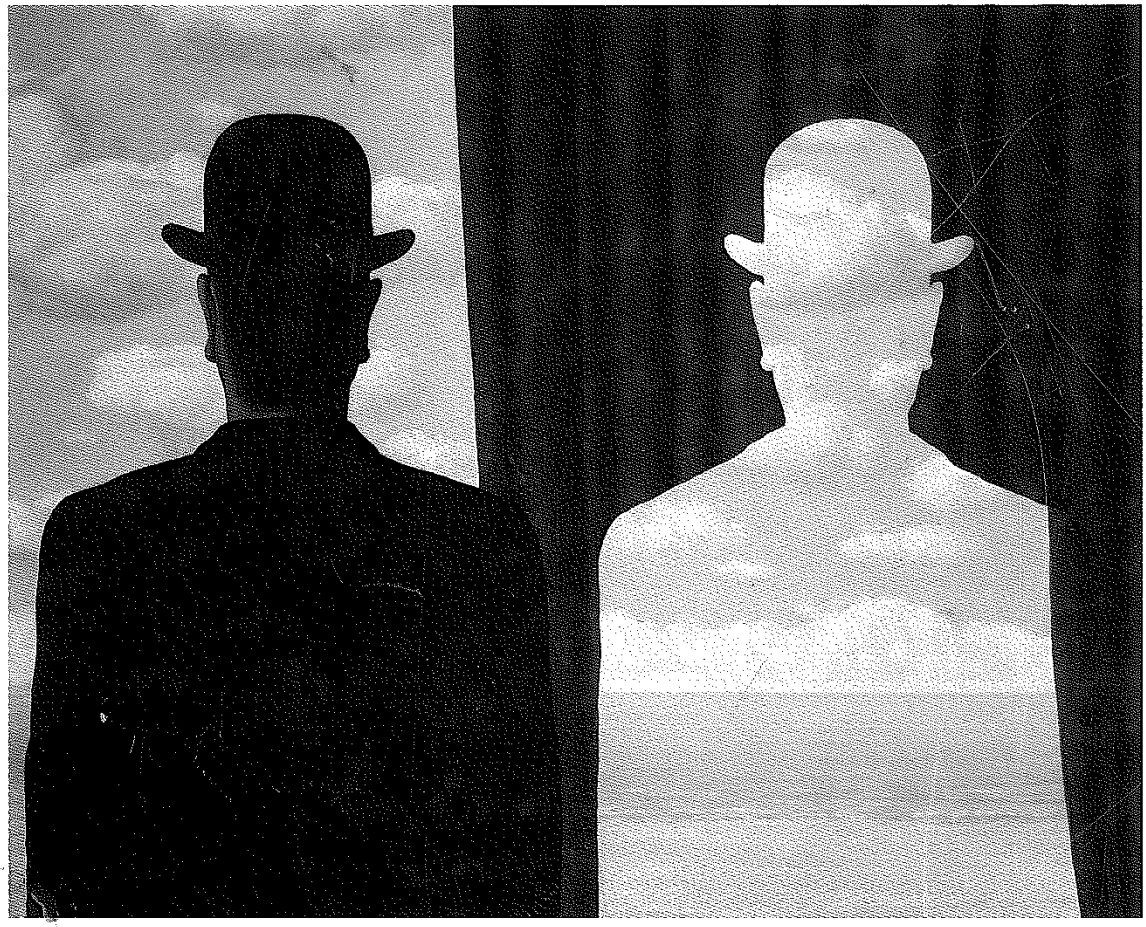
Petitioner’s Exhibit CSC 1014

Randy H. Katz

Contemporary Logic Design



Executive Editor: Dan Joraanstad
Sponsoring Editor: Jennifer Young
Developmental Editor: Jamie Spencer
Editorial Assistant: Laura Cheu
Marketing Manager: Mary Tudor
Production Editors: Megan Rundel, Brian Jones
Text and Cover Design: Rob Hugel, XXX Design
Copyeditor: Mary Prescott
Proofreader: Elizabeth Wiltsee
Illustrations: Rolin Graphics Inc.
Indexer: Ira Kleinberg
Composition: Rad Proctor, Proctor-Willenbacher
Film: The Courier Connection
Cover Printer: New England Book Components, Inc.
Text Printer and Binder: Courier/Westford
Senior Manufacturing Coordinator: Merry Free Osborn

The cover illustration is a reproduction of "Décalcomanie" by the twentieth-century Belgian surrealist painter René Magritte. We chose this particular painting by the author's favorite painter to reflect the duality of computer design. "Décalcomanie" © 1992 C. Herscovici/ARS, New York.

Library of Congress Cataloging-in-Publication Data

Katz, Randy H., 1955–
Contemporary logic design / Randy H. Katz.
p. cm.
Includes bibliographical references and index.
ISBN 0-8053-2703-7
1. Electronic digital computers—Circuits—Design. 2. Integrated circuits—Very large scale integration—Design—Data processing. 3. Logic design—Data processing. 4. Computer-aided design. I. Title.
TK7888.4.K36 1994
621.39'5—dc20 93-9013
CIP

ISBN 0-8053-2703-7

3 4 5 6 7 8 9 10-CRW-999897969594

The Benjamin/Cummings Publishing Company, Inc.
390 Bridge Parkway
Redwood City, California 94065

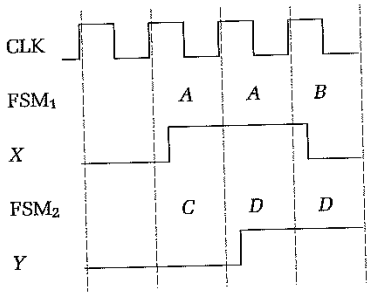


Figure 8.8 State and output changes associated with the FSM fragments of Figure 8.7.

Now that Y is 1, FSM_1 goes to state B on the next rising edge. In this state, it will output a 0, but this is too late to affect FSM_2's state change. It remains in state D.

8.2 Basic Design Approach

The counter design procedure presented in the last chapter forms the core of a more general procedure for arbitrary finite state machines. You will discover that the procedure must be significantly extended for the general case.

8.2.1 Finite State Machine Design Procedure

Step 1: *Understand the problem.* A finite state machine is often described in terms of an English-language specification of its behavior. It is important that you interpret this description in an unambiguous manner. For counters, it is sufficient simply to enumerate the sequence. For finite state machines, try some input sequences to be sure you understand the conditions under which the various outputs are generated.

Step 2: *Obtain an abstract representation of the FSM.* Once you understand the problem, you must place it in a form that is easy to manipulate by the procedures for implementing the finite state machine. A state diagram is one possibility. Other representations, to be introduced in the next section, include algorithmic state machines and specifications in hardware description languages.

Step 3: *Perform state minimization.* Step 2, deriving the abstract representation, often results in a description that has too many states. Certain paths through the state machine can be eliminated because their input/output behavior is duplicated by other function-

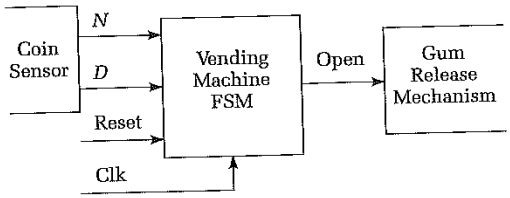


Figure 8.9 Vending machine block diagram.

the customer? Sometimes we have to make reasonable assumptions. For the first question, we assume that the coin sensor returns any coins it does not recognize, leaving N and D unasserted. For the latter, we assume that external logic resets the machine after the gum is delivered.

Abstract Representations Once you understand the behavior reasonably well, it is time to *map the specification into a more suitable abstract representation.* A good way to begin is by enumerating the possible unique sequences of inputs or configurations of the system. These will help define the states of the finite state machine.

For this problem, it is not too difficult to enumerate all the possible input sequences that lead to releasing the gum:

Three nickels in sequence: N, N, N

Two nickels followed by a dime: N, N, D

A nickel followed by a dime: N, D

A dime followed by a nickel: D, N

Two dimes in sequence: D, D

This can be represented as a state diagram, as shown in Figure 8.10. For example, the machine will pass through the states S_0, S_1, S_3, S_7 if the input sequence is three nickels.

To keep the state diagram simple and readable, we include only transitions that explicitly cause a state change. For example, in state S_0, if neither input N or D is asserted, we assume the machine remains in state S_0 (the specification allows us to assume that N and D are never asserted at the same time). Also, we include the output Open only in states in which it is asserted. Open is implicitly unasserted in any other state.

State Minimization This nine-state description isn't the "best" possible. For one thing, since states S_4, S_5, S_6, S_7, and S_8 have identical behavior, they can be combined into a single state.

To reduce the number of states even further, we can think of each state as representing the amount of money received so far. For example,

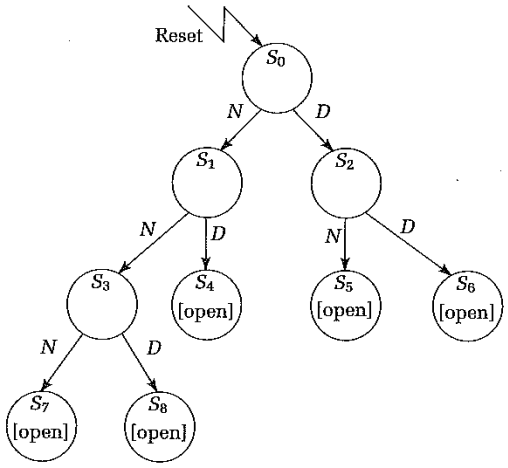


Figure 8.10 Vending machine state diagram.

it shouldn't matter whether the state representing 10 cents was reached through two nickels or one dime.

A state diagram derived in this way is shown in Figure 8.11. We capture the behavior in only four states, compared with nine in Figure 8.10. Also, as another illustration of a useful shorthand, notice the transition from state 10¢ to 15¢. We interpret the notation "*N, D*" associated with this transition as "go to state 15¢ if *N* is asserted OR *D* is asserted."

In the next chapter, we will examine formal methods for finding a state diagram with the minimum number of states. The process of minimizing the states in a finite state machine description is called *state minimization.*

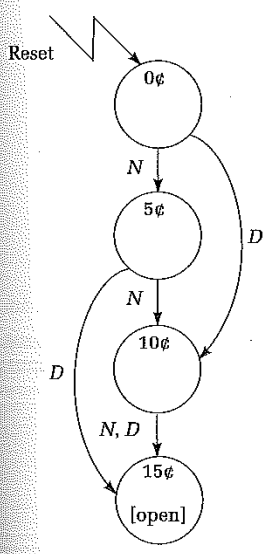


Figure 8.11 Minimized vending machine state diagram.

State Encoding At this point, we have a finite state machine with a minimum number of states, but it is still symbolic. See Figure 8.12 for the symbolic state transition table. The next step is *state encoding.*

The way you encode the state can have a major effect on the amount of hardware you need to implement the machine. A natural state assignment would encode the states in 2 bits: state 0¢ as 00, state 5¢ as 01, state 10¢ as 10, and state 15¢ as 11. A less obvious assignment could lead to reduced hardware. The *encoded* state transition table is shown in Figure 8.13.

In Chapter 9 we present a variety of methods and computer-based tools for finding an effective state encoding.

Implementation The next step is to implement the state transition table after choosing storage elements. We will look at implementations based on *D* and *J-K* flip-flops.

Present State	Inputs D	N	Next State	Output Open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Figure 8.12 Minimized vending machine symbolic state transition table.

Present State Q_1	Q_0	Inputs D	N	Next State D_1	D_0	Output Open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	X	X	X
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	X	X	X
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	X	X	X
1	1	0	0	1	1	1
		0	1	1	1	1
		1	0	1	1	1
		1	1	X	X	X

Figure 8.13 Encoded vending machine state transition table.

The K-maps for the D flip-flop implementation are shown in Figure 8.14. We filled these in directly from the encoded state transition table. The minimized equations for the flip-flop inputs and the output become

$$D_1 = Q_1 + D + Q_0 \bullet N$$

$$D_0 = N \bullet \overline{Q}_0 + Q_0 \bullet \overline{N} + Q_1 \bullet N + Q_1 \bullet D$$

$$\text{OPEN} = Q_1 \bullet Q_0$$

The logic implementation is shown in Figure 8.15. It uses eight gates and two flip-flops.

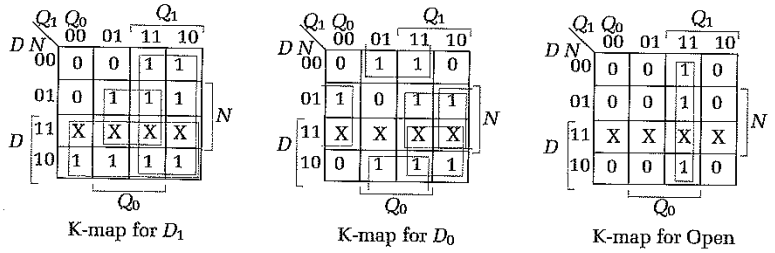


Figure 8.14 K-maps for *D* flip-flop implementation of vending machine.

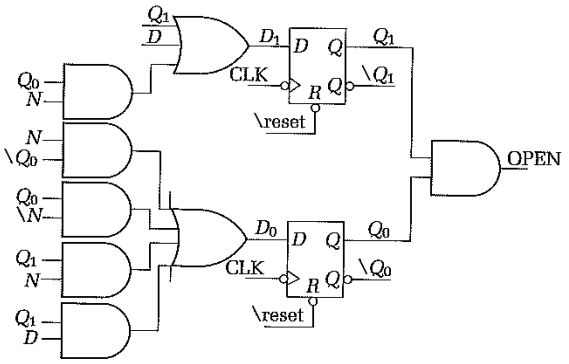


Figure 8.15 Vending machine FSM implementation based on *D* flip-flops.

To implement the state machine using *J-K* flip-flops, we must remap the next-state functions as in Chapter 7. The remapped state transition table for *J-K* flip-flop implementation is shown in Figure 8.16. We give the K-maps derived from this table in Figure 8.17. The minimized equations for the flip-flop inputs become

$$J_1 = D + Q_0 \bullet N \qquad K_1 = 0$$
$$J_0 = \overline{Q}_0 \bullet N + Q_1 \bullet D \qquad K_0 = \overline{Q}_1 \bullet N$$

Figure 8.18 shows the logic implementation. Using *J-K* flip-flops moderately reduced the hardware: seven gates and two flip-flops.

Discussion We briefly described the complete finite state machine design process and illustrated it by designing a simple vending machine controller. Starting with an English-language statement of the task, we first described the machine in a more formal representation. In this case, we used state diagrams.

Present State Q_1	Q_0	Inputs D	N	Next State D_1	D_0	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	X	0	X
		0	1	0	1	0	X	1	X
		1	0	1	0	1	X	0	X
		1	1	X	X	X	X	X	X
0	1	0	0	0	1	0	X	X	0
		0	1	1	0	1	X	X	1
		1	0	1	1	1	X	X	0
		1	1	X	X	X	X	X	X
1	0	0	0	1	0	X	0	0	X
		0	1	1	1	X	0	1	X
		1	0	1	1	X	0	1	X
		1	1	X	X	X	X	X	X
1	1	0	0	1	1	X	0	X	0
		0	1	1	1	X	0	X	0
		1	0	1	1	X	0	X	0
		1	1	X	X	X	X	X	X

Figure 8.16 Remapped next-state functions for the vending machine example.

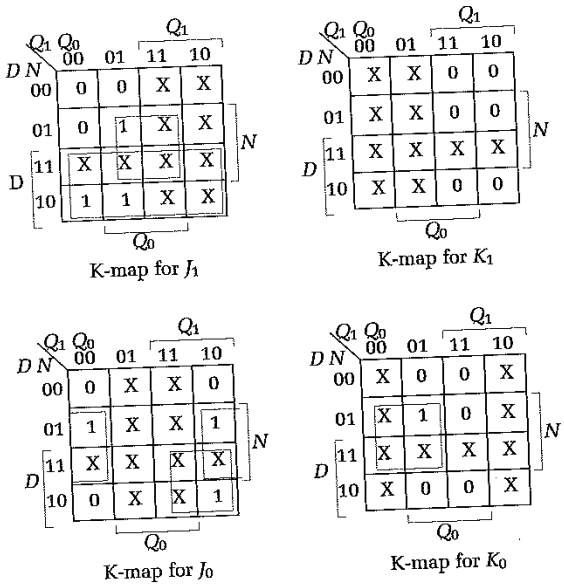


Figure 8.17 K-maps for *J-K* flip-flop implementation of vending machine.

Since more than one state diagram can lead to the same input/output behavior, it is important to find a description with as few states as possible. This usually reduces the implementation complexity of the finite state machine. For example, the state diagram of Figure 8.10 contains nine states and requires four flip-flops for its implementation. The minimized state

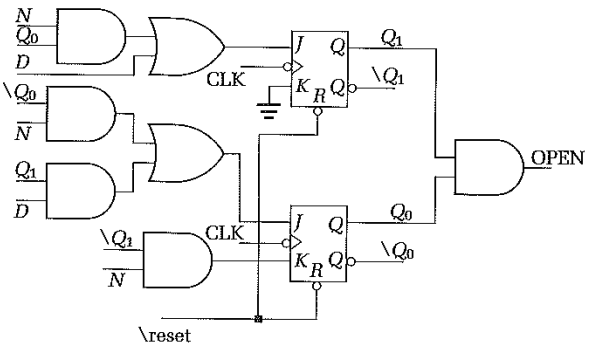


Figure 8.18 *J-K* flip-flop implementation for the vending machine example.

diagram of Figure 8.11 has four states and can be implemented with only two flip-flops.

Once we have obtained a minimum finite state description, the next step is to choose a good encoding of the states. The right choice can further reduce the logic for the next-state and output functions. In the example, we used only the most obvious state assignment.

The final step is to choose a flip-flop type for the state registers. In the example, the implementation based on *D* flip-flops was more straightforward. We did not need to remap the flip-flop inputs, but we used more gates than the *J-K* flip-flop implementation. This is usually the case.

Now we are ready to examine some alternatives to the state diagram for describing finite state machine behavior.

8.3 Alternative State Machine Representations

You have already seen how to describe finite state machines in terms of state diagrams and tables. However, it can be difficult to describe complex finite state machines in this way. Recently, hardware designers have shifted toward using alternative representations of FSM behavior that look more like software descriptions. In this section, we introduce algorithmic state machine (ASM) notation and hardware description languages (HDLs). ASMs are similar to program flowcharts, but they have a more rigorous concept of timing. HDLs look much like modern programming languages, but they explicitly support computations that can occur in parallel.

You may wonder what is wrong with state diagrams. The problem is that they do not adequately capture the notion of an *algorithm*—a well-defined sequence of steps that produce a desired sequence of actions based on input data. State diagrams are weak at capturing the structure behind complex sequencing. The representations discussed next do a better job of making this sequencing structure explicit.

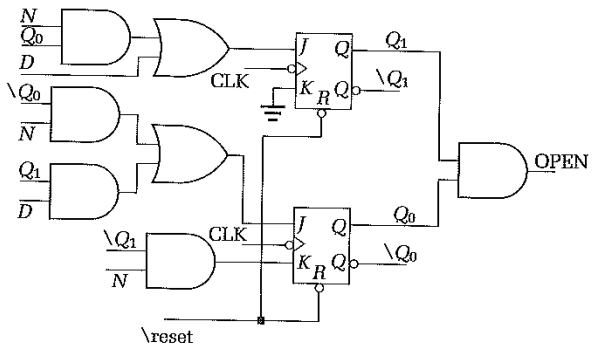


Figure 8.18 *J-K* flip-flop implementation for the vending machine example.

diagram of Figure 8.11 has four states and can be implemented with only two flip-flops.

Once we have obtained a minimum finite state description, the next step is to choose a good encoding of the states. The right choice can further reduce the logic for the next-state and output functions. In the example, we used only the most obvious state assignment.

The final step is to choose a flip-flop type for the state registers. In the example, the implementation based on *D* flip-flops was more straightforward. We did not need to remap the flip-flop inputs, but we used more gates than the *J-K* flip-flop implementation. This is usually the case.

Now we are ready to examine some alternatives to the state diagram for describing finite state machine behavior.

8.3 Alternative State Machine Representations

You have already seen how to describe finite state machines in terms of state diagrams and tables. However, it can be difficult to describe complex finite state machines in this way. Recently, hardware designers have shifted toward using alternative representations of FSM behavior that look more like software descriptions. In this section, we introduce algorithmic state machine (ASM) notation and hardware description languages (HDLs). ASMs are similar to program flowcharts, but they have a more rigorous concept of timing. HDLs look much like modern programming languages, but they explicitly support computations that can occur in parallel.

You may wonder what is wrong with state diagrams. The problem is that they do not adequately capture the notion of an *algorithm*—a well-defined sequence of steps that produce a desired sequence of actions based on input data. State diagrams are weak at capturing the structure behind complex sequencing. The representations discussed next do a better job of making this sequencing structure explicit.

8.3.1 Algorithmic State Machine Notation

The ASM notation consists of three primitive elements: the state box, the decision box, and the output box, as shown in Figure 8.19. Each major unit, called an ASM block, consists of a state box and, optionally, a network of condition and output boxes. A state machine is in exactly one state or ASM block during the stable portion of the state time.

State Boxes There is one state box per ASM block, reached from other ASM blocks through a single state entry path. In addition, for each combination of inputs there is a single unambiguous exit path from the ASM block. The state box is identified by a symbolic state name—in a circle—and a binary-encoded state code, and it contains an output signal list.

The output list describes the signals that are asserted whenever the state is entered. Because signals may be expressed in either positive or negative logic, it is customary to place an "L." or "H." prefix before the signal name, indicating whether it is asserted low or high. You can also specify whether the signal is asserted immediately (I) or is delayed (no special prefix) until the next clocking event. A signal not mentioned in the output list is left unasserted.

Condition Boxes The condition box tests an input to determine an exit path from the current ASM block to the block to be entered next. The order in which condition boxes are cascaded has no effect on the determination of the next ASM block. Figure 8.20(a) and (b) show functionally equivalent ASM blocks: state B is to be entered next if I_0 and I_1 are both 1; otherwise state C is next.

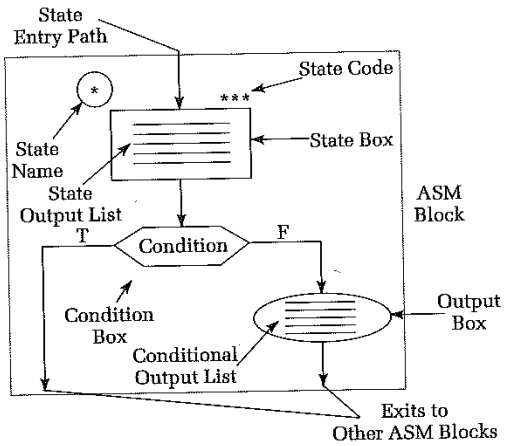


Figure 8.19 Elements of the ASM notation.

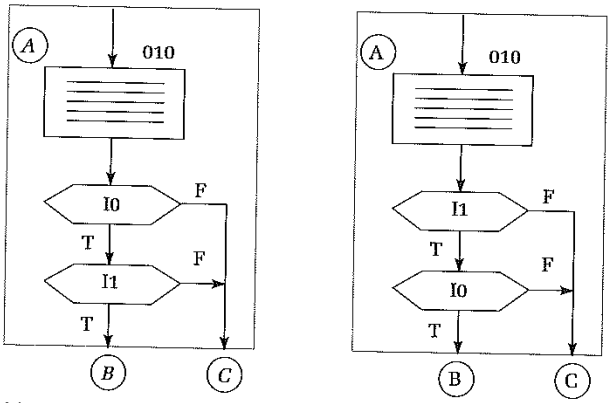


Figure 8.20 Functionally equivalent ASM blocks.

Output Boxes Any output boxes on the path from the state box to an exit contain signals that should be asserted along with the signals mentioned in the state box. The state machine advances from one state to the next in discrete rather than continuous steps. In this sense, ASM charts have different timing semantics than program flowcharts.

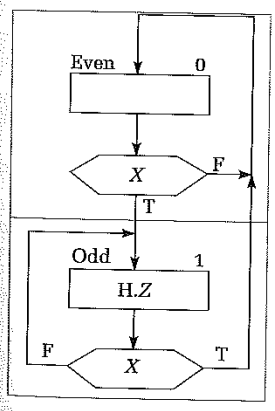


Figure 8.21 Parity checker ASM chart.

Example The Parity Checker As an example, we give the parity checker's ASM chart in Figure 8.21. It consists of two states, Even and Odd, encoded as 0 and 1, respectively. The input is the single bit X; the output is the single bit Z, asserted high when the finite state machine is in the Odd state.

We can derive the state transition table from the ASM chart. We simply list all the possible transition paths from one state to another and the input combinations that cause the transition to take place. For example, in state Even, when the input is 1, we go to state Odd. Otherwise we stay in state Even. For state Odd, if the input is 1, we advance to Even. Otherwise we remain in state Odd. The output Z is asserted only in state Odd. The transition table becomes:

Input X	Present State	Next State	Output Z
F	Even	Even	Not asserted
T	Even	Odd	Not asserted
F	Odd	Odd	Asserted
T	Odd	Even	Asserted

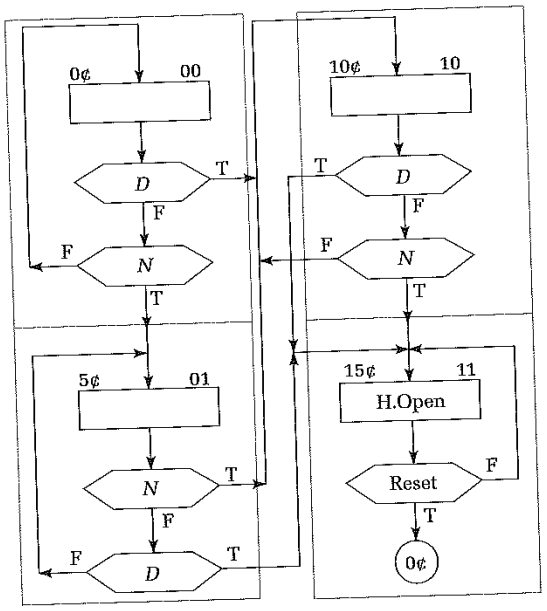


Figure 8.22 Vending machine ASM chart.

Example Vending Machine Controller We show the ASM chart for the vending machine in Figure 8.22. To extract the state transition table, we simply examine all exit paths from each state. For example, in the state 0¢, we advance to state 10¢ when input D is asserted. If N is asserted, we go to state 5¢. Otherwise, we stay in state 0¢. The rest of the table can be determined by looking at the remaining states in turn.

8.3.2 Hardware Description Languages: VHDL

Hardware description languages provide another way to specify finite state machine behavior. Such descriptions bear some resemblance to a program written in a modern structured programming language. But again, the concept of timing is radically different from that in a program written in a sequential programming language. Unlike state diagrams or ASM charts, specifications in a hardware description language can actually be simulated. They are executable descriptions that can be used to verify that the digital system they describe behaves as expected.

VHDL (VHSIC hardware description language) is an industry standard. Although its basic concepts are relatively straightforward, its detailed

syntax is beyond the scope of this text. However, we can illustrate its capabilities for describing finite state machines by examining a description of the parity checker written in VHDL:

```
ENTITY parity_checker IS
   PORT (
      x, clk: IN BIT;
      z: OUT BIT);
END parity_checker;

ARCHITECTURE behavioral OF parity_checker IS
   BEGIN
      main: BLOCK (clk = '1' and not clk'STABLE)

      TYPE state IS (Even, Odd);
      SIGNAL state_register: state := Even;

      BEGIN state_even:
         BLOCK ((state_register = Even) AND GUARD)
            BEGIN
            state_register <= Odd WHEN x = '1'
            ELSE Even
         END BLOCK state_even;

      BEGIN state_odd:
         BLOCK ((state_register = Odd) AND GUARD)
            BEGIN
            state_register <= Even WHEN x = '1'
            ELSE Odd;
         END BLOCK state_odd;

      z <= '0' WHEN state_register = Even ELSE
           '1' WHEN state_register = Odd;
   END BLOCK main;
END behavioral;
```

Every VHDL description has two components: an *interface description* and an *architectural body*. The former defines the input and output connections or "ports" to the hardware entity being designed; the latter describes the entity's behavior.

The architecture block defines the behavior of the finite state machine. The values the *state register* can take on are defined by the type *state*, consisting of the symbols *Even* and *Odd*. We write VHDL statements that assign new values to the state register and the output Z, depending on the current value of input X, whenever we detect a rising clock edge.

Checking for events like a clock transition is handled through the VHDL concept of the *guard*, an expression that enables certain statements in the description when it evaluates to true. For example, the expression

```
clk = '1' and not clk'stable
```

is a guard that evaluates to true whenever the clock signal has recently undergone a 0-to-1 transition. The main block is enabled for evaluation when this particular guard becomes true.

The description contains two subblocks, *state_even* and *state_odd*, that are enabled whenever the main guard is true and the machine is in the indicated state. Within each subblock, the state register receives a new assignment depending on the value of the input. Outside the subblocks, the output becomes 0 when the machine enters state Even and 1 when it enters state Odd.

8.3.3 ABEL Hardware Description Language

ABEL is a hardware description language closely tied to the specification of programmable logic components. It is also an industry standard and enjoys widespread use. The language is suitable for describing either combinational or sequential logic and supports hardware specification in terms of Boolean equations, truth tables, or state diagram descriptions. Although the detailed syntax and semantics of the language are beyond our scope, we can highlight its features with the parity checker finite state machine.

Let's look at the ABEL description of the parity checker:

```
module parity
title 'odd parity checker state machine
    Joe Engineer, Itty Bity Machines, Inc.'
u1   device   'p22v10';

"Input Pins
   clk, X, RESET pin 1, 2, 3;

"Output Pins
   Q, Z       pin 21, 22;
   Q, Z       istype 'pos,reg';

"State registers
SREG   =  [Q, Z];
EVEN   =  [0, 0]; " even number of 0's
ODD    =  [1, 1]; " odd number of 0's
```

```
equations
   [Q.ar, Z.ar] = RESET;"Reset to state S0

state_diagram SREG
state EVEN:
   if X then ODD
   else EVEN;
state ODD:
   if X then EVEN
   else ODD;

test_vectors ([clk, RESET, X] -> [SREG])
   [0,1,.X.] -> [EVEN];
   [.C.,0,1] -> [ODD];
   [.C.,0,1] -> [EVEN];
   [.C.,0,1] -> [ODD];
   [.C.,0,0] -> [ODD];
   [.C.,0,1] -> [EVEN];
   [.C.,0,1] -> [ODD];
   [.C.,0,0] -> [ODD];
   [.C.,0,0] -> [ODD];
   [.C.,0,0] -> [ODD];
end parity;
```

An ABEL description consists of several sections: *module*, *title*, *descriptions*, *equations*, *truth tables*, *state diagrams*, and *test vectors*, some of which are optional. Every ABEL description begins with a `module` statement and an optional `title` statement. These name the module and provide some basic documentation about its function.

These are followed by the `description` section. The elements of this section are the kind of device being programmed, the specification of inputs and outputs, and the declaration of which signals constitute the state of the finite state machine.

We must first describe the device selected for the implementation. It is a P22V10 PAL, with 12 inputs, 10 outputs, and embedded flip-flops associated with the outputs. For identification within the schematic, we call the device *u1*.

Next come the pin descriptions. The finite state machine's inputs are the clock *clk*, data *X*, and the *RESET* signal. The outputs are the state *Q* and the output *Z*. These are assigned to specific pins on the PAL. For example, pin 1 is connected to the clock inputs of the internal flip-flops.

Many of the attributes of a PAL are selectable, so the description may need to make explicit choices. The next line of the description tells ABEL that *Q* and *Z* are POSitive logic outputs of the PAL's internal flip-flops (REG) associated with particular output pins. The P22V10 PAL

also supports negative logic outputs as well as outputs that bypass the internal flip-flops.

The state of the finite state machine is represented by the outputs Q and Z. EVEN is defined as the state where Q and Z are 0. ODD is defined as the state where Q and Z are 1.

The `equation` section defines outputs in terms of Boolean equations of the inputs. In this case, the asynchronous reset (`.ar`) inputs of the Q and Z flip-flops are driven high when the RESET signal is asserted.

The `state_diagram` section describes the transitions among states using a programming language–like syntax. If we are in EVEN and the input X is asserted, we change to ODD. Otherwise we stay in EVEN. Similarly, if we are in ODD and X is asserted, we return to EVEN. Otherwise we stay in state ODD. ABEL supports a variety of control constructs, including such things as case statements.

The final section in this example is for `test_vectors`. This is a tabular listing of the expected input/output behavior of the finite state machine. The first entry describes what happens when RESET is asserted: independent of the current value of X, the machine is forced to EVEN. The rest of the entries describe the state sequence for the input string 111011000. The ABEL system simulates the description to ensure that the behavior matches the specified behavior of the test vectors.

The major weakness of an ABEL description is that it forces the designer to understand many low-level details about the target PAL. Nevertheless, the state diagram description is an intuitively simple way to describe the behavior of a state machine.

8.4 Moore and Mealy Machine Design Procedure

There are two basic ways to organize a clocked sequential network:

- *Moore machine*: The outputs depend only on the present state. See the block diagram in Figure 8.23. A combinational logic block maps the inputs and the current state into the necessary flip-flop inputs to store the appropriate next state. The outputs are computed by a combinational logic block whose only inputs are the flip-flops' state outputs. The outputs change *synchronously* with the state transition and the clock edge. The finite state machines you have seen so far are all Moore machines.
- *Mealy machine*: The outputs depend on the present state and the present value of the inputs. See Figure 8.24. The outputs can change immediately after a change at the inputs, independent of the clock. A Mealy machine constructed in this fashion has *asynchronous* outputs.

Moore outputs are synchronous with the clock, only changing with state transitions. Mealy outputs are asynchronous and can change in response to any changes in the inputs, independent of the clock. This

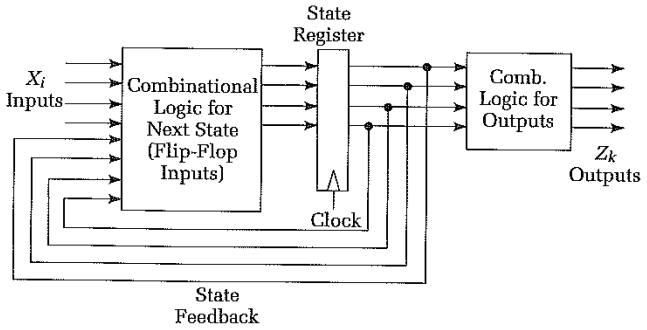


Figure 8.23 Moore machine block diagram.

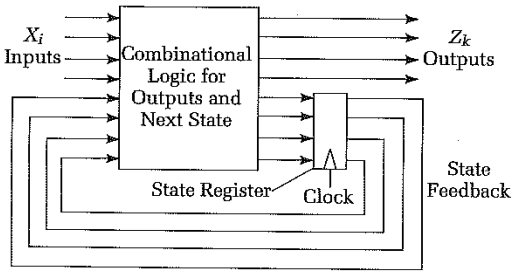


Figure 8.24 Mealy machine block diagram.

gives Moore machines an advantage in terms of disciplined timing methodology. However, there is a synchronous variation of the Mealy machine, which we describe later.

8.4.1 State Diagram and ASM Chart Representations

An ASM chart intended for Moore implementation would have no conditional output boxes. The necessary outputs are simply listed in the state box. Conditional output boxes in the ASM chart usually imply a Mealy implementation.

Figure 8.25 shows the notations for Mealy and Moore state diagrams, using the vending machine example. For Moore machines, the outputs are associated with the state in which they are asserted. Arcs are labeled with the input conditions that cause the transition from the state at the tail of the arc to the state at its head. Combinational logic functions are perfectly acceptable as arc labels.

In Mealy machines, the outputs are associated with the transition arcs rather than the state bubble. A slash separates the inputs from the

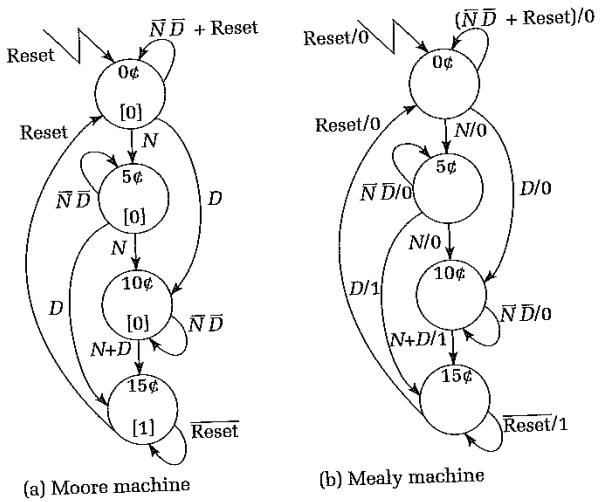


Figure 8.25 Moore and Mealy machine state diagrams for the vending machine FSM.

outputs. For example, if we are in state 10¢ and either N or D is asserted, Open will be asserted. Any glitch on N or D could cause the gum to be delivered by mistake.

The state diagrams in this figure are labeled more completely than our previous examples. For example, we make explicit the transitions that cause the machine to stay in the same state. We usually eliminate such transitions to simplify the state diagram. We also associate explicit output values with each transition in the Mealy state diagram and each state in the Moore state diagram. A common simplification places the output on the transition or in the state only when it is asserted. You should clarify your assumptions whenever you draw state diagrams.

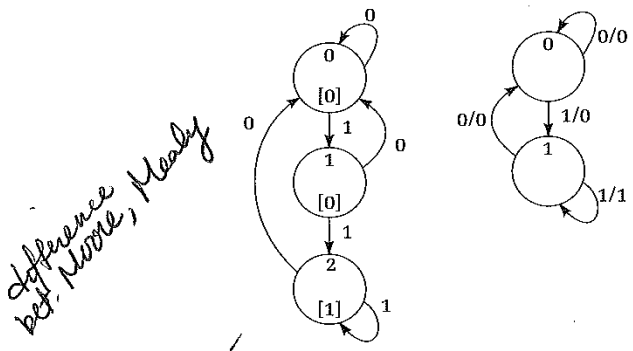
8.4.2 Comparison of the Two Machine Types

Because it can associate outputs with transitions, a Mealy machine can often generate the same output sequence in fewer states than a Moore machine.

Consider a finite state machine that asserts its single output whenever its input string has at least two 1's in sequence. The minimum Moore and Mealy state diagrams are shown in Figure 8.26. The equivalent ASM charts are in Figure 8.27.

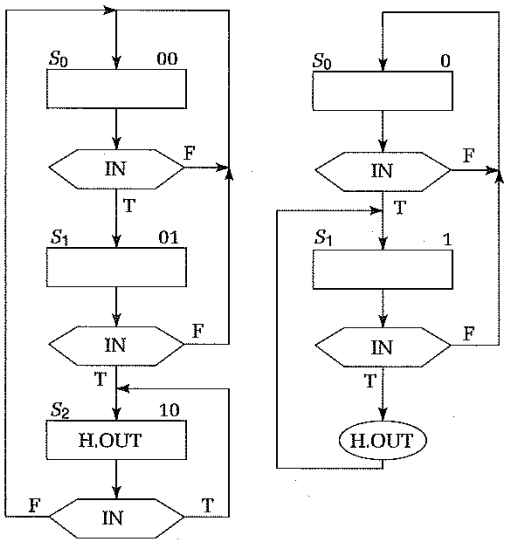
To represent the 1's sequence, the Moore machine requires two states to distinguish between the first and subsequent 1's. The first state has output 0, while the second has output 1. The Mealy machine accomplishes this

with a single state reached by two different transitions. For the first 1, the transition has output 0. For the second and subsequent 1's, the transition has output 1. Despite the Mealy machine's timing complexities, designers like its reduced state count.



(a) Moore machine (b) Mealy machine

Figure 8.26 Two state diagrams with the same I/O behavior but different number of states.



(a) Moore machine (b) Mealy machine

Figure 8.27 ASM equivalents of Figure 8.26.

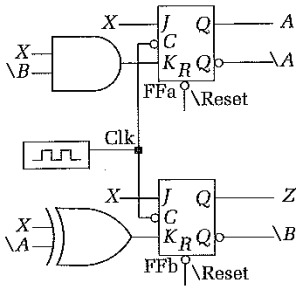


Figure 8.28 Mystery Moore finite state machine.

8.4.3 Examples of Moore and Mealy Machines

Example Moore Machine Description To better understand the timing behavior of Moore and Mealy machines, let's begin by *reverse engineering* some finite state machines. We will work backward from a circuit-level implementation of the finite state machine to derive an ASM chart or state diagram that describes the machine's behavior.

Figure 8.28 shows schematically a finite state machine with single data input X and output Z. The FSM is a Moore machine because the output is a combinational logic function (in this case a trivial one) of the state alone. The state register is implemented by two master/slave J-K flip-flops, named A and B, respectively. The machine can be in any one of up to four valid states. The output Z and the state bit B are the same.

Signal Trace Method There are two systematic approaches to determining the state transitions: *exhaustive signal tracing* and *extraction of the next state/output functions*. We examine the former here and the latter in the next subsection.

Signal tracing uses a collection of input sequences to exercise the various state transitions of the machine. To see how it works, let's start by generating a sample input sequence.

It is reasonable to assume that the FSM is initially reset and that it has been placed in state $A = 0$, $B = 0$. Figure 8.29 contains the timing waveform you would see after presenting the input sequence 1 0 1 0 1 0 to the machine. Because the FSM is implemented with master/slave flip-flops, the state time begins with the falling edge of the clock. Input X must be stable throughout the high time of the clock to guard against ones catching problems.

The sequence of events in Figure 8.29 is as follows. The asserted reset signal places the FSM in state 00. After the falling edge, input X goes high just after time step 20. At the next rising edge, the input is sampled and the next state is determined, but this is not presented to

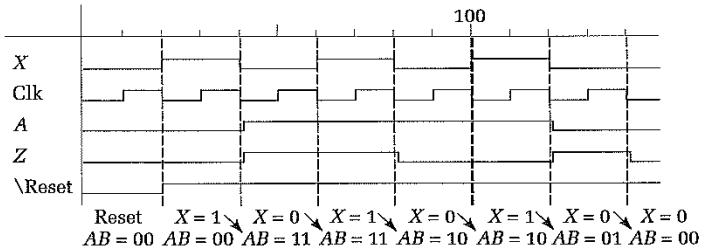


Figure 8.29 A timing trace of the mystery Moore machine.

A	B	X	A^+	B^+	Z
0	0	0	?	?	0
		1	1	1	0
0	1	0	0	0	1
		1	?	?	1
1	0	0	1	0	0
		1	0	1	0
1	1	0	1	1	1
		1	1	0	1

Figure 8.30 Partial state transition table derived from the signal trace.

the outputs until the falling edge at time step 40. After a short propagation delay, the state becomes 11. We express the transition as "a 1 input in state 00 leads to state 11."

During the next state time, X is 0, and the FSM stays in state 11 as seen at time step 60. X now changes to 1, and at the next falling edge the state changes to 10.

The input next changes to 0, causing the state machine to remain in state 10 at time step 100. A transition to 1 causes it to change to state 01 after time step 120. The final transition to 0 leaves the machine in state 00.

Figure 8.30 contains the partial transition table we deduce from this input sequence. We would have to generate additional input sequences to fill in the missing transitions. For example, an input sequence from the reset state starting with a 0 would fill in the missing transition from state 00. The sequence 1 1 1 1, tracing from state 00 to 11 to 10 to 01, would catch the remaining transition.

Next State/Output Function Analysis Signal tracing is acceptable for a small FSM, but it becomes intractable for more complex finite state machines. With a single input and 2 bits of state, the example FSM has eight different transitions, two from each of four states. And the number of combinations doubles for each additional input bit and doubles again for each state bit.

Our alternative method derives the next-state functions directly from the combinational logic equations at the flip-flop inputs and the output function from the flip-flop outputs. For the mystery machine, these are

$$J_a = X \qquad K_a = X \bullet \overline{B} \qquad Z = B$$
$$J_b = X \qquad K_b = X \oplus \overline{A}$$

We can now express the flip-flop outputs, A^+ and B^+, in terms of the excitation equations for the J-K flip-flop. We simply substitute the logic functions at the inputs into the excitation equations:

$$A^+ = J_a \bullet \overline{A} + \overline{K}_a \bullet A = X \bullet \overline{A} + (\overline{X} + B) \bullet A$$
$$B^+ = J_b \bullet \overline{B} + \overline{K}_b \bullet B = X \bullet \overline{B} + (X \bullet \overline{A} + \overline{X} \bullet A) \bullet B$$

The next-state functions, A^+ and B^+, are now expressed in terms of the current state, A and B, and the input X. We show the K-maps that correspond to these functions in Figure 8.31.

The missing state transitions are now obvious. In state 00 with input 0, the next state is $A^+ = 0$ and $B^+ = 0$. In state 01 with input 1, the next state is $A^+ = 1$, $B^+ = 1$. With its behavior no longer a mystery, we show the ASM chart for this finite state machine in Figure 8.32. In the figure, we assume the following symbolic state assignment: $S_0 = 00$, $S_1 = 01$, $S_2 = 10$, $S_3 = 11$.

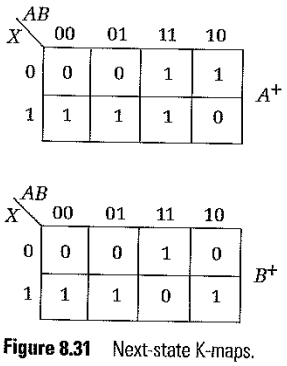


Figure 8.31 Next-state K-maps.

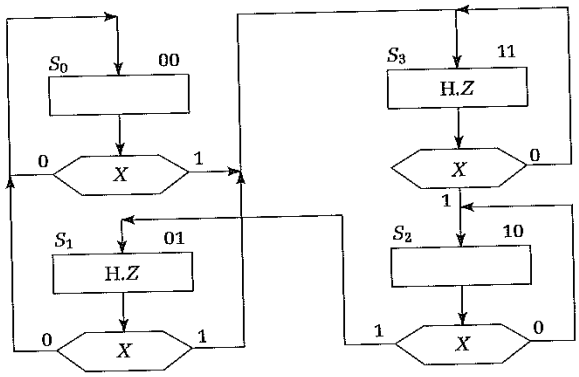


Figure 8.32 ASM chart for the mystery Moore machine.

Example **Mealy Machine Description** Continuing with our reverse engineering exercise, consider the circuit of Figure 8.33. Once again, the FSM has one input, *X*, and one output, *Z*. This time the output is a function of the current state, denoted by *A* and *B*, *and* the input *X*. The state register is implemented by one *D* flip-flop and one master/slave *J-K* flip-flop.

Before examining a signal trace, we must understand the conditions under which the Mealy machine's inputs are sampled and the outputs are valid. The next state is computed from the current state and the inputs, so exactly when are the inputs sampled? The answer depends on the kinds of flip-flops used to implement the state register. In the example, our use of a master/slave flip-flop dictates that the inputs must

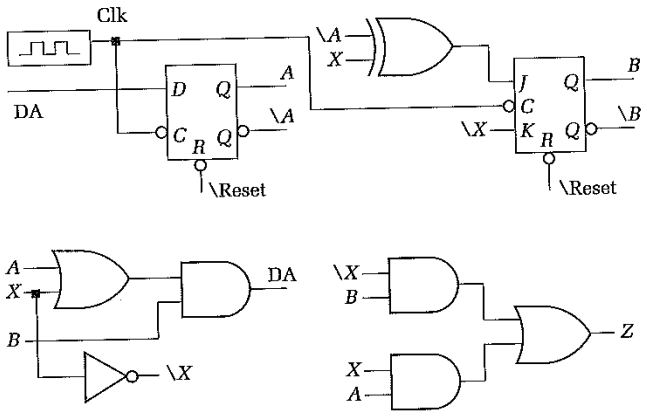


Figure 8.33 Circuit schematic of the mystery Mealy machine.

be stable during the high time of the clock (to avoid ones catching) and must be valid a setup time before the falling edge.

Technically, the outputs are valid only at the end of the state time, determined by the falling edge of the clock. In other words, the output for the current state is valid just as the machine enters its next state! If we are using a master/slave flip-flop and if the inputs do not change during the high time of the clock, then the outputs may also be valid during the clock high time.

Negative edge-triggered systems require that the inputs be stable before the falling edge that delineates the state times. This means that the outputs cannot be determined until just before the falling edge. The output remains valid only as long as it takes to compute a new output in the new state.

Similarly, for positive edge-triggered systems the outputs are valid at the rising edge. Again, the output is considered valid just before the clock edge that causes the machine to enter its new state.

Figure 8.34 gives the timing waveform that corresponds to the input sequence 10101, after a reset to state 00. In state 00, reading a 1 keeps the machine in state 00 (time step 40).

Reading a 0 then advances the machine to state 01 (time step 60). The waveform for output Z has a glitch. The valid output is determined only at the end of the state time. In this case, the output is 0.

A 1 in state 01 leads to state 11 (time step 80). Again, the output in this state is the value of Z at the falling edge and thus is 1.

Reading a 0 in state 11 moves us to state 10 (time step 100), with the output continuing to be asserted despite the momentary glitch.

A 1 in state 10 leads us to state 01 (time step 120). The output goes low and will stay that way as long as the input X stays high.

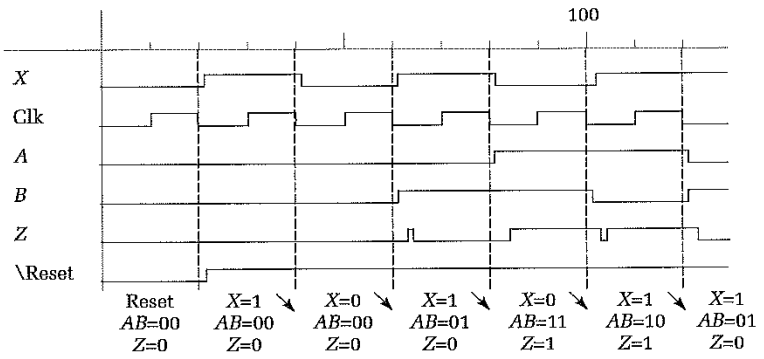


Figure 8.34 Signal trace of the mystery Mealy machine.

A	B	X	A^+	B^+	Z
0	0	0	0	1	0
		1	0	0	0
0	1	0	?	?	?
		1	1	1	0
1	0	0	?	?	?
		1	0	1	1
1	1	0	1	0	1
		1	?	?	?

Figure 8.35 Partial state transition table derived from the signal trace.

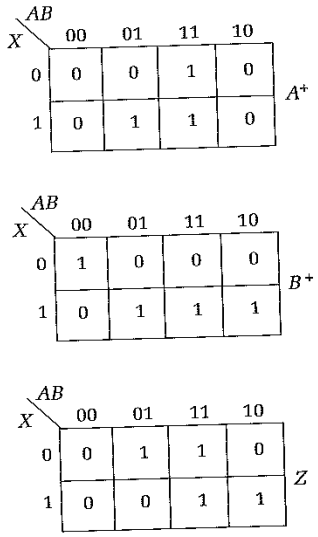


Figure 8.36 Next-state and output K-maps.

We show the partial state transition table in Figure 8.35. The input sequence produced only five of the eight state transitions. To complete the state diagram, we would have to generate additional sequences to traverse the missing transitions.

Alternatively, we can discover the complete set of transitions by analyzing the next-state and output functions directly, just as we did in the Moore machine:

$$A^+ = B\,(A + X) = A \bullet B + B \bullet X$$

$$B^+ = J_b \bullet \overline{B} + \overline{K}_b \bullet B = (\overline{A} \oplus X) \bullet \overline{B} + X \bullet B$$

$$= (\overline{A} \bullet \overline{X} + A \bullet X)\,\overline{B} + X \bullet B$$

$$= A \bullet \overline{B} \bullet X + \overline{A} \bullet \overline{B} \bullet \overline{X} + B \bullet X$$

$$Z = A \bullet X + B \bullet \overline{X}$$

Since A is a D flip-flop, the function for A^+ is exactly the combinational logic function at its input. B is a J-K flip-flop, so we determine the function for B^+ by substituting the logic functions at the J and K inputs into the J-K excitation function.

We give the next-state and output K-maps in Figure 8.36. The missing transitions are from state 01 to 00 on input 0, 10 to 00 on input 0, and 11 to 11 on input 1. The respective outputs are 1, 0, and 1. Assuming that S_0, S_1, S_2, and S_3 correspond to encoded states 00, 01, 10, 11, we show the ASM chart for the mystery machine in Figure 8.37.

States, Transitions, and Outputs in Mealy and Moore Machines Suppose that a given state machine has M inputs and N outputs and is being implemented using L flip-flops. You might ask a number of questions to bound the complexity of this state machine. For example, what are the minimum and maximum numbers of states that such a machine might have? With L flip-flops, the implementation has the power to represent 2^L states. But for a specific FSM as few as 1 and as many as 2^L of these might be valid states.

What are the minimum and maximum numbers of state transitions that can begin in a given state? Since there must be an exit transition for each possible input combination, the minimum and the maximum are the same: 2^M transitions.

A similar question involves the minimum and maximum numbers of state transitions that can end in a given state. Because we can have start-up states reachable only on reset, the minimum number of input transitions is 0. Since a single state could conceivably be the target of all the transitions of the finite state machine, the maximum number of input transitions is $2^M * 2^L$, the number of possible input combinations multiplied by the number of states.

A final question is the minimum and maximum numbers of patterns that can be observed on the machine's outputs. The minimum number of unique output patterns is 1, of course. Every state and every transition can be associated with the same pattern.

The maximum number depends on the kind of machine. For a Mealy machine, the maximum number of output patterns is the smaller of the number of transitions, $2^M * 2^L$, or the number of possible output patterns, 2^N. If the number of transitions exceeds the number of possible output patterns, then some must be repeated. In the Moore machine, the maximum is the smaller of the number of states, 2^L, and the number of possible output patterns, 2^N. If the number of states exceeds the number of output patterns, then some patterns will also need to be repeated.

As an example, consider a Moore machine with two inputs, one flip-flop, and three outputs. The state, transition, and output bounds are:

Minimum number of states: 1

Maximum number of states: 2

Minimum number of output transitions (per state): 4

Maximum number of output transitions (per state): 4

Minimum number of input transitions (per state): 0

Maximum number of input transitions (per state): 8

Minimum number of observed output patterns: 1

Maximum number of observed output patterns: 2

In this case, the output patterns are limited by the number of states.

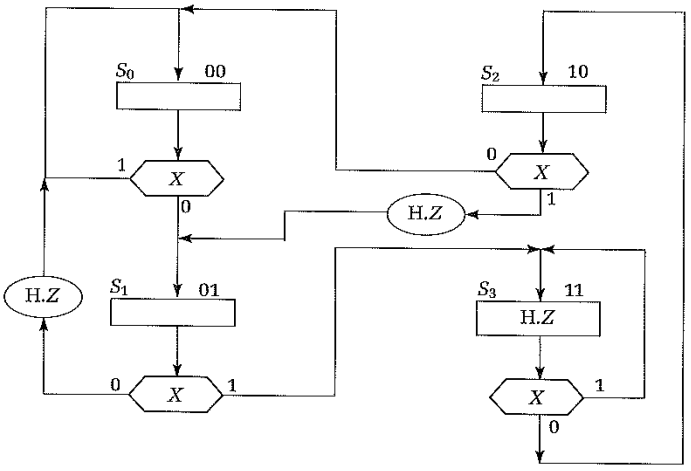


Figure 8.37 ASM chart for the mystery Mealy machine.

Synchronous Mealy Machines The glitches in the output in Figure 8.34 are inherent in the asynchronous nature of the Mealy machine. As you have already seen, glitches are undesirable in real hardware controllers. But because Mealy machines encode control in fewer states, saving on state register flip-flops, it is still desirable to use them.

This leads to alternative *synchronous* design styles for Mealy machines. Simply stated, the way to construct a synchronous Mealy machine is to break the direct connection between inputs and outputs by introducing storage elements.

One way to do this is to synchronize the Mealy machine outputs with output flip-flops. See Figure 8.38. The flip-flops are clocked with the same edge as the state register. This has the effect of converting the Mealy machine into a Moore machine, by making the outputs part of the state encoding! However, this machine does not have exactly the same input/output behavior as the original Mealy machine (can you figure out why?). We will have more to say about synchronous Mealy machines in Chapter 10.

Discussion In general, fully synchronous finite state machines are much easier to implement and debug than asynchronous machines. If you were using discrete TTL components, you would usually prefer the Moore machine organization, even though it may require more states. You should use edge-triggered flip-flops for the state registers.

Synchronous Mealy machines can be constructed in TTL logic, but the designer must be careful. The approach leads to more complex designs that may affect the input/output timing of the FSM. You should use asynchronous Mealy machines only after very careful analysis of the input/output timing behavior of the finite state machine.

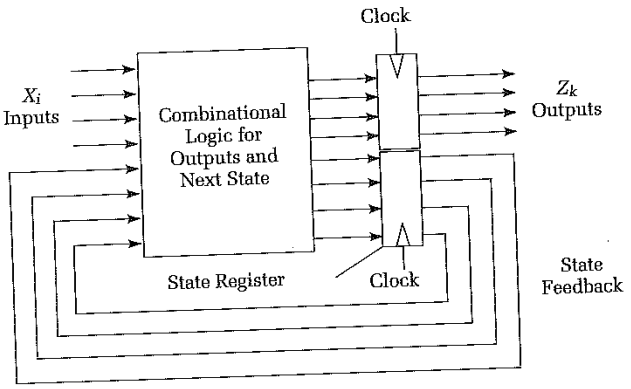


Figure 8.38 Synchronous Mealy machine block diagram.

9 Finite State Machine Optimization

Nature is as complex as it needs to be . . . and no more.
—A. Einstein

Introduction

We are now ready to complete the finite state machine design process introduced in Chapter 8. The steps of Chapter 8 yield an abstract description of the state machine. This may be a state diagram, an ASM chart, or a hardware description language specification. Deriving a symbolic state transition table from one of these is straightforward. In this chapter, we concentrate on state minimization, state assignment, and choice of flip-flops.

We show why finite state machine "optimization" (*improvement* might be a better word) is still important, even in today's era of very large scale integrated circuits. We present pencil-and-paper methods, as well as more formal techniques suitable for computer implementation, for reducing the number of states and for choosing a state encoding.

Then we examine the approaches for choosing the machine's flip-flops and how the choice affects the next-state and output combinational functions. The right choice of flip-flop leads to a smaller gate count and thus fewer components to implement the machine.

Finally, we develop techniques for partitioning complex finite state machines into simpler, smaller, communicating machines. You may be

forced to partition your state machine because it cannot fit into a given programmable logic component. This could arise, for example, because of limited logic resources, such as input/outputs, product terms, or flip-flops.

In this chapter, we emphasize the following techniques and concepts:

- *Procedures for optimizing a finite state machine.* You will learn the methods for state minimization and state assignment.
- *Application of modern computer-aided design tools for state assignment.* CAD tools make it possible for you to evaluate the implementation complexity of alternative state assignments very rapidly.
- *Partitioning methods.* You will learn the techniques for breaking finite state machines into smaller, communicating state machines that are well suited for implementation with programmable logic.

9.1 Motivation for Optimization

To review, the finite state machine design process consists of (1) understanding the problem, (2) obtaining a formal description (ultimately, a *symbolic state transition table*), (3) minimizing the number of states, (4) encoding the states, (5) choosing the flip-flops to implement the state registers, and finally (6) implementing the finite state machine's next-state and output functions. This chapter starts at step 3 and carries us through to the final implementation at step 6, using methods based on discrete logic gates. We discuss the use of programmable logic for finite state machine implementation in the next chapter.

9.1.1 Two State Diagrams, Same I/O Behavior

In the age of very large scale integrated circuits, why should we bother to minimize a finite state machine implementation? After all, as long as the input/output behavior of the machine is correct, it really doesn't matter how it is implemented. Or does it?

Figure 9.1 shows two different state diagrams for the odd parity checker of Section 8.2. They have identical output behavior for all input strings. You should try some inputs to convince yourself. We define *equivalence* of finite state machines as follows. Two machines are equivalent if their input/output behavior is identical for all possible input strings.

For a particular finite state machine, there are many equivalent forms. Rather than reusing states while deriving the state diagram, you could simply introduce a new state whenever you need one (to keep the number of states finite, you will need to reuse some of them, of course).

The two implementations of the state diagrams of Figure 9.1 are certainly not the same. The machine with more states requires more flip-flops and more complex next-state logic.

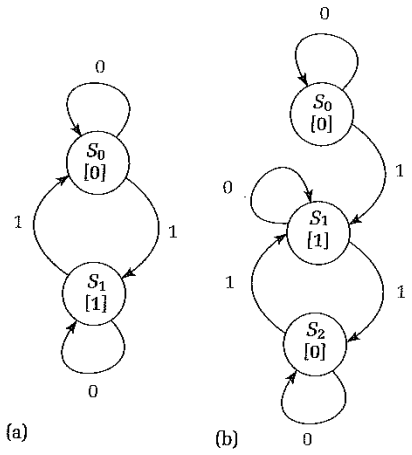


Figure 9.1 Two equivalent state diagrams for the odd parity checker.

9.1.2 Advantages of Minimum States

In general, you will find it is worthwhile to implement the finite state machine in as few states as possible. This usually reduces the number of logic gates and flip-flops you need for the machine's implementation.

Similarly, judicious mapping between symbolic and encoded states can reduce the implementation logic. For the parity checker, our implementation in Chapter 8 required no gates because we made a good state assignment that naturally matched the control input to the toggle flip-flop.

A state diagram with n states must be implemented with at least k flip-flops, where $2^{k-1} < n \leq 2^k$. By reducing the number of states to 2^{k-1} or less, you can save a flip-flop. For example, suppose you are given a finite state machine with five state flip-flops. This machine can represent up to 32 states. If you can reduce the number of states to 16 or less, you save a flip-flop.

Even when reducing the number of states is not enough to eliminate a flip-flop, it still has advantages. With fewer states, you introduce more don't-care conditions into the next-state and output functions, making their implementation simpler. Less logic usually means shorter critical timing paths and a higher clock rate for the system.

More important, today's programmable logic provides limited gate and flip-flop counts on a single programmable logic chip. A typical programmable logic part might have "2000 gate equivalents" (rarely approached in practice) yet provide only 64 flip-flops! An important goal of state reduction is to make the implementation "fit" in as few components as possible. The fewer components you use, the shorter the design time and the lower the manufacturing cost.

State reduction techniques also allow you to be sloppy in obtaining the initial finite state machine description. If you have introduced a few redundant states, you will find and eliminate them by using the state reduction techniques introduced next.

9.2 State Minimization/Reduction

State reduction identifies and combines states that have equivalent "behavior." Two states have *equivalent* behavior if, for all input combinations, their outputs are the same and they change to the same or equivalent next states.

For example, in Figure 9.1(b), states S_0 and S_2 are equivalent. Both states output a 0; both change to S_1 on a 1 and self-loop on a 0. Combining these into a single state leads to Figure 9.1(a). On all input strings, the output sequence of either state diagram is exactly the same.

Algorithms for state reduction begin with the symbolic state transition table. First, we group together states that have the same state outputs (Moore machine) or transition outputs (Mealy machine). These are potentially equivalent, since states cannot be equivalent if their outputs differ.

Next, we examine the transitions to see if they go to the same next state for every input combination. If they do, the states are equivalent and we can combine them into a renamed new state. We then change all transitions into the newly combined states. We repeat the process until no additional states can be combined.

In the following two subsections, we examine alternative algorithms for state reduction: *row matching* and *implication charts*. The former is a good pencil-and-paper method, but does not always obtain the best reduced state table. Implication charts are more complex to use by hand, but they are easy to implement via computer and do find the best solution.

We can always combine the two approaches. Row matching quickly reduces the number of states. The more complicated implication chart method, now working with fewer states, finds the equivalent states missed by row matching more rapidly.

9.2.1 Row-Matching Method

Let's begin our investigation of the row-matching method with a detailed example. We will see how to transform an initial state diagram for a simple sequence detector into a minimized, equivalent state diagram.

Four-Bit Sequence Detector: Specification and Initial State Diagram Let's consider a sequence-detecting finite state machine with the following specification. The machine has a single input X and output Z. The output is asserted after each 4-bit input sequence if it consists of one of the binary strings 0110 or 1010. The machine returns to the reset state after each 4-bit sequence.

We will assume a Mealy implementation. Some sample behavior of the finite state machine is

$X = 0010\ 0110\ 1100\ 1010\ 0011\ \ldots$

$Z = 0000\ 0001\ 0000\ 0001\ 0000\ \ldots$

The output is asserted only after the previous four serial inputs match one of the specified strings. Also, the input patterns do not overlap: the machine makes a decision to assert its output after each group of 4 bits.

Because this finite state machine recognizes finite length strings, we can place an upper bound on the number of states needed to recognize any particular binary string of length four. Figure 9.2 shows the state diagram. There are 16 unique paths through the state diagram, one for each possible 4-bit pattern. This adds up to 15 states and 30 transitions. We highlight the paths leading to recognition of the strings 0110 and 1010 in the figure. Only two of the transitions have a 1 output, representing the accepted strings.

Four-Bit Sequence Detector: State Table and Row-Matching Method We can combine many of the states in Figure 9.2 without changing the input/output behavior of the finite state machine. But how do we find these equivalent states in a systematic fashion?

First, we look at the state transition table, as shown in Figure 9.3. This table is in a slightly different format than we have seen so far. It contains one row per state, with multiple next-state and output columns based on the input combinations. It gives exactly the same information as a table with separate rows for each state and input combination.

The input sequence column is a documentation aid, describing the partial string as seen so far. When read from left to right, it describes the sequence of input bits that lead to the given state.

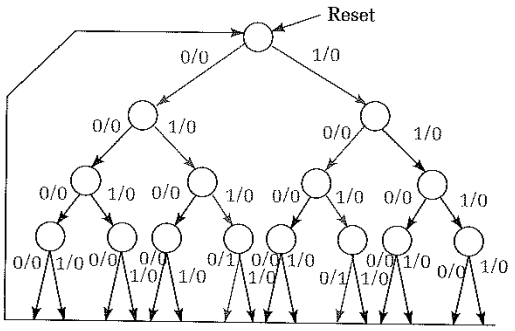


Figure 9.2 Original state diagram for 4-bit string recognizer.

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}	0	0
10	S_5	S_{11}	S_{12}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011	S_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
101	S_{12}	S_0	S_0	1	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

Figure 9.3 Initial state transition table for the 0110 or 1010 sequence detector.

Next we examine the rows of the state transition table to find any with identical next-state and output values (hence the term "row matching"). For this finite state machine, we can combine S_{10} and S_{12}. Let's call the new state S_{10}' and use it to rename all transitions to S_{10} or S_{12}. The revised state table is shown in Figure 9.4.

Four-Bit Sequence Detector: Row-Matching Iteration We continue matching rows until we can no longer combine any. In Figure 9.4, S_7, S_8, S_9, S_{11}, S_{13}, and S_{14} all have the same next states and outputs. We combine them into a renamed state S_7'. The table, with renamed transitions, is shown in Figure 9.5.

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}'	0	0
10	S_5	S_{11}	S_{10}'	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011 or 101	S_{10}'	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

Figure 9.4 Revised state transition table after S_{10} and S_{12} combined.

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7'	S_7''	0	0
01	S_4	S_7'	S_{10}'	0	0
10	S_5	S_7'	S_{10}'	0	0
11	S_6	S_7''	S_7''	0	0
not (011 or 101)	S_7''	S_0	S_0	0	0
011 or 101	S_{10}''	S_0	S_0	1	0

Figure 9.5 Revised state transition table after S_7, S_8, S_9, S_{11}, S_{13}, S_{14} combined.

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3'	S_4'	0	0
1	S_2	S_4'	S_3'	0	0
00 or 11	S_3'	S_7''	S_7''	0	0
01 or 10	S_4'	S_7''	S_{10}'	0	0
not (011 or 101)	S_7''	S_0	S_0	0	0
011 or 101	S_{10}'	S_0	S_0	1	0

Figure 9.6 Final reduced state transition table after S_3, S_6 and S_4, S_5 combined.

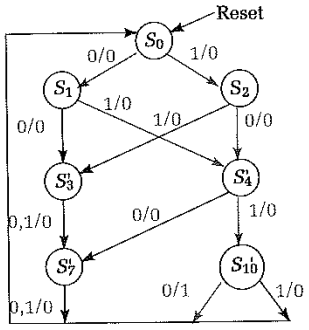


Figure 9.7 Reduced state diagram for 4-bit string recognizer.

Now states S_3 and S_6 can be combined, as can S_4 and S_5. We call the combined states S_3' and S_4', respectively. The final reduced state transition table is shown in Figure 9.6. In the process, we have reduced 15 states to just 7 states. This allows us encode the state in 3 bits rather than 4. The reduced state diagram is given in Figure 9.7.

Limitations of the Row-Matching Method Unfortunately, row matching does not always yield the most reduced state table. We can prove this with a simple counterexample. Figure 9.8 shows the state table for the three-state odd parity checker of Figure 9.1. Although states S_0 and S_2 have the same output, they do not have the same next state. Thus, they cannot be combined by simple row matching. The problem is the self-loop transitions on input 0. If we combined these two states, the self-loop would be maintained, but this is not found by row matching. We need another, more rigorous method for state reduction.

Present State	Next State X=0	Next State X=1	Output
S_0	S_0	S_1	0
S_1	S_1	S_2	1
S_2	S_2	S_1	0

Figure 9.8 State table for three-state odd parity checker.

9.2.2 Implication Chart Method

The implication chart method is a more systematic approach to finding the states that can be combined into a single reduced state. As you might suspect, the method is more complex and is better suited for machine implementation than hand use.

Three-Bit Sequence Detector: Specification and Initial State Table We illustrate its use with another example. Your goal is to design a binary sequence detector that will output a 1 whenever the machine has observed the serial sequence 010 or 110 at the inputs. We call this machine a 3-bit sequence detector. Figure 9.9 shows its initial state table.

Data Structure: The Implication Chart The method operates on a data structure that enumerates all possible combinations of states taken two at a time, called an *implication chart*. Figure 9.10(a) shows the chart with an entry for every pair of states. This form of the chart is more complicated than it needs to be. For example, the diagonal entries are not needed: it does not reduce states to combine a state with itself! And

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_0	S_0	0	0
01	S_4	S_0	S_0	1	0
10	S_5	S_0	S_0	0	0
11	S_6	S_0	S_0	1	0

Figure 9.9 Initial state transition table for the 3-bit sequence detector.

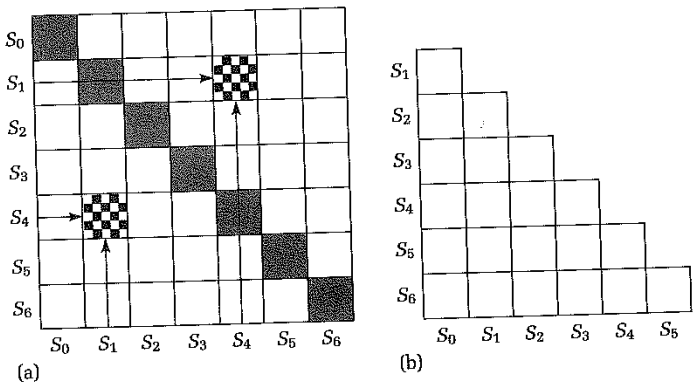


Figure 9.10 Matrix for state combinations and the corresponding implication chart.

the upper and lower triangles of entries are symmetric. The chart entry for S_i and S_j contains the same information as that for S_j and S_i. Thus, we work with the reduced structure of Figure 9.10(b).

We fill in the implication chart as follows. Let X_{ij} be the entry whose row is labeled by state S_i and whose column is labeled by state S_j. If we were able to combine states S_i and S_j, it would imply that their next-state transitions for each input combination must also be equivalent. The chart entry contains the next-state combinations that must be equivalent for the row and column states to be equivalent. If S_i and S_j have different outputs or next-state behavior, an X is placed in the entry. This indicates that the two states can never be equivalent.

Three-Bit Sequence Detector: Initial Implication Chart The implication chart for the example state table is shown in Figure 9.11. S_0, S_1, S_2, S_3, and S_5 have the same outputs and are candidates for being combined. Similarly, states S_4 and S_6 might also be combined. Any combination of states across the two groups, such as S_1 and S_4, is labeled by an X in the chart. Since their outputs are different, they can never be equivalent.

To fill in the chart entry for (row) S_1 and (column) S_0, we look at the next-state transitions. S_0 goes to S_1 on 0 and S_2 on 1, while S_1 goes to S_3 and S_4, respectively. We fill the chart in with S_1–S_3, the transitions on 0, and S_2–S_4, the transitions on 1. We call these groupings *implied state pairs*. The entry means that S_0 and S_1 cannot be equivalent unless S_1 is equivalent to S_3 and S_2 is equivalent to S_4. The rest of the entries are filled in similarly.

At this point, the chart already contains enough information to eliminate many impossible equivalent pairs. For example, we already know that S_2 and S_4 cannot be equivalent: they have different output behavior. Thus there is no way that S_0 can be equivalent to S_1.

Finding these cases is straightforward. We visit the entries in sequence. For example, start with the top square in the first column and advance from top to bottom and left to right. If square S_i,S_j contains the implied state pair S_m-S_n and square S_m,S_n contains an X, then mark S_i,S_j with an X as well.

Sequence Detector Example: First Marking Pass Figure 9.12 contains the results of this first marking pass. Entry S_2,S_0 is marked with an X because the chart entry for the implied state pair S_2-S_6 is already marked with an X. Entry S_3,S_0 is also marked, because entry S_1,S_0 (as well as S_2,S_0) has just been marked. The same is true for S_5,S_0. By the end of the pass, the only entries not marked are S_2,S_1; S_5,S_3; and S_6,S_4.

Sequence Detector Example: Second Marking Pass We now make a second pass through the chart to see if we can add any new markings. Entry S_2,S_1 remains unmarked. Nothing in the chart refutes that S_3 and S_5 are equivalent. The same is true of S_4 and S_6.

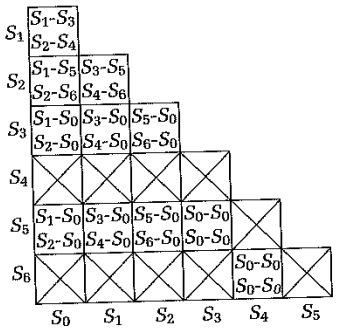


Figure 9.11 Initial implication chart for the 3-bit sequence detector.

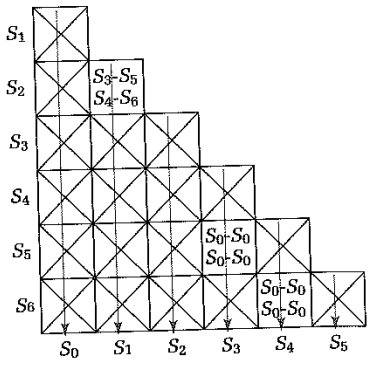


Figure 9.12 Results of first marking pass.

Continuing, S_3,S_5 and S_4,S_6 are now obviously equivalent. They have identical outputs and transfer to the same next state (S_0) for all input combinations.

Since no new markings have been added, the algorithm stops. The unmarked entries represent equivalences between the row and column indices: S_1 is equivalent to S_2, S_3 to S_5, and S_4 to S_6. The final reduced state table is shown in Figure 9.13.

Multi-Input Example: State Diagram and Transition Table We can generalize the procedure for finite state machines with more than one input. The only difference is that there are more implied state pairs: one for each input combination.

Let's consider the state diagram for a two-input Moore machine shown in Figure 9.14. Each state has four next-state transitions, one for each possible input condition. The derived state transition table is given in Figure 9.15.

Multi-Input Example: Implication Chart Processing Figure 9.16 shows the implication chart derived from the state transition table. Let's see how some of the entries are filled in. Since S_1 and S_0 have different state outputs, we place X in entry S_1,S_0. For the S_2,S_0 entry, we list the implied state pairs under the input conditions 00, 01, 10, 11. Because S_0

		Next State		Output	
Input Sequence	Present State	X=0	X=1	X=0	X=1
Reset	S_0	S_1'	S_1'	0	0
0 or 1	S_1'	S_3'	S_4'	0	0
00 or 10	S_3'	S_0	S_0	0	0
01 or 11	S_4'	S_0	S_0	1	0

Figure 9.13 Final reduced state transition table for the 3-bit sequence detector.

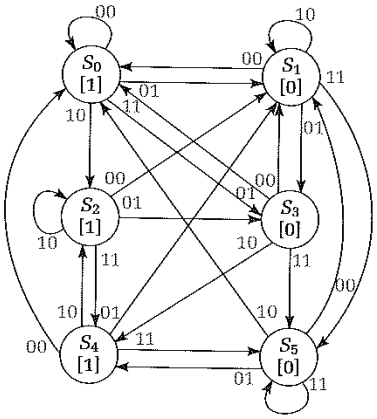


Figure 9.14 Multiple-input state diagram.

Present State	Next State 00	01	10	11	Output
S_0	S_0	S_1	S_2	S_3	1
S_1	S_0	S_3	S_1	S_5	0
S_2	S_1	S_3	S_2	S_4	1
S_3	S_1	S_0	S_4	S_5	0
S_4	S_0	S_1	S_2	S_5	1
S_5	S_1	S_4	S_0	S_5	0

Figure 9.15 Multiple-input state transition table.

stays in S_0 on input 00, while S_2 goes to S_1 on 00, we add the implied state pair S_0-S_1 to the entry. On input 01, S_0 goes to S_1, S_2 goes to S_3, and we add S_1-S_3 to the entry. Similarly, we add the pairs S_2-S_2 on 10 and S_3-S_4 on 11 to the entry and fill in the rest of the entries.

Now we begin the marking pass. Working down the columns, we cross out entry S_2-S_0 because S_0,S_1 is already crossed out. The same thing happens to the entries S_3,S_1; S_5,S_1; and S_4,S_2. This leaves S_4,S_0 and S_5,S_3 unmarked (these are highlighted in the figure). Their being unmarked implies that S_4 is equivalent to S_0 (renamed S_0') and S_3 is equivalent to S_5 (S_3'). The reduced state table is given in Figure 9.17.

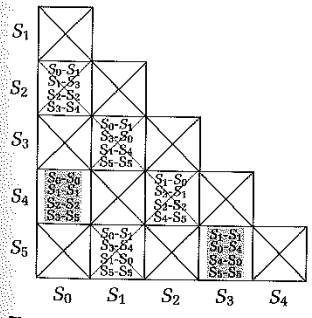


Figure 9.16 Multiple-input implication chart.

Present State	Next State 00	01	10	11	Output
S_0'	S_0'	S_1	S_2	S_3'	1
S_1	S_0'	S_3'	S_1	S_3'	0
S_2	S_1	S_3'	S_2	S_0'	1
S_3'	S_1	S_0'	S_0'	S_3'	0

Figure 9.17 Multiple-input reduced state table.

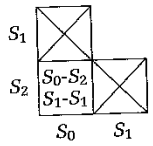


Figure 9.18 Implication chart for three-state odd parity checker.

Example State Reduction of Parity Checker Finite State Machine The row-matching method could not combine states S_0 and S_2 in the three-state parity checker of Figure 9.1. Can the implication chart method do the job?

The implication chart for the state transition table of Figure 9.8 is given in Figure 9.18. S_1,S_0 and S_2,S_1 are marked immediately because their outputs differ. The remaining square is left unmarked, implying that S_0 and S_2 are equivalent. This is the correct reduced state transition table.

Implication Chart Summary The algorithm for state reduction using the implication chart method consists of the following steps:

1. Construct the implication chart, consisting of one square for each possible combination of states taken two at a time.
2. For each square labeled by states S_i and S_j, if the outputs of the states differ, mark the square with an X; these states are *not* equivalent. Otherwise, they may be equivalent. Within the square, write implied pairs of next states for all input combinations.
3. Systematically advance through the squares of the implication chart. If the square labeled by states S_i,S_j contains an implied pair S_m-S_n and square S_m,S_n is marked with an X, then mark S_i,S_j with an X. Since S_m and S_n are not equivalent, neither are S_i and S_j.
4. Continue executing step 3 until no new squares are marked with an X.
5. For each remaining unmarked square S_i,S_j, you can conclude that states S_i and S_j are equivalent.

9.3 State Assignment

The number of gates needed to implement a sequential logic network depends strongly on how we assign *encoded* Boolean values to *symbolic* states. Unfortunately, the only way to obtain the best possible assignment is to try every choice for the encoding, an extremely large number for real state machines. For example, a four-state finite state machine, such as the traffic light controller of the last chapter, has 4! (4 factorial) = 4 * 3 * 2 * 1 = 24 different encodings (see Figure 9.19).

9.3.1 Traffic Light Controller

To illustrate the impact of state encoding on the next-state and output logic, let's use the symbolic state transition table for the traffic light controller, shown in Figure 9.20. The input combinations that cause the state transitions are shown at the left of the table. The symbolic state names HG, HY, FG, FY represent the states highway green/farmroad red, highway yellow/farmroad red, highway red/farmroad green, and highway red/farmroad yellow. We have already encoded the traffic light outputs: 00 = Green, 01 = Yellow, and 10 = Red.

HG	HY	FG	FY
00	01	10	11
00	01	11	10
00	10	01	11
00	10	11	01
00	11	01	10
00	11	10	01
01	00	10	11
01	00	11	10
01	10	00	11
01	10	11	00
01	11	00	10
01	11	10	00

HG	HY	FG	FY
10	00	01	11
10	00	11	01
10	01	00	11
10	01	11	00
10	11	00	01
10	11	01	00
11	00	01	10
11	00	10	01
11	01	00	10
11	01	10	00
11	10	00	01
11	10	01	00

Figure 9.19 Alternative state encodings of the traffic light controller.

Inputs			Present State	Next State	Outputs		
C	TL	TS	Q_1 Q_0	P_1 P_0	ST	H_1 H_0	F_1 F_0
0	X	X	HG	HG	0	00	10
X	0	X	HG	HG	0	00	10
1	1	X	HG	HY	1	00	10
X	X	0	HY	HY	0	01	10
X	X	1	HY	FG	1	01	10
1	0	X	FG	FG	0	10	00
0	X	X	FG	FY	1	10	00
X	1	X	FG	FY	1	10	00
X	X	0	FY	FY	0	10	01
X	X	1	FY	HG	1	10	01

Figure 9.20 Traffic light controller symbolic state transition table.

We can use *espresso* to examine the alternative state assignments rapidly. Figure 9.21 shows the generic truth table description that is input to *espresso*. We simply replace the symbolic state names HG, HY, FG, and FY with a particular encoding. Before we do a state assignment

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 10
0-- HG HG 00010
-0- HG HG 00010
11- HG HY 10010
--0 HY HY 00110
--1 HY FG 10110
10- FG FG 01000
0-- FG FY 11000
-1- FG FY 11000
--0 FY FY 01001
--1 FY HG 11001
.e
```

Figure 9.21 *Espresso* input for the traffic light controller.

and two-level minimization, the finite state machine requires 10 unique product terms (one for each row of Figure 9.21).

Figure 9.22 and Figure 9.23 show the results of *espresso* runs with the state assignments HG = 00, HY = 01, FG = 11, FY = 10 and HG = 00, HY = 10, FG = 01, FY = 11 respectively. A cursory glance shows that the second encoding uses fewer product terms, eight versus nine, and fewer literals, 21 versus 26.

Comparison of the Two Encodings Let's look at the relative complexity of the two implementations. The logic equations implied by the two alternative encodings are the following.

First encoding:

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 10
0-- 00 00 00010
-0- 00 00 00010
11- 00 01 10010
--0 01 01 00110
--1 01 11 10110
10- 11 11 01000
0-- 11 10 11000
-1- 11 10 11000
--0 10 10 01001
--1 10 00 11001
.e
```

(a) *Espresso* input

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 9
11-00 0110000
10-11 1101000
--101 1010000
--010 1001001
---01 0100100
--110 0011001
---0- 0000010
0--11 1011000
-1-11 1011000
.e
```

(b) *Espresso* output

Figure 9.22 First encoding.

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 10
0-- 00 00 00010
-0- 00 00 00010
11- 00 10 10010
--0 10 10 00110
--1 10 01 10110
10- 01 01 01000
0-- 01 11 11000
-1- 01 11 11000
--0 11 11 01001
--1 11 00 11001
.e
```

(a) *Espresso* input

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 8
11-0- 1010000
--010 1000100
0--01 1010000
--110 0110100
--111 0011001
----0 0000010
---01 0101000
--011 1101001
.e
```

(b) *Espresso* output

Figure 9.23 Second encoding.

$$P_1 = C \bullet \overline{TL} \bullet Q_1 \bullet Q_0 + TS \bullet \overline{Q_1} \bullet Q_0 + \overline{TS} \bullet Q_1 \bullet \overline{Q_0} + \overline{C} \bullet Q_1 \bullet Q_0 + TL \bullet Q_1 \bullet Q_0$$

$$P_0 = C \bullet TL \bullet \overline{Q_1} \bullet \overline{Q_0} + C \bullet \overline{TL} \bullet Q_1 \bullet Q_0 + \overline{Q_1} \bullet Q_0$$

$$ST = C \bullet TL \bullet \overline{Q_1} \bullet \overline{Q_0} + TS \bullet \overline{Q_1} \bullet Q_0 + TS \bullet Q_1 \bullet \overline{Q_0} + \overline{C} \bullet Q_1 \bullet Q_0 + TL \bullet Q_1 \bullet Q_0$$

$$H_1 = C \bullet \overline{TL} \bullet Q_1 \bullet Q_0 + \overline{TS} \bullet Q_1 \bullet \overline{Q_0} + TS \bullet Q_1 \bullet \overline{Q_0} + \overline{C} \bullet Q_1 \bullet Q_0 + TL \bullet Q_1 \bullet Q_0$$

$$H_0 = \overline{Q_1} \bullet Q_0$$

$$F_1 = \overline{Q_1}$$

$$F_0 = \overline{TS} \bullet Q_1 \bullet \overline{Q_0} + TS \bullet Q_1 \bullet \overline{Q_0}$$

With conventional gate logic, the encoding requires 3 five-input gates, 2 four-input gates, 6 three-input gates, and 2 two-input gates, a total of 13 gates. We assume that variables and their complements are available to the network.

Second encoding:

$$P_1 = C \bullet TL \bullet \overline{Q_1} + \overline{TS} \bullet Q_1 \bullet \overline{Q_0} + \overline{C} \bullet \overline{Q_1} \bullet Q_0 + \overline{TS} \bullet Q_1 \bullet Q_0$$

$$P_0 = TS \bullet Q_1 \bullet \overline{Q_0} + \overline{Q_1} \bullet Q_0 + \overline{TS} \bullet Q_1 \bullet Q_0$$

$$ST = C \bullet TL \bullet \overline{Q_1} + \overline{C} \bullet \overline{Q_1} \bullet Q_0 + TS \bullet Q_1 \bullet \overline{Q_0} + TS \bullet Q_1 \bullet Q_0$$

$$H_1 = TS \bullet Q_1 \bullet Q_0 + \overline{Q_1} \bullet Q_0 + \overline{TS} \bullet Q_1 \bullet Q_0$$

$$H_0 = \overline{TS} \bullet Q_1 \bullet \overline{Q_0} + TS \bullet Q_1 \bullet \overline{Q_0}$$

$$F_1 = \overline{Q_0}$$

$$F_0 = TS \bullet Q_1 \bullet Q_0 + \overline{TS} \bullet Q_1 \bullet Q_0$$

This encoding requires 2 four-input gates, 8 three-input gates, and 3 two-input gates, for a total of 13 gates. This implementation uses the same number of gates, but it makes more extensive use of gates with smaller fan-ins. This reduces overall wiring and is one reason why it is often more useful to count literals than gates in comparing circuit complexity.

In the next two subsections, we present methods for finding good state encodings. These are suitable for pencil and paper, as well as computer-aided design tools.

9.3.2 Pencil-and-Paper Methods

Without computer-aided design tools, there is little you can do to generate a good encoding. Hand enumeration using trial and error becomes tedious even for a relatively small number of states. An n-state finite state machine has $n!$ different encodings. And this is only the lower

bound. If the state is not densely encoded in the fewest number of bits, even more encodings are possible.

To make the problem more tractable when you must use hand methods, designers have developed a collection of heuristic "guidelines." These try to reduce the distance in Boolean n-space between related states. For example, if state Y is reached by a transition from state X, then the encodings should differ by as few bits as possible. The next-state logic will be minimized if you follow such guidelines. We examine them in this section.

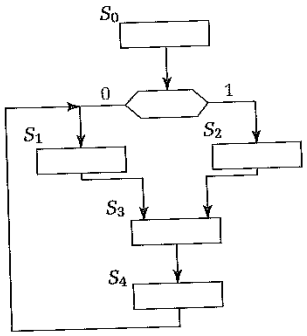


Figure 9.24 Five-state ASM chart.

State Maps *State maps*, similar in concept to K-maps, provide a means of observing adjacencies in state assignments. The squares of the state map are indexed by the binary values of state bits; the state given that encoding is placed in the map square. Obviously the technique is limited to situations in which a K-map can be used, that is, up to six variables.

Figure 9.24 presents an ASM chart for a five-state finite state machine. Figure 9.25 gives two alternative state assignments and their representations in state maps.

Minimum-Bit-Change Heuristic One heuristic strategy assigns states so that the number of bit changes for all state transitions is minimized. For example, the assignment of Figure 9.25(a) is not as good as the one in Figure 9.25(b) under this criterion:

Transition	First Assignment Bit Changes	Second Assignment Bit Changes
S_0 to S_1:	2	1
S_0 to S_2:	3	1
S_1 to S_3:	3	1
S_2 to S_3:	2	1
S_3 to S_4:	1	1
S_4 to S_1:	2	2

The first assignment leads to 13 different bit changes in the next-state function, the second only 7 bit changes.

We derived the first assignment completely at random and the second assignment with minimum transition distance in mind. Here is how we did it. We made the assignment for S_0 first. Because of the way reset logic works, it usually makes sense to assign all zeros to the starting state. We make assignments for S_1 and S_2 next, placing them next to S_0 because they are targets of transitions out of the starting state.

Note how we used the edge adjacency of the state map. This is so we can place S_3 between the assignments for S_1 and S_2, since it is the target of transitions from both of these states.

Finally, we place S_4 adjacent to S_3, since it is the destination of S_3's only transition. It would be perfect if S_4 could also be placed distance 1

State Name	Assignment Q_2	Q_1	Q_0
S_0	0	0	0
S_1	1	0	1
S_2	1	1	1
S_3	0	1	0
S_4	0	1	1

Assignment

Q_2 \ $Q_1 Q_0$	00	01	11	10
0	S_0		S_4	S_3
1		S_1	S_2	

State Map

(a) First state assignment and map

State Name	Assignment Q_2	Q_1	Q_0
S_0	0	0	0
S_1	0	0	1
S_2	0	1	0
S_3	0	1	1
S_4	1	1	1

Assignment

Q_2 \ $Q_1 Q_0$	00	01	11	10
0	S_0	S_1	S_3	S_2
1			S_4	

State Map

(b) Second state assignment and map

Figure 9.25 Five-state finite state machine.

from S_0, but it is not possible to do this and satisfy the other desired adjacencies.

The resulting assignment exhibits only seven bit transitions. There may be many other assignments with the same number of bit transitions, and perhaps an assignment that needs even fewer.

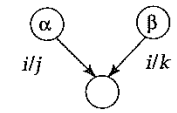
The minimum-bit-change heuristic, although simple, is not likely to achieve the best assignment. For a finite state machine like the traffic light controller, cycling through its regular sequence of states, the minimum transition distance is obtained by a Gray code assignment: HG = 00, HY = 01, FG = 11, FY = 10. This was the first state assignment we tried in the previous subsection, and it was not as good as the second assignment, even though the latter did not involve a minimum number of bit changes.

Guidelines Based on Next State and Input/Outputs Although the criterion of minimum transition distance is simple, it suffers by not considering the input and output values in determining the next state. A second set of heuristic guidelines makes an effort to consider this in the assignment of states:

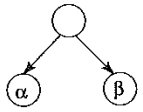
Highest priority: States with the same next state for a given input transition should be given adjacent assignments in the state map.

Medium priority: Next states of the same state should be given adjacent assignments in the state map.

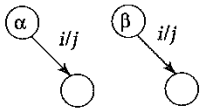
Lowest priority: States with the same output for a given input should be given adjacent assignments in the state map.



Highest Priority



Medium Priority



Lowest Priority

Figure 9.26 Adjacent assignment priorities.

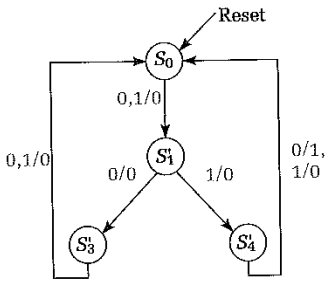


Figure 9.27 Reduced state diagram for 3-bit sequence detector.

The guidelines, illustrated in Figure 9.26 for the candidate states α and β, are ranked from highest to lowest priority. The first two rules attempt to group together ones in the next-state maps, while the third rule performs a similar grouping function for the output maps. We do a state assignment by listing all state adjacencies implied by the guidelines, satisfying as many of these as possible.

Example Applying the Guidelines Consider the state transition table of Figure 9.13 for the 3-bit sequence detector. The corresponding state diagram is shown in Figure 9.27. Let's apply the state assignment guidelines to this state diagram.

The highest-priority constraint for adjacent assignment applies to states that share a common next state on the same input. In this case, states S_3' and S_4' both have S_0 as their next state. No other states share a common next state.

The medium-priority assignment is for states that have a common ancestor state. Again, S_3' and S_4' are the only states that fit this description.

The lowest-priority assignments are made for states that have the same output behavior for a given input. S_0, S_1', and S_3' all output 0 when the input is 0. Similarly, S_0, S_1', S_3', and S_4' output 0 when the input is 1.

The constraints on the assignments can be summarized as follows:

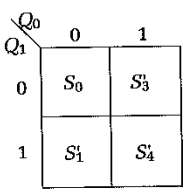
Highest priority: (S_3', S_4');

Medium priority: (S_3', S_4');

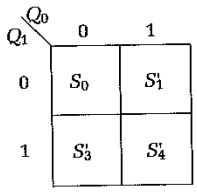
Lowest priority: 0/0: (S_0, S_1', S_3');
1/0: (S_0, S_1', S_3', S_4');

Since the finite state machine has four states, we can make the assignment onto two state bits. In general, it is a good idea to assign the reset state to state map square 0. Figure 9.28 shows two possible assignments. Both assign S_0 to 00 and place S_3' and S_4' adjacent to each other.

Example Applying the Guidelines in a More Complicated Case As another example, let's consider the more complicated state diagram of



(a) First state assignment



(b) Second state assignment

Figure 9.28 Two possible state assignments.

the 4-bit string recognizer of Figure 9.7. Applying the guidelines yields the following set of assignment constraints:

Highest priority: (S_3', S_4'), (S_7', S_{10}');

Medium priority: (S_1, S_2), $2 \times (S_3', S_4')$, (S_7', S_{10}');

Lowest priority: 0/0: $(S_0, S_1, S_2, S_3', S_4', S_7')$;
1/0: $(S_0, S_1, S_2, S_3', S_4', S_7', S_{10}')$;

Figure 9.29 shows two alternative assignments that meet most of these constraints. We start with Figure 9.29(a) and first assign the reset state to the encoding for 0. Since (S_3', S_4') is both a high-priority and medium-priority adjacency, we make their assignments next. S_3' is assigned 011 and S_4' is assigned 111.

We assign (S_7', S_{10}') next because this pair also appears in the high- and medium-priority lists. We assign them the encodings 010 and 110, respectively. Besides giving them adjacent assignments, this places S_7 near S_0, S_3', and S_4', which satisfies some of the lower-priority adjacencies.

The final adjacency is (S_1, S_2). We give them the assignments 001 and 101. This satisfies a medium-priority placement as well as the lowest-priority placements.

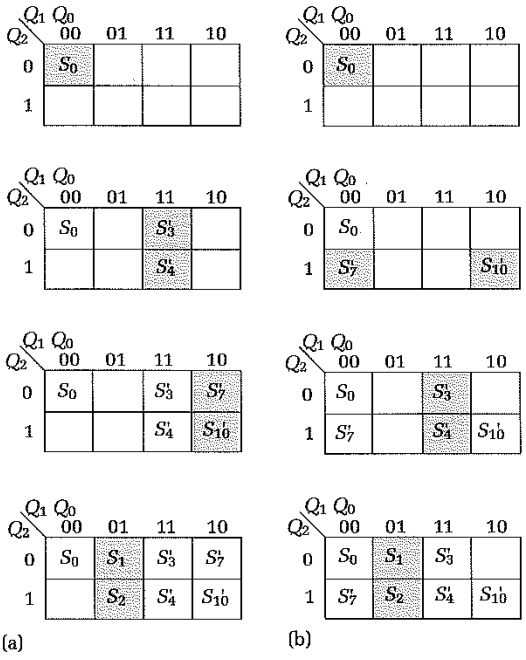


Figure 9.29 State assignment example.

The second assignment is shown in Figure 9.29(b). We arrived at it by a similar line of reasoning, except that we assigned S_7' and S_{10}' the states 100 and 110. The second assignment does about as good a job as the first, satisfying all of the high- and medium-priority guidelines, as well as most of the lowest-priority ones.

Applying the Guidelines: Why They Work The state assignment guidelines attempt to maximize the adjacent groupings of 1's in the next-state and output functions. Let P_2, P_1, and P_0 be the next-state functions, expressed in terms of the current state Q_2, Q_1, Q_0 and the input X. To see how effective the guidelines were, let's compare the assignment of Figure 9.29(a) with a more naive assignment: $S_0 = 000$, $S_1 = 001$, $S_2 = 010$, $S_3' = 011$, $S_4' = 100$, $S_7' = 101$, $S_{10}' = 110$.

Figure 9.30 compares the encoded next-state tables and K-maps for the two encodings. The 1's are nicely clustered in the next state K-maps for the assignment derived from the guidelines. We can implement P_2 with three product terms and P_1 and P_2 with one each.

In the second assignment, the 1's are spread throughout the K-maps, since we made the assignment with no attempt to cluster the 1's usefully. In this implementation, the next-state functions P_2, P_1, and P_0 each require three product terms, with a considerably larger number of literals overall.

9.3.3 One Hot Encodings

So far, our goal has been *dense* encodings: state encodings in as few bits as possible. An alternative approach introduces additional flip-flops, in the hope of reducing the next-state and output logic.

One form of this method is called *one hot encoding*. A machine with n states is encoded using exactly n flip-flops. Each state is represented by an n-bit binary code in which exactly 1 bit is asserted. This is the origin of the term "one hot."

Let's consider the traffic light finite state machine described earlier in this section. The following would be a possible one hot encoding of the machine's state:

$HG = 0001$

$HY = 0010$

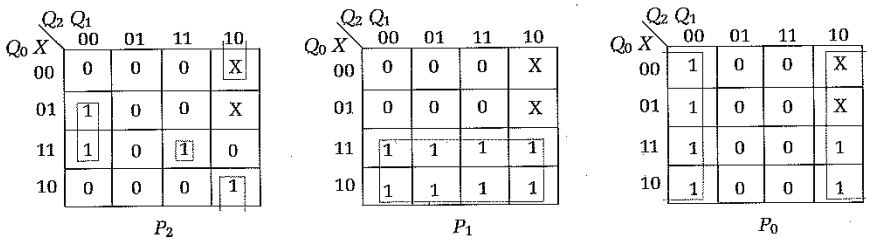
$FG = 0100$

$FY = 1000$

The state is encoded in four flip-flops rather than two, and only 1 bit is asserted in each of the states.

Figure 9.31 shows the *espresso* inputs and outputs for this encoding. It yields eight product terms, as good as the result of Figure 9.23. However, the logic is considerably more complex:

Current State	Next State $X=0$	Next State $X=1$
(S_0) 000	001	101
(S_1) 001	011	111
(S_2) 101	111	011
(S_3') 011	010	010
(S_4) 111	010	110
(S_7') 010	000	000
(S_{10}') 110	000	000



Current State	Next State $X=0$	Next State $X=1$
(S_0) 000	001	010
(S_1) 001	011	100
(S_2) 010	100	011
(S_3') 011	101	101
(S_4') 100	101	110
(S_7') 101	000	000
(S_{10}') 110	000	000

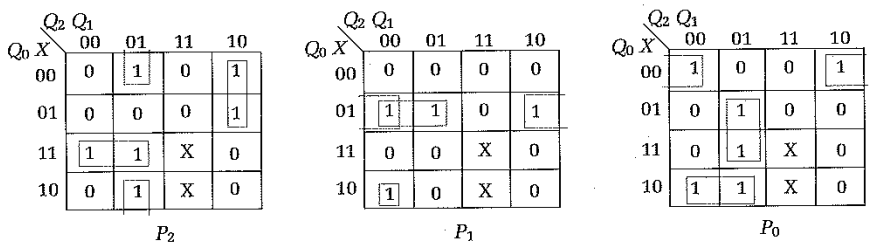


Figure 9.30 State assignment example.

$$P_3 = \overline{C} \bullet \overline{Q}_3 \bullet Q_2 \bullet \overline{Q}_1 \bullet \overline{Q}_0 + TL \bullet \overline{Q}_3 \bullet Q_2 \bullet \overline{Q}_1 \bullet \overline{Q}_0 + \overline{TS} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$P_2 = C \bullet \overline{TL} \bullet \overline{Q}_3 \bullet Q_2 \bullet \overline{Q}_1 \bullet \overline{Q}_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$P_1 = C \bullet TL \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + \overline{TS} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$P_0 = \overline{TL} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + \overline{C} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$ST = C \bullet TL \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + \overline{C} \bullet \overline{Q}_3 \bullet Q_2 \bullet \overline{Q}_1 \bullet \overline{Q}_0 + TL \bullet \overline{Q}_3 \bullet Q_2 \bullet \overline{Q}_1 \bullet \overline{Q}_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$H_1 = C \bullet TL \bullet Q_3 \bullet Q_2 \bullet Q_1 \bullet Q_0 + C \bullet Q_3 \bullet Q_2 \bullet Q_1 \bullet Q_0 + TL \bullet \overline{Q}_3 \bullet Q_2 \bullet \overline{Q}_1 \bullet \overline{Q}_0 + \overline{TS} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$H_0 = \overline{TS} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$F_1 = C \bullet TL \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + \overline{TL} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + \overline{C} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet \overline{Q}_1 \bullet Q_0 + \overline{TS} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

$$F_0 = \overline{TS} \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0 + TS \bullet \overline{Q}_3 \bullet \overline{Q}_2 \bullet Q_1 \bullet \overline{Q}_0$$

```
.i 7
.o 9
.ilb c tl ts q3 q2 q1 q0
.ob p3 p2 p1 p0 st h1 h0 f1 f0
.p 10
0-- 0001 0001 00010
-0- 0001 0001 00010
11- 0001 0010 10010
--0 0010 0010 00110
--1 0010 0100 10110
10- 0100 0100 01000
0-- 0100 1000 11000
-1- 0100 1000 11000
--0 0010 1000 01001
--1 0010 0001 11001
.e
```

(a) *Espresso* input

```
.i 7
.o 9
.ilb c tl ts q3 q2 q1 q0
.ob p3 p2 p1 p0 st h1 h0 f1 f0
.p 8
10-0100 010001000
11-0001 001010010
-0-0001 000100010
0--0001 000100010
0--0100 100011000
-1-0100 100011000
--00010 101001111
--10010 010111111
.e
```

(b) *Espresso* output

Figure 9.31 *Espresso* input/output for the one hot encoding of the traffic light state machine.

The product terms are all five and six variables, with two to five terms per output. This is rather complex for discrete logic but would not cause problems for a PLA-based design.

9.3.4 Computer Tools: *Nova, Mustang, Jedi**

The previous subsections described various heuristic approaches for obtaining a good state encoding. None are guaranteed to obtain the best result. In this section, we examine three programs for computer-generated state assignments.

*This subsection requires access to software developed at the University of California, Berkeley. If you do not have access to it, you may want to skim this section, or skip it altogether.

```
0-- HG HG 00010
-0- HG HG 00010
11- HG HY 10010
--0 HY HY 00110
--1 HY FG 10110
10- FG FG 01000
0-- FG FY 11000
-1- FG FY 11000
--0 FY FY 01001
--1 FY HG 11001
```

Figure 9.32 *Nova* inputs for the traffic light controller.

```
# .start_codes
.code HG 00
.code HY 11
.code FG 01
.code FY 10
# .end_codes
#
BLIF Representation
.i 5
.o 7
.p 9
10-01 0101000
11-00 1110000
--01- 1000000
--11- 0010000
---00 0000010
---10 0001001
0--01 1011000
-1-01 1011000
---11 0100110
.e
#
```

Figure 9.33 Abbreviated *Nova* outputs.

The state assignment guidelines of Section 9.3.2 place related states together in the state map, thereby clustering the 1's in those functions. However, we cannot evaluate an assignment fully without actually minimizing the functions. When it comes to hand techniques, this means the K-map method. If we have to use K-maps to minimize the functions for each possible assignment, we are not likely to examine very many of them!

Tools are available that follow the basic kinds of heuristics described above. Unlike hand methods, they are not limited to six state bits, they can generate many alternative assignments rapidly, and they can evaluate the derived assignments by invoking minimization tools like *espresso* or *misII* automatically. Three tools that provide this function are *nova*, for two-level logic implementations, and *mustang* and *jedi*, for multilevel logic. *Jedi* is somewhat more general; it can be used for general-purpose symbolic encodings, such as outputs, as well as next states. We examine each of these programs in the following subsections.

***Nova:* State Assignment for Two-Level Implementation** The inputs to *nova* are similar to the truth table format already described for *espresso*. The input file consists of the truth table entries for the finite state machine description. The latter are of the form:

inputs current_state next_state outputs

Figure 9.32 shows the format for our example traffic light controller. The states are symbolic; the inputs and outputs are binary encoded. It is important to separate each section with a space. We interpret the first line as "if input 1 (car sensor) is unasserted and the state is HG (highway green), then the next state is HG (highway green) and the outputs are 00010 ($ST = 0$, $H_1{:}H_0 = 00$ "green," $F_1{:}F_0 = 10$ "red").

Figure 9.33 gives the abbreviated *nova* output associated with the state machine of Figure 9.32, assuming you have requested a "greedy" state assignment. The "codes" section shows the state assignment: HG = 00, HY = 11, FG = 01, and FY = 10. The *espresso* truth table is included in the output, indicating that it takes nine product terms to implement the state machine.

Intuitively, the state assignment algorithms used by *nova* are much like the assignment guidelines of Section 9.3.2. States that are mapped by some input into the same next state and that assert the same output are partitioned into groups. In the terminology of state assignment, these are called *input constraints*. *Nova* attempts to assign adjacent encodings within the smallest Boolean cube to states in the same group. A related concept is *output constraints*. States that are next states of a common predecessor state are given adjacent assignments.

Nova implements a wide range of state-encoding strategies, any of which you can select when you invoke the program:

- *Greedy:* makes its state assignment based on satisfying as many of the input constraints as it can. When using a greedy approach, *nova* looks only at new assignments that strictly improve on those it has already examined.
- *Hybrid:* also makes its state assignment based on satisfying the input constraints. However, *nova* will examine some assignments that start off looking worse, but may eventually yield a better assignment. This yields better assignments than the greedy strategy.
- *I/O Hybrid:* similar to hybrid, except it tries to satisfy input and output constraints. Its results are usually better than hybrid's.
- *Exact:* obtains the best encoding that satisfies the input constraints. Since the output constraints are not considered, this is still not the best possible encoding.
- *Input Annealing:* similar to hybrid, except it uses a more sophisticated method for improving the state assignment.
- *One-Hot:* uses a one-hot encoding. This rarely yields a minimum product term assignment, but it may dramatically reduce the complexity of the product terms (in other words, the literal count may be greatly reduced).
- *Random:* uses a randomly generated encoding. *Nova* will generate the specified number of random assignments. It will report on the best assignment it has found (the one requiring the smallest number of product terms).

For the traffic light controller, the various encoding algorithms yield the following assignments:

	HG	HY	FG	FY	Number of product terms	PLA Area
Greedy:	00	11	01	10	9	153
Hybrid:	00	11	10	01	9	153
Exact:	11	10	01	00	10	170
IO Hybrid:	00	01	11	10	9	153
I Annealing:	01	10	11	10	9	153
Random:	11	00	01	10	9	153

None of the assignments found by *nova* match the eight-product-term encoding we found in Figure 9.23. But we shouldn't despair just because the tool did not find the best possible assignment. The advantage of the

```
- S0 S1 0
0 S1 S3 0
1 S1 S4 0
- S3 S0 0
0 S4 S0 1
1 S4 S0 0
```

Figure 9.34 *Nova* inputs for the 3-bit string recognizer.

computer-based tool is that it finds reasonable solutions rapidly. Thus you can examine a variety of encodings and choose the one that best reduces your implementation task.

As another example, let's consider the 3-bit sequence recognizer of Figure 9.27. Its *nova* input file is shown in Figure 9.34. The state assignments found by *nova* are the following:

	S_0	S_1'	S_3'	S_4'	Number of product terms	PLA Area
Greedy:	00	01	11	10	4	36
Hybrid:	00	01	10	11	4	36
Exact:	00	01	10	11	4	36
IO Hybrid:	00	10	01	11	4	36
I Annealing:	00	01	11	10	4	36
Random:	00	11	10	01	4	36

Several of the assignments correspond to the ones found in Figure 9.28(a) and (b).

```
0 S0 S1 0
1 S0 S2 0
0 S1 S3 0
1 S1 S4 0
0 S2 S4 0
1 S2 S3 0
- S3 S7 0
0 S4 S7 0
1 S4 S10 0
- S7 S0 0
0 S10 S0 1
1 S10 S0 0
```

Figure 9.35 *Nova* inputs for the 4-bit string recognizer.

The last example is the 4-bit recognizer of Figure 9.7. The *nova* input file is given in Figure 9.35. *Nova* produced the following assignments:

	S_0	S_1	S_2	S_3'	S_4'	S_7'	S_{10}'	Number of product terms
Greedy:	100	110	010	011	111	000	001	7
Hybrid:	101	110	111	001	011	000	010	7
Exact:	101	110	111	001	011	000	010	7
IO Hybrid:	110	011	001	100	101	000	010	7
I Annealing:	100	101	001	111	110	000	010	6
Random:	011	100	101	110	111	000	001	7

None of these assignments match those derived in Figure 9.29, since S_0 has not been assigned 000. Figure 9.36 shows that the adjacencies that were highly desirable based on the guidelines—that is, S_3' adjacent to S_4', S_7' adjacent to S_{10}', and S_1 adjacent to S_2—are satisfied by the input annealing assignment.

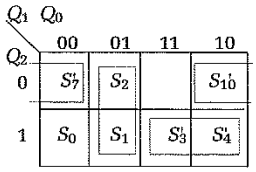


Figure 9.36 State map for the best *Nova* assignment of the finite state machine of Figure 9.7.

Mustang The minimum number of product terms is a good criterion if you are going to implement the logic in a two-level form, such as a PLA. This is what *nova* uses. Literal count is better if you plan to implement the logic in multiple levels. *Mustang* takes this approach. Its optimization criterion is to minimize the number of literals in the multi-level factored form of the next-state and output functions.

```
.i 3
.o 5
.s 2
0--   HG HG 00010
-0-   HG HG 00010
11-   HG HY 10010
--0   HY HY 00110
--1   HY FG 10110
10-   FG FG 01000
0--   FG FY 11000
-1-   FG FY 11000
--0   FY FY 01001
--1   FY HG 11001
```

Figure 9.37 *Mustang* input file for traffic light controller.

Mustang has an input format similar to that of *nova*. The only difference is that the number of inputs, outputs, and state bits must be explicitly declared with `.i`, `.o`, and `.s` directives. For example, the traffic light input file is shown in Figure 9.37.

Mustang implements several alternative strategies for state assignment, specified by the user on the command line. These include:

- *Random:* chooses a random state encoding.
- *Sequential:* assigns the states binary codes in a sequential order.
- *One-Hot:* makes a one-hot state assignment.
- *Fan-in:* works on the input and fan-in of each state. Next states that are produced by the same inputs from similar sets of present states are given adjacent state assignments. The assignment is chosen to maximize the common subexpressions in the next-state function. This is much like the high-priority assignment guideline of Section 9.3.2.
- *Fan-out:* works on the output and fan-out of each state, without regard to inputs. Present states with the same outputs that produce similar sets of next states are given adjacent state assignments. Again, this maximizes the common subexpressions in the output and next-state functions. This approach works well for finite state machines with many outputs but few inputs. It is much like the medium- and low-priority assignment guidelines from Section 9.3.2.

Mustang derives the following assignments for the traffic light state machine:

	HG	HY	FG	FY	Number of product terms
Random:	01	10	11	00	9
Sequential:	01	10	11	00	9
Fan-in:	00	01	10	11	8
Fan-out:	10	11	00	01	8

The eight term encodings are actually better than any of those found by *nova*. To determine the multilevel implementation, you must invoke *misII* on the *espresso* file created by *mustang*.

For example, the *misII* output for the fan-in encoding of the traffic light controller is the following:

$$P_1 = H_0 \bullet TS + F_0 \bullet \overline{TS} + F_0 \bullet Q_1$$

$$P_0 = P_1 \bullet \overline{C} \bullet Q_1 + C \bullet TL \bullet \overline{Q}_0 + \overline{TS} \bullet Q_0$$

$$ST = P_0 \bullet \overline{Q}_0 + \overline{P}_0 \bullet Q_0$$

$$H_1 = F_0 + Q_1 \bullet \overline{Q}_0$$
$$H_0 = \overline{Q}_1 \bullet Q_0$$
$$F_1 = \overline{Q}_1$$
$$F_0 = \overline{H}_0 \bullet Q_0$$

For comparison with *nova*, *mustang* obtains the following encodings for the 3-bit string recognizer:

	S_0	S_1	S_3'	S_4'	Number of product terms
Random:	01	10	11	00	5
Sequential:	01	10	11	00	5
Fan-in:	10	11	00	01	4
Fan-out:	10	11	00	01	4

The number of product terms to implement an encoding is comparable to the number needed for the *nova* encodings. However, don't forget that the goal of *mustang* is to reduce literal count rather than product terms.

As a final example, let's look at the *mustang* encodings for the 4-bit recognizer:

	S_0	S_1	S_2	S_3'	S_4'	S_7'	S_{10}'	Number of product terms
Random:	101	010	011	110	111	001	000	8
Sequential:	001	010	011	100	101	110	111	8
Fan-in:	100	010	011	000	001	101	110	8
Fan-out:	110	010	011	100	101	000	001	6

It is interesting that in all three cases, *mustang* obtained an encoding that is as good as any of the best encodings found by *nova*.

Jedi The final encoding program we shall examine is *jedi*. It is similar to *mustang* in that its goal is to obtain a good encoding for a multilevel implementation. It is more powerful than *mustang* because it can solve *general encoding problems*: *jedi* can find good encodings for the outputs as well as the states.

Like the other programs, *jedi* implements several alternative encoding strategies that can be selected on the command line. Besides random, one hot, and straightforward, the program supports input dominant, output dominant, modified output dominant, and input/output combination algorithms.

The *jedi* input format is similar to, but slightly different from, the *mustang* input format. We can illustrate this best with an example. Figure 9.38 shows the *jedi* input file for the traffic light controller. The present state and next states each count as a single input, even though they may be encoded by several bits. The `.enum States` line tells *jedi* that there are four states and that they should be encoded in 2 bits and then gives the state names. The `.enum Colors` line tells *jedi* that there are three output colors and that these should also be encoded in 2 bits. You should think of these as enumerated types. The `.itype` and `.otype` lines define the types of the inputs and outputs, respectively.

The encodings obtained from *jedi* for the traffic light controller are:

	HG	HY	FG	FY	Grn	Yel	Red	Number of product terms
Input:	00	10	11	01	11	01	00	9
Output:	00	01	11	10	10	11	01	9
Combination:	00	10	11	01	10	00	01	9
Output':	01	00	10	11	10	00	01	10

For the 3-bit string recognizer, the state assignments are:

	S_0	S_1	S_3'	S_4'	Number of product terms
Input:	01	00	11	10	4
Output:	11	01	00	10	4
Combination:	10	00	11	01	4
Output':	11	01	00	10	4

Finally, for the 4-bit string recognizer, they are:

	S_0	S_1	S_2	S_3'	S_4'	S_7'	S_{10}'	Number of product terms
Input:	111	101	100	010	110	011	001	7
Output:	101	110	100	010	000	111	011	7
Combination:	100	011	111	110	010	000	101	7
Output':	001	100	101	010	011	000	111	6

Let's look at one head-to-head comparison between *mustang* and *jedi*. We will use the *mustang* encoding in which HG = 00, HY = 01, FG = 10, FY = 11, Green = 00, Yellow = 01, and Red = 10 and the *jedi* encoding in which HG = 00, HY = 01, FG = 11, FY = 10, Green = 10, Yellow = 11, and Red = 01. The first encoding used eight product terms in a two-level implementation; the second used nine.

```
.i 4
.o 4
.enum States 4 2 HG HY FG FY
.enum Colors 3 2 GREEN RED YELLOW
.itype Boolean Boolean Boolean States
.otype States Boolean Colors Colors
0 - - HG HG 0 GREEN RED
- 0 - HG HG 0 GREEN RED
1 1 - HG HY 1 GREEN RED
- - 0 HY HY 0 YELLOW RED
- - 1 HY FG 1 YELLOW RED
1 0 - FG FG 0 RED GREEN
0 - - FG FY 1 RED GREEN
- 1 - FG FY 1 RED GREEN
- - 0 FY FY 0 RED YELLOW
- - 1 FY HG 1 RED YELLOW
```

Figure 9.38 *Jedi* input file.

The multilevel implementation for the *mustang* assignment was already shown in the *mustang* section. It requires 26 literals. The *jedi* multilevel implementation is

$$P_1 = H_1 \bullet C \bullet TL + H_0 + F_1 \bullet C \bullet TL$$

$$P_0 = H_0 \bullet TS + F_1 + F_0 \bullet \overline{TS}$$

$$ST = P_1 \bullet H_1 + \overline{P}_1 \bullet F_1 + \overline{H}_1 \bullet \overline{F}_0 \bullet TS$$

$$H_1 = \overline{Q}_1 \bullet \overline{Q}_0$$

$$H_0 = Q_1 \bullet \overline{Q}_0$$

$$F_1 = \overline{H}_0 \bullet Q_1$$

$$F_0 = \overline{H}_1 \bullet \overline{Q}_1$$

It has 27 literals. In terms of straight literal count, the *mustang* encoding is better. If we examine the wiring complexity, the *mustang* encoding is also slightly better.

9.4 Choice of Flip-Flops

After state reduction and state assignment, the next step in the design process is to choose flip-flop types for the state registers. The issues are identical to those in the counter case studies of Chapter 7.

Usually, we have to decide whether to use *J-K* flip-flops or *D* flip-flops. *J-K* devices tend to reduce the gate count but increase the number of connections. *D* flip-flops simplify the implementation process and are well suited for VLSI implementations, where connections are at more of

a premium than gates. Because the CAD tools mentioned in the previous section were developed to assist in VLSI implementations, it is not surprising that they implicitly assume *D* flip-flops as the targets of the assignment. Their best assignment may not lead to the minimum logic for a *J-K* flip-flop implementation.

The following procedure completes the finite state machine implementation, given a particular choice of flip-flops:

1. Given the state assignments, derive the next-state maps from the state transition table.
2. Remap the next-state maps given the excitation tables for the flip-flops chosen to implement the state bits.
3. Minimize the remapped next-state function.

9.4.1 Flip-Flop Choice for the Four-Bit Sequence Detector

Let's illustrate the procedure with the 4-bit sequence detector, using the state assignment of Figure 9.39, the encoded state transition table. Each state has been replaced by its binary encoding given by the state assignment. Figure 9.40 is the encoded next-state map, organized according to the standard binary sequence and showing the don't cares.

***D* Implementation** To obtain the direct form for determining the state machine implementation with *D* flip-flops, represent the encoded next-state functions as K-maps. Figure 9.41 contains the four-variable K-maps for the next-state functions Q_2^+, Q_1^+, Q_0^+, given the current state Q_2, Q_1, Q_0 and the input I. The reduced equations that describe the inputs to the *D* flip-flops are

$$D_{Q2+} = \overline{Q}_2 \bullet Q_1 + Q_0$$
$$D_{Q1+} = Q_1 \bullet Q_0 \bullet I + \overline{Q}_2 \bullet \overline{Q}_0 \bullet \overline{I} + \overline{Q}_2 \bullet \overline{Q}_1$$
$$D_{Q0+} = \overline{Q}_2 \bullet Q_1 + \overline{Q}_2 \bullet \overline{I}$$

Present State	Next State I=0	Next State I=1	Output I=0	Output I=1
000 (S_0)	011 (S_1)	010 (S_2)	0	0
011 (S_1)	101 (S_3')	111 (S_4')	0	0
010 (S_2)	111 (S_4')	101 (S_3')	0	0
101 (S_3')	100	100 (S_7')	0	0
111 (S_4')	(S_7')	110	0	0
100 (S_7')	100	(S_{10}')	0	0
110 (S_{10}')	(S_7')	000	1	0

Figure 9.39 Encoded state transition table for 4-bit sequence detector.

Present State	Next State I=0	Next State I=1
000	011	010
001	XXX	XXX
010	111	101
011	101	111
100	000	000
101	100	100
110	000	000
111	100	110

Figure 9.40 Encoded next-state map.

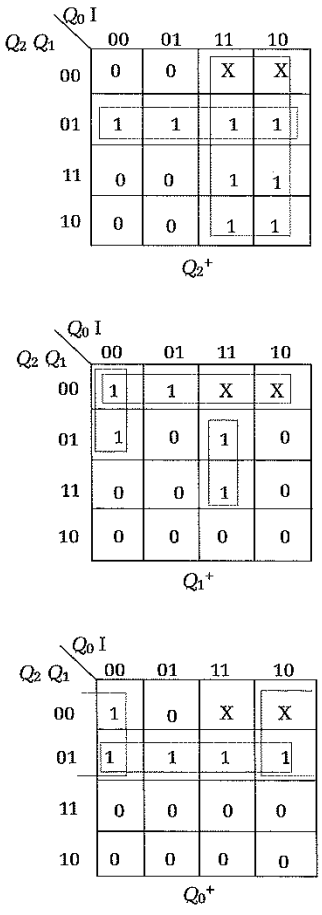


Figure 9.41 Next-state K-maps.

There are six unique product terms and 15 literals. In terms of discrete gates, the implementation requires 3 three-input gates, 5 two-input gates, and 4 inverters, a total of 12 gates.

***J-K* Implementation** For the *J-K* implementation, we begin by remapping the inputs based on the *J-K* excitation tables. Figure 9.42 gives the remapped next-state table, and Figure 9.43 shows the K-maps. The *J-K* logic equations become

$$J_{Q2+} = Q_1 \qquad K_{Q2+} = \overline{Q}_0$$
$$J_{Q1+} = \overline{Q}_2 \qquad K_{Q1+} = \overline{Q}_0 \bullet I + Q_0 \bullet \overline{I} + Q_2 \bullet \overline{Q}_0$$
$$J_{Q0+} = \overline{Q}_2 \bullet Q_1 + \overline{Q}_2 \bullet \overline{I} \qquad K_{Q0+} = Q_2$$

This implementation requires nine unique terms and 14 literals. The gate count is 1 three-input gate, 6 two-input gates, and 3 inverters, a total of 10 gates. This is slightly fewer than the *D* flip-flop implementation. However, when you use structured logic such as a PLA to implement the functions, the option with fewer product terms is better. In this case, it would be the *D* implementation.

9.5 Finite State Machine Partitioning

In the preceding sections, we described the design process for a single monolithic finite state machine. The approach is reasonable for many strategies for implementing a finite state machine, such as using discrete gates.

However, when using some forms of programmable logic, we may need to partition the machine. In some cases we cannot implement a complex finite state machine with a single programmable logic component. The machine might require too many inputs or outputs, or the number of terms to describe the next-state or output functions might be too large, even after state reduction and Boolean minimization.

	Next State		Remapped Next State			
			J	*K*	*J*	*K*
Present State	*I*=0	*I*=1	*I*=0	*I*=0	*I*=1	*I*=1
000	011	010	011	XXX	010	XXX
001	XXX	XXX	XXX	XXX	XXX	XXX
010	111	101	1X1	X0X	1X1	X1X
011	101	111	1XX	X10	1XX	X00
100	000	000	X00	1XX	X00	1XX
101	100	100	X0X	0X1	X0X	0X1
110	000	000	XX0	11X	XX0	11X
111	100	110	XXX	011	XXX	001

Figure 9.42 Remapped next-state table for *J-K* flip-flops.

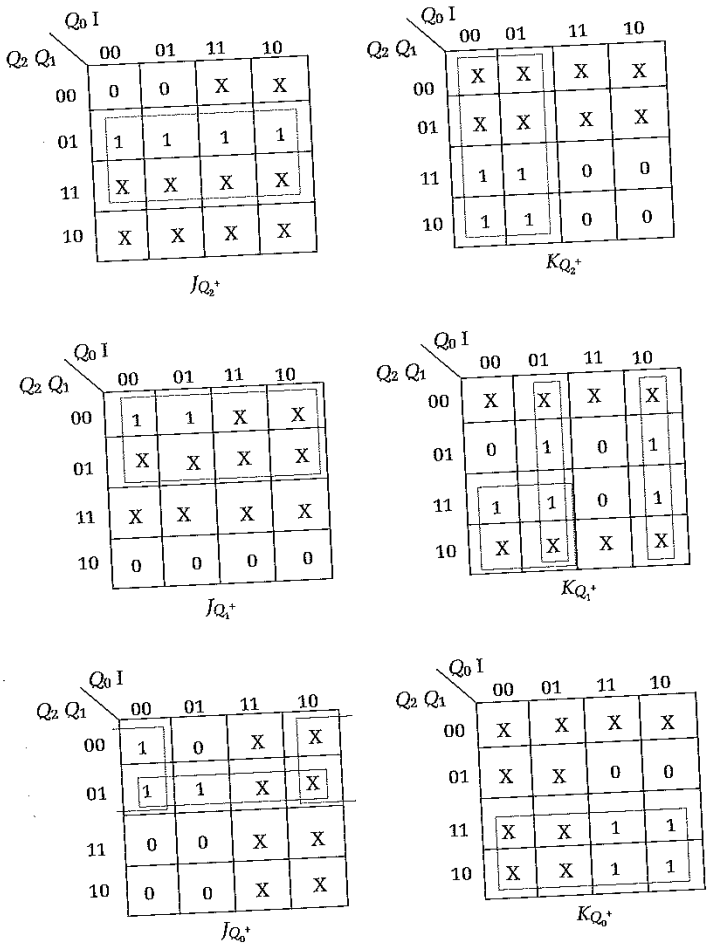


Figure 9.43 Remapped K-maps.

Example 1 FSM Partitioning To illustrate the value of state machine partitioning, suppose we have a finite state machine with 20 inputs and 10 outputs (including next-state outputs). But we only have programmable logic components with 15 inputs and 5 outputs. We cannot implement this finite state machine with a single component.

Suppose we can arrange the outputs in two sets of five, each of which can be computed from different 15-element subsets of the original 20 inputs. Then we could partition the output functions among two programmable logic components, as shown in Figure 9.44. Of course, it isn't

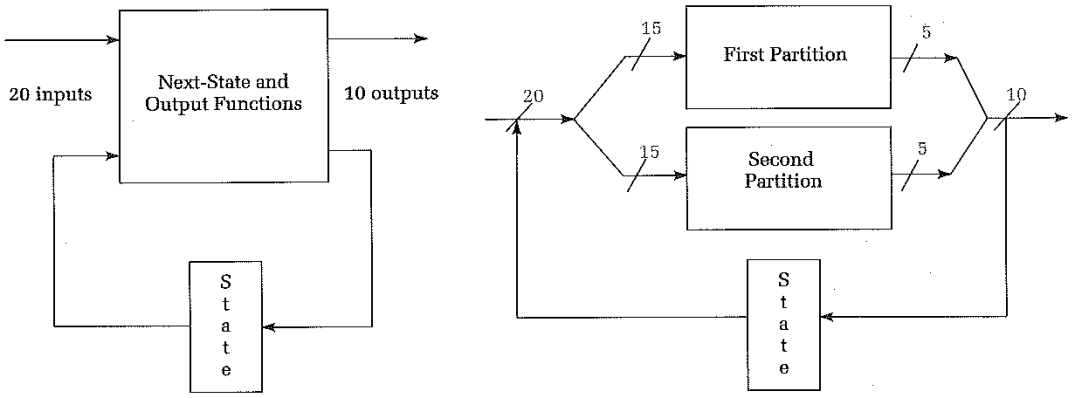


Figure 9.44 Finite state machine partitioning on inputs and outputs.

always possible to find such a fortuitous partitioning. For example, every output might be a function of 16 inputs.

If we cannot reduce the complexity of the finite state machine by simple input/output partitioning, another way to "make it fit" is to partition the single finite state machine into smaller, less complex, communicating finite state machines. We examine this approach in the next subsection.

9.5.1 Finite State Machine Partitioning by Introducing Idle States

Partitioning the finite state machine makes sense if the next-state logic is too complex to implement with the programmable logic components at hand. The problem is that PALs provide a fixed number of product terms per output function. We can make a trade-off between the number of flip-flops needed to encode the state and the complexity of these next-state functions. Our idea is to introduce additional "idle" states into the finite state machine in the hope of reducing the number of terms in the next-state functions.

Example 2 FSM Partitioning For example, Figure 9.45 shows a subset of a state diagram. We have chosen to partition the state diagram into two separate machines, containing states S_1, S_2, S_3 and S_4, S_5, S_6, respectively. The symbols C_i associated with the transitions represent the Boolean conditions under which the transition takes place.

What happens if we partition the state diagram, but a transition must take place between the two pieces? We need to introduce idle states to synchronize the activity between the two finite state machines. In essence, the machine at the left hands control off to the machine at the

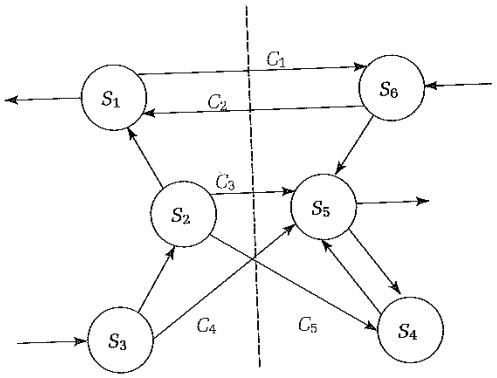


Figure 9.45 State diagram fragment before partitioning.

right when a transition from S_1 to S_6 takes place. The left machine must idle in some new state until it regains control, such as when there is a transition from S_6 back to S_1. In this event, the machine on the right must remain idle until it regains control.

The revised state diagrams are shown in Figure 9.46. We have introduced two new states, S_A and S_B, to synchronize the transitions across the partition boundary. Here is how it works for the state sequence S_1 to S_6 and back to S_1. Initially, the machines are in states S_1 and S_B. If condition C_1 is true, then the left-hand state machine exits S_1 and enters its idle state, S_A. At the same time, the right-hand machine exits S_B and enters S_6.

Suppose that the right-hand machine sequences through some states, eventually returning to S_6. Throughout this time, the left-hand machine

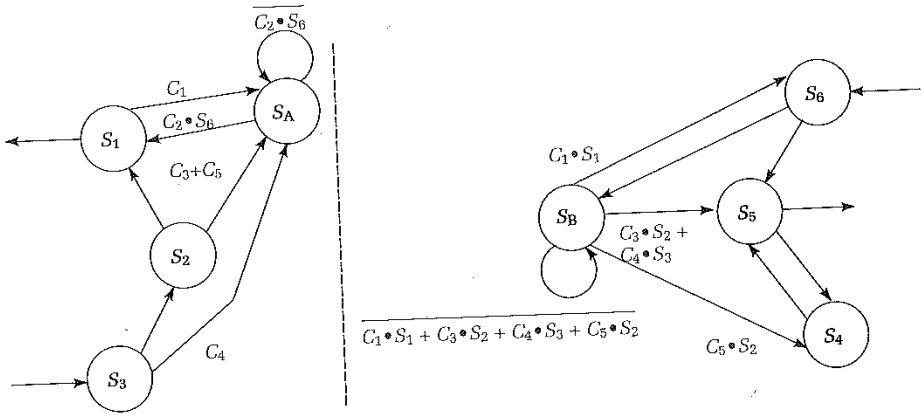
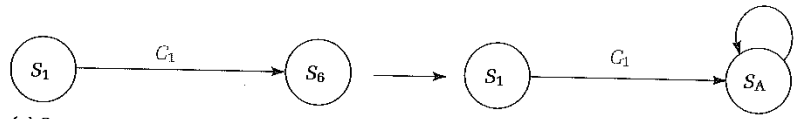


Figure 9.46 State diagram fragment after partitioning.

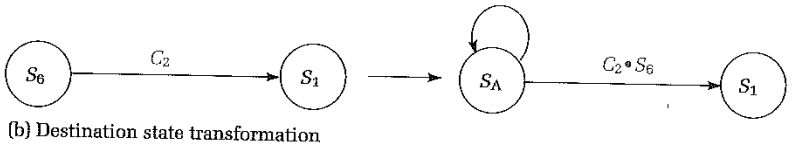
remains in its idle state. If the right-hand machine is in S_6 and C_2 is true, it next enters its idle state, S_B. At the same instant, the left-hand machine exits S_A, returning to S_1. While the left-hand machine sequences through states, the right-hand machine idles in S_B.

Rules for Partitioning We are ready to describe the rules for introducing idle states into a partitioned finite state machine. We illustrate each rule with an example from the partitioned state machine of the previous subsection. All the rules involve transitions that cross the partition boundary.

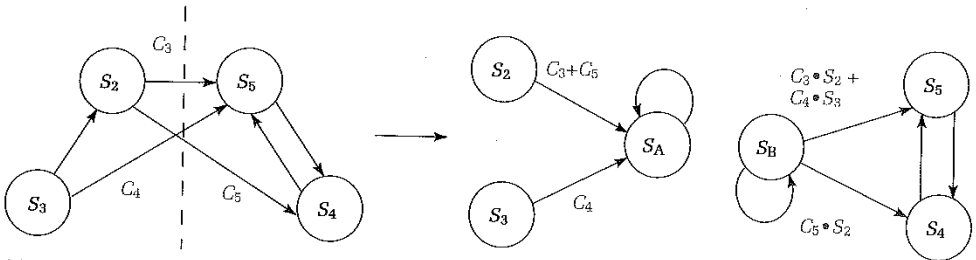
The first rule applies for a state that is the source of a transition that crosses the boundary. The case is shown in Figure 9.47(a). The cross-boundary transition is replaced by a transition to the idle state, labeled by



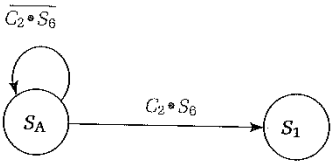
(a) Source state transformation



(b) Destination state transformation



(c) Multiple transitions with same source or destination



(d) Hold condition for idle state

Figure 9.47 Rules for partitioning.

the same exit condition as the original transition. For example, the S_1-to-S_6 transition is replaced by a transition with the same condition to S_A.

The second rule applies to the destination of a transition that crosses the partition boundary. This is shown in Figure 9.47(b). The transition is replaced with an exit transition from the idle state, labeled with the original condition ANDed with the source state. For example, the transition from S_6 to S_1 is replaced with a transition from S_A. We exit the idle state when both C_2 is true and the right-hand state machine is in S_6. Hence, the transition is labeled with the condition $C_2 \bullet S_6$.

The third rule applies when multiple transitions share the same source or destination. This case is illustrated in Figure 9.47(c). If a state is the source of multiple transitions across the partition boundary, all of these are collapsed into a single transition to the idle state. The exit conditions are ORed together to label the new transition. For example, S_2 has transitions to states S_5 and S_4. These are replaced with a single transition to S_A, labeled $C_3 + C_5$.

If a state is the target of multiple transitions across the boundary, a single transition is added from the idle state to this state. The transition is labeled with the OR of the conditions associated with the individual transitions in the original state machine. This case is illustrated by the transitions from S_2 and S_3 to S_5. These are replaced by a single transition from S_B to S_5, labeled $C_3 \bullet S_2 + C_4 \bullet S_3$.

When all these rules have been applied, the final rule describes the self-loop ("hold") condition for the idle states. Simply form the OR of all of the exit conditions and invert it. This is shown in Figure 9.47(d). Consider the idle state S_A. Its only exit condition is $C_2 \bullet S_6$. So its hold condition is the inverse of this, namely $\overline{C_2 \bullet S_6}$.

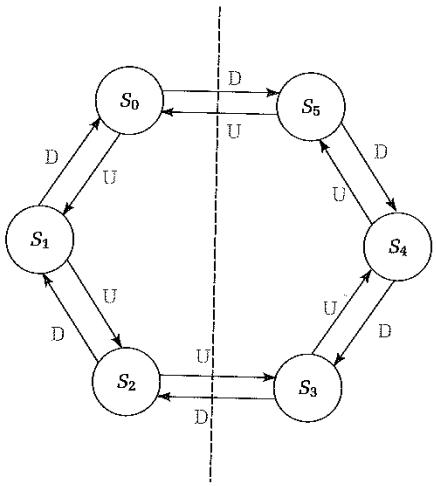
Example 3 FSM Partitioning Consider the six-state finite state machine of Figure 9.48(a). The machine implements a simple up/down counter. When the input U is asserted, the machine counts up. When D is asserted, it counts down. Otherwise the machine stays in its current state.

The goal is to partition the machine into two communicating four-state finite state machines. We might need to do this because the underlying logic primitives provide support for two flip-flops within the logic block, as in the Xilinx CLB to be introduced in the next chapter.

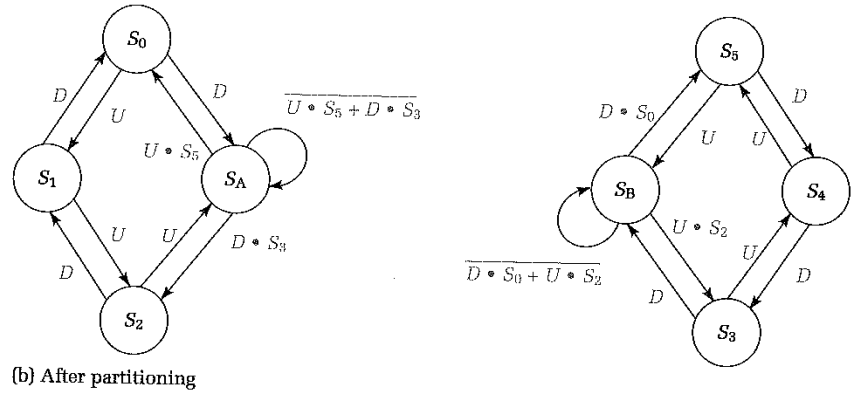
Figure 9.48(b) shows the result of the partitioning. States S_0, S_1, and S_2 form the core of one machine and S_3, S_4, and S_6 form the other. We also introduce the two idle states, S_A and S_B.

The machine at the left enters its idle state S_A when it is in S_0 and D is asserted or when it is in S_2 and U is asserted. It exits the idle state when the machine at the right is in S_5 with U asserted or in S_3 with D asserted. Otherwise it stays in its idle state. The machine at the right works similarly.

To see how the machines communicate, let's consider an up-count sequence from S_0 to S_5 and back to S_0. On reset, the machine on the left



(a) Before partitioning



(b) After partitioning

Figure 9.48 Partitioning example.

enters S_0 while the machine on the right enters S_B. With U asserted, the left machine advances from S_0 to S_1 to S_2 to S_A. It will idle in this state until the right machine is ready to exit S_5.

Meanwhile, the right machine holds in S_B until the left machine enters S_2. At the same time that the left machine changes to S_A, the right one exits S_B to S_3. On subsequent clock transitions, it advances from S_3 to S_4 to S_5 to S_B, where it holds. When the right machine changes from S_5 to S_B, the left machine exits S_A to S_0, and the process repeats itself. Down-count sequences work in an analogous way.

Chapter Review

This chapter has concentrated on the optimization of finite state machines. We have emphasized the methods for state reduction, state assignment, choice of flip-flops, and state machine partitioning. For state reduction, we introduced the row matching and implication chart methods. These can be used to identify and eliminate redundant states, thus reducing the number of flip-flops needed to implement a particular finite state machine.

We then examined heuristic methods for state assignment, aimed at reducing the number of product terms or literals needed to implement the next-state and output functions. Since paper-and-pencil methods are not particularly effective, we introduced computer-aided design tools for state assignment that do a much better job in a fraction of the time: *nova*, *mustang*, and *jedi*.

The latter part of the chapter focused on choosing flip-flops for implementing the state registers of the finite state machine. *J-K* flip-flops tend to be most effective in reducing the logic, but they require logical remapping of the next-state functions and more wires than the simpler *D* flip-flops.

Finally, we discussed state machine partitioning methods, in particular partitioning based on inputs and outputs and partitioning by introducing idle states. These techniques are needed when we cannot implement a finite state machine with a single programmable logic component.

In the next chapter, we will examine implementation strategies in more detail. In particular, we will look at the methods for implementing finite state machines based on structured logic methods, such as ROM, programmable logic, and approaches based on MSI components.

Further Reading

The traffic light controller example used extensively in this chapter is borrowed from the famous text by C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1979. C. Roth's book, *Fundamentals of Logic Design*, West Publishing, St. Paul, MN, 1985, has an extensive discussion of state assignment guidelines that formed the basis of our Section 9.3.2. *Modern Logic Design* by D. Green, Addison-Wesley, 1986, has a highly readable, short, direct description of state assignment (pp. 40–43).

Nova's approach to state assignment is described in T. Villa and A. Sangiovanni-Vincentelli's paper "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations," given at the 26th Design Automation Conference, Miami, FL (June 1989). A revised and expanded version of the paper appeared in *IEEE Transactions on Computer-Aided Design* in September 1990 (vol. 9, no. 9, pp. 1326–1334). *Mustang*'s method is described in "MUSTANG: State Assignment of Finite State Machines Targeting Multi-level Logic Implementations," by S. Devadas, B. Ma,

R. Newton, and A. Sangiovanni-Vincentelli, in *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 12 (December 1988). *Jedi*'s method for symbolic assignment is described by Lin and Newton in "Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages," which appeared in the *Proceedings of the VLSI'89 Conference*, Munich, West Germany, in August 1989.

These tools (along with *espresso* and *misII*) are available for a very modest charge from the Industrial Liaison Program Office of the Electrical Engineering and Computer Science Department, University of California, Berkeley. Detailed descriptions of how to invoke the tools, as well as examples of their use, can be found in the most current OCTTOOLS Manual distributed by that office.

Finite state machine partitioning is a topic that waxes and wanes in importance. The original work was done in the late 1950s, became less interesting during the era of VLSI, and is becoming more important again with pervasive use of programmable logic in digital designs. The topic is not well covered by most of today's textbooks. One exception is M. Bolton's book, *Digital System Design with Programmable Logic*, Addison-Wesley, Wokingham, England, 1990, which offers a section on the topic. The partitioning rules introduced in Section 9.5.1 were obtained from an applications note in the *Altera Applications Handbook*, Altera Corporation, Santa Clara, CA, 1988.

Exercises

9.1 *(State Reduction)* Use the implication chart method to reduce the 4-bit string recognizer state diagram of Figure 9.2.

9.2 *(State Reduction)* Given the state diagram in Figure Ex9.2, obtain an equivalent reduced state diagram containing a minimum number of states. You may use row matching or implication charts. Put your final answer in the form of a state diagram rather than a state table. Make it clear which states have been combined.

9.3 *(State Reduction)* Given the state diagram in Figure Ex9.3, determine which states should be combined to determine the reduced state diagram. You may use row matching or implication charts.

9.4 *(State Reduction)* Given the state diagram in Figure Ex9.4, draw the fully reduced state diagram. State succinctly what strings cause the recognizer to output a 1.

9.5 *(State Reduction)* Starting with the state diagram of Figure Ex9.5, use the implication chart method to find the minimum state diagram. Which of the original states are combined?