9. Integrated Device Technology Inc. "16 × 16 parallel CMOS multipliers IDT7216L/IDT7217L." data sheet, Santa Clara, CA, August 1995.
10. Johnson, K. T., A. R. Hurson, and B. Shirazi. "General Purpose Systolic Arrays." *IEEE Computer,* vol. 26 (November 1993) pp. 20–31.
11. Kampe, T. W. "The Design of a General-Purpose Microprogrammable Computer with Elementary Structure." *IEEE Transactions on Electronic Computers,* vol. EC-9 (June 1960), pp. 208–13.
12. Kane, G. and J. Heinrich. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
13. Kogge, P. M. *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
14. Koren, I. *Computer Arithmetic Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
15. Mick, J. and J. Brick. *Bit-Slice Microprocessor Design*. New York: McGraw-Hill, 1980.
16. Motorola Inc. *MC68881/MC68882 Floating-Point Coprocessor User's Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
17. Robertson, J. E. "Twos Complement Multiplication in Binary Parallel Computers." *IRE Transactions on Electronic Computers,* vol. EC-4 (September 1955) pp. 118–19.
18. Stone, H. S. *High-Performance Computer Architecture*. 3rd ed. Reading, MA: Addison-Wesley, 1993.
19. Texas Instruments Inc. *The TTL Logic Data Book*. Dallas, TX, 1988.
20. Volder, J. E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers,* vol. EC-8 (September 1959) pp. 330–34.
21. Wallace, C. S. "A Suggestion for a Fast Multiplier." *IEEE Transactions on Electronic Computers,* vol. EC-13 (February 1964) pp. 14–17.

CHAPTER 5

Control Design

In this chapter we study the register-level design of the control part of an instruction-set processor; the data-processing part was covered in Chapter 4. The two basic approaches to control-unit design—hardwired and microprogrammed—are discussed in detail. The complex task of controlling pipelined and superscalar processors is also examined.

5.1 BASIC CONCEPTS

First we discuss the general structure and behavior of control units. Then we examine the design of hardwired controllers, which are characterized by the use of fixed (nonprogrammable) logic circuits.

5.1.1 Introduction

We saw in section 2.1.1 that it is useful to separate a digital system into two parts: a datapath (data processing) unit and a control unit. The datapath is a network of functional and storage units capable of performing certain (micro) operations on data words. The purpose of the control unit is to issue control signals to the datapath. These control signals enter the datapath at "control points," where they select the functions to be performed at specific times and route the data through the appropriate parts of the datapath unit. In other words, the control unit logically reconfigures the datapath to implement some specified instruction or program.

A CPU's datapath contains circuits to perform arithmetic and logical operations on words such as fixed-point or floating-point numbers. The internal struc-

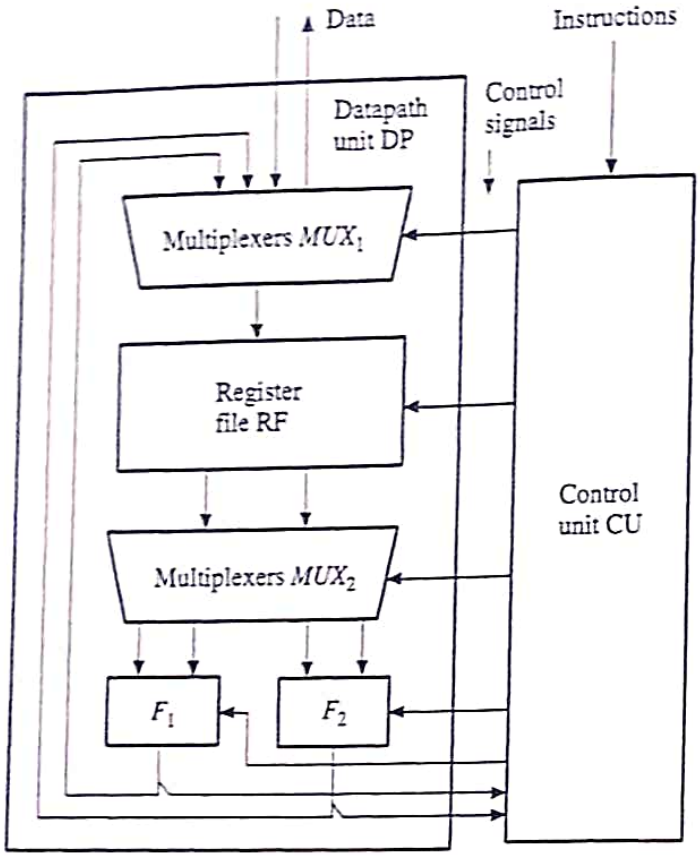


Figure 5.1
Processor composed of a datapath unit DP and a control unit CU.

ture of the datapath circuit DP of a small microprocessor is depicted in Figure 5.1. It contains a register file RF for temporary storage of operands, two functional units F_1 and F_2 responsible for data processing, and multiplexers to allow the data to be steered through DP. Typical functional units are an ALU performing addition, subtraction, and logical operations; a shifter; or a multiplier. The control unit CU receives external instructions or commands, which it converts into a sequence of control signals that the CU applies to DP to implement a sequence of register-transfer operations.

Figure 5.2 shows the control signals that implement an addition instruction of the form ADD A,B, which we write as

$$A := A + B; \tag{5.1}$$

in our HDL notation. Assume that this operation can be executed in a single clock cycle, whose timing details are not of concern at this level of abstraction. The input variables A and B are obtained from registers of the same name in RF, and the result is stored back into register A. Observe that the registers of RF permit their contents to be read from and written into in the same clock cycle, a basic property of the (edge-triggered) flip-flops from which such registers are constructed. RF is configured with one input and two output ports to support operations like (5.1) with two or three addresses. Besides selecting the data registers to be used, the control unit CU must also select the operation to be performed on the data, in this case, functional unit F_1's ADD operation. Finally the necessary logical connections for the data to flow through DP must be established by applying appropriate control signals to the multiplexers.

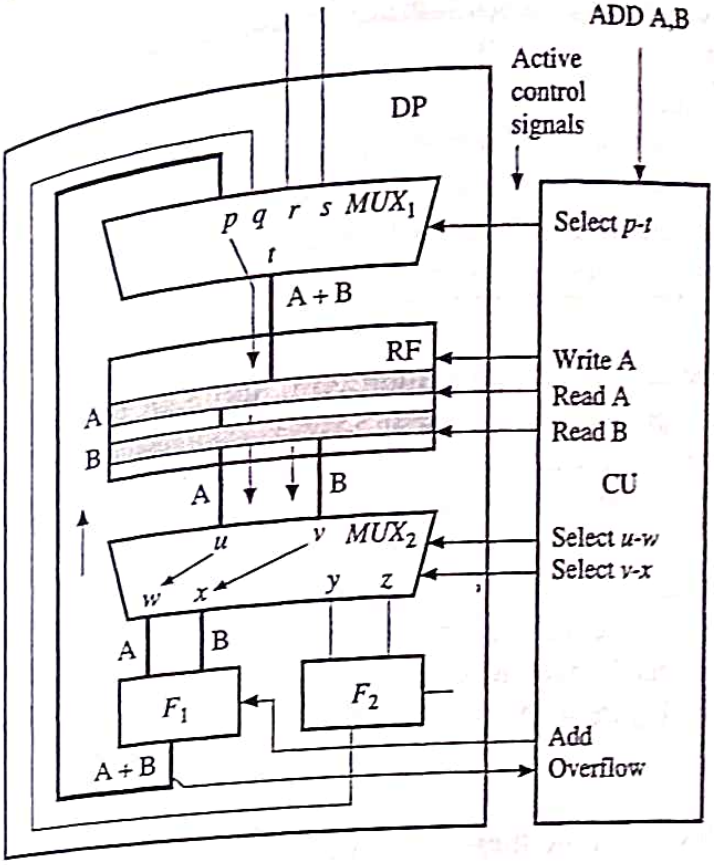


Figure 5.2
The processor of Figure 5.1 configured to implement the add operation A := A + B.

Thus we see that CU must activate the following three types of control signals during the clock cycle in which the ADD A,B instruction is executed.

- Function select: Add.
- Storage control: Read A, Read B, Write A.
- Data routing: Select *p-t*, Select *u-w*, Select *v-x*.

There is usually some feedback of control information from DP to CU to indicate exceptional conditions encountered during instruction execution. In the example of Figure 5.2, the functional unit F_1 performing the addition sends an overflow signal to CU whenever the sum A + B exceeds the normal word size.

Multicycle operations. Many types of instructions are executed in a single clock cycle—indeed, single-cycle execution is a central goal of RISC design. Some instructions require more than one clock cycle for their execution, however. For example, double-precision addition can be implemented by a two-instruction sequence (program) of the form

```
ADD     AL, BL          (5.2)
ADDC    AH, BH
```

which involves two double-word operands A and B. The first (ADD) instruction in (5.2) adds the low-order half (right word) of B to the low-order half of A, implicitly generating and storing a carry-out signal C. The second (ADDC) instruction adds the high-order half of B to the high-order half of A along with the carry C, thus ensuring that carries are propagated across the full double-length result. This short program is implemented in two consecutive clock cycles by activating the control signals listed below, not all of which appear in Figure 5.2.

Cycle	Function select	Storage control	Data routing
1	Add	Read AL, Read BL, Write AL	Select *p-t*, Select *u-w*, Select *v-x*
2	Add with carry	Read AH, Read BH, Write AH	Select *p-t*, Select *u-w*, Select *v-x*

(5.3)

This low-level description of the double-precision addition in terms of the control signals to be activated is an example of a *microprogram* and is contrasted with the higher-level *program* for the same operation appearing in (5.2). Each line of (5.3) is an example of a *microinstruction* specifying a set of low-level *microoperations*.

A further complication arises when the execution of a microoperation is conditional on the values of certain data or control signals. For example, the various sequential multiplication algorithms covered in the preceding chapters are specified by multistep algorithms that can be viewed as multicycle microprograms. The Booth multiplication algorithm (Figure 4.15), for instance, has statements of the following type:

```
LOOP:   if COND1 = true then ADD A,B
          else SUB A,B;
        if COND2 = true then go to OUTPUT
          else LOOP;
OUTPUT: ...
```

We can expand the microinstruction format of (5.3) to accommodate conditional operations in the following straightforward (but inefficient) way:

Current address	Condition select *C*	Next address		Function select		Storage control	Data routing
		C = true	*C* ≠ true	*C* = true	*C* ≠ true		
ADR1	COND1	ADR2	ADR2	ADD	SUB	...	...
ADR2	COND2	ADR3	ADR1	...	...	...	...
ADR3	...	...	...	...	...	...	...

(5.4)

Here we are introducing some new fields to specify a condition *C* to be tested, as well as alternative control signals to be activated depending on the current value of *C*. Typically, *C* corresponds to a status control signal from DP, or to a special signal generated within CU, such as an end-of-loop condition. If, for instance, *C* = COND1 in the preceding example, then one of the two function-select signals, ADD or SUB, is activated. To vary the order in which the microinstructions are executed, a pair of next-address fields is also provided, one of which is selected by the current value of *C*. This technique requires attaching an address to every microinstruction, thus completing the analogy between microinstructions and higher-level (assembly language) formats.

Implementation methods. Historically, two general approaches to control unit design have evolved. One approach views the controller as a sequential logic circuit or finite-state machine that generates specific sequences of control signals in response to externally supplied instructions; see Figure 5.3*a*. It is designed with the usual goals of minimizing the number of components used and maximizing the speed of operation. Once the unit is constructed, the only way to implement changes in control-unit behavior is by redesigning the entire unit. Such a circuit is therefore said to be *hardwired*. The format of (5.4) is essentially similar to the state-table format for describing the behavior of a (hardwired) sequential circuit, as illustrated in (5.5).

Current state	Current input	Next state	Current outputs		
			Function	Storage	Routing
ADR1	COND1 = 1	ADR2	ADD	...	...
ADR1	COND1 = 0	ADR2	SUB	...	...
ADR2	COND2 = 1	ADR3	...	...	...
ADR2	COND2 = 0	ADR1	...	...	...
ADR3	...	...	...	...	...

(5.5)

Microprogramming provides an alternative method of designing program control units. A *microprogrammed control unit* has the structure shown in Figure 5.3*b*. It is built around a storage unit called a *control memory*, where all the control signals are stored in a programlike format resembling (5.4). The control memory stores a set of microprograms designed to implement or *emulate* the behavior of the given instruction set. Each instruction causes the corresponding microprogram to be fetched and its control information extracted in a manner that resembles the fetching and execution of a program from the computer's main memory.

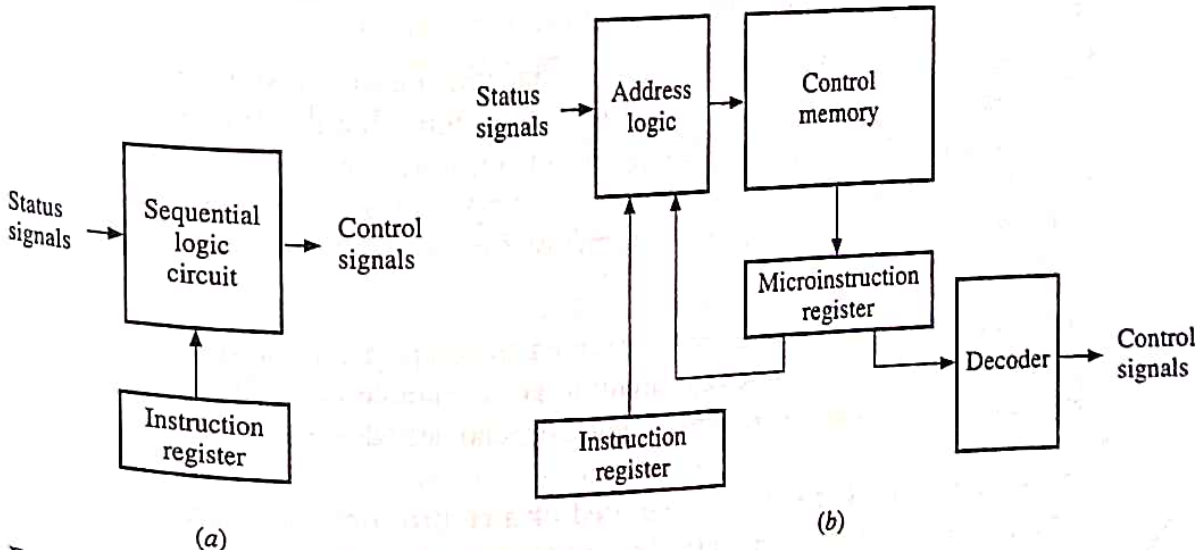


Figure 5.3
General structure of (*a*) a hardwired and (*b*) a microprogrammed control unit.

Microprogramming makes control unit design more systematic by organizing control signals into formatted words (microinstructions). Since the control signals are embedded in a kind of low-level software—this is referred to as *firmware*—design changes can be easily made just by altering the contents of the control memory. On the negative side, microprogrammed control units are more costly to manufacture than hardwired units due to the presence of the control memory and its access circuitry. Microprogrammed units also tend to be slower because of the extra time required to fetch microinstructions from the control memory. RISC processors, with their emphasis on small, fast instruction sets, favor the use of hardwired control units.

CPU control units, both hardwired and microprogrammed, are often organized as (*micro*) *instruction pipelines* in order to improve their performance. As we saw in section 4.3.2, pipelining is a relatively low-cost way of increasing a processor's throughput by decomposing its operation into a sequence of relatively independent steps. Program control naturally involves a sequence of steps (instruction fetching and decoding, input operand fetching, operation execution, and result storage) that can be carried out concurrently with different instructions. Modern CPUs make extensive use of pipelines to increase their effective instruction execution rate [Stone 1993].

5.1.2 Hardwired Control

Next we examine the design of control units that use fixed logic circuits to interpret instructions and generate control signals from them.

Design methods. Control-unit design involves various trade-offs between the amount of hardware used, the speed of operation, and the cost of the design process itself. To illustrate these issues, we consider two systematic approaches to the design of hardwired controllers [Hayes 1993; Baranov 1994]. These methods are representative of those used in practice, but by themselves are suitable only for small control units such as might be encountered in simple RISC processors or application-specific controllers.

- Method 1: The *classical method* of sequential circuit design, which was discussed briefly in section 2.1.3. It attempts to minimize the amount of hardware, in particular, by using only $\lceil \log_2 P \rceil$ flip-flops to realize a P-state circuit.
- Method 2: An approach that uses one flip-flop per state and is known as the *one-hot method*. While expensive in terms of flip-flops, this method simplifies CU design and debugging.

In practice, processor control units are often so complex that no one design method by itself can yield a satisfactory circuit at an acceptable cost. The most acceptable design may consist of several linked, but independently designed, sequential circuits.

State tables. The behavior required of a control unit, like that of any finite-state machine, can be represented by a state table of the general type shown in Figure 5.4*a*. The rows of the state table correspond to the set of internal states $\{S_i\}$. These states are determined by the information stored in the machine at discrete points of time (clock cycles). Let X and Z denote the input and output variables.

	Inputs			
State	I_1	I_2	...	I_m
S_1	$S_{1,1}, O_{1,1}$	$S_{1,2}, O_{1,2}$	...	$S_{1,m}, O_{1,m}$
S_2	$S_{2,1}, O_{2,1}$	$S_{2,2}, O_{2,2}$	...	$S_{2,m}, O_{2,m}$
			...	
S_n	$S_{n,1}, O_{n,1}$	$S_{n,2}, O_{n,2}$	...	$S_{n,m}, O_{n,m}$

(*a*)

	Inputs				
State	I_1	I_2		I_m	Outputs
S_1	$S_{1,1}$	$S_{1,2}$	...	$S_{1,m}$	O_1
S_2	$S_{2,1}$	$S_{2,2}$	...	$S_{2,m}$	O_2
			...		
S_n	$S_{n,1}$	$S_{n,2}$	...	$S_{n,m}$	O_n

(*b*)

Figure 5.4
State tables for a finite-state machine: (*a*) Mealy type and (*b*) Moore type.

The columns correspond to the combinations of the X signals that can be applied to the machine and are denoted here by $\{I_j\}$. The entry in row S_i and column I_j has the form $S_{i,j}$,$O_{i,j}$, where $S_{i,j}$ is the next state of the machine that results from the application of input combination I_j, and $O_{i,j}$ denotes the output signals that appear on Z whenever the machine is in state S_i with input I_j applied. In general, an entry in the state table defines a specific, one-cycle transition between two states.

Control units have a feature that favors a slightly different style of state table: Their output signal values often depend on the current state S_i only and so are independent of the input combination I_j. If all outputs are of this type, the circuit is called a *Moore machine*, in contrast with the more general *Mealy machine* of Figure 5.4*a*. (These names honor G. H. Mealy and E. F. Moore who were early researchers into finite-state machine theory [Mealy 1955; Moore 1956].) The state table of Figure 5.4*a* becomes a Moore machine if for every row i, we have $O_{i,j} = O_{i,k} = O_i$ for all j, $k = 1,2,\ldots,m$. In that case we can represent the machine's behavior in the more compact format of Figure 5.4*b*, where the output signals associated with each row are placed in a separate column.

GCD processor. To illustrate the classical and one-hot approaches to control-unit design, we will apply them to a special-purpose processor that computes the greatest common divisor $gcd(X,Y)$ of two positive integers X and Y; $gcd(X,Y)$ is defined as the largest integer that divides exactly into both X and Y. For example, $gcd(12,18) = 6$, and $gcd(12,17) = 1$. It is customary to assume that $gcd(0,0) = 0$.

We use a variant of Euclid's algorithm [Cormen, Leiserson, and Rivest 1990] to calculate $gcd(X,Y)$. Figure 5.5 gives an HDL description of this method.

```
gcd(in: X,Y; out: Z);
  register XR, YR, TEMPR;
  XR := X;                          {Input the data}
  YR := Y;
  while XR > 0 do begin
    if XR ≤ YR then begin           {Swap XR and YR}
      TEMPR := YR;
      YR := XR;
      XR := TEMPR; end
      XR := XR – YR;                {Subtract YR from XR}
  end
  Z := YR;                          {Output the result}
end gcd;
```

Figure 5.5
Procedure *gcd* to compute the greatest common divisor of two numbers.

The basic idea is to subtract the smaller of the two numbers from the other repeatedly—recall that division corresponds to repeated subtraction—until we obtain a number that divides the other. For example, with $X = 20$ and $Y = 12$, our *gcd* algorithm proceeds as follows:

Conditions		Actions	
		$XR := 20$; $YR := 12$;	
$XR > 0$:	$XR > YR$:	$XR := XR - YR = 8$;	
$XR > 0$:	$XR \leq YR$:	$YR := 8$; $XR := 12$;	$XR := XR - YR = 4$;
$XR > 0$:	$XR \leq YR$:	$YR := 4$; $XR := 8$;	$XR := XR - YR = 4$;
$XR > 0$	$XR \leq YR$:	$YR := 4$; $XR := 4$;	$XR := XR - YR = 0$;
$XR \leq 0$:		$Z := 4$;	

Hence we conclude that $gcd(20,12) = 4$.

Analysis of the *gcd* procedure suggests that its datapath unit DP should contain a pair of registers *XR* and *YR* to store the corresponding variables, one or more functional units to perform subtraction and magnitude comparison, and multiplexers for data routing, as indicated in Figure 5.6. We do not need to include a register for the "temporary" variable *TEMPR*, as we would in a typical programmed implementation, because we can read from and write to a register in the same clock cycle. The swap operation can therefore be done without conflict in one cycle thus:

$$X := Y, \quad Y := X; \tag{5.6}$$

The control unit CU generates control signals *Load XR* and *Load YR* to load each register independently with the input data *X* and *Y*. A control signal *Select XY* routes *X* and *Y* to *XR* and *YR*, respectively. Another signal *Swap* controls the swap operation defined by (5.6), which requires routing the outputs of the *XR* and *YR* registers to each other's inputs. A final signal *Subtract* is assumed to control the subtraction $XR := XR - YR$ by routing the output of the subtracter to *XR*. The input signals to CU are an asynchronous *Reset* signal, two comparison signals ($XR \geq YR$) and ($XR > 0$) generated by DP, and the usual, implicit clock signal.

We can identify a set of states for CU by examining the behavior defined in the HDL specification (refer to Figure 5.5)—a simple process here, but one that is

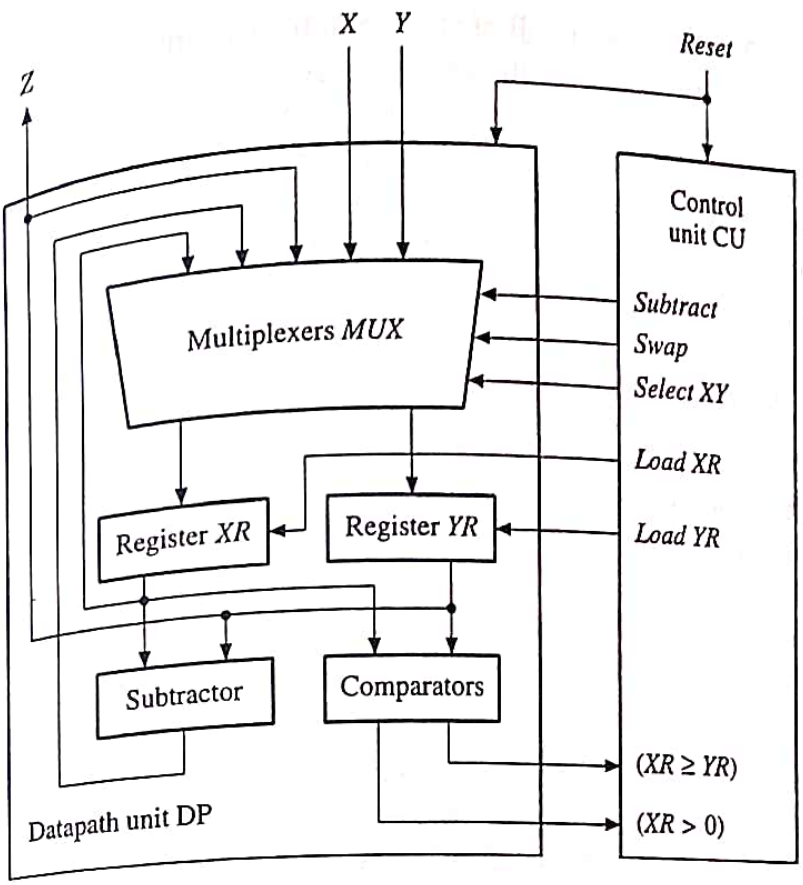


Figure 5.6
Hardware needed to implement the *gcd* procedure.

tedious and error-prone in the case of large control units. A start state S_0 is entered when *Reset* becomes 1; this state also loads X and Y into the DP registers. The subsequent actions of the *gcd* processor are either a swap or a subtraction, for which we define the states S_1 and S_2, respectively. A final state S_3 is entered when $gcd(X,Y)$ has been computed. Figure 5.7 gives a Moore-type state table defining the CU's behavior. Each state transition is deduced directly from the HDL description. If the input control signal $(XR > 0) = 0$, indicating that the **while** loop should be skipped, a transition is made from S_0 to S_3; this yields the first next-state entry in the top row of Figure 5.7. If, on the other hand, $(XR > 0) = 1$, the **while** loop is entered, and a transition is made to S_1 to perform a swap if $(XR \geq YR) = 0$; otherwise, the transition is to S_2 to perform a subtraction. The latter case defines to the third entry of the state table, whose input combination is $(XR > 0)(XR \geq YR) = 11$.

	Inputs $(XR > 0)$ $(XR \geq YR)$			**Outputs**				
State	**0–**	**10**	**11**	*Subtract*	*Swap*	*Select XY*	*Load XR*	*Load YR*
S_0 (Begin)	S_3	S_1	S_2	0	0	1	1	1
S_1 (Swap)	S_2	S_2	S_2	0	1	0	1	1
S_2 (Subtract)	S_3	S_1	S_2	1	0	0	1	0
S_3 (End)	S_3	S_3	S_3	0	0	0	0	0

Figure 5.7
State table defining the control unit of the *gcd* processor.

Since a subtraction always follows a swap, all next-state entries in the second row are S_2. The corresponding active outputs are the two register-load signals *Load XR* and *Load YR*, along with *Swap*, which route the outputs of *XR* and *YR* to *YR* and *XR*, respectively. The next states for state S_2 are the same as those for S_0; the active outputs are *Subtract*, which routes the output *XR* – *YR* of the subtracter to *XR*, and *Load XR*. The final state S_3 is assumed to be a "dead" state that is unaffected by all inputs (except *Reset*) and produces no active outputs.

Classical method. The major steps of the classical design method are as follows:

1. Construct a P-row state table that defines the desired input-output behavior.
2. Select the minimum number p of D-type flip-flops and assign a p-bit binary code to each state.
3. Design a combinational circuit C that generates the primary output signals $\{z_i\}$ and the secondary outputs $\{D_i\}$ that must be applied to the flip-flops.

We now apply this method to the design of the control unit CU for the *gcd* processor. We have already constructed the necessary state table (Figure 5.7). Since there are four states, we require two flip-flops, whose outputs $D_1D_0 = y_1y_0$ define CU's internal states. We assign the binary patterns to the four states in the following obvious way:

$$\begin{aligned} S_0 &= 0\,0 \\ S_1 &= 0\,1 \\ S_2 &= 1\,0 \\ S_3 &= 1\,1 \end{aligned} \tag{5.7}$$

We note in passing that the state assignment pattern affects the complexity of the circuit in subtle ways.

At this point we can construct a binary version of the state table, the *excitation table*, as shown in Figure 5.8. The D flip-flop's characteristic equation $D_i^+(t+1) = D_i(t)$ defines the inputs D_1^+ and D_0^+ to the flip-flops. CU's combinational logic C can now be derived from the excitation table using any available manual or automatic method. Suppose, for instance, that we use two-level sum-of-products (SOP) minimization. It is easily checked that C is defined by the following SOP equations, which lead directly to the design of Figure 5.9. Note that all gates in an AND-OR SOP circuit can be changed to NANDs to produce a NAND-NAND realization of the original function.

$$\begin{aligned} D_1^+ &= (\overline{XR > 0}) + (XR \geq YR) + D_0 \\ D_0^+ &= D_1 \cdot D_0 + (\overline{XR \geq XR}) \cdot \overline{D}_0 + (\overline{XR > 0}) \cdot \overline{D}_0 \\ Subtract &= D_1 \cdot \overline{D}_0 \\ Swap &= \overline{D}_1 \cdot D_0 \\ Select\,XY &= \overline{D}_1 \cdot \overline{D}_0 \\ Load\,XR &= \overline{D}_0 + \overline{D}_1 \\ Load\,YR &= \overline{D}_1 \end{aligned} \tag{5.8}$$

Inputs		Present state		Next state		Outputs				
$(XR > 0)$	$(XR \geq YR)$	D_1	D_0	D_1^+	D_0^+	*Subtract*	*Swap*	*Select XY*	*Load XR*	*Load YR*
0	*d*	0	0	1	1	0	0	1	1	1
0	*d*	0	1	1	0	0	1	0	1	1
0	*d*	1	0	1	1	1	0	0	1	0
0	*d*	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	0	0	1	1	1
1	0	0	1	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	0	1	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	1	1	1
1	1	0	1	1	0	0	1	0	1	1
1	1	1	0	1	0	1	0	0	1	0
1	1	1	1	1	1	0	0	0	0	0

Figure 5.8
Excitation table for the control unit of the *gcd* processor.

One-hot method. While the classical design method minimizes a control unit's memory elements, its effect on the amount of combinational logic C is less obvious. Furthermore, control units designed by this technique tend to have a complicated, "random" structure, which makes design debugging and subsequent maintenance of the circuit difficult. An alternative approach that simplifies the

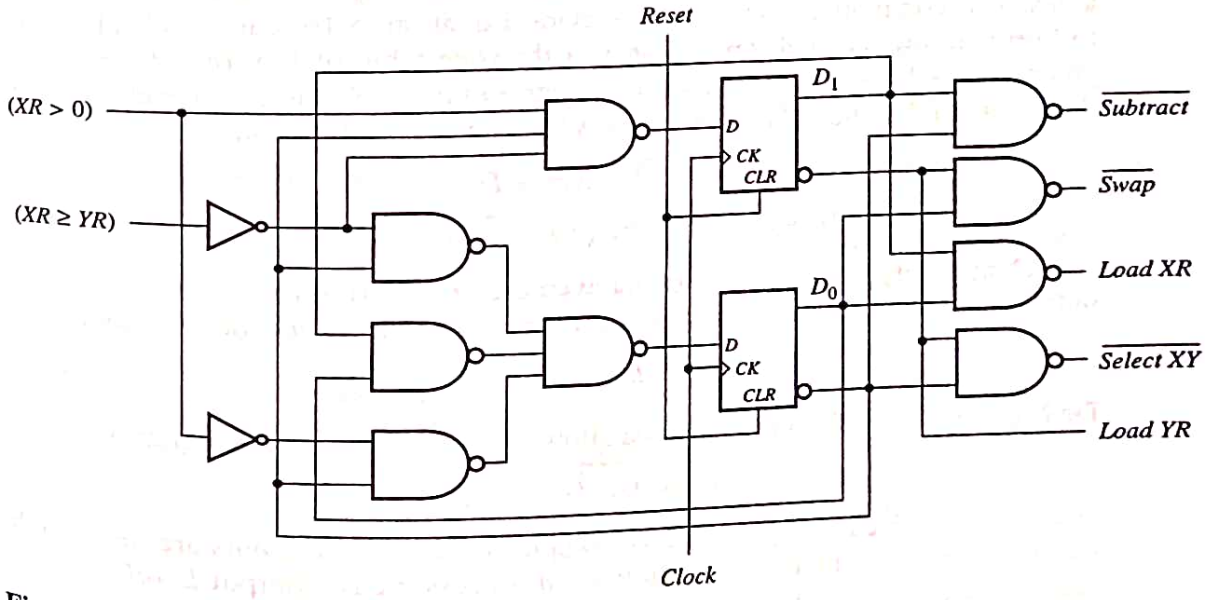


Figure 5.9
All-NAND classical design for the control unit of the *gcd* processor.

design process and gives C a regular and predictable structure, is the one-hot method, so called because its binary state assignment always contains a single 1—the "hot" bit—while all the remaining bits are 0. Thus the state assignment for a four-state machine like the *gcd* processor takes the following form:

$$\begin{aligned} S_0 &= 0\,0\,0\,1 \\ S_1 &= 0\,0\,1\,0 \\ S_2 &= 0\,1\,0\,0 \\ S_3 &= 1\,0\,0\,0 \end{aligned} \tag{5.9}$$

In general, P flip-flops are needed to represent P states, so the one-hot method is restricted to fairly small values of P.

A key feature of this technique is that the next-state and output equations have a simple, systematic form and can be written down directly from the control unit's original symbolic state table. Because the binary pattern assigned to each state is, in effect, fully decoded, we can find out whether the machine is in state S_i merely by inspecting the corresponding hot state variable D_i. The classical method requires us to check all state variables to get this information.

Suppose that state S_i in a one-hot design has the hot variable D_i. Further, suppose that $I_{j,1}, I_{j,2}, \ldots, I_{j,n_j}$ denote all input combinations that cause a state transition from S_j to S_i. Then each AND combination of the form $D_j \cdot I_{j,k}$ must make $D_i = 1$. Hence, considering all such combinations that cause transitions to S_i, we can write

$$D_i^+ = D_1(I_{1,1} + I_{1,2} + \ldots + I_{1,n_1}) + D_2(I_{2,1} + I_{2,2} + \ldots + I_{2,n_2}) + \ldots \tag{5.10}$$

This immediately yields the SOP form

$$D_i^+ = D_1 I_{1,1} + D_1 I_{1,2} + \ldots + D_1 I_{1,n_1} + D_2 I_{2,1} + D_2 I_{2,2} + \ldots + D_2 I_{2,n_2} + \ldots$$

which is practical to implement by an AND-OR or NAND-NAND circuit, provided that each state transition is determined by relatively few states and input variables, as is common in control-unit behavior. Equation (5.10) can also lead directly to fairly simple factored forms. Consider the state table of Figure 5.7 for the *gcd* processor's CU. State S_1 appears as a next state only for S_0 and S_2, in each case with the input combination $(XR > 0)\cdot(\overline{XR \geq XR})$. Hence (5.10) becomes

$$\begin{aligned} D_1^+ &= D_0 \cdot (XR > 0) \cdot (\overline{XR \geq XR}) + D_2 \cdot (XR > 0) \cdot (\overline{XR \geq XR}) \\ &= (D_0 + D_2) \cdot (XR > 0) \cdot (\overline{XR \geq XR}) \end{aligned}$$

The primary output equations are even easier to derive for one-hot designs. If output signal z_k is 1 (active) only in rows k,h for $h = 1,2,\ldots,m_k$, then we have

$$z_k = D_{k,1} + D_{k,2} + \ldots + D_{k,m_k} \tag{5.11}$$

De Morgan's law of Boolean algebra allows us to rewrite this OR equation as

$$z_k = \overline{\bar{D}_{k,1}\bar{D}_{k,2}\ldots\bar{D}_{k,m_k}}$$

in which form it can be generated by a single NAND whose inputs are the complemented outputs of the flip-flops. In the *gcd* processor case, output *Load YR* = 1 in states S_0 and S_1 only; therefore

$$Load\ YR = D_0 + D_1 = \overline{\bar{D}_0\bar{D}_1}$$

The entire set of next-state and output equations obtained by applying (5.10) and (5.11) to the *gcd* processor's CU follows.

$$D_0^+ = 0$$

$$D_1^+ = D_0 \cdot (XR > 0) \cdot \overline{(XR \geq XR)} + D_2 \cdot (XR > 0) \cdot \overline{(XR \geq XR)}$$

$$D_2^+ = D_0 \cdot (XR > 0) \cdot (XR \geq XR) + D_1 + D_2 \cdot (XR > 0) \cdot (XR \geq XR)$$

$$D_3^+ = D_0 \cdot \overline{(XR > 0)} + D_2 \cdot \overline{(XR > 0)} + D_3$$

$$Subtract = D_2$$

$$Swap = D_1$$

$$Select\ XY = D_0$$

$$Load\ XR = D_0 + D_1 + D_2$$

$$Load\ YR = D_0 + D_1$$

A NAND implementation of these equations appears in Figure 5.10. Note that the asynchronous *Reset* line must set D_0 to 1 and all other state variables to 0.

The steps of the one-hot design method for a Moore machine can be summarized as follows:

1. Construct a P-row state table that defines the desired input-output behavior.
2. Associate a separate D-type flip-flop D_i with each state S_i, and assign the P-bit one-hot binary code $D_1,D_2,\ldots,D_{i-1},D_i,D_{i+1},\ldots,D_P = 0,0,\ldots,0,1,0,\ldots,0$ to S_i.
3. Design a combinational circuit C that generates the primary and secondary output signals $\{D_i\}$ and $\{z_k\}$, respectively. D_i^+ is defined by the logic equation

$$D_i^+ = \sum_{i=1}^{P} D_i(I_{j,1} + I_{j,2} + \ldots + I_{j,n_j})$$

where $I_{j,1},I_{j,2},\ldots,I_{j,n_j}$ denote all input combinations that cause a transition from S_j to S_i. If $z_k = 1$ (active) only in rows k,h for $h = 1,2,\ldots,m_k$, then z_k is defined by

$$z_k = D_{k,1} + D_{k,2} + \ldots + D_{k,m_k} = \overline{\bar{D}_{k,1}\bar{D}_{k,2}\ldots\bar{D}_{k,m_k}}$$

We next present an example that illustrates the application of the one-hot method to a computer's IO interface, specifically to a direct-memory access (DMA) controller, which handles data transfers between main memory and high-speed IO devices. (DMA communication is discussed in Chapter 7.)

EXAMPLE 5.1 DESIGN OF A DMA CONTROLLER. This problem, which is adapted from [Actel 1994], is representative of control units that link several interacting systems—in this case, main memory and a set of IO devices. The target machine is the control part of a four-channel DMA controller of the kind found in the IO subsystem of most computers. It is a six-state Moore-type machine with four input and five output signals, which are identified as follows:

Inputs:	*IOREQ*	Any of four data-transfer request signals
	CONT	Continue (indicates pending, unprocessed requests)

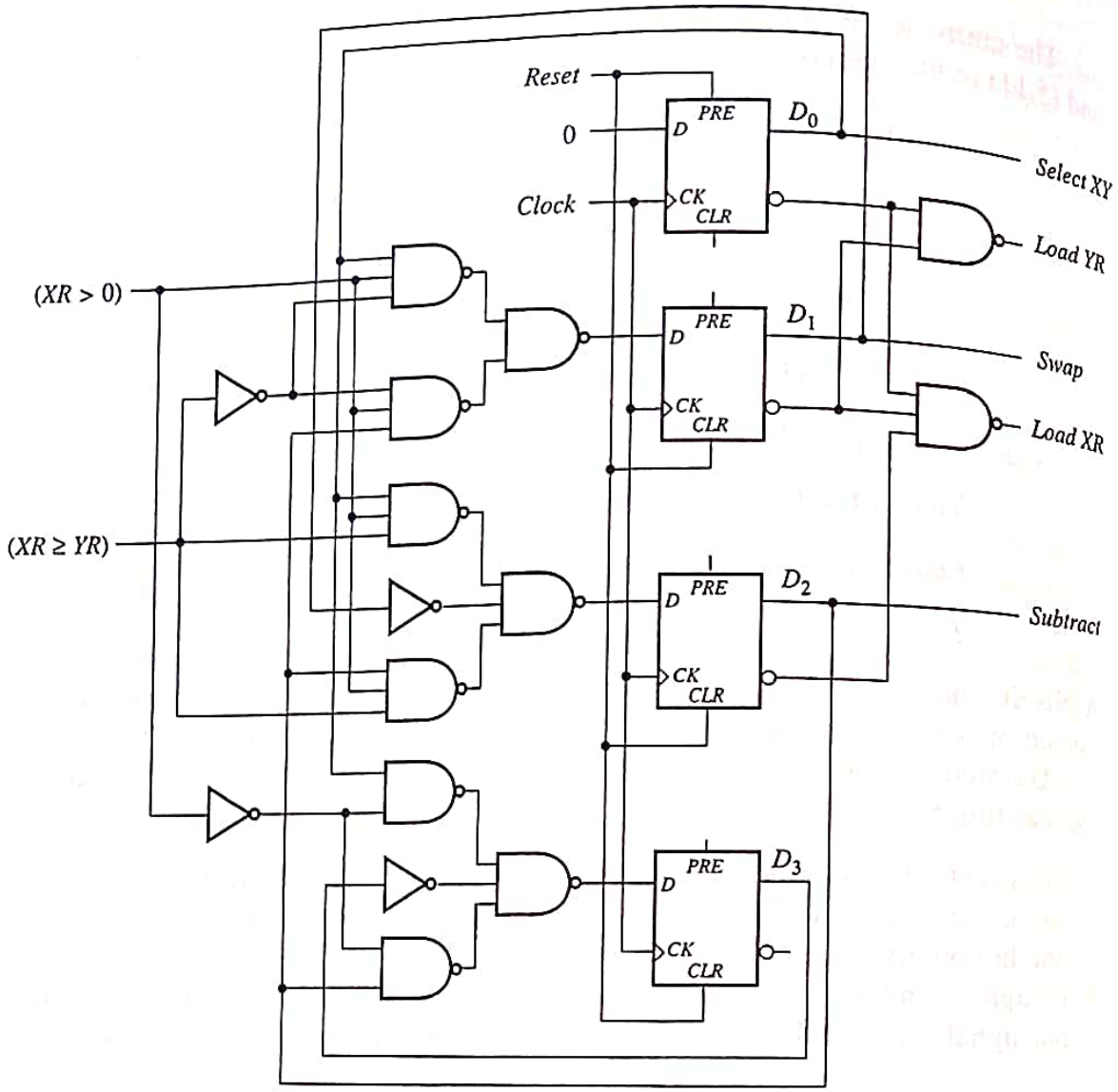
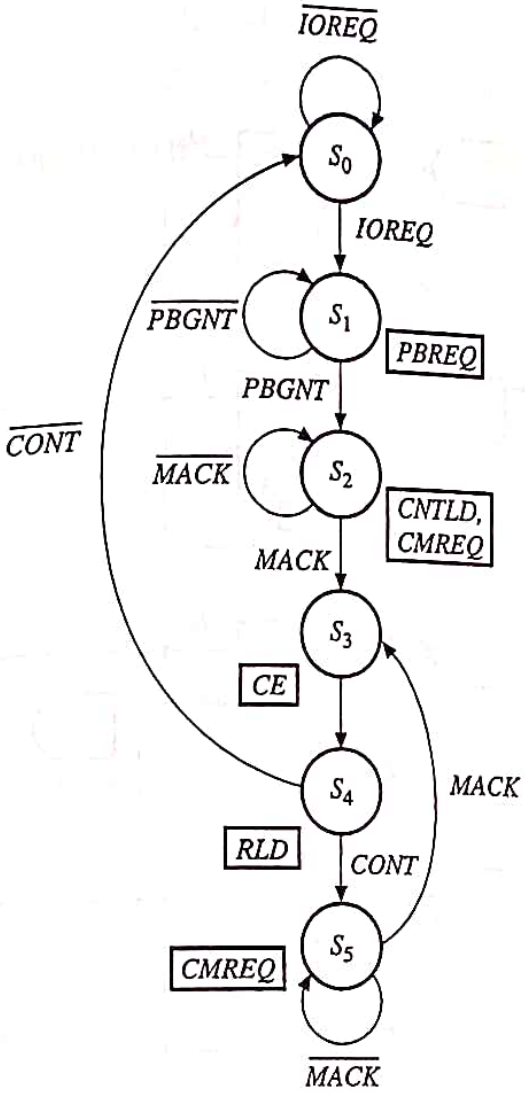


Figure 5.10
All-NAND one-hot design for the control unit of the *gcd* processor.

	MACK	Memory transfer acknowledgment
	PBGNT	Processor bus grant (indicates availability of data-transfer bus)
Outputs:	*CE*	Count enable (bookkeeping function)
	CMREQ	Channel memory request
	CNTLD	Counter load (bookkeeping function)
	RLD	Register load (bookkeeping function)
	PBREQ	Processor bus request for control of data-transfer bus

The behavior of the DMA controller is given by the state transition diagram of Figure 5.11*a*. Each transition is marked with the corresponding active input control signals. Since every transition is triggered by only one such signal, this notation is quite compact. Each state is marked with the (boxed) name of the output control signals that it activates—the number of such signals ranges from zero to two. A state table in the style of Figure 5.4 that is equivalent to Figure 5.11*a* is easy to construct, but it is large because of the many possible input combinations. Noting that most input signals do not affect a given state transition, and so are assigned the don't-care value *d*, we can condense the state table into the compact form of Figure 5.11*b*.



(*a*)

Inputs				Present state	Next state	Outputs				
IOREQ	*CONT*	*MACK*	*PBGNT*			*PBREQ*	*CNTLD*	*CMREQ*	*RLD*	*CE*
0	*d*	*d*	*d*	S_0	S_0	0	0	0	0	0
1	*d*	*d*	*d*	S_0	S_1	0	0	0	0	0
d	*d*	*d*	0	S_1	S_1	1	0	0	0	0
d	*d*	*d*	1	S_1	S_2	1	0	0	0	0
d	*d*	0	*d*	S_2	S_2	0	1	1	0	0
d	*d*	1	*d*	S_2	S_3	0	1	1	0	0
d	*d*	*d*	*d*	S_3	S_4	0	0	0	0	1
d	0	*d*	*d*	S_4	S_0	0	0	0	1	0
d	1	*d*	*d*	S_4	S_5	0	0	0	1	0
d	*d*	0	*d*	S_5	S_5	0	0	1	0	0
d	*d*	1	*d*	S_5	S_3	0	0	1	0	0

(*b*)

Figure 5.11
State behavior of the DMA controller: (*a*) state transition graph and (*b*) condensed state table.

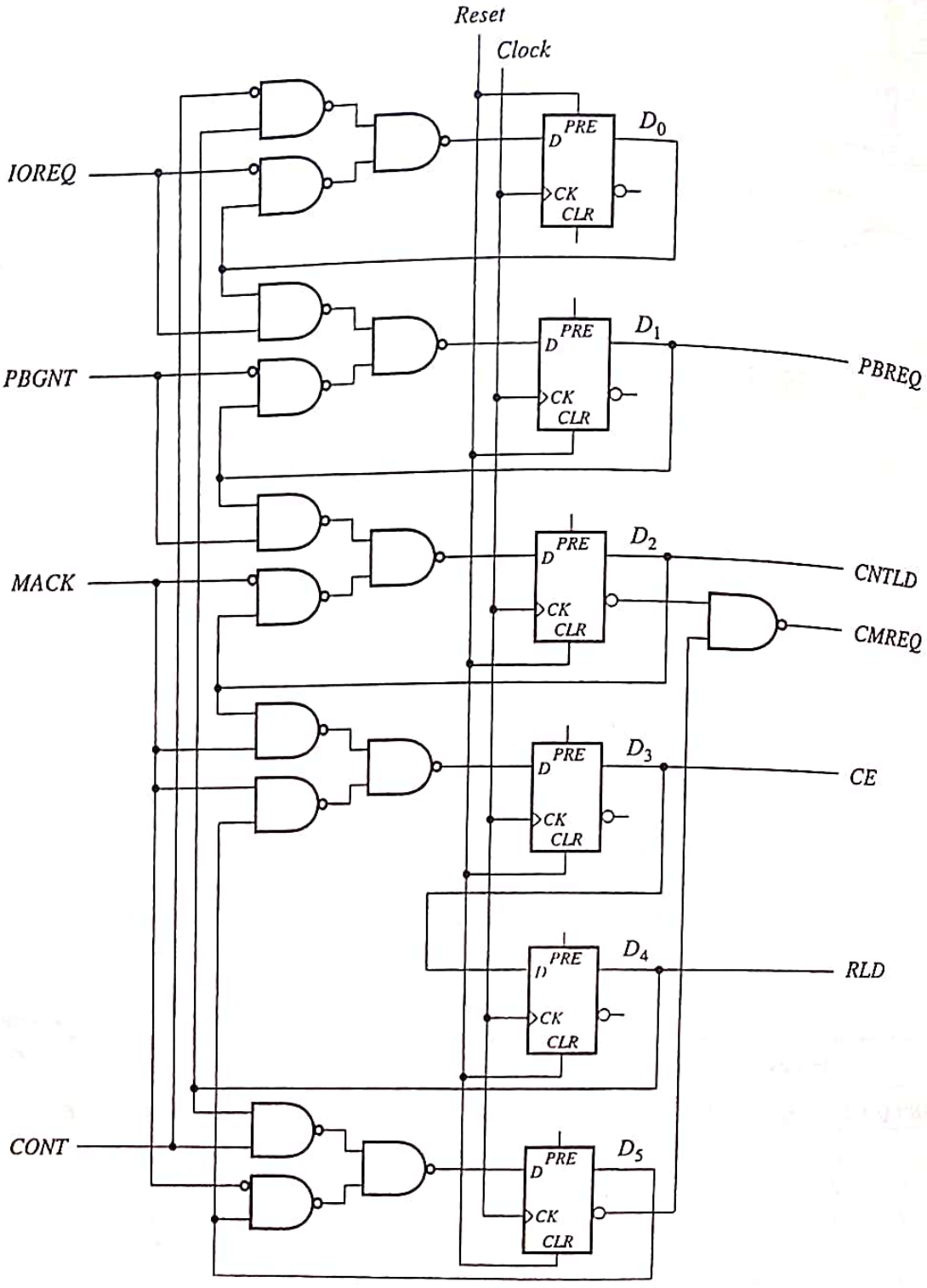


Figure 5.12
One-hot design for the DMA controller.

Now we assign a D flip-flop D_i to each state S_i and write down the six state-transition equations directly from Figure 5.11.

$$D_0^+ = D_0 \cdot \overline{IOREQ} + D_4 \cdot \overline{CONT}$$

$$D_1^+ = D_0 \cdot IOREQ + D_1 \cdot \overline{PBGNT}$$

$$D_2^+ = D_1 \cdot PBGNT + D_2 \cdot \overline{MACK}$$

$$D_3^+ = D_2 \cdot MACK + D_5 \cdot MACK$$

$$D_4^+ = D_3$$

$$D_5^+ = D_4 \cdot CONT + D_5 \cdot \overline{MACK}$$

The output equations are also immediately obtained from Figure 5.11.

$$CE = D_3$$

$$CMREQ = D_2 + D_5$$

$$CNTLD = D_2$$

$$RLD = D_4$$

$$PBREQ = D_1$$

Figure 5.12 shows an all-NAND circuit derived from these equations. Note the regular structure of the combinational logic, which is typical of one-hot designs. An equivalent classical design has three flip-flops but a much more irregular combinational part.

5.1.3 Design Examples

This section presents some examples to illustrate the foregoing methods for designing hardwired control units. We will use these examples again in our discussion of microprogrammed control.

Multiplier control. First consider the design of a control unit CU for the twos-complement (Robertson) multiplier introduced in Example 4.2 (section 4.1.2). The block diagram of the multiplier's datapath unit DP (Figure 4.12) is redrawn in expanded form in Figure 5.13 to show a set of control points, which represent abstractly the control signals and associated logic circuits needed to link CU and DP. These control signals are derived from the multiplication algorithm in Figure 4.13 and are listed in Figure 5.14. In general, a control point can be associated with each distinct action (register-transfer operation) op_i appearing in the algorithm being implemented. Its enabling control signal c_i is inserted into the component or interconnections associated with op_i. Operations that take place simultaneously may be able to share control signals. (Procedures to eliminate redundant control signals are considered later.) The statement labeled BEGIN in Figure 4.13, for instance, requires the registers A, COUNT, and F to be reset simultaneously to the all-zero state. A single control signal c_{10} is therefore provided for this purpose. It can be connected directly to the CLEAR inputs of the three registers in question, so no additional logic is needed to implement the c_{10} control point. Control signals c_8 and c_9 transfer a data word from the input bus INBUS to registers Q and M, respectively, and are shown in the corresponding data paths of Figure 5.13; these signals may be connected to the registers' (parallel) LOAD inputs. Control signal c_5 serves to change the function performed by the parallel adder from addition to subtraction for the correction step; it also resets Q[0] to 0. The remaining control signals of Figure 5.14 are defined similarly. Figure 5.13 introduces a control signal called COUNT7, which is set to 1 when COUNT = 111_2 and is set to 0 otherwise. COUNT7, the right-most bit Q[0] of the multiplier register Q and the external BEGIN signal serve as the primary inputs to CU.

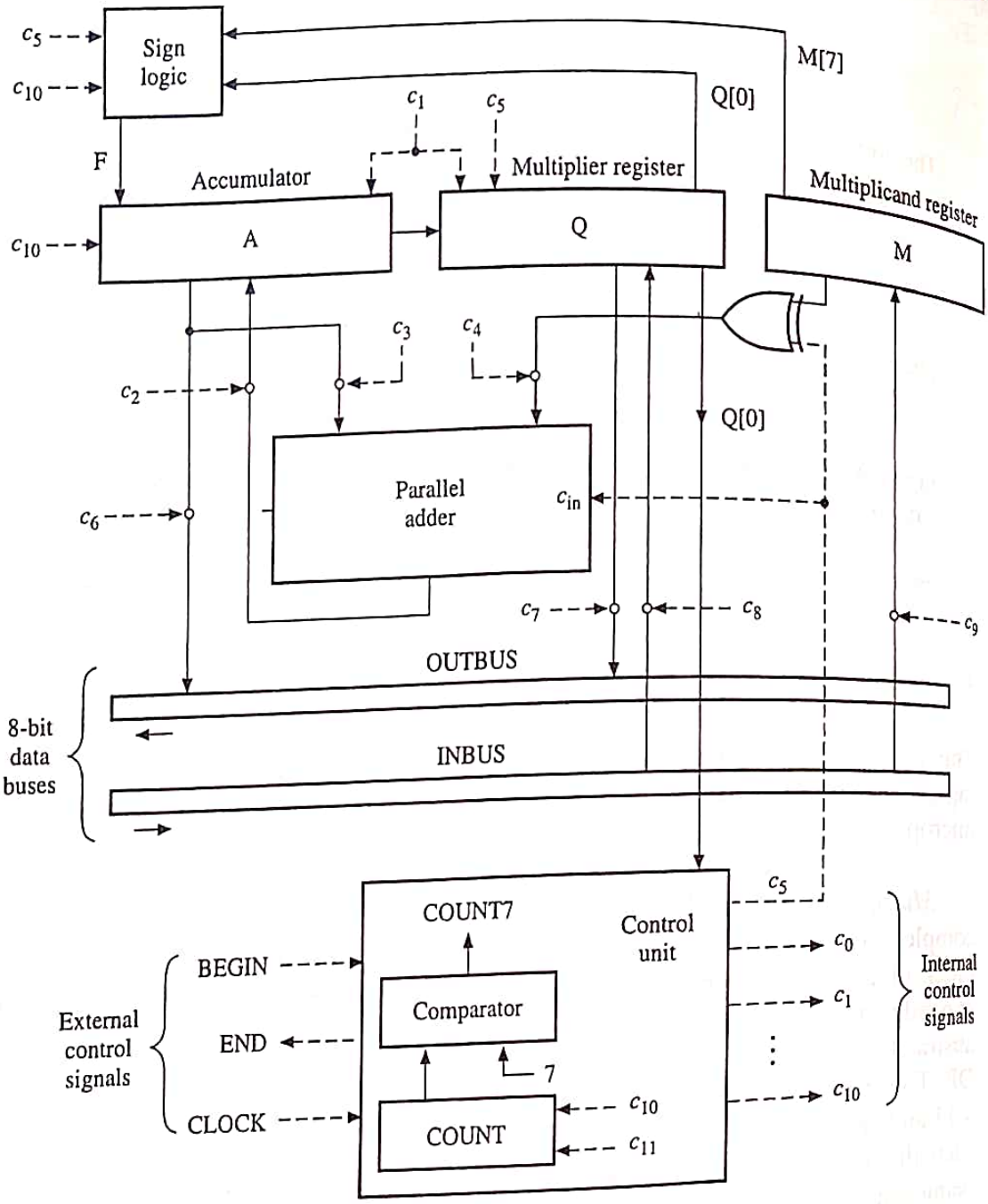


Figure 5.13
Twos-complement multiplier with a set of control points.

The multiplication algorithm is reformulated as a flowchart in Figure 5.15 to display the control signals from Figure 5.14 and indicate when each one is activated. The flowchart resembles a state transition graph that describes the behavior of both the control and datapath units. To obtain a state table for the control unit CU, we associate a state S_i with every operation block in Figure 5.15, leading to the seven states labeled S_1:S_7. An additional state S_0 represents the reset or waiting condition of the control unit. CU has three primary input signals—BEGIN, Q[0], and COUNT7; hence there are eight possible input combinations. Figure 5.16 shows an eight-state Moore-type state table in the style of Figure 5.7, which is derived directly from Figure 5.15. This state table is not necessarily the smallest such table defining the desired control function. In fact, the 13 output control signals c_0:c_{11}, END can be immediately reduced to 8 because several sets are equivalent in that they are always activated together, specifically $c_0 = c_1 = c_{11}$, $c_2 = c_3 = c_4$, and $c_9 =$

Control Signal	Operation controlled
	Set sign bit of A to F.
c_0	Right-shift register-pair A.Q.
c_1	Transfer adder output to A.
c_2	Transfer A to left input of adder.
c_3	Transfer M to right input of adder.
c_4	Perform subtraction (correction). Clear Q[0].
c_5	Transfer A to output bus.
c_6	Transfer Q to output bus.
c_7	Transfer word on input bus to Q.
c_8	Transfer word on input bus to M.
c_9	Clear A, COUNT, and F registers.
c_{10}	Increment COUNT.
c_{11}	Completion signal (CU idle).
END	

Figure 5.14
Control signals for the twos-complement multiplier.

c_{10}. Methods also exist that attempt to reduce the number of states by merging "compatible" states; see Problem 5.8. To eliminate a flip-flop in this case, we would need to reduce the number of states of CU from eight to four or fewer, which is not possible.

In the following example, we apply the two basic hardwired design techniques, classical and one hot, to the multiplier, taking Figures 5.15 and 5.16 as starting points.

EXAMPLE 5.2 IMPLEMENTING A MULTIPLIER CONTROL UNIT. The multiplier control unit CU is small enough that the classical design approach can be applied to it. This technique uses the minimum number of flip-flops, in this case three, whose outputs $D_2D_1D_0$ denote CU's internal state. We make the natural assignment of the bit-pattern $D_2D_1D_0 = i$ to state S_i, yielding $S_0 = 000$, $S_1 = 001$, $S_2 = 010$, and so on. The remaining problem is to design the combinational logic circuit portion C of CU, a straightforward but fairly tedious task, because C has six inputs BEGIN, Q[0], COUNT7,D_2,D_1,D_0 and 11 outputs: D_2^+,D_1^+,D_0^+,c_0,c_2,c_5,c_6,c_7,c_8,c_{11},END.

The behavior of CU's combinational logic C is defined by an excitation table, which is obtained by replacing the symbolic states of Figure 5.16 with the corresponding 3-bit patterns. It remains to design C. Figure 5.17 shows the results of applying the two-level optimization program *espresso* to this problem. The *espresso* program implements an efficient algorithm that finds a minimum or near-minimum number of product terms (prime implicants), in this case only 13, needed to define C. The input description (Figure 5.17*a*) is a 64-row excitation table for C written in *espresso*'s PLA-style format. For example, the last row of Figure 5.17*a*

111111 00000001000

specifies the state transition from S_7 to S_0, with input combination BEGIN Q[0] COUNT7 = 111 and active output $c_7 = 1$. The last row of *espresso*'s solution (Figure 5.17*b*)

---110 11100010000

specifies the transition from S_6 to S_7, with input combination BEGIN Q[0] COUNT7 = --- denoting don't cares, and active output signal $c_6 = 1$. This captures the fact that S_6 is always followed by S_7, independent of CU's primary input signals. A NAND realization of the multiplier CU appears in Figure 5.18, which directly realizes the minimum SOP form obtained via *espresso*.

We can also implement CU directly from the state table of Figure 5.16, or, equally easily, from the flowchart of Figure 5.15 by the one-hot method. Eight flip-flops are needed to accommodate CU's eight states S_0:S_7. Equations (5.10) and (5.11) give the next-state and output equations. The next-state equations are

$$D_0^+ = D_0 \cdot \overline{BEGIN} + D_7$$
$$D_1^+ = D_0 \cdot BEGIN$$
$$D_2^+ = D_1$$
$$D_3^+ = D_2 \cdot Q[0] + D_4 \cdot Q[0] \cdot \overline{COUNT7}$$
$$D_4^+ = D_2 \cdot \overline{Q[0]} + D_3 + D_4 \cdot \overline{Q[0]} \cdot \overline{COUNT7}$$
$$D_5^+ = D_4 \cdot Q[0] \cdot COUNT7$$
$$D_6^+ = D_5 + D_4 \cdot \overline{Q[0]} \cdot \overline{COUNT7}$$
$$D_7^+ = D_6$$

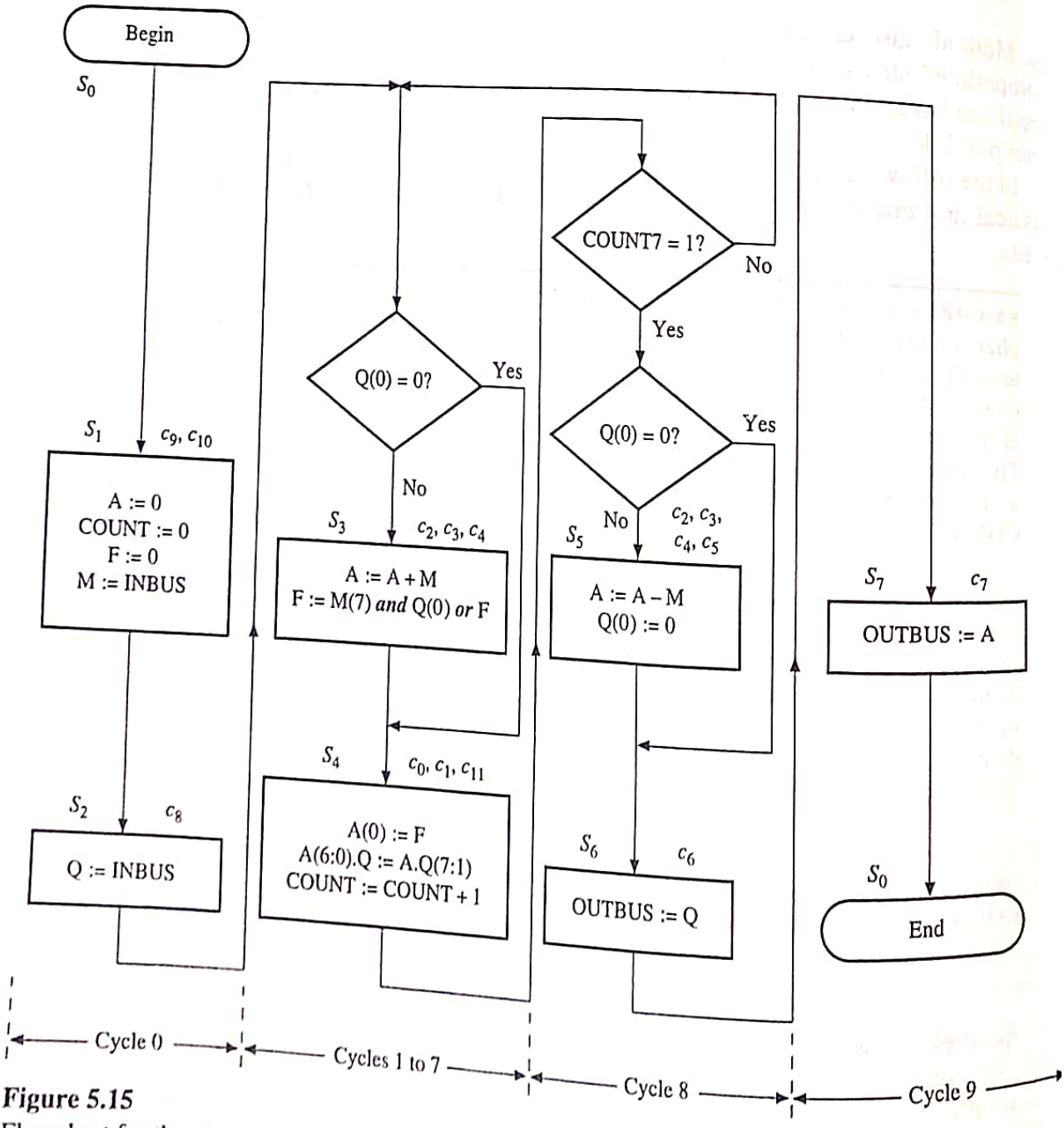


Figure 5.15
Flowchart for the twos-complement multiplier.

State	Inputs: BEGIN Q[0] Count7								Outputs												
	000	001	010	011	100	101	110	111	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	END
S_0	S_0	S_0	S_0	S_0	S_1	S_1	S_1	S_1	0	0	0	0	0	0	0	0	0	0	0	0	1
S_1	S_2	S_2	S_2	S_2	S_2	S_2	S_2	S_2	0	0	0	0	0	0	0	0	0	1	1	0	0
S_2	S_4	S_4	S_3	S_3	S_4	S_4	S_3	S_3	0	0	0	0	0	0	0	0	1	0	0	0	0
S_3	S_4	S_4	S_4	S_4	S_4	S_4	S_4	S_4	0	0	1	1	1	0	0	0	0	0	0	0	0
S_4	S_4	S_6	S_3	S_5	S_4	S_6	S_3	S_5	1	1	0	0	0	0	0	0	0	0	0	1	0
S_5	S_6	S_6	S_6	S_6	S_6	S_6	S_6	S_6	0	0	1	1	1	1	0	0	0	0	0	0	0
S_6	S_7	S_7	S_7	S_7	S_7	S_7	S_7	S_7	0	0	0	0	0	0	1	0	0	0	0	0	0
S_7	S_0	S_0	S_0	S_0	S_0	S_0	S_0	S_0	0	0	0	0	0	0	0	1	0	0	0	0	0

Figure 5.16
State table for the multiplier control unit.

```
.model  mult007.pla
.inputs  BGN Q0 CT7 D2 D1 D0
.outputs  D2+ D1+ D0+ c0 c2 c5 c6
          c7 c8 c9 END

.i 6
.o 11
.p 64
000000  00000000001
001000  00000000001
010000  00000000001
011000  00000000001
100000  00100000001
101000  00100000001
110000  00100000001
111000  00100000001
000001  01000000010
001001  01000000010
010001  01000000010
011001  01000000010
100001  01000000010
101001  01000000010
110001  01000000010
111001  01000000010
000010  10000000100
001010  10000000100
  .  .  .  .  .  .
101111  00000001000
110111  00000001000
111111  00000001000
.end
```

(*a*)

```
model  mult005.pla
.inputs  BGN Q0 CT7 D2 D1 D0
.outputs  D2+ D1+ D0+ c0 c2 c5 c6
          c7 c8 c9 END
.i 6
.o 11
.p 13
-011-0  01000000000
1--000  00100000000
-10100  01110000000
-11100  10110000000
---111  00000001000
-0-010  10000000100
-0-100  10010000000
---011  10001000000
-1-010  01100000100
---000  00000000001
---001  01000000010
---101  11001100000
---110  11100010000
.e
```

(*b*)

Figure 5.17
Design of combinational part of the multiplier control unit by *espresso*: (*a*) input data (excitation table) and (*b*) output data (optimized SOP specification).

The output equations are

$$c_0 = c_1 = c_{11} = D_4$$

$$c_2 = c_3 = c_4 = D_3 + D_5$$

$$c_5 = D_5$$

$$c_6 = D_6$$

$$c_7 = D_7$$

$$c_8 = D_2$$

$$c_9 = c_{10} = D_1$$

$$END = D_0$$

The NAND circuit implied by these equations appears in Figure 5.19.

Despite having more flip-flops, the one-hot design is better in many ways than the classical design. The one-hot design has fewer and generally smaller gates in its combi-

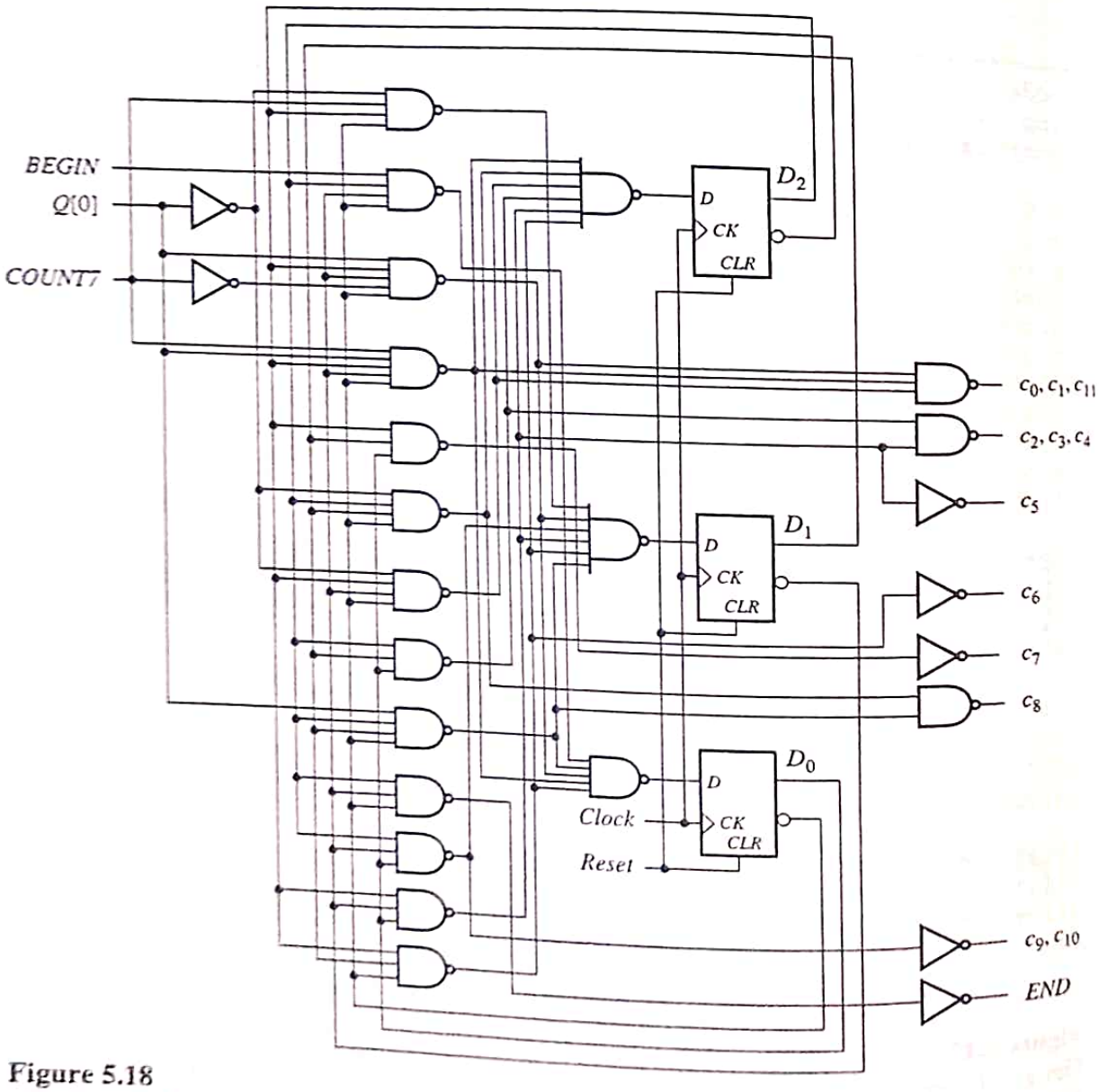


Figure 5.18
All-NAND classical design for the multiplier control unit.

national logic because entry to a particular state depends on a small number of primary and secondary (state) input variables—as few as one state variable in several cases. This dependence also holds for the output functions, since most of the primary output signals can be taken directly from the corresponding hot-state variable. Another point worth noting is that the one-hot CU's structure closely follows that of its state behavior, as exemplified by the flowchart specification (Figure 5.15). Consequently, the one-hot design is easier to understand and easier to modify (should that be necessary) than the classical design.

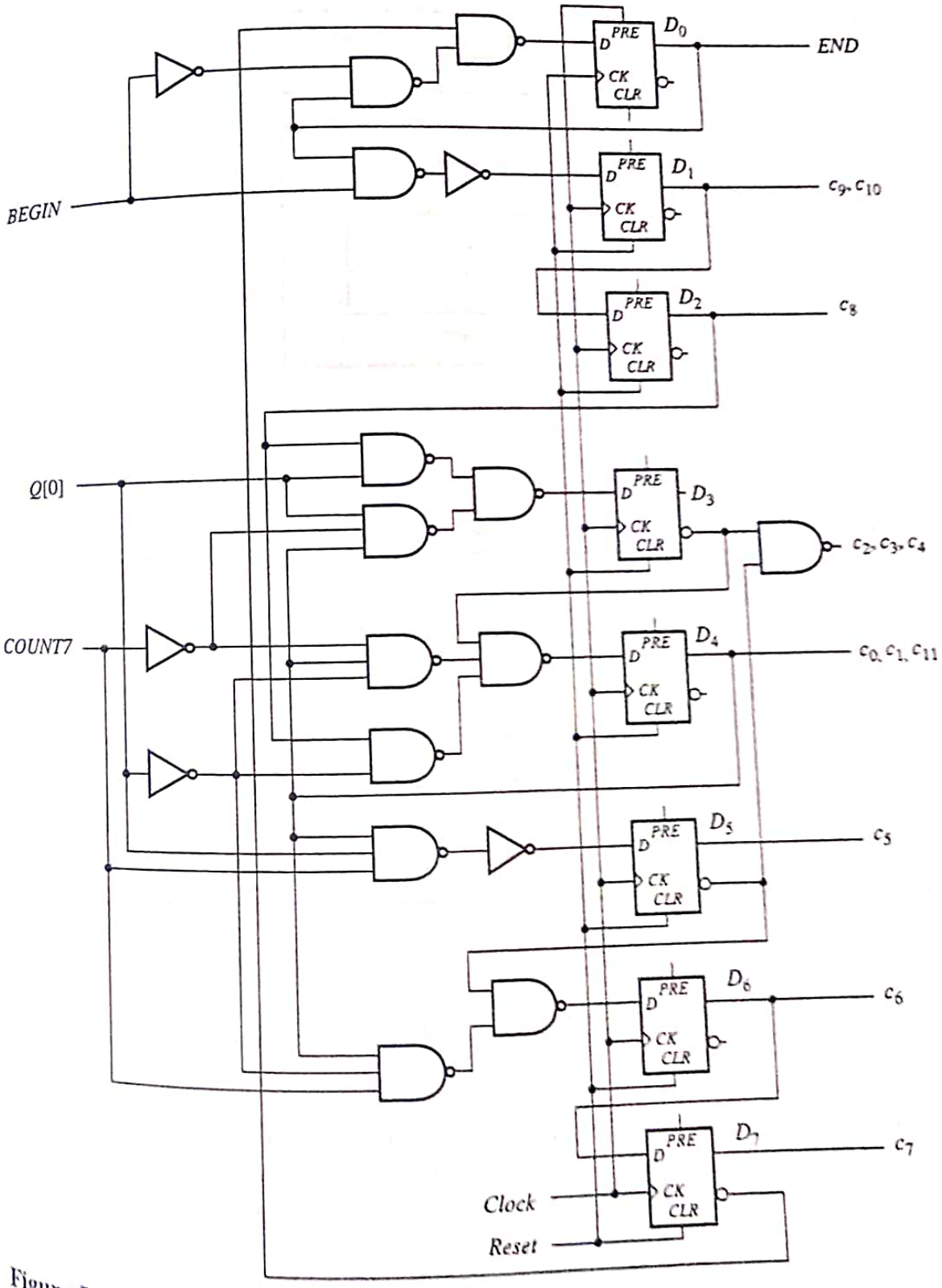
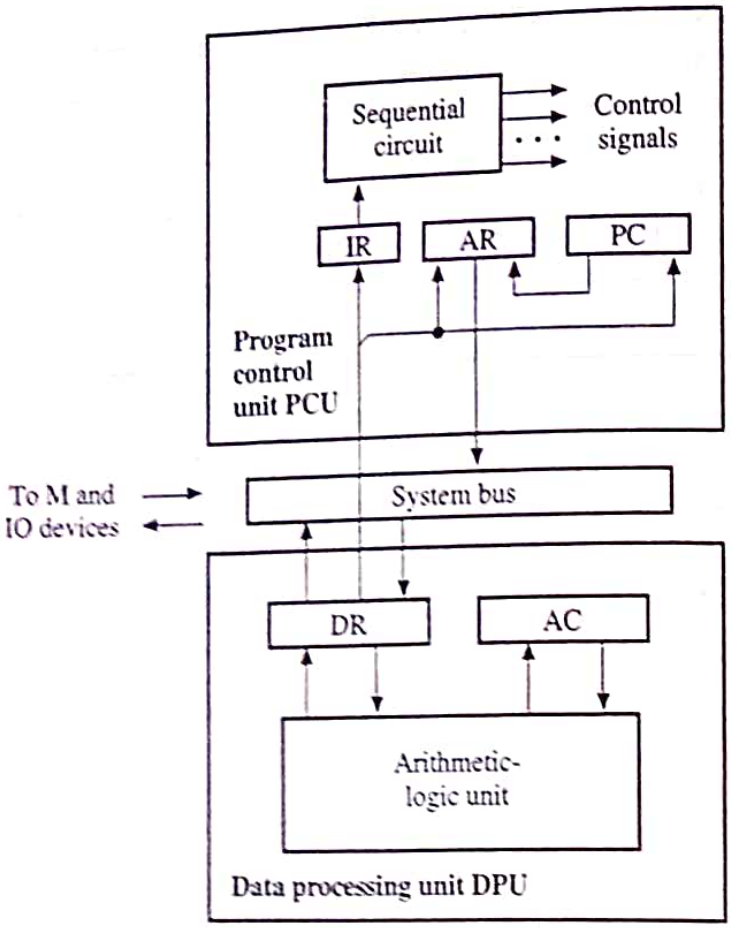


Figure 5.19
All-NAND one-hot design for the multiplier control unit.



(*a*)

Type	HDL format	Assembly format	Comment
Data transfer	AC := M(X)	LD X	Load X from M into AC
	M(X) := AC	ST X	Store contents of AC in M as X
	DR := AC	MOV DR, AC	Copy contents of AC to DR
	AC := DR	MOV AC, DR	Copy contents of DR to AC
Data processing	AC := AC + DR	ADD	Add DR to AC
	AC := AC – DR	SUB	Subtract DR from AC
	AC := AC *and* DR	AND	And DR to AC
	AC := *not* AC	NOT	Complement contents of AC
Program control	PC := M(*adr*)	BRA adr	Jump to instruction with address *adr*
	if AC = 0 then PC := M(*adr*)	BZ adr	Jump to instruction *adr* if AC = 0

(*b*)

Figure 5.20
An accumulator-based CPU: (*a*) organization and (*b*) instruction set.

CPU control unit. The design of the CU for a basic, nonpipelined CPU differs mainly in degree—the CPU is a multifunction unit that can contain hundreds of control lines—but not in kind from the multiplier control unit. Here we examine a few of the design issues involved, using an accumulator-based CPU as an example. In section 5.3.3 we will discuss the complex control problems associated with pipelined CPUs.

The accumulator-based CPU introduced in section 3.1.1 has the overall organization depicted in Figure 5.20*a* (which repeats Figure 3.3). This CPU consists of a datapath unit DPU designed to execute the set of 10 basic single-address instructions listed in Figure 5.20*b*. The instructions are assumed to be of fixed length and to act on data words of the same fixed length, say 32 bits. The program control unit PCU is responsible for managing the control signals linking the PCU to the DPU, as well as the control signals between the CPU and the external memory M.

To design the PCU, we must first identify the relevant control actions (microoperations) needed to process the given instruction set using the hardware from Figure 5.20*a*. A flowchart description of the behavior of the CPU appears in Figure 5.21, which is similar in form to the multiplier's flowchart (Figure 5.15). All instructions require a common instruction-fetch step, followed by an execution step that varies with each instruction type. The fetch step copies the contents of the program counter PC to the memory address register AR. A memory-read operation is then executed, which transfers the instruction word *I* to memory data register DR; this is expressed by DR := M(AR). *I*'s opcode is transferred to the instruction register IR, where it is decoded; at the same time PC is incremented to point to the next consecutive instruction in M.

The subsequent operations depend on the opcode pattern. For example, the store instruction ST *X* is executed in three steps: the address field of ST *X* is transferred to AR, the contents of the accumulator AC are transferred to DR, and finally the memory write operation M(AR) := DR is performed. The branch-on-zero instruction BZ *adr* is executed by first testing AC. If AC $\neq$ 0, no action is taken; if AC = 0, the address field *adr*, which is in DR(ADR), is transferred to PC, thus effecting the branch operation. Figure 5.21 implies that instruction fetching takes three cycles, while instruction execution takes from one to three cycles. As we will see later, RISC processors are usually designed so that all instruction execution times are equalized to one CPU clock period T_C in length, making the cycles associated with the register-transfer operations in Figure 5.21 into subcycles of T_C.

The microoperations appearing in the flowchart implicitly determine the control signals and control points needed by the CPU. Figure 5.22*b* lists a suitable set of control signals for the CPU and their functions, while Figure 5.22*a* shows the approximate positions of the corresponding control points in both the PCU and DPU. These control lines can be placed in the three basic groups defined earlier.

- Function select: c_2, c_9, c_{10}, c_{11}, c_{12}.
- Storage control: c_1, c_8.
- Data routing: c_0, c_3, c_4, c_5, c_6, c_7.

Here *storage control* refers to the external memory M. Many of the control signals transfer information between the CPU's internal data and control registers.

Control unit design. The overall organization of a hardwired control unit that implements the flowchart of Figure 5.21 appears in Figure 5.23. It is assumed that the opcode stored in the instruction register IR is decoded into 10 signals, one per instruction type, which along with BEGIN and a status signal (AC = 0) form the inputs to the main sequential circuit FSM that generates the control signals c_0:c_{12}. Hence FSM has 12 primary inputs and 13 primary outputs. The number of internal states can be estimated from Figure 5.21. If each distinct action box is assigned to a different state,

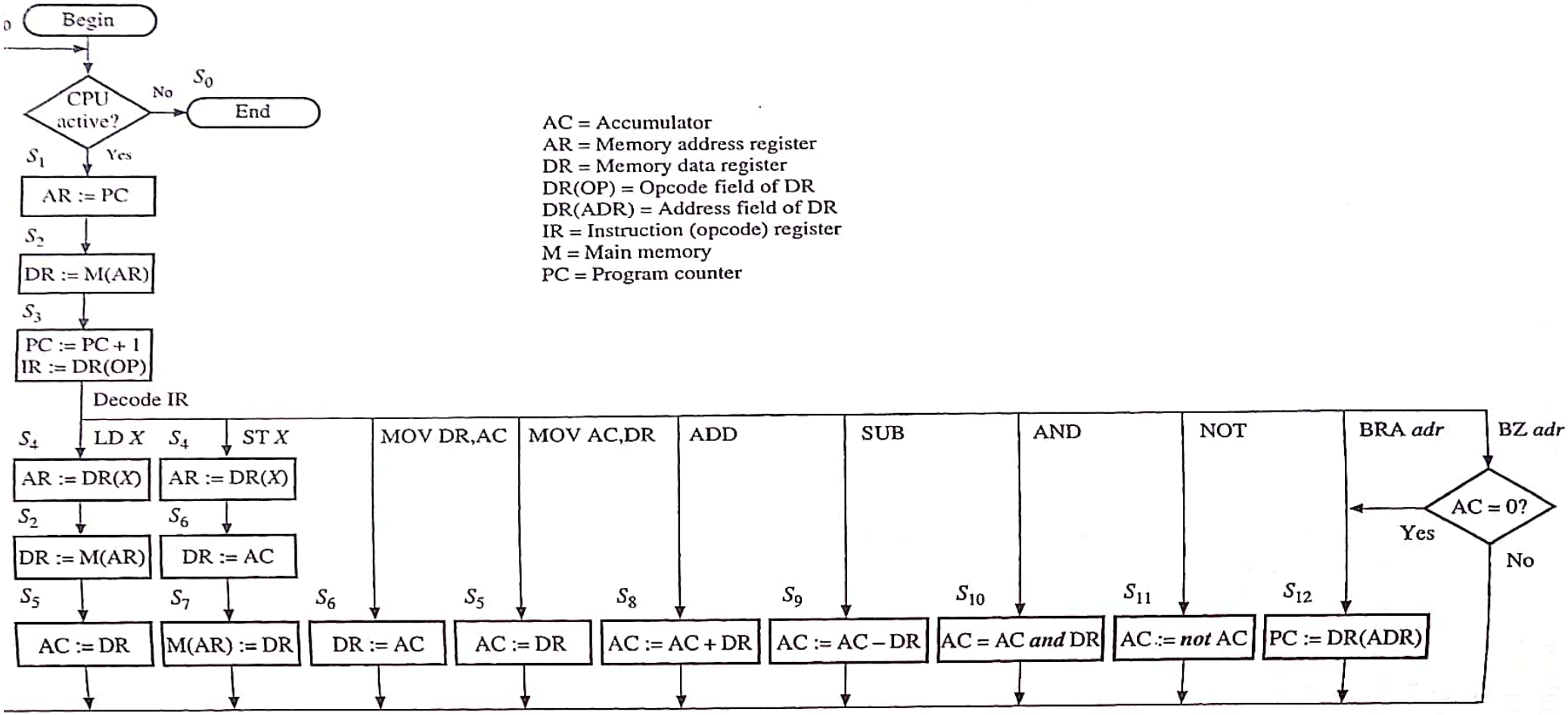
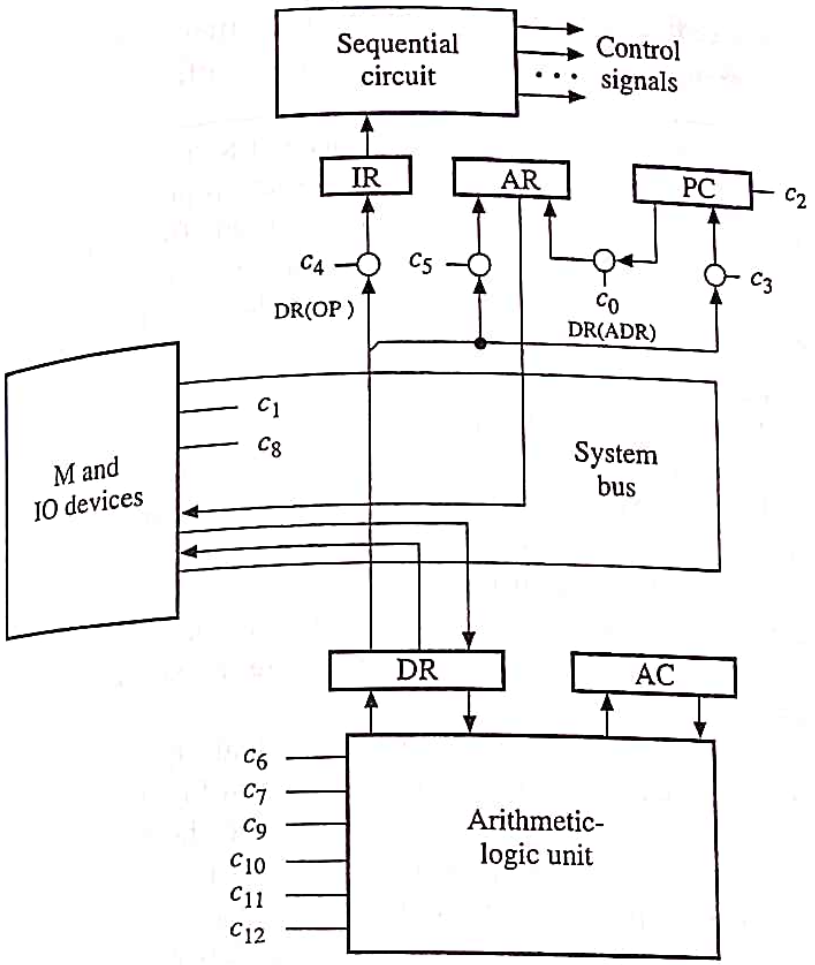


Figure 5.21
Flowchart of the accumulator-based CPU.



(*a*)

Control signal	Operation controlled
c_0	AR := PC
c_1	DR := M(AR)
c_2	PC := PC + 1
c_3	PC := DR(ADR)
c_4	IR := DR(OP)
c_5	AR := DR(ADR)
c_6	DR := AC
c_7	AC := DR
c_8	M(AR) := DR
c_9	AC := AC + DR
c_{10}	AC := AC – DR
c_{11}	AC := AC ***and*** DR
c_{12}	AC := ***not*** AC

(*b*)

Figure 5.22
(*a*) Control points and (*b*) control signal definitions for the accumulator-based CPU.

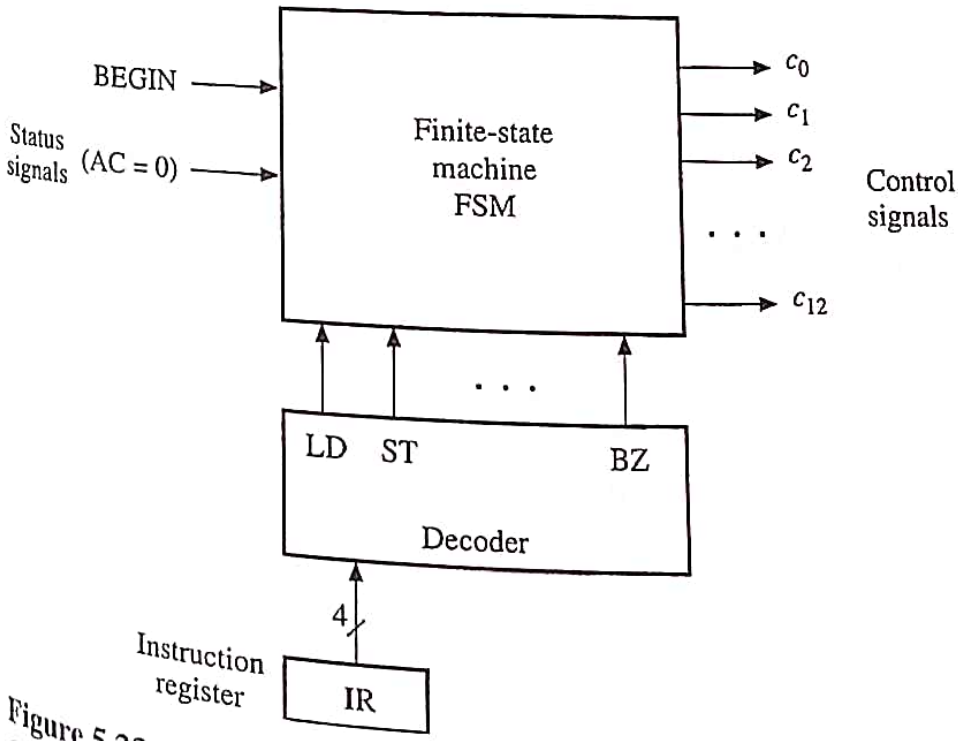


Figure 5.23
Organization of a hardwired control unit for the accumulator-based CPU

then there are 13 states S_0:S_{12}, as indicated on the flowchart. Implementing the resulting 13-state machine via the classical or one-hot methods is straightforward.

EXAMPLE 5.3 IMPLEMENTING A PROGRAM CONTROL UNIT. The circuit FSM of Figure 5.23 issues the control signals governing instruction processing in the 10-instruction accumulator-based CPU. Its behavior is defined by the flowchart in Figure 5.21. We will implement FSM using a minor variant of our earlier one-hot method that reduces the number of flip-flops needed, while maintaining the simplicity of a one-hot design. Most of the states S_4:S_{12} identified in Figure 5.21 for the execution phase of the instruction can be distinguished by the opcode-type signals LD, ST, and so on, which are primary inputs of FSM. If we do not require FSM to be a Moore machine, we can coalesce the states into a smaller set whose output actions (the control signals they activate) are determined by FSM's primary inputs as well as its states. Specifically, we can replace the states in the execution phase by just three states S_4^*,S_5^*,S_6^*, all of which are visited in sequence by the load and store instructions, but which reduce to a single state S_4^* for the remaining instructions. We can also reduce the instruction-fetch phase to a sequence of three states by merging S_0 and S_1 into one state S_1^* so that, whether active or inactive, FSM performs the operation AR := PC.

The resulting machine has the state behavior depicted by a Mealy-type state transition graph in Figure 5.24. This figure follows the condensed style of Figure 5.11 where only the signals that directly affect or are affected by each state are shown. For example, the state transition graph implies that when in state S_5^*, a transition is made to S_6^* with output $c_1 = 1$ (the only active output) if the current instruction is of type LD. This event is indicated by the label LD/c_1 on the S_5^*-to-S_6^* transition arrow. If the current instruction is ST, however, then only c_6 becomes 1, as indicated by the label ST/c_6. No other instruction types allow FSM to enter state S_5^*. When the transition from state S_i to S_j is automatic, that is, it is independent of the primary input signals, we used a label of the form $\mathcal{Q}/O_j$.

Let us implement FSM using six D flip-flops, with the output D_i of the ith flip-flop forming the hot variable for state S_i or S_i^*. We can now write down a set of logic equations directly from Figure 5.24 that define FSM.

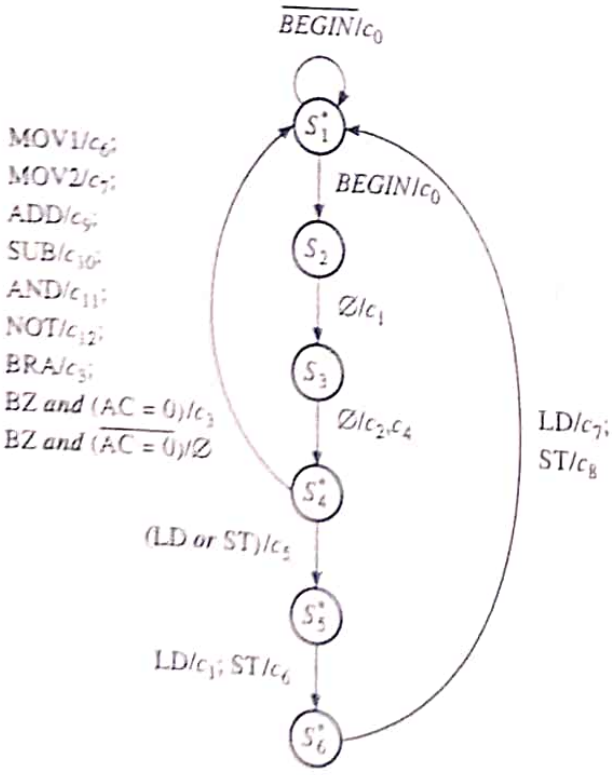


Figure 5.24
State transition graph for the accumulator-based CPU.

$$D_1^+ = D_1 \cdot \overline{BEGIN} + D_4 \cdot (\text{MOV1} + \text{MOV2} + \text{ADD} + \text{SUB} + \text{AND} + \text{NOT} + \text{BRA} + \text{BZ}) + D_6$$
$$D_2^+ = D_1 \cdot BEGIN$$
$$D_3^+ = D_2$$
$$D_4^+ = D_3$$
$$D_5^+ = D_4 \cdot (\text{LD} + \text{ST})$$
$$D_6^+ = D_5$$

The output equations also follow immediately from Figure 5.24.

$$c_0 = D_1 \qquad c_1 = D_2 + D_5 \cdot \text{LD}$$
$$c_2 = c_4 = D_3 \qquad c_3 = D_4 \cdot (\text{BRA} + \text{BZ} \cdot (\text{AC} = 0))$$
$$c_5 = D_4 \cdot (\text{LD} + \text{ST}) \qquad c_6 = D_4 \cdot \text{MOV1} + D_5 \cdot \text{ST}$$
$$c_7 = D_4 \cdot \text{MOV2} + D_6 \cdot \text{LD} \qquad c_8 = D_6 \cdot \text{ST}$$
$$c_9 = D_4 \cdot \text{ADD} \qquad c_{10} = D_4 \cdot \text{SUB}$$
$$c_{11} = D_4 \cdot \text{AND} \qquad c_{12} = D_4 \cdot \text{NOT}$$

These equations lead to the logic circuit in Figure 5.25.

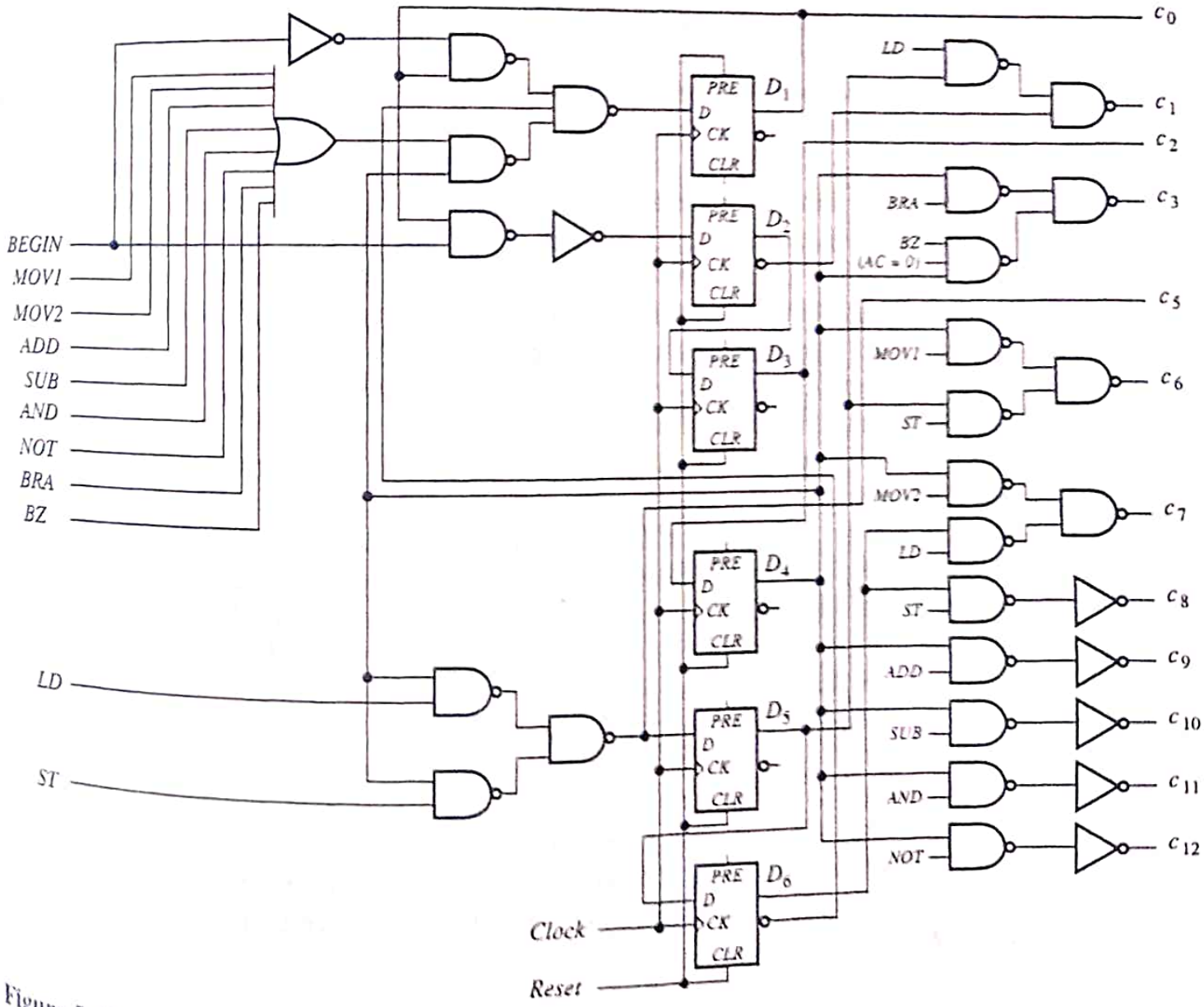


Figure 5.25
One-hot implem[illegible] of Figure 5.24

5.2 MICROPROGRAMMED CONTROL

We turn next to the design of control units that use microprograms to select, interpret, and execute a processor's instruction set.

5.2.1 Basic Concepts

An instruction is implemented by a sequence of one or more sets of concurrent microoperations. Each microoperation is associated with a group of control lines that must be activated in a prescribed sequence to trigger the microoperations. As the number of instructions and control lines can be in the hundreds, a hardwired control unit is difficult to design and verify, even with good CAD tool support. Furthermore, such a control unit is inherently inflexible in the sense that changes, for example, to correct design errors or update the instruction set, require that the control unit be redesigned.

Microprogramming [Lynch 1993] is a method of control-unit design in which the control signal selection and sequencing information is stored in a ROM or RAM called a *control memory* CM. The control signals to be activated at any time are specified by a *microinstruction*, which is fetched from CM in much the same way an instruction is fetched from main memory. Each microinstruction also explicitly or implicitly specifies the next microinstruction to be used, thereby providing the necessary information for microoperation sequencing. A set of related microinstructions forms a *microprogram*. Microprograms can be changed relatively easily by changing the contents of CM; hence microprogramming yields control units that are more flexible than their hardwired counterparts. This flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry. There is also a performance penalty due to the time required to access the microinstructions from CM. These disadvantages have discouraged the use of microprogramming in RISCs and other high-speed processors, where chip area and circuit delay must both be minimized. Microprogramming continues to be used in such CISCs as the Pentium and 680X0.

In a microprogrammed CPU, each machine instruction is executed by a microprogram which acts as a real-time interpreter for the instruction. The set of microprograms that interpret a particular instruction set or machine language L is called an *emulator* for L. A microprogrammed computer C_1 can be made to execute programs written in the machine language L_2 of another, very similar computer C_2 by placing an emulator for L_2 in the control memory of C_1. In that case C_1 is said to be able to *emulate* C_2.

As a design activity, microprogramming can be compared with assembly-language programming; however, the microprogrammer requires a more detailed knowledge of the processor hardware than the assembly-language programmer. Symbolic languages similar to assembly languages are used to write microprograms: these are called *microassembly languages*. A *microassembler* is necessary to translate microprograms into executable programs that can be stored in the control memory.

Control unit organization. In its simplest form a microinstruction has two parts: a set of *control fields* that specify the control signals to be activated and an

address field that contains the address in CM of the next microinstruction to be executed. In the original scheme proposed by Maurice V. Wilkes, the inventor of microprogramming, each bit k_i of a control field corresponds to a distinct control line c_i [Wilkes 1951]. When $k_i = 1$ in the current microinstruction, c_i is activated; otherwise c_i remains inactive. Figure 5.26 shows a microprogrammed control unit designed in this style. The control memory CM is implemented by a ROM of the type discussed in section 2.2.2. The left part (AND plane) of the ROM decodes an address obtained from the *control memory address register* (CMAR). Each address selects a particular row in the right part (OR plane) of the ROM that contains a microinstruction composed (in this small example) of a 6-bit control field and a 3-bit address field. When the top-most row in Figure 5.26, which represents the microinstruction with address 000, is selected, the control signals c_0, c_2, and c_4 are activated, as indicated by the ×s in the control field. At the same time, the contents of the address field $a_2a_1a_0 = 001$ are sent to the CMAR, where they are stored and used to address the next microinstruction to be executed.

As Figure 5.26 indicates, the CMAR can be loaded from an external source as well as from the address field of a microinstruction. The external source typically provides the starting address of a microprogram in the CM. A specific microprogram prestored in CM executes (interprets) each instruction of a microprogrammed

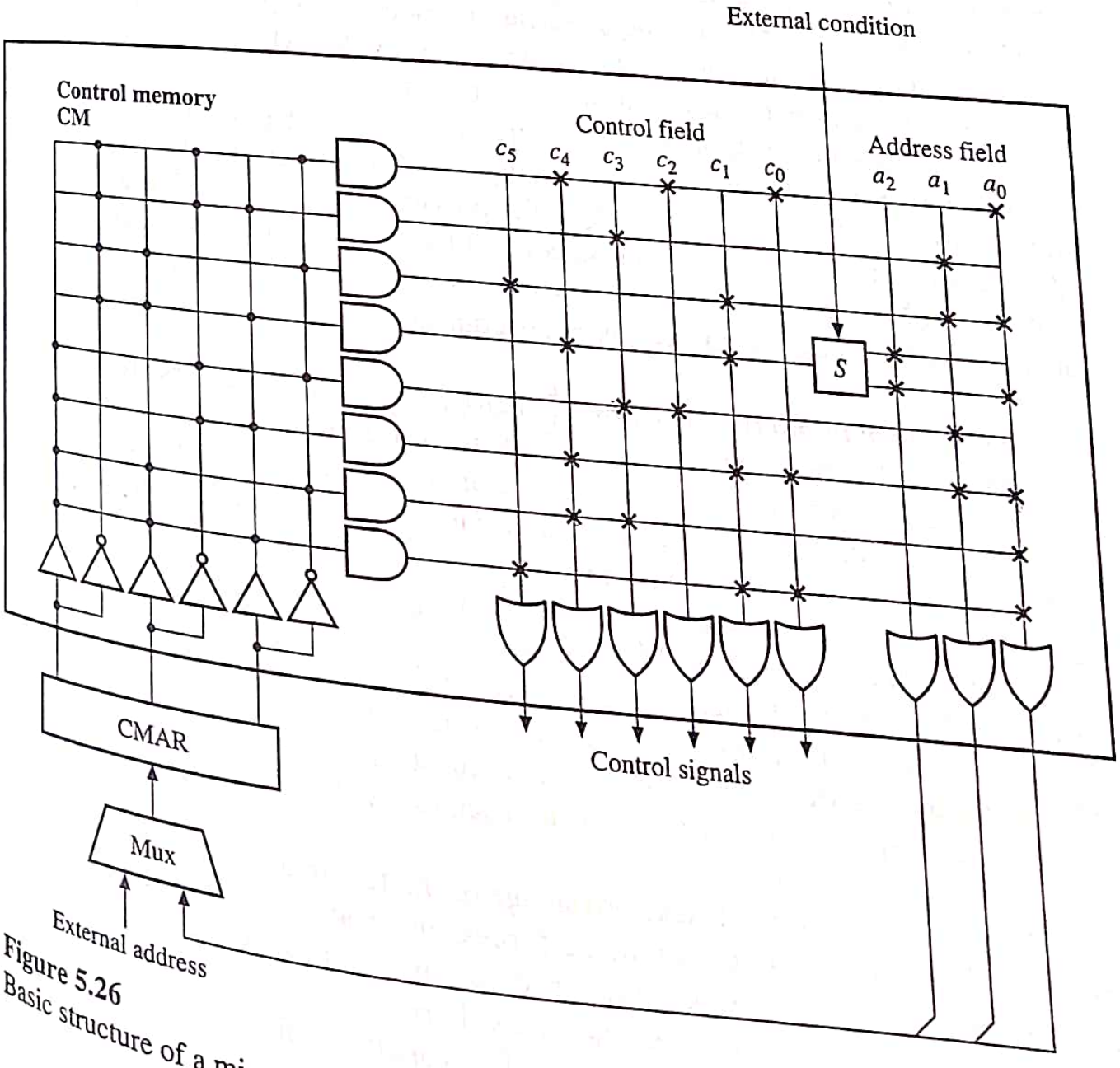


Figure 5.26
Basic structure of a mi