$$

Random-access memories can be very large, so large that their equivalent number of logic elements (which we see from the above lemma is proportional to the storage capacity of the memory) is much larger than the tens to hundreds of thousands of logic elements in the CPUs to which they are attached.

3.6 Computational Inequalities for the RAM

We now state computational inequalities that apply for all computations on the bounded-memory RAM. Since this machine consists of two interconnected synchronous FSMs, we invoke the inequalities of Theorem 3.1.3, which require bounds on the size and depth of the next-state and output functions for the CPU and the random-access memory.

From Section 3.10.6 we see that size and depth of these functions for the CPU grow slowly in the word length b and number of memory words m. In Section 3.5 we designed an FSM modeling an S-bit random-access memory and showed that the size and depth of its next-state and output functions are proportional to S and $\log S$, respectively. Combining these results, we obtain the following computational inequalities.

THEOREM 3.6.1 *Let f be a subfunction of $f_{\mathrm{RAM}}^{(T,m,b)}$, the function computed by the m-word, b-bit RAM with storage capacity $S = mb$ in T steps. Then the following bounds hold simultaneously over the standard basis Ω_0 for logic circuits:*

$$C_{\Omega_0}(f) = O(ST)$$
$$D_{\Omega_0}(f) = O(T \log S)$$

The discussion in Section 3.1.2 of computational inequalities for FSMs applies to this theorem. In addition, this theorem demonstrates the importance of the space-time product, ST, as well as the product $T \log S$. While intuition may suggest that ST is a good measure of the resources needed to solve a problem on the RAM, this theorem shows that it is a fundamental quantity because it directly relates to another fundamental complexity measure, namely, the size of the smallest circuit for a function f. Similar statements apply to the second inequality.

It is important to ask how tight the inequalities given above are. Since they are both derived from the inequalities of Theorem 3.1.1, this question can be translated into a question about the tightness of the inequalities of this theorem. The technique given in Section 3.2 can be used to tighten the second inequality of Theorem 3.1.1 so that the bounds on circuit depth can be improved to logarithmic in T without sacrificing the linearity of the bound on circuit size. However, the coefficients on these bounds depend on the number of states and can be very large.

3.7 Turing Machines

The Turing machine model is the classical model introduced by Alan Turing in his famous 1936 paper [337]. No other model of computation has been found that can compute functions that a Turing machine cannot compute. The Turing machine is a canonical model of computation used by theoreticians to understand the limits on serial computation, a topic that is explored in Chapter 5. The Turing machine also serves as the primary vehicle for the classification of problems by their use of space and time. (See Chapter 8.)

The (deterministic) one-tape, bounded-memory **Turing machine (TM)** consists of two interconnected FSMs, a **control unit** and a **tape unit** of potentially unlimited storage capacity.

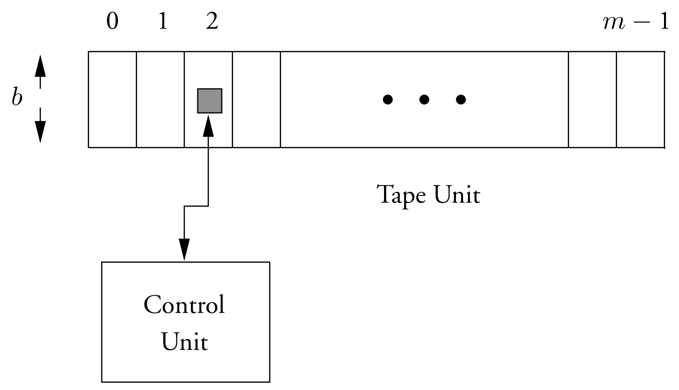


Figure 3.22 A bounded-memory one-tape Turing machine.

(It is shown schematically in Fig. 3.22.) At each unit of time the control unit accepts input from the tape unit and supplies output to it. The tape unit produces the value in the cell under the head, a b-bit word, and accepts and writes a b-bit word to that cell. It also accepts commands to move the head one cell to the left or right or not at all. The bounded-memory tape unit is an array of m b-bit cells and has a **storage capacity** of $S = mb$ bits. A formal definition of the one-tape deterministic Turing machine is given below.

DEFINITION 3.7.1 *A* **standard Turing machine (TM)** *is a six-tuple* $M = (\Gamma, \beta, Q, \delta, s, h)$*, where* Γ *is the* **tape alphabet** *not containing the* **blank symbol** β*,* Q *is the finite* **set of states***,* $\delta : Q \times (\Gamma \cup \{\beta\}) \mapsto (Q \cup \{h\}) \times (\Gamma \cup \{\beta\}) \times \{\mathbf{L}, \mathbf{N}, \mathbf{R}\}$ *is the* **next-state function***,* s *is the* **initial state***, and* $h \notin Q$ *is the* **accepting halt state***. A TM cannot exit from* h*. If* M *is in state* q *with letter* a *under the tape head and* $\delta(q, a) = (q', a', \mathbf{C})$*, its control unit enters state* q' *and writes* a' *in the cell under the head, and moves the head left (if possible), right, or not at all if* $\mathbf{C}$ *is* $\mathbf{L}$*,* $\mathbf{R}$*, or* $\mathbf{N}$*, respectively.*

The TM M **accepts the input string** $\boldsymbol{w} \in \Gamma^*$ *(it contains no blanks) if, when started in state* s *with* $\boldsymbol{w}$ *placed left-adjusted on its otherwise blank tape and the tape head at the leftmost tape cell, the last state entered by* M *is* h*. If* M *has other halting states (states from which it does not exit) these are rejecting states. Also,* M *may not halt on some inputs.*

M **accepts the language** $L(M)$ *consisting of all strings accepted by* M*. If a Turing machine halts on all inputs, we say that it* **recognizes the language** *that it accepts. For simplicity, we assume that when* M *halts during language acceptance it writes the letter 1 in its first tape cell if its input string is accepted and 0 otherwise.*

The **function computed by a Turing machine** *on input string* $\boldsymbol{w}$ *is the string* $\boldsymbol{z}$ *written leftmost on the non-blank portion of the tape after halting. The function computed by a TM is* **partial** *if the TM fails to halt on some input strings and* **complete** *otherwise.*

Thus, a TM performs a computation on input string $\boldsymbol{w}$, which is placed left-adjusted on its tape by placing its head over the leftmost symbol of $\boldsymbol{w}$ and repeatedly reading the symbol under the tape head, making a state change in its control unit, and producing a new symbol for the tape cell and moving the head left or right by one cell or not at all. The head does not move left from the leftmost tape cell. If a TM is used for language acceptance, it accepts $\boldsymbol{w}$ by halting in the accepting state h. If the TM is used for computation, the result of a computation on input $\boldsymbol{w}$ is the string $\boldsymbol{z}$ that remains on the non-blank portion of its tape.

We require that M store the letter 1 or 0 in its first tape cell when halting during language acceptance to simplify the construction of a circuit simulating M in Section 3.9.1. This requirement is not essential because the fact that M has halted in state h can be detected with a simple circuit.

The **multi-tape Turing machine** is a generalization of this model that has multiple tape units. (These models and limits on their ability to solve problems are examined in Chapter 5, where it is shown that the multi-tape TM is no more powerful than the one-tape TM.) Although in practice a TM uses a bounded number of memory locations, the full power of TMs is realized only when they have access to an unbounded number of tape cells.

Although the TM is much more limited than the RAM in the flexibility with which it can access memory, given sufficient time and storage capacity they both compute exactly the same set of functions, as we show in Section 3.8.

A very important class of languages recognized by TMs is the class **P** of polynomial-time languages.

DEFINITION 3.7.2 *A language $L \subseteq \Gamma^*$ is in **P** if there is a Turing machine M with tape alphabet Γ and a polynomial $p(n)$ such that, for every $\boldsymbol{w} \in \Gamma^*$, a) M halts in $p(|\boldsymbol{w}|)$ steps and b) M accepts $\boldsymbol{w}$ if and only if it is in L.*

The class **P** is said to contain all the "feasible" languages because any language requiring more than a polynomial number of steps for its recognition is thought to require so much time for long strings as not to be recognizable in practice.

A second important class of languages is **NP**, the languages accepted in polynomial time by nondeterministic Turing machines. To define this class we introduce the nondeterministic Turing machines.

3.7.1 Nondeterministic Turing Machines

A **nondeterministic Turing machine (NDTM)** is identical to the standard TM except that its control unit has an external choice input. (See Fig. 3.23.)

DEFINITION 3.7.3 *A **non-deterministic Turing machine (NDTM)** is the extension of the TM model by the addition of a choice input to its control unit. Thus an NDTM is a seven-tuple $M = (\Sigma, \Gamma, \beta, Q, \delta, s, h)$, where Σ is the **choice input alphabet**, Γ is the **tape alphabet** not containing the **blank symbol** β, Q is the finite **set of states**, s is the **initial state**, and $h \notin Q$ is the **accepting halt state**. A TM cannot exit from h. When M is in state q with letter a under the tape head, reading choice input c, its **next-state function** $\delta : Q \times \Sigma \times (\Gamma \cup \{\beta\}) \mapsto (Q \cup \{h\}) \times (\Gamma \cup \{\beta\}) \times \{\mathbf{L}, \mathbf{R}, \mathbf{N}\} \cup \perp$ has value $\delta(q, c, a)$. If $\delta(q, c, a) = \perp$, there is no successor to the current state with choice input c and tape symbol a. If $\delta(q, c, a) = (q', a', \mathbf{C})$, M's control unit enters state q', writes a' in the cell under the head, and moves the head left (if possible), right, or not at all if $\mathbf{C}$ is $\mathbf{L}$, $\mathbf{R}$, or $\mathbf{N}$, respectively. The choice input selects possible transitions on each time step.*

*An NDTM M reads one character of its **choice input string** $\boldsymbol{c} \in \Sigma^*$ on each step. An NDTM M **accepts string** $\boldsymbol{w}$ if there is some choice string $\boldsymbol{c}$ such that the last state entered by M is h when M is started in state s with $\boldsymbol{w}$ placed left-adjusted on its otherwise blank tape and the tape head at the leftmost tape cell. We assume that when M halts during language acceptance it writes the letter 1 in its first tape cell if its input string is accepted and 0 otherwise.*

*An NDTM M **accepts the language** $L(M) \subseteq \Gamma^*$ consisting of those strings $\boldsymbol{w}$ that it accepts. Thus, if $\boldsymbol{w} \notin L(M)$, there is no choice input for which M accepts $\boldsymbol{w}$.*

Note that the choice input $\boldsymbol{c}$ associated with acceptance of input string $\boldsymbol{w}$ is selected with full knowledge of $\boldsymbol{w}$. Also, note that an NDTM does not accept any string not in $L(M)$; that is, for no choice inputs does it accept such a string.

The NDTM simplifies the characterization of languages. It is used in Section 8.10 to characterize the class **NP** of languages accepted in nondeterministic polynomial time.

DEFINITION 3.7.4 *A language $L \subseteq \Gamma^*$ is in **NP** if there is a nondeterministic Turing machine M and a polynomial $p(n)$ such that M accepts L and for each $\boldsymbol{w} \in L$ there is a choice input $\boldsymbol{c}$ such that M on input $\boldsymbol{w}$ with this choice input halts in $p(|\boldsymbol{w}|)$ steps.*

A choice input is said to "verify" membership of a string in a language. The particular string provided by the choice agent is a verifier for the language. The languages in **NP** are thus

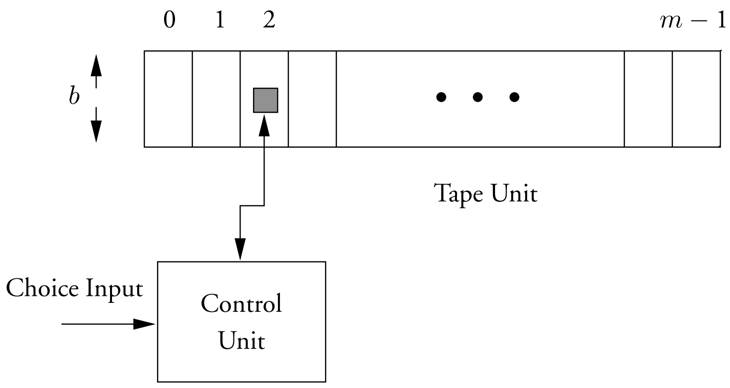


Figure 3.23 A nondeterministic Turing machine modeled as a deterministic one whose control unit has an external choice input that disambiguates the value of its next state.

easy to verify: they can be verified in a polynomial number of steps by a choice input string of polynomial length.

The class **NP** contains many important problems. The **Traveling Salesperson Problem (TSP)** is in this class. TSP is a set of strings of the following kind: each string contains an integer n, the number of vertices (cities) in an undirected graph G, as well as distances between every pair of vertices in G, expressed as integers, and an integer k such that there is a path that visits each city once, returning to its starting point (a **tour**), whose length is at most k. A verifier for TSP is an ordering of the vertices such that the total distance traveled is no more than k. Since there are $n!$ orderings of the n vertices and $n!$ is approximately $\sqrt{2\pi n}n^n e^{-n}$, a verifier can be found in a number of steps exponential in n; the actual verification itself can be done in $O(n^2)$ steps. (See Problem 3.24.) **NP** also contains many other important languages, in particular, languages defining important combinatorial problems.

While it is obvious that **P** is a subset of **NP**, it is not known whether they are the same. Since for each language L in **NP** there is a polynomial p such that for each string $\boldsymbol{w}$ in L there is a verifying choice input $\boldsymbol{c}$ of length $p(|\boldsymbol{w}|)$, a polynomial in the length of $\boldsymbol{w}$, the number of possible choice strings $\boldsymbol{c}$ to be considered in search of a verifying string is at most an exponential in $|\boldsymbol{w}|$. Thus, for every language in **NP** there is an exponential-time algorithm to recognize it.

Despite decades of research, the question of whether **P** is equal to **NP**, denoted $\mathbf{P} \overset{?}{=} \mathbf{NP}$, remains open. It is one of the great outstanding questions of computer science today. The approach taken to this question is to identify **NP**-complete problems (see Section 8.10), the hardest problems in **NP**, and then attempt to determine problems whether or not such problems are in **P**. TSP is one of these **NP**-complete problems.

3.8 Universality of the Turing Machine

We show the existence of a universal Turing machine in two senses. On the one hand, we show that there is a Turing machine that can simulate any RAM computation. Since every Turing machine can be simulated by the RAM, the Turing machine simulating a RAM is universal for the set of all Turing machines.

Also, because there is a Turing machine that can simulate any RAM computation, every RAM program can be simulated on this Turing machine. Since it is not hard to see that every Turing machine can be described by a RAM program (see Problem 3.29), it follows that the RAM programs are exactly the programs computed by Turing machines. Consequently, the RAM is also universal.

The following theorem demonstrates that RAM computations can be simulated by Turing-machine computations and vice versa when each operates with bounded memory. Note that all halting computations are bounded-memory computations. A direct proof of the existence of a universal Turing machine is given in Section 5.5.

THEOREM 3.8.1 *Let $S = mb$ and $m \geq b$. Then for every m-word, b-bit Turing machine M_{TM} (with storage capacity S) there is an $O(m)$-word, b-bit RAM that simulates a time T computation of M_{TM} in time $O(T)$ and storage $O(S)$. Similarly, for every m-word, b-bit RAM M_{RAM} there is an $O((m/b)\log m)$-word, $O(b)$-bit Turing machine that simulates a T-time, S-storage computation of M_{RAM} in time $O(ST\log^2 S)$ and storage $O(S\log S)$.*

Proof We begin by describing a RAM that simulates a TM. Consider a b-bit RAM program to simulate an m-word, b-bit TM. As shown in Theorem 3.4.1, a RAM program can be written to simulate one step of an FSM. Since a TM control unit is an FSM, it suffices to exhibit a RAM program to simulate a tape unit (also an FSM); this is straightforward, as is combining the two programs. If the RAM has storage capacity proportional to that of the TM, then the RAM need only record with one additional word the position of the tape head. This word, which can be held in a RAM register, is incremented or decremented as the head moves. The resulting program runs in time proportional to the running time of the TM.

We now describe a b^*-bit TM that simulates a RAM, where $b^* = \lceil \log m \rceil + b + c$ for some constant c, an assumption we examine later. Let RAM words and their corresponding addresses be placed in individual cells on the tape of the TM, as suggested in Fig. 3.24. Let the address **addr** of the RAM CPU program counter be placed on the tape of the TM to the left, as suggested by the shading in the figure. (It is usually assumed that, unlike the RAM, the TM holds words of size no larger than $O(b)$ in its control unit.) The TM simulates a RAM by simulating the RAM fetch-and-execute cycle. This means it fetches a word at

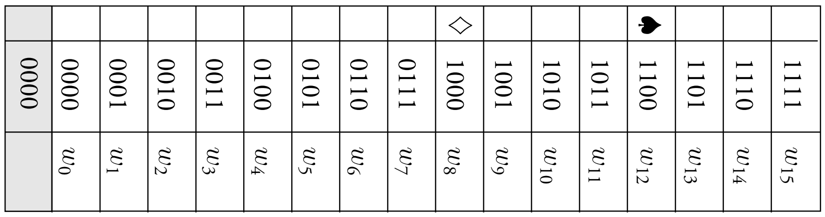

0000
◇
♠
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 w_{10} w_{11} w_{12} w_{13} w_{14} w_{15}

Figure 3.24 Organization of a tape unit to simulate a RAM. Each RAM memory word w_j is accompanied by its address j in binary.

address **addr** in the simulated RAM memory unit, interprets it as an instruction, and then executes the instruction (which might require a few additional accesses to the memory unit to read or write data). We return to the simulation of the RAM CPU after we examine the simulation of the RAM memory unit.

The TM can find a word at location **addr** as follows. It reads the most significant bit of **addr** and moves right on its tape until it finds the first word with this most significant bit. It leaves a marker at this location. (The symbol $\diamondsuit$ in Fig. 3.24 identifies the first place a marker is left.) It then returns to the left-hand end of the tape and obtains the next most significant bit of **addr**. It moves back to the marker $\diamondsuit$ and then carries this marker forward to the next address containing the next most significant bit (identified by the marker $\spadesuit$ in Fig. 3.24). This process is repeated until all bits of **addr** have been visited, at which point the word at location **addr** in the simulated RAM is found. Since m tape unit cells are used in this simulation, at most $O(m \log m)$ TM steps are taken for this purpose.

The TM must also simulate internal RAM CPU computations. Each addition, subtraction, and comparison of b-bit words can be done by the TM control unit in a constant number of steps, as can the logical vector operations. (For simplicity, we assume that the RAM does not use its I/O registers. To simulate these operations, either other tapes would be used or space would be reserved on the single tape to hold input and output words.) The jump instructions as well as the incrementing of the program counter require moving and incrementing $\lceil \log m \rceil$-bit addresses. These cannot be simulated by the TM control unit in a constant number of steps since it can only operate on b-bit words. Instead, they are simulated on the tape by moving addresses in b-bit blocks. If two tape cells are separated by $q-1$ cells, $2q$ steps are necessary to move each block of b bits from the first cell to the second. Thus, a full address can be moved in $2q\lceil \lceil \log m \rceil / b \rceil$ steps. An address can also be incremented using ripple addition in $\lceil \lceil \log m \rceil / b \rceil$ steps using operations on b-bit words, since the blocks of an address are contiguous. (See Section 2.7 for a discussion of ripple addition.) Thus, both of these address-manipulation operations can be done in at most $O(m\lceil \lceil \log m \rceil / b \rceil)$ steps, since no two words are separated by more than $O(m)$ cells.

Now consider the general case of a TM with word size comparable to that of the RAM, that is, a size too small to hold an address as well as a word. In particular, consider a TM with $\hat{b}$-bit tape alphabet where $\hat{b} = cb$, $c > 1$ a constant. In this case, we divide addresses into $\left\lceil \lceil \log m \rceil / \hat{b} \right\rceil$ $\hat{b}$-bit words and place these words in locations that precede the value of the RAM word at this address, as suggested in Fig. 3.40. We also place the address **addr** at the beginning of the tape in the same number of tape words. A total of $O((m/b)(\log m))$ $O(b)$-bit words are used to store all this data. Now assume that the TM can carry the contents of a $\hat{b}$-bit word in its control unit. Then, as shown in Problem 3.26, the extra symbols in the TM's tape alphabet can be used as markers to find a word with a given address in at most $O((m/b)(\log^2 m))$ TM steps using storage $O((m/b) \log m)$. Hence each RAM memory access translates into $O((m/b)(\log^2 m))$ TM steps on this machine.

Simulation of the CPU on this machine is straightforward. Again, each addition, subtraction, comparison, and logical vector operation on b-bit words can be done in a constant number of steps. Incrementing of the program counter can also be done in $\lceil \lceil \log m \rceil / b \rceil$ operations since the cells containing this address are contiguous. However, since a jump operation may require moving an address by $O(m)$ cells in the b^*-bit TM, it may now require moving it by $O(m(\log m)/b)$ cells in the $\hat{b}$-bit TM in $O\left(m\left((\log m)/b\right)^2\right)$ steps.

Combining these results, we see that each step of the RAM may require as many as $O((m((\log m)/b)^2)$ steps of the $\hat{b}$-bit TM. This machine uses storage $O((m/b)\log m)$. Since $m = S/b$, the conclusion of the theorem follows. ■

This simulation of a bounded-memory RAM by a Turing machine assumes that the RAM has a fixed number of memory words. Although this may appear to prevent an unbounded-memory TM from simulating an unbounded-memory RAM, this is not the case. If the Turing machine detects that an address contains more than the number of bits currently assumed as the maximum number, it can increase by 1 the number of bits allocated to each memory location and then resume computation. To make this adjustment, it will have to space out the memory words and addresses to make space for the extra bits. (See Problem 3.28.)

Because a Turing machine with no limit on the length of its tape can be simulated by a RAM, this last observation demonstrates the existence of **universal Turing machines**, Turing machines with unbounded memory (but with fixed-size control units and bounded-size tape alphabets) that can simulate arbitrary Turing machines. This matter is also treated in Section 5.5.

Since the RAM can execute RAM programs, the same is true of the Turing machines. As mentioned above, it is not hard to see that every Turing machine can be simulated by a RAM program. (See Problem 3.29.) As a consequence, the RAM programs are exactly the programs that can be computed by a Turing machine.

While the above remarks apply to the one-tape Turing machine, they also apply to all other Turing machine models, such as double-ended and multi-tape Turing machines, because each of these can also be simulated by the one-tape Turing machine. (See Section 5.2.)

3.9 Turing Machine Circuit Simulations

Just as every T-step finite-state machine computation can be simulated by a circuit, so can every T-step Turing machine computation. We give two circuit simulations, a simple one that demonstrates the concept and another more complex one that yields a smaller circuit. We use these two simulations in Sections 3.9.5 and 3.9.6 to establish computational inequalities that must hold for Turing machines. With a different interpretation they provide examples of **P**-complete and **NP**-complete problems. (See also Sections 8.9 and 8.10.) These results illustrate the central role of circuits in theoretical computer science.

3.9.1 A Simple Circuit Simulation of TM Computations

We now design a circuit simulating a computation of a Turing machine M that uses m memory cells and T steps. Since the only difference between a deterministic and nondeterministic Turing machine is the addition of a choice input to the control unit, we design a circuit for a nondeterministic Turing machine.

For deterministic computations, the circuit simulation provides computational inequalities that must be satisfied by computational resources, such as space and time, if a problem is to be solved by M. Such an inequality is stated at the end of this section.

With the proper interpretation, the circuit simulation of a deterministic computation is an instance of a **P**-complete problem, one of the hardest problems in **P** to parallelize. Here **P** is the class of polynomial-time languages. A first **P**-complete problem is stated in the following section. This topic is studied in detail in Section 8.9.

For nondeterministic computations, the circuit simulation produces an instance of an **NP**-complete problem, a hardest problem to solve in **NP**. Here **NP** is the class of languages accepted in polynomial time by a nondeterministic Turing machine. A first **NP**-complete problem is stated in the following section. This topic is studied in detail in Section 8.10.

THEOREM 3.9.1 *Any computation performed by a one-tape Turing machine M, deterministic or nondeterministic, on an input string $\boldsymbol{w}$ in T steps using m b-bit memory cells can be simulated by a circuit $\mathcal{C}_{M,T}$ over the standard complete basis Ω of size and depth $O(ST)$ and $O(T \log S)$, respectively, where $S = mb$ is the storage capacity in bits of M's tape. For the deterministic TM the inputs to this circuit consist of the values of $\boldsymbol{w}$. For the nondeterministic TM the inputs consist of $\boldsymbol{w}$ and the Boolean choice input variables whose values are not set in advance.*

Proof To construct a circuit $\mathcal{C}_{M,T}$ simulating T steps by M is straightforward because M is a finite-state machine now that its storage capacity is limited. We need only extend the construction of Section 3.1.1 and construct a circuit for the next-state and output functions

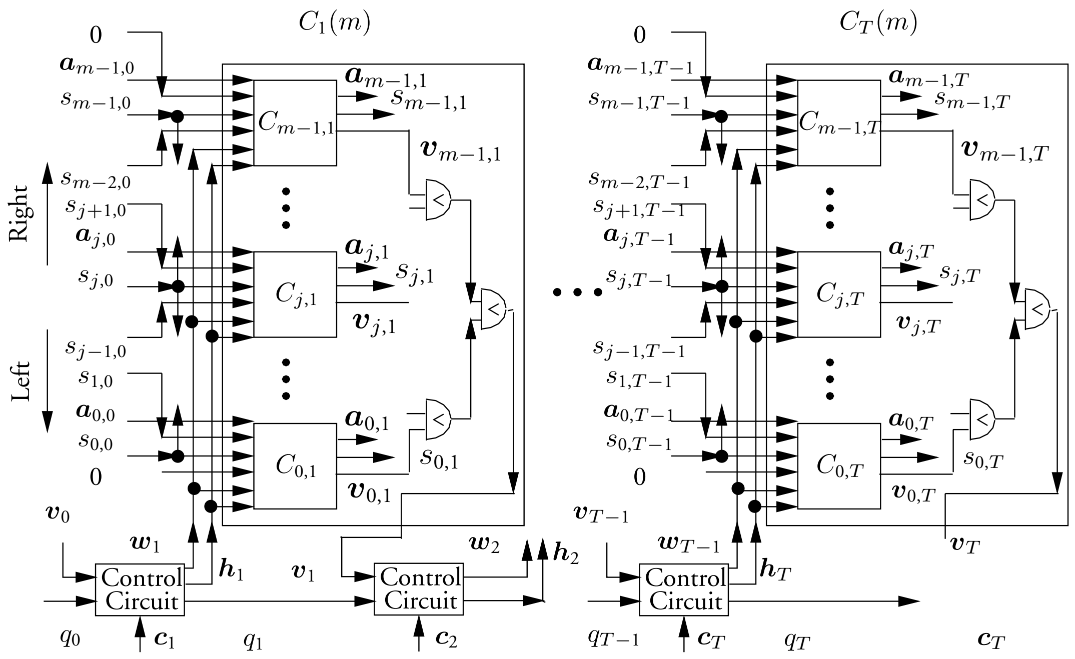


Figure 3.25 The circuit $\mathcal{C}_{M,T}$ simulates an m-cell, T-step computation by a nondeterministic Turing machine M. It contains T copies of M's control unit circuit and T column circuits, $\mathcal{C}_t$, each containing cell circuits $C_{j,t}$, $0 \le j \le m-1$, $1 \le t \le T$, simulating the jth tape cell on the tth time step. q_t and $\boldsymbol{c}_t$ are M's state on the tth step and its tth set of choice variables. Also, $\boldsymbol{a}_{j,t}$ is the value in the jth cell on the tth step, $s_{j,t}$ is 1 if the head is over cell j at the tth time step, and $\boldsymbol{v}_{j,t}$ is $\boldsymbol{a}_{j,t}$ if $s_{j,t} = 1$ and $\mathbf{0}$ otherwise. $\boldsymbol{v}_t$, the vector OR of $\boldsymbol{v}_{j,t}$, $0 \le j \le m-1$, supplies the value under the head to the control unit, which computes head movement commands, $\boldsymbol{h}_t$, and a new word, $\boldsymbol{w}_t$, for the current cell in the next simulated time step. The value of the function computed by M resides on its tape after the Tth step.

of M. As shown in Fig. 3.25, it is convenient to view M as a pair of synchronous FSMs (see Section 3.1.4) and design separate circuits for M's control and tape units. The design of the circuit for the control unit is straightforward since it is an unspecified NFSM. The **tape circuit**, which realizes the next-state and output functions for the tape unit, contains m **cell circuits**, one for each cell on the tape. We denote by $\mathcal{C}_t(m)$, $1 \leq t \leq T$, the tth tape circuit. We begin by constructing a tape circuit and determining its size and depth.

For $0 \leq j \leq m$ and $1 \leq t \leq T$ let $C_{j,t}$ be the jth cell circuit of the tth tape circuit, $\mathcal{C}_t(m)$. $C_{j,t}$ produces the value $\boldsymbol{a}_{j,t}$ contained in the jth cell after the jth step as well as $s_{j,t}$, whose value is 1 if the head is over the jth tape cell after the tth step and 0 otherwise. The value of $\boldsymbol{a}_{j,t}$ is either $\boldsymbol{a}_{j,t-1}$ if $s_{j,t} = 0$ (the head is not over this cell) or $\boldsymbol{w}$ if $s_{j,t} = 1$ (the head is over the cell). Subcircuit SC_2 of Fig. 3.26 performs this computation.

Subcircuit SC_1 in Fig. 3.26 computes $s_{j,t}$ from $s_{j-1,t-1}$, $s_{j,t-1}$, $s_{j+1,t-1}$ and the triple $\boldsymbol{h}_t = (h_t^{-1}, h_t^0, h_t^{+1})$, where $h_t^{-1} = 1$ if the head moves to the next lower-numbered cell, $h_t^{+1} = 1$ if it moves to the next higher-numbered cell, or $h_t^0 = 1$ if it does not move. Thus, $s_{j,t} = 1$ if $s_{j+1,t-1} = 1$ and $h_t^{-1} = 1$, or if $s_{j-1,t-1}$ and $h_t^{+1} = 1$, or if $s_{j,t-1} = 1$ and $h_t^0 = 1$. Otherwise, $s_{j,t} = 0$.

Subcircuit SC_3 of cell circuit $C_{j,t}$ generates the b-bit word $\boldsymbol{v}_{j,t}$ that is used to provide the value under the head on the tth step. $\boldsymbol{v}_{j,t}$ is $\boldsymbol{a}_{j,t}$ if the head is over the jth cell on the

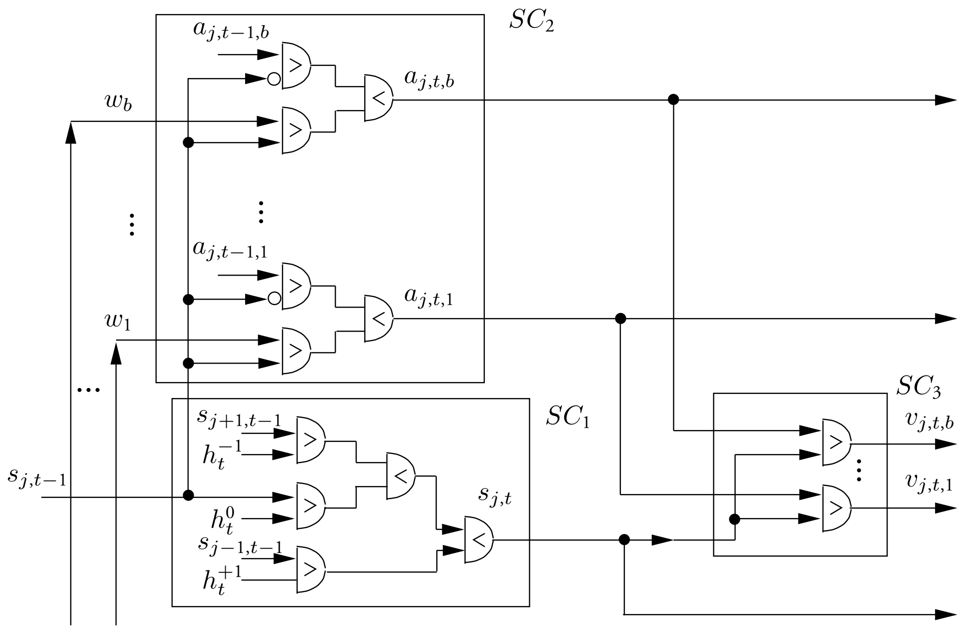


Figure 3.26 The cell circuit $C_{j,t}$ has three components: SC_1, a circuit to compute the new value for the head location bit $s_{j,t}$ from the values of this quantity on the preceding step at neighboring cells and the head movement vector $\boldsymbol{h}_t$, SC_2, a circuit to replace the value in the jth cell on the t step with the input $\boldsymbol{w}$ if the head is over the cell on the $(t-1)$st step ($s_{j,t-1} = 1$), and SC_3, a circuit to produce the new value in the jth cell at the tth step if the head is over this cell ($s_{j,t} = 1$) and the zero vector otherwise. The circuit $C_{j,t}$ has $5(b+1)$ gates and depth 4.

tth step ($s_{j,t} = 1$) and **0** otherwise. The vector-OR of $\boldsymbol{v}_{j,t}$, $0 \leq j \leq m-1$, is formed using the tree circuit shown in Fig. 3.25 to compute the value of the b-bit word $\boldsymbol{v}_t$ under the head after the tth step. (This can be done by b balanced binary OR trees, each with size $m-1$ and depth $\lceil \log_2 m \rceil$.) $\boldsymbol{v}_t$ is supplied to the tth copy of the control unit circuit, which also uses the previous state of the control unit, q_t, and the choice input $\boldsymbol{c}_t$ (a tuple of Boolean variables) to compute the next state, q_{t+1}, the new b-bit word $\boldsymbol{w}_{t+1}$ for the current tape cell, and the head movement command $\boldsymbol{h}_{t+1}$.

Summarizing, it follows that the tth tape circuit, $\mathcal{C}_t(m)$, uses $O(S)$ gates (here $S = mb$) and has depth $O(\log S/b)$.

Let C_{control} and D_{control} be the size and depth of the circuit simulating the control unit. It follows that the circuit simulating T computation steps by a Turing machine M has $T\, C_{\text{control}}$ gates in the T copies of the control unit and $O(ST)$ gates in the tape circuits for a total of $O(ST)$ gates. Since the longest path through the circuit of Fig. 3.26 passes through each control and tape circuit, the depth of this circuit is $O(T(D_{\text{control}} + \log S/b)) = O(T \log S)$.

The simulation of M is completed by placing the head over the zeroth cell by letting $s_{0,0} = 1$ and $s_{j,0} = 0$ for $j \neq 0$. The inputs to M are fixed by setting $\boldsymbol{a}_{j,0} = w_j$ for $0 \leq j \leq n-1$ and to the blank symbol for $j \geq n$. Finally, $\boldsymbol{v}_0$ is set equal to $\boldsymbol{a}_{j,0}$, the value under the head at the start of the computation. The choice inputs are sets of Boolean variables under the control of an outside agent and are treated as variables of the circuit simulating the Turing machine M. ■

We now give two interpretations of the above simulation. The first establishes that the circuit complexity for a function provides a lower bound to the time required by a computation on a Turing machine. The second provides instances of problems that are **P**-complete and **NP**-complete.

3.9.2 Computational Inequalities for Turing Machines

When the simulation of Theorem 3.9.1 is specialized to a deterministic Turing machine M, a circuit is constructed that computes the function f computed by M in T steps with S bits of memory. It follows that $C_\Omega(f)$ and $D_\Omega(f)$ cannot be larger than those given in this theorem, since this circuit also computes f. From this observation we have the following computational inequalities.

THEOREM 3.9.2 *The function f computed by an m-word, b-bit one-tape Turing machine in T steps can also be computed by a circuit whose size and depth satisfy the following bounds over any complete basis Ω, where $S = mb$ is the storage capacity used by this machine:*

$$C_\Omega(f) = O(ST)$$
$$D_\Omega(f) = O(T \log S)$$

Since $S = O(T)$ (at most $T + 1$ cells can be visited in T steps), we have the following corollary. It demonstrates that the time T to compute a function f with a Turing machine is at least the square root of its circuit size. As a consequence, circuit size complexity can be used to derive lower bounds on computation time on Turing machines.

COROLLARY 3.9.1 *Let the function f be computed by an m-word, b-bit one-tape Turing machine in T steps, b fixed. Then, over any complete basis Ω the following inequality must hold:*

$$C_\Omega(f) = O\left(T^2\right)$$

There is no loss in assuming that a language L is a set of strings over a binary alphabet; that is, $L \subseteq \mathcal{B}^*$. As explained in Section 1.2.3, a language can be defined by a family $\{f_1, f_2, f_3, \dots\}$ of characteristic (Boolean) functions, $f_n : \mathcal{B}^n \mapsto \mathcal{B}$, where a string $\boldsymbol{w}$ of length n is in L if and only if $f_n(\boldsymbol{w}) = 1$.

Theorem 3.9.2 not only establishes a clear connection between Turing time complexity and circuit size complexity, but it also provides a potential means to resolve the question $\mathbf{P} \overset{?}{=} \mathbf{NP}$ of whether **P** and **NP** are equal or not. Circuit complexity is currently believed to be the most promising tool to examine this question. (See Chapter 9.)

3.9.3 Reductions from Turing to Circuit Computations

As shown in Theorem 3.9.1, a circuit $\mathcal{C}_{M,T}$ can be constructed that simulates a time- and space-bounded computation by either a deterministic or a nondeterministic Turing machine M. If M is deterministic and accepts the binary input string $\boldsymbol{w}$, then $\mathcal{C}_{M,T}$ has value 1 when supplied with the value of $\boldsymbol{w}$. If M is nondeterministic and accepts the binary input string $\boldsymbol{w}$, then for some values of the binary choice variables $\boldsymbol{c}$, $\mathcal{C}_{M,T}$ on inputs $\boldsymbol{w}$ and $\boldsymbol{c}$ has value 1.

The language of strings describing circuits with fixed inputs whose value on these inputs is 1 is called CIRCUIT VALUE. When the circuits also have variable inputs whose values can be chosen so that the circuits have value 1, the language of strings describing such circuits is called CIRCUIT SAT. (See Section 3.9.6.) The languages CIRCUIT VALUE and CIRCUIT SAT are examples of **P**-complete and **NP**-complete languages, respectively.

The **P**-complete and **NP**-complete languages play an important role in complexity theory: they are prototypical hard languages. The **P**-complete languages can all be recognized in polynomial time on serial machines, but it is not known how to recognize them on parallel machines in time that is a polynomial in the logarithm of the length of strings (this is called **poly-logarithmic time**), which should be possible if they are parallelizable. The **NP**-complete languages can be recognized in exponential time on deterministic serial machines, but it is not known how to recognize them in polynomial time on such machines. Many important problems have been shown to be **P**-complete or **NP**-complete.

Because so much effort has been expended without success in trying to show that the **NP**-complete (**P**-complete) languages can be solved serially (in parallel) in polynomial (poly-logarithmic) time, it is generally believed they cannot. Thus, showing that a problem is **NP**-complete (**P**-complete) is considered good evidence that a problem is hard to solve serially (in parallel).

To obtain such results, we exhibit a program that writes the description of the circuit $\mathcal{C}_{M,T}$ from a description of the TM M and the values written initially on its tape. The time and space needed by this program are used to classify languages and, in particular, to identify the **P**-complete and **NP**-complete languages.

The simple program $\mathcal{P}$ shown schematically in Fig. 3.27 writes a description of the circuit $\mathcal{C}_{M,T}$ of Fig. 3.25, which is deterministic or nondeterministic depending on the nature of M. (Textual descriptions of circuits are given in Section 2.2. Also see Problem 3.8.) The first loop of this program reads the value of ith input letter w_i of the string $\boldsymbol{w}$ written on

```
**for** $i$ := 0 **to** $n - 1$
    READ_VALUE($w_i$)
    WRITE_INPUT($i, w_i$)
**for** $j$ := $n$ **to** $m - 1$
    WRITE_INPUT($j, \beta$)
**for** $t$ := 1 **to** $T$
    WRITE_CONTROL_UNIT($t, \boldsymbol{c}_t$)
    WRITE_OR($t$, $m$)
    **for** $j$ := 0 **to** $m - 1$
        WRITE_CELL_CIRCUIT($j, t$)
```

Figure 3.27 A program $\mathcal{P}$ to write the description of a circuit $\mathcal{C}_{M,T}$ that simulates T steps of a nondeterministic Turing machine M and uses m memory words. It reads the n inputs supplied to M, after which it writes the input steps of a straight-line program that reads these n inputs as well as $m - n$ blanks β into the first copy of a tape unit. It then writes the remaining steps of a straight-line program consisting of descriptions of the T copies of the control unit and the mT cell circuits simulating the T copies of the tape unit.

the input tape of T, after which it writes a fragment of a straight-line program containing the value of w_i. The second loop sets the remaining initial values of cells to the blank symbol β. The third outer loop writes a straight-line program for the control unit using the procedure WRITE_CONTROL_UNIT that has as arguments t, the index of the current time step, and $\boldsymbol{c}_t$, the tuple of Boolean choice input variables for the tth step. These choice variables are not used if M is deterministic. In addition, this loop uses the procedure WRITE_OR to write a straight-line program for the vector OR circuit that forms the contents $\boldsymbol{v}_t$ of the cell under the head after the tth step. Its inner loop uses the procedure WRITE_CELL_CIRCUIT with parameters j and t to write a straight-line program for the jth cell circuit in the tth tape.

The program $\mathcal{P}$ given in Fig. 3.27 is economical in its use of space and time, as we show. Consider a language L in **P**; that is, for L there is a deterministic Turing machine M_L and a polynomial $p(n)$ such that on an input string $\boldsymbol{w}$ of length n, M_L halts in $T = p(n)$ steps. It accepts $\boldsymbol{w}$ if it is in L and rejects it otherwise. Since $\mathcal{P}$ uses space logarithmic in the values of n and T and $T = p(n)$, $\mathcal{P}$ uses space logarithmic in n. (For example, if $p(n) = n^6$, $\log_2 p(n) = 6 \log_2 n = O(\log n)$.) Such programs are called **log-space programs**.

We show in Theorem 8.8.1 that the composition of two log-space programs is a log-space program, a non-obvious result. However, it is straightforward to show that the composition of two polynomial-time programs is a polynomial-time program. (See Problems 3.2 and 8.19.) Since $\mathcal{P}$'s inner and outer loops each execute a polynomial number of steps, it follows that $\mathcal{P}$ is a polynomial-time program.

If M is nondeterministic, $\mathcal{P}$ continues to be a log-space, polynomial-time program. The only difference is that it writes a circuit description containing references to choice variables whose values are not specified in advance. We state these observations in the form of a theorem.

THEOREM 3.9.3 *Let $L \in$ **P** ($L \in$ **NP**). Then for each string $\boldsymbol{w} \in \Gamma^*$ a deterministic (nondeterministic) circuit $\mathcal{C}_{M,T}$ can be constructed by a program in logarithmic space and polynomial time in $n = |\boldsymbol{w}|$, the length of $\boldsymbol{w}$, such that the output of $\mathcal{C}_{M,T}$, the value in the first tape cell, is (can be) assigned value 1 (for some values of the choice inputs) if $\boldsymbol{w} \in L$ and* 0 *if $\boldsymbol{w} \notin L$.*

The program of Fig. 3.27 provides a translation (or **reduction**) from any language in **NP** (or **P**) to a language that we later show is a hardest language in **NP** (or **P**).

We now use Theorem 3.9.3 and the above facts to give a brief introduction to the **P**-complete and **NP**-complete languages, which are discussed in more detail in Chapter 8.

3.9.4 Definitions of **P**-Complete and **NP**-Complete Languages

In this section we identify languages that are hardest in the classes **P** and **NP**. A language L_0 is **hardest** in one of these classes if a) L_0 is itself in the class and b) for every language L in the class, a test for the membership of a string $\boldsymbol{w}$ in L can be constructed by translating $\boldsymbol{w}$ with an algorithm to a string $\boldsymbol{v}$ and testing for membership of $\boldsymbol{v}$ in L_0. If the class is **P**, the algorithm must use at most space logarithmic in the length of $\boldsymbol{w}$, whereas in the case of **NP**, the algorithm must use time at most a polynomial in the length of $\boldsymbol{w}$. Such a language L_0 is said to be a **complete language** for this complexity class. We begin by defining the **P**-complete languages.

DEFINITION 3.9.1 *A language $L \subseteq \mathcal{B}^*$ is* **P-complete** *if it is in* **P** *and if for every language $L_0 \subseteq \mathcal{B}^*$ in* **P***, there is a log-space deterministic program that translates each $\boldsymbol{w} \in \mathcal{B}^*$ into a string $\boldsymbol{w}' \in \mathcal{B}^*$ such that $\boldsymbol{w} \in L_0$ if and only if $\boldsymbol{w}' \in L$.*

The **NP**-complete languages have a similar definition. However, instead of requiring that the translation be log-space, we ask only that it be polynomial-time. It is not known whether all polynomial-time computations can be done in logarithmic space.

DEFINITION 3.9.2 *A language $L \subseteq \mathcal{B}^*$ is* **NP-complete** *if it is in* **NP** *and if for every language $L_0 \subseteq \mathcal{B}^*$ in* **NP***, there is a polynomial-time deterministic program that translates each $\boldsymbol{w} \in \mathcal{B}^*$ into a string $\boldsymbol{w}' \in \mathcal{B}^*$ such that $\boldsymbol{w} \in L_0$ if and only if $\boldsymbol{w}' \in L$.*

Space precludes our explaining the important role of the **P**-complete languages. We simply report that these languages are the hardest languages to parallelize and refer the reader to Sections 8.9 and 8.14.2. However, we do explain the importance of the **NP**-complete languages.

As the following theorem states, if an **NP**-complete language is in **P**; that is, if membership of a string in an **NP**-complete language can be determined in polynomial time, then the same can be done for every language in **NP**; that is, **P** and **NP** are the same class of languages. Since decades of research have failed to show that **P** $=$ **NP**, a determination that a problem is **NP**-complete is a testimonial to but not a proof of its difficulty.

THEOREM 3.9.4 *If an* **NP***-complete language is in* **P***, then* **P** $=$ **NP***.*

Proof Let L be **NP**-complete and let L_0 be an arbitrary language in **NP**. Because L is **NP**-complete, there is a polynomial-time program that translates an arbitrary string $\boldsymbol{w}$ into a string $\boldsymbol{w}'$ such that $\boldsymbol{w}' \in L$ if and only if $\boldsymbol{w} \in L_0$. If $L \in$ **P**, then testing of membership of strings in L_0 can be done in polynomial time in the length of the string. It follows that there exists a polynomial-time program to determine membership of a string in L_0. Thus, every language in **NP** is also in **P**. ■

3.9.5 Reductions to **P**-Complete Languages

We now formally define CIRCUIT VALUE, our first **P**-complete language.

CIRCUIT VALUE
Instance: A circuit description with fixed values for its input variables and a designated output gate.
Answer: "Yes" if the output of the circuit has value 1.

THEOREM 3.9.5 *The language* CIRCUIT VALUE *is* **P***-complete.*

Proof To show that CIRCUIT VALUE is **P**-complete, we must show that it is in **P** and that every language in **P** can be translated to it by a log-space program. We have already shown the second half of the proof in Theorem 3.9.1. We need only show the first half, which follows from a simple analysis of the obvious program. Since a circuit is a graph of a straight-line program, each step depends on steps that precede it. (Such a program can be produced by a pre-order traversal of the circuit starting with its output vertex.) Now scan the straight-line program and evaluate and store in an array the value of each step. Successive steps access this array to find their arguments. Thus, one pass over the straight-line program suffices to evaluate it; the evaluating program runs in linear time in the length of the circuit description. Hence CIRCUIT VALUE is in **P**. ■

When we wish to show that a new language L_1 is **P**-complete, we first show that it is in **P**. Then we show that every language $L \in \mathbf{P}$ can be translated to it in logarithmic space; that is, for each string $\boldsymbol{w}$, there is an algorithm that uses temporary space $O(\log |\boldsymbol{w}|)$ (as does the program in Fig. 3.27) that translates $\boldsymbol{w}$ into a string $\boldsymbol{v}$ such that $\boldsymbol{w}$ is in L if and only if $\boldsymbol{v}$ is in L_1. (This is called a **log-space reduction**. See Section 8.5 for a discussion of temporary space.)

If we have already shown that a language L_0 is **P**-complete, we ask whether we can save work by using this fact to show that another language, L_1, in **P** is **P**-complete. This is possible because the composition of two deterministic log-space algorithms is another log-space algorithm, as shown in Theorem 8.8.1. Thus, if we can translate L_0 into L_1 with a log-space algorithm, then every language in **P** can be translated into L_1 by a log-space reduction. (This idea is suggested in Fig. 3.28.) Hence, the task of showing L_1 to be **P**-complete is reduced to showing that L_1 is in **P** and that L_0, which is **P**-complete, can be translated to L_1 by a log-space algorithm. Many **P**-complete languages are exhibited in Section 8.9.

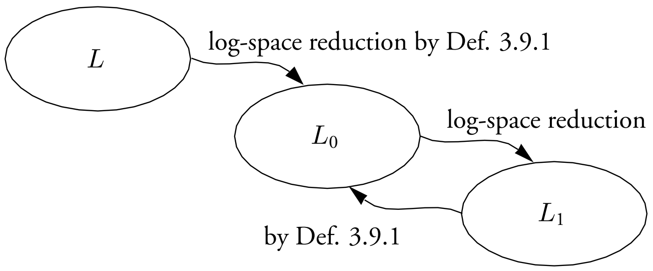


Figure 3.28 A language L_0 is shown **P**-complete by demonstrating that L_0 is in **P** and that every language L in **P** can be translated to it in logarithmic space. A new language L_1 is shown **P**-complete by showing that it is in **P** and that L_0 can be translated to it in log-space. Since L can be L_1, L_1 can also be translated to L_0 in log-space.

3.9.6 Reductions to **NP**-Complete Languages

Our first **NP**-complete language is CIRCUIT SAT, a language closely related to CIRCUIT VALUE.

> CIRCUIT SAT
> *Instance:* A circuit description with n input variables $\{x_1, x_2, \dots, x_n\}$ for some integer n and a designated output gate.
> *Answer:* "Yes" if there is an assignment of values to the variables such that the output of the circuit has value 1.

THEOREM 3.9.6 *The language* CIRCUIT SAT *is* **NP***-complete.*

Proof To show that CIRCUIT SAT is **NP**-complete, we must show that it is in **NP** and that every language in **NP** can be translated to it by a polynomial-time program. We have already shown the second half of the proof in Theorem 3.9.1. We need only show the first half. As discussed in the proof of Theorem 3.9.5, each circuit can be organized so that all steps on which a given step depends precede it. We assume that a string in CIRCUIT SAT meets this condition. Design an NTM which on such a string uses choice inputs to assign values to each of the variables in the string. Then invoke the program described in the proof of Theorem 3.9.5 to evaluate the circuit. For some assignment to the variables $x_1, x_2, \dots, x_n$, this nondeterministic program can accept each string in CIRCUIT SAT but no string not in CIRCUIT SAT. It follows that CIRCUIT SAT is in **NP**. ■

The model used to show that a language is **P**-complete directly parallels the model used to show that a language L_1 is **NP**-complete. We first show that L_1 is in **NP** and then show that every language $L \in$ **NP** can be translated to it in polynomial time. That is, we show that there is a polynomial p and algorithm that on inputs of length n runs in time $p(n)$, and that for each string $\boldsymbol{w}$ the algorithm translates $\boldsymbol{w}$ into a string $\boldsymbol{v}$ such that $\boldsymbol{w}$ is in L if and only if $\boldsymbol{v}$ is in L_1. (This is called a **polynomial-time reduction**.) Since any algorithm that uses log-space (as does the program in Fig. 3.27) runs in polynomial time (see Theorem 8.5.8), a log-space reduction can be used in lieu of a polynomial-time reduction.

If we have already shown that a language L_0 is **NP**-complete, we can show that another language, L_1, in **NP** is **NP**-complete by translating L_0 into L_1 with a polynomial-time algorithm. Since the composition of two polynomial-time algorithms is another polynomial-time algorithm (see Problem 3.2), every language in **NP** can be translated in polynomial time into L_1 and L_1 is **NP**-complete. The diagram shown in Fig. 3.28 applies when the reductions are polynomial-time and the languages are members of **NP** instead of **P**. Many **NP**-complete languages are exhibited in Section 8.10.

We apply this idea to show that SATISFIABILITY is **NP**-complete. Strings in this language consist of strings representing the POSE (product-of-sums expansion) of a Boolean function. Thus, they consist of clauses containing literals (a variable or its negation) with the property that for some value of the variables at least one literal in each clause is satisfied.

> SATISFIABILITY
> *Instance:* A set of **literals** $X = \{x_1, \overline{x}_1, x_2, \overline{x}_2, \dots, x_n, \overline{x}_n\}$ and a sequence of **clauses** $C = (c_1, c_2, \dots, c_m)$ where each clause c_i is a subset of X.
> *Answer:* "Yes" if there is a (satisfying) assignment of values for the variables $\{x_1, x_2, \dots, x_n\}$ over the set $\mathcal{B}$ such that each clause has at least one literal whose value is 1.

THEOREM 3.9.7 *SATISFIABILITY is* **NP**-*complete.*

Proof SATISFIABILITY is in **NP** because for each string $\boldsymbol{w}$ in this language there is a satisfying assignment for its variables that can be verified by a polynomial-time program. We sketch a deterministic RAM program for this purpose. This program reads as many choice variables as there are variables in $\boldsymbol{w}$ and stores them in memory locations. It then evaluates each literal in each clause in $\boldsymbol{w}$ and declares this string satisfied if all clauses evaluate to 1. This program, which runs in time linear in the length of $\boldsymbol{w}$, can be converted to a Turing-machine program using the construction of Theorem 3.8.1. This program executes in a time cubic in the time of the original program on the RAM. We now show that every language in **NP** can be reduced to SATISFIABILITY via a polynomial-time program.

Given an instance of CIRCUIT SAT, as we now show, we can convert the circuit description, a straight-line program (see Section 2.2), into an instance of SATISFIABILITY such that the former is a "yes" instance of CIRCUIT SAT if and only if the latter is a "yes" instance of SATISFIABILITY. Shown below are the different steps of a straight-line program and the clauses used to replace them in constructing an instance of SATISFIABILITY. A deterministic TM can be designed to make these translations in time proportional to the length of the circuit description. Clearly the instance of SATISFIABILITY that it produces is a satisfiable instance if and only if the instance of CIRCUIT SAT is satisfiable.

Step Type				*Corresponding Clauses*		
$(i$	READ	$x)$		$(\overline{g}_i \vee x)$	$(g_i \vee \overline{x})$	
$(i$	NOT	$j)$		$(\overline{g}_i \vee \overline{g}_j)$	$(g_i \vee g_j)$	
$(i$	OR	j	$k)$	$(g_i \vee \overline{g}_j)$	$(g_i \vee \overline{g}_k)$	$(\overline{g}_i \vee g_j \vee g_k)$
$(i$	AND	j	$k)$	$(\overline{g}_i \vee g_j)$	$(\overline{g}_i \vee g_k)$	$(g_i \vee \overline{g}_j \vee \overline{g}_k)$
$(i$	OUTPUT	$j)$		(g_j)		

For each gate type it is easy to see that each of the corresponding clauses is satisfiable only for those gate and argument values that are consistent with the type of gate. For example, a NOT gate with input g_j has value $g_i = 1$ when g_j has value 0 and $g_i = 0$ when g_j has value 1. In both cases, both of the clauses $(\overline{g_i} \vee \overline{g_j})$ and $(g_i \vee g_j)$ are satisfied. However, if g_i is equal to g_j, at least one of the clauses is not satisfied. Similarly, if g_i is the AND of g_j and g_k, then examining all eight values for the triple (g_i, g_j, g_k) shows that only when g_i is the AND of g_j and g_k are all three clauses satisfied. The verification of the above statements is left as a problem for the reader. (See Problem 3.36.) Since the output clause (g_j) is true if and only if the circuit output has value 1, it follows that the set of clauses are all satisfiable if and only if the circuit in question has value 1; that is, it is satisfiable.

Given an instance of CIRCUIT SAT, clearly a deterministic TM can produce the clauses corresponding to each gate using a temporary storage space that is logarithmic in the length of the circuit description because it need deal only with integers that are linear in the length of the input. Thus, each instance of CIRCUIT SAT can be translated into an instance of SATISFIABILITY in a number of steps polynomial in the length of the instance of CIRCUIT SAT. Since it is also in **NP**, it is **NP**-complete. ■

3.9.7 An Efficient Circuit Simulation of TM Computations*

In this section we construct a much more efficient circuit of size $O(Tb \log m)$ that simulates a computation done in T steps by an m-word, b-bit one-tape TM. A similar result on circuit depth is shown.

THEOREM 3.9.8 *Let an m-word, b-bit Turing machine compute in T steps the function f, a projection of $f_{\text{TM}}^{(T,m,b)}$, the function computed by the TM in T steps. Then the following bounds on the size and depth of f over the complete basis Ω must be satisfied:*

$$C_\Omega(f) = O\left(T(\log[\min(bT, S)]\right)$$
$$D_\Omega(f) = O(T)$$

Proof The circuit $\mathcal{C}_{M,T}$ described in Theorem 3.9.1 has size proportional to $O(ST)$, where $S = mb$. We now show that a circuit computing the same function, $N(1, T, m)$, can be constructed whose size is $O\left(T(\log[\min(bT, S)]\right)$. This new circuit is obtained by more efficiently simulating the tape unit portion of a Turing machine. We observe that if the head never reaches a cell, the cell circuit of Fig. 3.26 can be replaced by wires that pass its inputs to its output. It follows that the number of gates can be reduced if we keep the head near the center of a simulated tape by "centering" it periodically. This is the basis for the circuit constructed here.

It simplifies the design of $N(1, T, m)$ to assume that the tape unit has cells indexed from $-m$ to m. Since the head is initially placed over the cell indexed with 0, it is over the middle cell of the tape unit. (The control unit is designed so that the head never enters cells whose index is negative.) We construct $N(1, T, m)$ from a subcircuit $N(c, s, n)$ that simulates s steps of a tape unit containing n b-bit cells under the assumption that the tape head is initially over one of the middle c cells where c and n are odd. Here $n \geq c + 2s$, so that in s steps the head cannot move from one of the middle c cells to positions that are not simulated by this circuit. Let $C(c, s, n)$ and $D(c, s, n)$ be the size and depth of $N(c, s, n)$.

As base cases for our recursive construction of $N(c, s, n)$, consider the circuits $N(1, 1, 3)$ and $N(3, 1, 5)$. They can be constructed from copies of the tape circuit $\mathcal{C}_t(3)$ and $\mathcal{C}_t(5)$ since they simulate one step of tape units containing three and five cells, respectively. In fact, these circuits can be simplified by removing unused gates. Without simplification $\mathcal{C}_t(n)$ contains $5(b + 1)$ gates in each of the n cell circuits (see Fig. 3.26) as well as $(n - 1)b$ gates in the vector OR circuit, for a total of at most $6n(b + 1)$ gates. It has depth $4 + \lceil \log_2 n \rceil$. Thus, $N(1, 1, 3)$ and $N(3, 1, 5)$ each can be realized with $O(b)$ gates and depth $O(1)$.

We now give a recursive construction of a circuit that simulates a tape unit. The $N(1, 2q, 4q + 1)$ circuit simulates $2q$ steps of the tape unit when the head is over the middle cell. It can be decomposed into an $N(1, q, 2q + 1)$ circuit simulating the first q steps and an $N(2q + 1, q, 4q + 1)$ circuit simulating the second q steps, as shown in Fig. 3.29. In the $N(1, q, 2q + 1)$ circuit, the head may move from the middle position to any one of $2q + 1$ positions in q steps, which requires that $2q + 1$ of the inputs be supplied to it. In the $N(2q + 1, q, 4q + 1)$ circuit, the head starts in the middle $2q + 1$ positions and may move to any one of $4q + 1$ middle positions in the next q steps, which requires that $4q + 1$ inputs be supplied to it. The size and depth of our $N(1, 2q, 4q + 1)$ circuit satisfy the following recurrences:

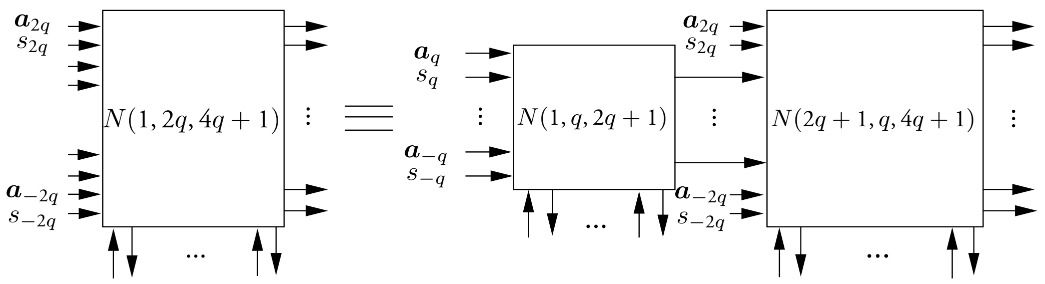


Figure 3.29 A decomposition of an $N(1, 2q, 4q + 1)$ circuit.

$$\begin{aligned} C(1,2q,4q+1) &\leq C(1,q,2q+1) + C(2q+1,q,4q+1) \\ D(1,2q,4q+1) &\leq D(1,q,2q+1) + D(2q+1,q,4q+1) \end{aligned} \tag{3.4}$$

When the number of tape cells is bounded, the above construction and recurrences can be modified. Let $m = 2^p$ be the maximum number of cells used during a T-step computation by the TM. We simulate this computation by placing the head over the middle of a tape with $2m + 1$ cells. It follows that at least m steps are needed to reach each of the reachable cells. Thus, if $T \leq m$, we can simulate the computation with an $N(1, T, 2T + 1)$ circuit. If $T \geq m$, we can simulate the first m steps with an $N(1, m, 2m + 1)$ circuit and the remaining $T - m$ steps with $\lceil (T - m)/m \rceil$ copies of an $N(2m+1, m, 4m+1)$ circuit. This follows because at the end of the first m steps the head is over the middle $2m + 1$ of $4m + 1$ cells (of which only $2m + 1$ are used) and remains in this region after m steps due to the limitation on the number of cells used by the TM.

From the above discussion we have the following bounds on the size $C(T, m)$ and depth $D(T, m)$ of a simulating circuit for a T-step, m-word TM computation:

$$C(T,m) \leq \begin{cases} C(1,T,2T+1) & T \leq m \\ C(1,m,2m+1) + \left(\left\lceil \frac{T}{m} \right\rceil - 1\right) C(2m+1,m,4m+1) & T \geq m \end{cases}$$
$$\tag{3.5}$$
$$D(T,m) \leq \begin{cases} D(1,T,2T+1) & T \leq m \\ D(1,m,2m+1) + \left(\left\lceil \frac{T}{m} \right\rceil - 1\right) D(2m+1,m,4m+1) & T \geq m \end{cases}$$

We complete the proof of Theorem 3.9.8 by bounding $C(1, 2q, 4q + 1)$, $C(2q + 1, q, 4q + 1)$, $D(1, 2q, 4q + 1)$, and $D(2q + 1, q, 4q + 1)$ appearing in (3.4) and combining them with the bounds of (3.5).

We now give a recursive construction of an $N(2q + 1, q, 4q + 1)$ circuit from which these bounds are derived. Shown in Fig. 3.30 is the recursive decomposition of an $N(4t + 1, 2t, 8t+1)$ circuit in terms of two copies of $N(2t+1, t, 4t+1)$ circuits. The t-centering circuits detect whether the head is in positions $2t, 2t-1, \ldots, 1, 0$ or in positions $-1, \ldots, -2t$. In the former case, this circuit cyclically shifts the $8t + 1$ inputs inputs down by t positions; in the latter, it cyclically shifts them up by t positions. The result is that the head is centered in the middle $2t+1$ positions. The OR of $s_{-1}, \ldots, s_{-2t}$ can be used as a signal to determine

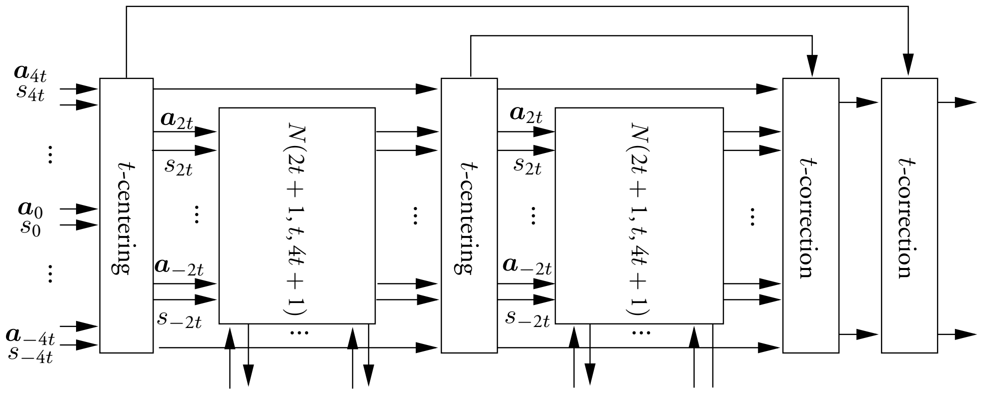


Figure 3.30 A recursive decomposition of $N(4t+1, 2t, 8t+1)$.

which shift to take. After centering, t steps are simulated, the head is centered again, and another t steps are again simulated. Two t-correction circuits cyclically shift the results in directions that are the reverse of the first two shifts. This circuit correctly simulates the tape computation over $2t$ steps and produces an $N(4t+1, 2t, 8t+1)$ circuit.

A t-centering circuit can be realized as a single stage of the cyclic shift circuit described in Section 2.5.2 and shown in Fig. 2.8. A t-correction circuit is just a t-centering circuit in which the shift is in the reverse direction. The four shifting circuits can be realized with $O(tb)$ gates and constant depth. The two OR trees to determine the direction of the shift can be realized with $O(t)$ gates and depth $O(\log t)$. From this discussion we have the following bounds on the size and depth of $N(4t+1, 2t, 8t+1)$:

$$\begin{aligned}
C(4t+1, 2t, 8t+1) &\leq 2C(2t+1, t, 4t+1) + O(bt) \\
C(3, 1, 5) &\leq O(b) \\
D(4t+1, 2t, 8t+1) &\leq 2D(2t+1, t, 4t+1) + 2\lceil \log_2 t \rceil \\
D(3, 1, 5) &\leq O(1)
\end{aligned}$$

We now solve this set of recurrences. Let $\mathcal{C}(k) = C(2t+1, t, 4t+1)$ and $\mathcal{D}(k) = D(2t+1, t, 4t+1)$ when $t = 2^k$. The above bounds translate into the following recurrences:

$$\begin{aligned}
\mathcal{C}(k+1) &\leq 2\mathcal{C}(k) + K_1 2^k + K_2 \\
\mathcal{C}(0) &\leq K_3 \\
\mathcal{D}(k+1) &\leq 2\mathcal{D}(k) + 2k + K_4 \\
\mathcal{D}(0) &\leq K_5
\end{aligned}$$

for constants K_1, K_2, K_3, K_4, and K_5. It is straightforward to show that $\mathcal{C}(k+1)$ and $\mathcal{D}(k+1)$ satisfy the following inequalities:

$$\begin{aligned}
\mathcal{C}(k) &\leq 2^k (K_1 k/2 + K_2 + K_3) - K_2 \\
\mathcal{D}(k) &\leq 2^k (K_5 + K_4 + 2) - 2k - (K_4 + 2)
\end{aligned}$$

We now derive explicit upper bounds to (3.4). Let $\Lambda(k) = C(1, q, 2q+1)$ and $\Delta(k) = D(1, q, 2q+1)$ when $q = 2^k$. Then, the inequalities of (3.4) become the following:

$$\begin{aligned}\Lambda(k+1) &\leq \Lambda(k) + \mathcal{C}(k)\\ \Lambda(0) &\leq K_6\\ \Delta(k+1) &\leq \Delta(k) + \mathcal{D}(k)\\ \Delta(0) &\leq K_7\end{aligned}$$

where $K_6 = C(1, 1, 3) = 7b + 3$ and $K_7 = D(1, 1, 3) = 4$. The solutions to these recurrences are given below.

$$\begin{aligned}\Lambda(k) &\leq \sum_{j=0}^{k-1} \mathcal{C}(j)\\ &= 2^k(K_1k/2 + K_2 + K_3 - K_1) - kK_2 + (K_6 - (K_2 + K_3 - K_1))\\ &= O(k2^k)\\ \Delta(k) &\leq \sum_{j=0}^{k-1} \mathcal{D}(j)\\ &= 2^k(K_5 + K_4 + 2) - k^2 + (1 - (K_4 + 2))k + (K_7 - (K_5 + K_4 + 2))\\ &= O(2^k)\end{aligned}$$

Here we have made use of the identity in Problem 3.1. From (3.5) and (3.6) we establish the result of Theorem 3.9.8. ■

3.10 Design of a Simple CPU

In this section we design an eleven-instruction CPU for a general-purpose computer that has a random-access memory with 2^{12} 16-bit memory words. We use this design to illustrate how a general-purpose computer can be assembled from gates and binary storage devices (flip-flops). The design is purposely kept simple so that basic concepts are made explicit. In practice, however, CPU design can be very complex. Since the CPU is the heart of every computer, a high premium is attached to making them fast. Many clever ideas have been developed for this purpose, almost all of which we must for simplicity ignore here.

Before beginning, we note that a typical complex instruction set (CISC) CPU, one with a rich set of instructions, contains several tens of thousands of gates, while as shown in the previous section, a random-access memory unit has a number of equivalent gates proportional to its memory capacity in bits. (CPUs are often sold with caches, small random-access memory units that add materially to the number of equivalent gates.) The CPUs of reduced instruction set (RISC) computers have many fewer gates. By contrast, a four-megabyte memory has the equivalent of several tens of millions of gates. As a consequence, the size and depth of the next-state and output functions of the random-access memory, δ_{RMEM} and λ_{RMEM}, typically dominate the size and depth of the next-state and output functions, δ_{CPU} and λ_{CPU}, of the CPU, as shown in Theorem 3.6.1.

3.10.1 The Register Set

A CPU is a sequential circuit that repeatedly reads and executes an instruction from its memory in what is known as the fetch-and-execute cycle. (See Sections 3.4 and 3.10.2.) A machine-language program is a set of instructions drawn from the instruction set of the CPU. In our simple CPU each instruction consists of two parts, an **opcode** and an **address**, as shown schematically below.

1 4	5 16
Opcode	Address

Since our computer has eleven instructions, we use a 4-bit opcode, a length sufficient to represent all of them. Twelve bits remain in the 16-bit word, providing addresses for 4,096 16-bit words in a random-access memory.

We let our CPU have eight special registers: the 16-bit **accumulator** (AC), the 12-bit **program counter** (PC), the 4-bit **opcode register** (OPC), the 12-bit **memory address register** (MAR), the 16-bit **memory data register** (MDR), the 16-bit **input register** (INR), the 16-bit **output register** (denoted OUTR), and the **halt register** (HLT). These registers are shown schematically together with the random-access memory in Fig. 3.31.

The program counter PC contains the address from which the next instruction will be fetched. Normally this is the address following the address of the current instruction. However, if some condition is true, such as that the contents of the accumulator AC are zero, the program might place a new address in the PC and jump to this new address. The memory address register MAR contains the address used by the random-access memory to fetch a word. The memory data register MDR contains the word fetched from the memory. The halt register HLT contains the value 0 if the CPU is halted and otherwise contains 1.

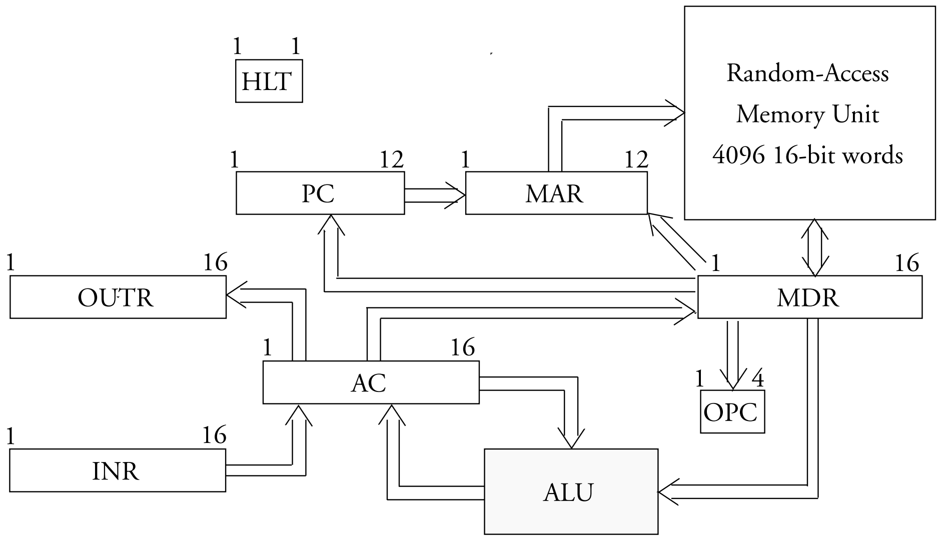


Figure 3.31 Basic registers of the simple CPU and the paths connecting them. Also shown is the arithmetic logic unit (ALU) containing circuits for AND, addition, shifting, and Boolean complement.

3.10.2 The Fetch-and-Execute Cycle

The fetch-and-execute cycle has a fetch portion and an execution portion. The fetch portion is always the same: the instruction whose address is in the PC is fetched into the MDR and the opcode portion of this register is copied into the OPC. At this point the action of the CPU diverges, based on the instruction denoted by the value of the OPC. Suppose, for example, that the OPC denotes a load accumulator instruction. The action required is to copy the word specified by the address part of the instruction into the accumulator. Fig. 3.32 contains a decomposition of the **load accumulator** instruction into eight **microinstructions** executed in six **microcycles**. During each microcycle several microinstructions can be executed concurrently, as shown in the table for the second and fourth microcycles. In Section 3.10.5 we describe implementations of the fetch-and-execute cycle for each of the instructions of our computer.

It is important to note that a realistic CPU must do more than fetch and execute instructions: it must be interruptable by a user or an external device that demands its attention. After fetching and executing an instruction, a CPU typically examines a small set of flip-flops to see if it must break away from the program it is currently executing to handle an **interrupt**, an action equivalent to fetching an instruction associated with the interrupt. This action causes an interrupt routine to be run that responds to the problem associated with the interrupt, after which the CPU returns to the program it was executing when it was interrupted. It can do this by saving the address of the next instruction of this program (the value of the PC) at a special location in memory (such as address 0). After handling the interrupt, it branches to this address by reloading PC with the old value.

3.10.3 The Instruction Set

Figure 3.33 lists the eleven instructions of our simple CPU. The first group consists of arithmetic (see Section 2.7), logic, and shift instructions (see Section 2.5.1). The circulate instruction executes a cyclic shift of the accumulator by one place. The second group consists of instructions to move data between the accumulator and memory. The third set contains a conditional jump instruction: when the accumulator is zero, it causes the CPU to resume fetching instructions at a new address, the address in the memory data register. This address is moved to the program counter before fetching the next instruction. The fourth set contains input/output instructions. The fifth set contains the halt instruction. Many more instruc-

Cycle	*Microinstruction*	*Microinstruction*
1	Copy contents of PC to MAR.	
2	Fetch word at address MAR into MDR.	Increment PC.
3	Copy opcode part of MDR to OPC.	
4	Interpret OPC	Copy address part of MDR to MAR.
5	Fetch word at address MAR into MDR.	
6	Copy MDR into AC.	

Figure 3.32 Decomposition of the **load accumulator** instruction into eight microinstructions in six microcycles.

	Opcode	*Binary*	*Description*
Arithmetic	ADD	0000	Add memory word to AC
Logic	AND	0001	AND memory word to AC
	CLA	0010	Clear (set to zero) the accumulator
	CMA	0011	Complement AC
	CIL	0100	Circulate AC left
Memory	LDA	0101	Load memory word into AC
	STA	0110	Store AC into memory word
Jump	JZ	0111	Jump to address if AC zero
I/O	IN	1000	Load INR into AC
	OUT	1001	Store AC into OUTR
Halt	HLT	1010	Halt computer

Figure 3.33 Instructions of the simple CPU.

tions could be added, including ones to simplify the execution of subroutines, handle loops, and process interrupts. Each instruction has a mnemonic opcode, such as CLA, and a binary opcode, such as 0010.

Many other operations can be performed using this set, including subtraction, which can be realized through the use of ADD, CMA, and two's-complement arithmetic (see Problem 3.18). Multiplication is also possible through the use of CIL and ADD (see Problem 3.38). Since multiple CILs can be used to rotate right one place, division is also possible. Finally, as observed in Problem 3.39, every two-input Boolean function can be realized through the use of AND and CMA. This implies that every Boolean function can be realized by this machine if it is designed to address enough memory locations.

Each of these instructions is a **direct memory instruction**, by which we mean that all addresses refer directly to memory locations containing the **operands** (data) on which the program operates. Most CPUs also have **indirect memory instructions** (and are said to support **indirection**). These are instructions in which an address is interpreted as the address at which to find the address containing the needed operand. To find such an indirect operand, the CPU does two memory fetches, the first to find the address of the operand and the second to find the operand itself. Often a single bit is added to an opcode to denote that an instruction is an indirect memory instruction.

An instruction stored in the memory of our computer consists of sixteen binary digits, the first four denoting the opcode and the last twelve denoting an address. Because it is hard for humans to interpret such **machine-language** statements, mnemonic opcodes and assembly languages have been devised.

3.10.4 Assembly-Language Programming

An assembly-language program consists of a number of lines each containing either a real or pseudo-instruction. Real instructions correspond exactly to machine-language instructions except that they contain mnemonics and symbolic addresses instead of binary sequences. Pseudoinstructions are directions to the **assembler**, the program that translates an assembly-language program into machine language. A typical pseudo-instruction is ORG 100, which instructs the assembler to place the following lines at locations beginning with location 100. Another example is the DAT pseudo-instruction that identifies a word containing only data. The END pseudo-instruction identifies the end of the assembly-language program.

Each assembly-language instruction fits on one line. A typical instruction has the following fields, some or all of which may be used.

Symbolic_Address	Mnemonic	Address	Indirect Bit	Comment

If an instruction has a Symbolic_Address (a string of symbols), the address is converted to the physical address of the instruction by the assembler and substituted for all uses of the symbolic address. The Address field can contain one or more symbolic or real addresses, although the assembly language used here allows only one address. The Indirect Bit specifies whether or not indirection is to be used on the address in question. In our CPU we do not allow indirection, although we do allow it in our assembly language because it simplifies our sample program.

Let's now construct an assembly-language program whose purpose is to boot up a computer that has been reset. The **boot program** reads another program provided through its input port and stores this new program (a sequence of 16-bit words) in the memory locations just above itself. When it has finished reading this new program (determined by reading a zero word), it transfers control to the new program by jumping to the first location above itself. When computers are turned off at night they need to be rebooted, typically by executing a program of this kind.

Figure 3.34 shows a program to boot up our computer. It uses three symbolic addresses, ADDR_1, ADDR_2, ADDR_3, and one real address, 10. We assume this program resides

	ORG	0		Program is stored at location 0.
ADDR_1	IN			Start of program.
	JZ	10		Transfer control if AC zero.
	STA	ADDR_2	I	Indirect store of input.
	LDA	ADDR_2		Start incrementing ADDR_2.
	ADD	ADDR_3		Finish incrementing of ADDR_2.
	STA	ADDR_2		Store new value of ADDR_2.
	CLA			Clear AC.
	JZ	ADDR_1		Jump to start of program.
ADDR_2	DAT	10		Address for indirection.
ADDR_3	DAT	1		Value for incrementing.
	END			

Figure 3.34 A program to reboot a computer.

permanently in locations 0 through 9 of the memory. After being reset, the CPU reads and executes the instruction at location 0 of its memory.

The first instruction of this program after the ORG statement reads the value in the input register into the accumulator. The second instruction jumps to location 10 if the accumulator is zero, indicating that the last word of the second program has been written into the memory. If this happens, the next instruction executed by the CPU is at location 10; that is, control is transferred to the second program. If the accumulator is not zero, its value is stored indirectly at location ADDR_2. (We explain the indirect STA in the next paragraph.) On the first execution of this command, the value of ADDR_2 is 10, so that the contents of the accumulator are stored at location 10. The next three steps increment the value of ADDR_2 by placing its contents in the accumulator, adding the value in location ADDR_3 to it, namely 1, and storing the new value into location ADDR_2. Finally, the accumulator is zeroed and a JZ instruction used to return to location ADDR_1, the first address of the boot program.

The indirect STA instruction in this program is not available in our computer. However, as shown in Problem 3.42, this instruction can be simulated by a self-modifying subprogram. While it is considered bad programming practice to write self-modifying programs, this exercise illustrates the power of self-modification as well as the advantage of having indirection in the instruction set of a computer.

3.10.5 Timing and Control

Now that the principles of a CPU have been described and a programming example given, we complete the description of a sequential circuit realizing the CPU. To do this we need to describe circuits controlling the combining and movement of data. To this end we introduce the assignment notation in Fig. 3.35. Here the expression AC $\leftarrow$ MDR means that the contents of MDR are copied into AC, whereas AC $\leftarrow$ AC + MDR means that the contents of AC and MDR are added and the result assigned to AC. In all cases the left arrow, $\leftarrow$, signifies that the result or contents on the right are assigned to the register on the left. However, when the register on the left contains information of a particular type, such as an address in the case of PC or an opcode in the case of OPC, and the register on the right contains more information, the assignment notation means that the relevant bits of the register on the right are loaded into the register on the left. For example, the assignment PC $\leftarrow$ MDR means that the address portion of MDR is copied to PC.

Register transfer notation uses these assignment operations as well as timing information to break down a machine-level instruction into microinstructions that are executed in succes-

Notation	*Explanation*
AC $\leftarrow$ MDR	Contents of MDR loaded into AC.
AC $\leftarrow$ AC + MDR	Contents of MDR added to AC.
MDR $\leftarrow$ M	Contents of memory location MAR loaded into MDR.
M $\leftarrow$ MDR	Contents of MDR stored at memory location MAR.
PC $\leftarrow$ MDR	Address portion of MDR loaded into PC.
MAR $\leftarrow$ PC	Contents of PC loaded into MAR.

Figure 3.35 Microinstructions illustrating assignment notation.

Timing	*Microinstructions*
t_1	MAR $\leftarrow$ PC
t_2	MDR $\leftarrow$ M, PC $\leftarrow$ PC+1
t_3	OPC $\leftarrow$ MDR

Figure 3.36 The microcode for the fetch portion of each instruction.

sive microcycles. The j*th* microcycle is specified by the **timing variable** t_j, $1 \leq j \leq k$. That is, t_j is 1 during the j*th* microcycle and is zero otherwise. It is straightforward to show that these timing variables can be realized by connecting a decoder to the outputs of a counting circuit, a circuit containing the binary representation of an integer that increments the integer modulo some other integer on each clock cycle. (See Problem 3.40.)

Since the fetch portion of each instruction is the same, we write a few lines of register transfer notation for it, as shown in Fig. 3.36. On the left-hand side of each line is timing variable indicating the cycle during which the microinstruction is executed.

The microinstructions for the execute portion of each instruction of our computer are shown in Fig. 3.37. On the left-hand side of each line is a timing variable that must be ANDed with the indicated **instruction variable**, such as c_{ADD}, which is 1 if that instruction is in

Control		*Microcode*
ADD		
c_{ADD}	t_4	MAR $\leftarrow$ MDR
c_{ADD}	t_5	MDR $\leftarrow$ M
c_{ADD}	t_6	AC $\leftarrow$ AC + MDR
AND		
c_{AND}	t_4	MAR $\leftarrow$ MDR
c_{AND}	t_5	MDR $\leftarrow$ M
c_{AND}	t_6	AC $\leftarrow$ AC AND MDR
CLA		
c_{CLA}	t_4	AC $\leftarrow$ 0
CIL		
c_{CIL}	t_4	AC $\leftarrow$ Shift(AC)
LDA		
c_{LDA}	t_4	MAR $\leftarrow$ MDR
c_{LDA}	t_5	MDR $\leftarrow$ M
c_{LDA}	t_6	AC $\leftarrow$ MDR

Control		*Microcode*
STA		
c_{STA}	t_4	MAR $\leftarrow$ MDR
c_{STA}	t_4	MDR $\leftarrow$ AC
c_{STA}	t_5	M $\leftarrow$ MDR
CMA		
c_{CMA}	t_4	AC $\leftarrow \neg$ AC
JZ		
c_{JZ}	t_4	if (AC = 0) PC $\leftarrow$ MDR
IN		
c_{IN}	t_4	AC $\leftarrow$ INR
OUT		
c_{OUT}	t_4	OUTR $\leftarrow$ AC
HLT		
c_{HLT}	t_4	$t_j \leftarrow 0$ for $1 \leq j \leq k$

Figure 3.37 The execute portions of the microcode of instructions.

the opcode register OPC and 0 otherwise. These instruction variables can be generated by a decoder attached to the output of OPC. Here ¬A denotes the complement of the accumulator.

Now that we understand how to combine microinstructions in microcycles to produce macroinstructions, we use this information to define control variables that control the movement of data between registers or combine the contents of two registers and assign the result to another register. This information will be used to complete the design of the CPU.

We now introduce notation for **control variables**. If a microinstruction results in the movement of data from register B to register A, denoted $A \leftarrow B$ in our assignment notation, we associate the control variable $L(A, B)$ with it. If a microinstruction results in the combination of the contents of registers B and C with the operation $\odot$ and the assignment of the result to register A, denoted $A \leftarrow B \odot C$ in our assignment notation, we associate the control variable $L(A, B \odot C)$ with it. For example, inspection of Figs. 3.36 and 3.37 shows that we can write the following expressions for the control variables $L(\text{OPC}, \text{MDR})$ and $L(\text{AC}, \text{AC+MDR})$:

$$\begin{aligned} L(\text{OPC}, \text{MDR}) &= t_3 \\ L(\text{AC}, \text{AC+MDR}) &= c_{\text{ADD}} \wedge t_6 \end{aligned}$$

Thus, OPC is loaded with the contents of MDR when $t_3 = 1$, and the contents of AC are added to those of MDR and copied into AC when $c_{\text{ADD}} \wedge t_6 = 1$.

The complete set of control variables can be obtained by first grouping together all the microinstructions that affect a given register, as shown in Fig. 3.38, and then writing expressions for the control variables. Here M denotes the memory unit and HLT is a special register that must be set to 1 for the CPU to run. Inspection of Fig. 3.38 leads to the following expressions for control variables:

$$\begin{aligned}
L(\text{AC}, \text{AC} + \text{MDR}) &= c_{\text{ADD}} \wedge t_6 \\
L(\text{AC}, \text{AC AND MDR}) &= c_{\text{AND}} \wedge t_6 \\
L(\text{AC}, 0) &= c_{\text{CLA}} \wedge t_4 \\
L(\text{AC}, \text{Shift(AC)}) &= c_{\text{CIL}} \wedge t_4 \\
L(\text{AC}, \text{MDR}) &= c_{\text{LDA}} \wedge t_6 \\
L(\text{AC}, \text{INR}) &= c_{\text{IN}} \wedge t_4 \\
L(\text{AC}, \neg\,\text{AC}) &= c_{\text{CMA}} \wedge t_4 \\
L(\text{MAR}, \text{PC}) &= t_1 \\
L(\text{MAR}, \text{MDR}) &= (c_{\text{ADD}} \vee c_{\text{AND}} \vee c_{\text{LDA}} \vee c_{\text{STA}}) \wedge t_4 \\
L(\text{MDR}, \text{M}) &= t_2 \vee (c_{\text{ADD}} \vee c_{\text{AND}} \vee c_{\text{LDA}}) \wedge t_5 \\
L(\text{MDR}, \text{AC}) &= c_{\text{STA}} \wedge t_4 \\
L(\text{M}, \text{MDR}) &= c_{\text{STA}} \wedge t_5 \\
L(\text{PC}, \text{PC+1}) &= t_2 \\
L(\text{PC}, \text{MDR}) &= (AC = 0) \wedge c_{\text{JZ}} \wedge t_4 \\
L(\text{OPC}, \text{MDR}) &= t_3 \\
L(\text{OUTR}, \text{AC}) &= c_{\text{OUT}} \wedge t_4 \\
L(t_j) &= c_{\text{HLT}} \wedge t_4 \text{ for } 1 \leq j \leq 6
\end{aligned}$$

Control		*Microcode*
AC		
c_{ADD}	t_6	AC ← AC + MDR
c_{AND}	t_6	AC ← AC AND MDR
c_{CLA}	t_4	AC ← 0
c_{CIL}	t_4	AC ← Shift(AC)
c_{LDA}	t_6	AC ← MDR
c_{CMA}	t_4	AC ← ¬ AC
c_{IN}	t_4	AC ← INR
MAR		
	t_1	MAR ← PC
c_{ADD}	t_4	MAR ← MDR
c_{AND}	t_4	MAR ← MDR
c_{LDA}	t_4	MAR ← MDR
c_{STA}	t_4	MAR ← MDR
MDR		
	t_2	MDR ← M
c_{ADD}	t_5	MDR ← M
c_{AND}	t_5	MDR ← M
c_{LDA}	t_5	MDR ← M
c_{STA}	t_4	MDR ← AC

Control		*Microcode*
M		
c_{STA}	t_5	M ← MDR
PC		
	t_2	PC ← PC+1
c_{JZ}	t_4	if (AC = 0) PC ← MDR
OPC		
	t_3	OPC ← MDR
OUTR		
c_{OUT}	t_4	OUTR ← AC
HLT		
c_{HLT}	t_4	$t_j \leftarrow 0$ for $1 \le j \le k$

Figure 3.38 The microinstructions affecting each register.

The expression (AC = 0) denotes a Boolean variable whose value is 1 if all bits in the AC are zero and 0 otherwise. This variable is the AND of the complement of each component of register AC.

To illustrate the remaining steps in the design of the CPU, we show in Fig. 3.39 the circuits used to provide input to the accumulator AC. Shown are registers AC, MDR, and INR as well as circuits for the functions f_{add} (see Section 2.7) and f_{and} that add two binary numbers and take their AND, respectively. Also shown are multiplexer circuits f_{mux} (see Section 2.5.5). They have three control inputs, L_0, L_1, and L_2, and can select one of eight inputs to place on their output lines. However, only seven inputs are needed: the result of adding AC and MDR, the result of ANDing AC and MDR, the zero vector, the result of shifting AC, the contents of MDR or INR, and the complement of AC. The three control inputs encode the seven control variables, $L(\text{AC}, \text{AC} + \text{MDR})$, $L(\text{AC}, \text{AC AND MDR})$, $L(\text{AC}, 0)$, $L(\text{AC}, \text{Shift(AC)})$, $L(\text{AC}, \text{MDR})$, $L(\text{AC}, \text{INR})$, and $L(\text{AC}, \neg\text{AC})$. Since at most one of these control variables has value 1 at any one time, the encoder circuit of Section 2.5.3 can be used to encode these seven control variables into the three bits L_0, L_1, and L_2 shown in Fig. 3.39.

The logic circuit to supply inputs to AC has size proportional to the number of bits in each register. Thus, if the word size of the CPU were scaled up, the size of this circuit would scale linearly with the word size.

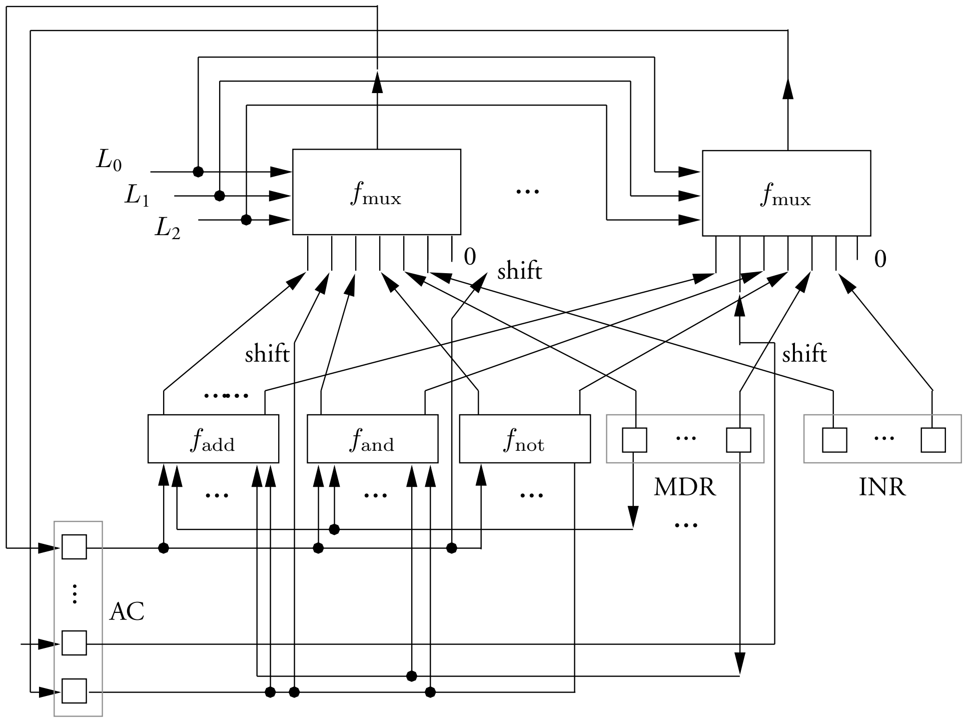


Figure 3.39 Circuits providing input to the accumulator AC.

The circuit for the program counter PC can be designed from an adder, a multiplexer, and a few additional gates. Its size is proportional to $\lceil \log_2 m \rceil$. The circuits to supply inputs to the remaining registers, namely MAR, MDR, OPC, INR, and OUTR, are less complex to design than those for the accumulator. The same observations apply to the control variable to write the contents of the memory. The complete design of the CPU is given as an exercise (see Problem 3.41).

3.10.6 CPU Circuit Size and Depth

Using the design given above for a simple CPU as a basis, we derive upper bounds on the size and depth of the next-state and output functions of the RAM CPU defined in Section 3.4.

All words on which the CPU operates contain b bits except for addresses, which contain $\lceil \log m \rceil$ bits where m is the number of words in the random-access memory. We assume that the CPU not only has an $\lceil \log m \rceil$-bit program counter but can send the contents of the PC to the MAR of the random-access memory in one unit of time. When the CPU fetches an instruction that refers to an address, it may have to retrieve multiple b-bit words to create an $\lceil \log m \rceil$-bit address. We assume the time for such operations is counted in the number T of steps that the RAM takes for the computation.

The arithmetic operations supported by the RAM CPU include addition and subtraction, operations realized by circuits with size and depth linear and logarithmic respectively in b, the length of the accumulator. (See Section 2.7.) The same is true for the logical vector and the shift operations. (See Section 2.5.1.) Thus, circuits affecting the accumulator (see Fig. 3.39) have size $O(b)$ and depth $O(\log b)$. Circuits affecting the opcode and output registers and the memory address and data registers are simple and have size $O(b)$ and depth $O(\log b)$. The circuits affecting the program counter not only support transfer of data from the accumulator to the program counter but also allow the program counter to be incremented. The latter function can be performed by an adder circuit whose size is $O(\lceil \log m \rceil)$ and depth is $O(\log \lceil \log m \rceil)$. It follows that

$$\begin{aligned} C_\Omega(\delta_{\mathrm{CPU}}) &= O(b + \lceil \log m \rceil) \\ D_\Omega(\delta_{\mathrm{CPU}}) &= O(\log b + \log \lceil \log m \rceil) \end{aligned}$$

3.10.7 Emulation

In Section 3.4 we demonstrated that whatever computation can be done by a finite-state machine can be done by a RAM when the latter has sufficient memory. This universal nature of the RAM, which is a model for the CPU we have just designed, is emphasized by the problem of emulation, the simulation of one general-purpose computer by another.

Emulation of a target CPU by a host CPU means reading the instructions in a program for the target CPU and executing host instructions that have the same effect as the target instructions. In Problem 3.44 we ask the reader to sketch a program to emulate one CPU by another. This is another manifestation of universality, this time for unbounded-memory RAMs.

Problems

MATHEMATICAL PRELIMINARIES

3.1 Establish the following identity:

$$\sum_{j=0}^{k} j2^j = 2\left((k-1)2^k + 1\right)$$

3.2 Let $p : \mathbb{N} \mapsto \mathbb{N}$ and $q : \mathbb{N} \mapsto \mathbb{N}$ be polynomial functions on the set $\mathbb{N}$ of non-negative integers. Show that $p(q(n))$ is also a polynomial in n.

FINITE-STATE MACHINES

3.3 Describe an FSM that compares two binary numbers supplied as concurrent streams of bits in descending order of importance and enters a rejecting state if the first string is smaller than the second and an accepting state otherwise.

3.4 Describe an FSM that computes the threshold-two function on n Boolean inputs that are supplied sequentially to the machine.

3.5 Consider the full-adder function $f_{FA}(x_i, y_i, c_i) = (c_{i+1}, s_i)$ defined below where $+$ denotes integer addition:

$$2c_{i+1} + s_i = x_i + y_i + c_i$$

Show that the subfunction of f_{FA} obtained by fixing $c_i = 0$ and deleting c_{i+1} is the EXCLUSIVE OR of the variables x_i and y_i.

3.6 It is straightforward to show that every Moore FSM is a Mealy FSM. Given a Mealy FSM, show how to construct a Moore FSM whose outputs for every input sequence are identical to those of the Mealy FSM.

3.7 Find a deterministic FSM that recognizes the same language as that recognized by the nondeterministic FSM of Fig. 3.8.

3.8 Write a program in a language of your choice that writes the straight-line program described in Fig. 3.3 for the FSM of Fig. 3.2 realizing the EXCLUSIVE OR function.

SHALLOW FSM CIRCUITS

3.9 Develop a representation for states in the m-word, b-bit random-access memory so that its next-state mappings form a semigroup.

Hint: Show that the information necessary to update the current state can be succinctly described.

3.10 Show that matrix multiplication is associative.

SEQUENTIAL CIRCUITS

3.11 Show that the circuit of Fig. 3.15 computes the functions defined in the tables of Fig. 3.14.

Hint: Section 2.2 provides a method to produce a circuit from a tabular description of a binary function.

3.12 Design a sequential circuit (an **electronic lock**) that enters an accepting state only when it receives some particular four-bit sequence that you specify.

3.13 Design a sequential circuit (a **modulo-p counter**) that increments a binary number by one on each step until it reaches the integer value p, at which point it resets its value to zero. You should assume that p is not a power of 2.

3.14 Give an efficient design of an **incrementing/decrementing counter**, a sequential circuit that increments or decrements a binary number modulo 2^n. Specify the machine as an FSM and determine the number of gates in the sequential circuit in terms of n.

RANDOM-ACCESS MACHINES

3.15 Given a straight-line program for a Boolean function, describe the steps taken to compute it during fetch-and-execute cycles of a RAM. Determine whether jump instructions are necessary to execute such programs.

3.16 Consulting Theorem 3.4.1, determine whether jump instructions are necessary for all RAM computations. If not, what advantage accrues to using them?

3.17 Sketch a RAM program using time and space $O(n)$ that recognizes strings of the form $\{0^m1^m \mid 1 \leq m \leq n\}$.

ASSEMBLY-LANGUAGE PROGRAMMING

3.18 Write an assembly-language program in the language of Fig. 3.18 to subtract two integers.

3.19 The assembly-language instructions of Fig. 3.18 operate on integers. Show that the operations AND, OR, and NOT can be realized on Boolean variables with these instructions. Show also that these operations on vectors can be implemented.

3.20 Write an assembly-language program in the language of Fig. 3.18 to form x^y for integers x and y.

3.21 Show that the assembly-language instructions CLR R_i, $\mathrm{R}_i \leftarrow \mathrm{R}_j$, JMP_+ N_i, and JMP_- N_i can be realized from the assembly-language instructions INC, DEC, CONTINUE, R_j JMP_+ N_i, and R_j JMP_- N_i.

TURING MACHINES

3.22 In a standard Turing machine the tape unit has a left end but extends indefinitely to the right. Show that allowing the tape unit to be infinite in both directions does not add power to the Turing machine.

3.23 Describe in detail a Turing machine with unlimited storage capacity that recognizes the language $\{0^m 1^m | 1 \leq m\}$.

3.24 Sketch a proof that in $O(n^4)$ steps a Turing machine can verify that a particular tour of n cities in an instance of the Traveling Salesperson Problem satisfies the requirement that the total distance traveled is less than or equal to the limit k set on this instance of the Traveling Salesperson Problem.

3.25 Design the additional circuitry needed to transform a sequential circuit for a random-access memory into one for a tape memory. Give upper bounds on the size and depth of the next-state and output functions that are simultaneously achievable.

3.26 In the proof of Theorem 3.8.1 it is assumed that the words and their addresses in a RAM memory unit are placed on the tape of a Turing machine in order of increasing addresses, as suggested by Fig. 3.40. The addresses, which are $\lceil \log m \rceil$ bits in length, are organized as a collection of $\lceil \lceil \log m \rceil / b \rceil$ b-bit words. (In the example, $b = 1$.) An address is written on tape cells that immediately precede the value of the corresponding RAM word. A RAM address addr is stored on the tape to the left in the shaded region.

Assume that markers can be placed on cells. (This amounts to enlarging the tape alphabet by a constant factor.) Show that markers can be used to move from the first word whose RAM address matches the ib most significant bits of the address $\boldsymbol{a}$ to the

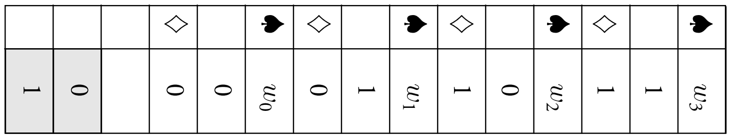


Figure 3.40 A TM tape with markers on words and the first bit of each address.

next one that matches the $(i+1)b$ most significant bits. Show that this procedure can be used to find the RAM word whose address matches `addr` in $O((m/b)(\log m)^2)$ Turing machine steps by a machine that can store in its control unit only one b-bit subword of `addr`.

3.27 Extend Problem 3.26 by demonstrating that the simulation can be done with a binary tape symbol alphabet.

3.28 Extend Theorem 3.8.1 to show that there exists a Turing machine that can simulate an unbounded-memory RAM.

3.29 Sketch a proof that every Turing machine can be simulated by a RAM program of the kind described in Section 3.4.3.

Hint: Because such RAM programs can only have a finite number of registers, encode the contents of the TM tape as a number to be stored in one register.

COMPUTATIONAL INEQUALITIES FOR TURING MACHINES

3.30 Show that a one-tape Turing machine needs time exponential in n to compute most Boolean functions $f : \mathcal{B}^n \mapsto \mathcal{B}$ on n variables, regardless of how much memory is allocated to the computation.

3.31 Apply Theorem 3.2.2 to the one-tape Turing machine that executes T steps. Determine whether the resulting inequalities are weaker or stronger than those given in Theorem 3.9.2.

3.32 Write a program in your favorite language for the procedure WRITE_OR(t, m) introduced in Fig. 3.27.

3.33 Write a program in your favorite language for the procedure WRITE_CELL_CIRCUIT(t, m) introduced in Fig. 3.27.

Hint: See Problem 2.4.

FIRST P-COMPLETE AND NP-COMPLETE PROBLEMS

3.34 Show that the language MONOTONE CIRCUIT VALUE defined below is **P**-complete.

MONOTONE CIRCUIT VALUE
Instance: A description for a monotone circuit with fixed values for its input variables and a designated output gate.
Answer: "Yes" if the output of the circuit has value 1.

Hint: Using dual-rail logic, find a way to translate (reduce) a string in the language CIRCUIT VALUE to a string in MONOTONE CIRCUIT VALUE by converting in logarithmic space (in the length of the string) a circuit over the standard basis to a circuit over the monotone basis. Note that, as stated in the text, the composition of two logarithmic-space reductions is a logarithmic-space reduction. To simplify the conversion from non-monotone circuits to monotone circuits, use even integers to index vertices in the non-monotone circuits so that both even and odd integers can be used in the monotone case.

3.35 Show that the language FAN-OUT 2 CIRCUIT SAT defined below is **NP**-complete.

FAN-OUT 2 CIRCUIT SAT
Instance: A description for a circuit of fan-out 2 with free values for its input variables and a designated output gate.
Answer: "Yes" if the output of the circuit has value 1.

Hint: To reduce the fan-out of a vertex, replace the direct connections between a gate and its successors by a binary tree whose vertices are AND gates with their inputs connected together. Show that, for each gate of fan-out more than two, such trees can be generated by a program that runs in polynomial time.

3.36 Show that clauses given in the proof of Theorem 3.9.7 are satisfied only when their variables have values consistent with the definition of the gate type.

3.37 A circuit with n input variables $\{x_1, x_2, \dots, x_n\}$ is satisfiable if there is an assignment of values to the variables such that the output of the circuit has value 1. Assume that the circuit has only one output and the gates are over the basis $\Omega = \{\text{AND}, \text{OR}, \text{NOT}\}$.

a) Describe a nondeterministic procedure that accepts as input the description of a circuit in POSE and returns 1 if the circuit is satisfiable and 0 otherwise.

b) Describe a deterministic procedure that accepts as input the description of a circuit in POSE and returns 1 if the circuit is satisfiable and 0 otherwise. What is the running time of this procedure when implemented on the RAM?

c) Describe an efficient (polynomial-time) deterministic procedure that accepts as input the description of a circuit in SOPE and returns 1 if the circuit is satisfiable and 0 otherwise.

d) By using Boolean algebra, we can convert a circuit from POSE to SOPE. We can then use the result of the previous question to determine if the circuit is satisfiable. What is the drawback of this approach?

CENTRAL PROCESSING UNIT

3.38 Write an assembly-language program to multiply two binary numbers using the simple CPU of Section 3.10. How large are the integers that can be multiplied without producing numbers that are too large to be recorded in registers?

3.39 Assume that the simple CPU of Section 3.10 is modified to address an unlimited number of memory locations. Show that it can realize any Boolean function by demonstrating that it can compute the Boolean operations AND, OR, and NOT.

3.40 Design a circuit to produce the timing variables t_j, $1 \le j \le k$, of the simple CPU. They must have the property that exactly one of them has value 1 at a time and they successively become 1.

Hint: Design a circuit that counts sequentially modulo k, an integer. That is, it increments a binary number until it reaches k, after which it resets the number to zero. See Problem 3.13.

3.41 Complete the design of the CPU of Section 3.10 by describing circuits for PC, MAR, MDR, OPC, INR, and OUTR.

3.42 Show that an indirect store operation can be simulated by the computer of Section 3.10.

Hint: Construct a program that temporarily moves the value of AC aside, fetches the address containing the destination for the store, and uses Boolean operations to modify a STA instruction in the program so that it contains the destination address.

3.43 Write an assembly-language program that repeatedly examines the input register until it is nonzero and then moves its contents to the accumulator.

3.44 Sketch an assembly-language program to emulate a target CPU by a host CPU under the assumption that each CPU's instruction set supports indirection. Provide a skeleton program that reads an instruction from the target instruction set and decides which host instruction to execute. Also sketch the particular host instructions needed to emulate a target add instruction and a target jump-on-zero instruction.

Chapter Notes

Although the concept of the finite-state machine is fully contained in the Turing machine model (Section 3.7) introduced in 1936 [337], the finite-state machine did not become a serious object of study until the 1950s. Mealy [214] and Moore [222] introduced models for finite-state machines that were shown to be equivalent. The Moore model is used in Section 3.1. Rabin and Scott [265] introduced the nondeterministic machine, although not defined in terms of external choice inputs as it is defined here.

The simulation of finite-state machines by logic circuits exhibited in Section 3.1.1 is due to Savage [284], as is its application to random-access (Section 3.6) and deterministic Turing machines (Section 3.9.1) [285]. The design of a simple CPU owes much to the early simple computers but is not tied to any particular architecture. The assembly language of Section 3.4.3 is borrowed from Smith [311].

The shallow circuits simulating finite-state machines described in Section 3.2 are due to Ladner and Fischer [185] and the existence of a universal Turing machine, the topic of Section 3.7, was shown by Turing [337].

Cook [74] identified the first **NP**-complete problem and Karp [158] demonstrated that a large number of other problems are **NP**-complete, including the Traveling Salesperson problem. About this time Levin [198] (see also [334]) was led to similar concepts for combinatorial problems. Our construction in Section 3.9.1 of a satisfiable circuit follows the general outline given by Papadimitriou [234] (who also gives the reduction to SATISFIABILITY) as well as the construction of a circuit simulating a deterministic Turing machine given by Savage [285]. Cook also identified the first **P**-complete problem [75,79]. Ladner [184] observed that the circuit of Theorem 3.9.1 could be written by a program using logarithmic space, thereby showing that CIRCUIT VALUE is **P**-complete. More information on **P**-complete and **NP**-complete problems can be found in Chapter 8.

The more sophisticated simulation of a circuit by a Turing machine given in Section 3.9.7 is due to Pippenger and Fischer [251] with improvements by Schnorr [300] and Savage, as cited by Schnorr.