

Computer Systems Organization & Architecture

JOHN D. CARPINELLI

New Jersey Institute of Technology

Copyright © 2001 by Pearson Education, Inc.
This edition is published by arrangement with Pearson Education, Inc. and Dorling Kindersley
Publishing, Inc.

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold,
distributed, or otherwise circulated without the publisher's prior written consent in any form of binding or
cover other than that in which it is published and without a similar condition including this condition being
imposed on the subsequent purchaser and without limiting the rights under copyright reserved above, no
part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in
any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior
written permission of both the copyright owner and the above-mentioned publisher of this book.

ISBN 978-81-7728-767-2

First Impression, 2000
Second Impression, 2007
Third Impression, 2008
Fourth Impression, 2009

This edition is manufactured in India and is authorized for sale only in India, Bangladesh, Pakistan,
Pakistan, Nepal, Sri Lanka and the Maldives. Circulation of this edition outside of these territories is
UNAUTHORIZED.

Published by Dorling Kindersley (India) Pvt. Ltd., licensee of Pearson Education in South Asia.

Head Office: 482, F.I.E., Patparganj, Delhi
Registered Office: 14 Local Shopping Centre, New Delhi 110 017, India

PEARSON
Education

Printed in India by Special Print-N-Print

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Copyright © 2001 by Pearson Education, Inc.

This edition is published by arrangement with Pearson Education, Inc. and Dorling Kindersley Publishing, Inc.

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of both the copyright owner and the above-mentioned publisher of this book.

ISBN 978-81-7758-767-8

First Impression, 2006

Second Impression, 2007

Third Impression, 2008

Fourth Impression, 2009

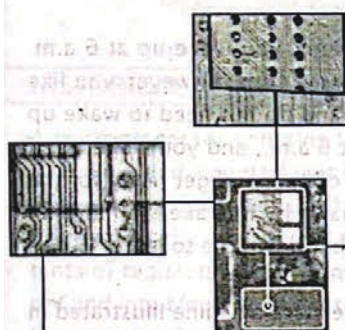
This edition is manufactured in India and is authorized for sale only in India, Bangladesh, Bhutan, Pakistan, Nepal, Sri Lanka and the Maldives. Circulation of this edition outside of these territories is UNAUTHORIZED.

Published by Dorling Kindersley (India) Pvt. Ltd., licensees of Pearson Education in South Asia.

Head Office: 482, F.I.E., Patparganj, Delhi 110 092, India.

Registered Office: 14 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India.

Printed in India by Sheel Print-N-Pack.



2

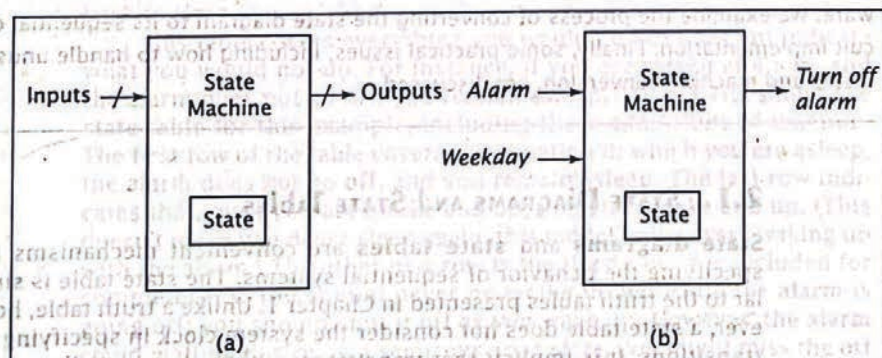
INTRODUCTION TO FINITE STATE MACHINES

The combinational circuits described in Chapter 1 can be designed in a relatively straightforward manner. We create functions and then minimize them using Karnaugh maps or some other method, such as a logic design computer program. We then implement the minimized function in hardware. It is not so easy, however, to design sequential circuits. A **finite state machine** is a tool to model the desired behavior of a sequential system. The designer first develops a finite state model of the system behavior and then designs a circuit to implement this model.

A finite state machine consists of several **states**. **Inputs** to the machine are combined with the current state of the machine to determine the new state, or **next state** of the machine. Depending on the type of machine, **outputs** are generated based on either the state of the machine, or the state of the machine and the machine inputs. This is illustrated in Figure 2.1(a).

Figure 2.1

State machine models: (a) generic and (b) alarm clock system



Consider a greatly simplified example. As a student, you wake up at 6 a.m. every weekday so you won't miss your first class. On weekends, however, you like to catch up on the sleep you lost while studying late, and do not need to wake up that early. You are asleep, your alarm clock goes off at 6 a.m., and you wake up. If this is a weekday, you turn off the alarm clock, get out of bed, and get ready for the day ahead. However, if it is a weekend, you set the alarm by mistake. You angrily turn off the alarm clock (throw it across the room, flush it down the toilet, or otherwise disable it), and go back to sleep.

We can model this sequence of events by the finite state machine illustrated in Figure 2.1(b). The state machine is actually you. You can be in one of three states: asleep, awake but still in bed, or awake and up. You receive two inputs: the alarm, which wakes you up, and whether or not the current day is a weekday, which determines how you react to the alarm. In this example, the only output is to turn off the alarm clock. (This assumes that you don't really destroy the clock; if you do, then this example would also need a "destroy alarm clock" output.)

This is the first step in going from a description of the behavior of a system to its implementation as a sequential circuit. Next, this behavior would be encoded in a state table and diagram, then designed and implemented as a sequential digital circuit.

Although finite state machines can be used for very simple digital circuits, such as one for this example, they are especially useful for designing more complex systems. In computer architecture, they are well suited to the task of designing microprocessors. (See the following Historical Perspective for more information.) Chapters 6 and 7 apply the finite state machine approach to the design of microprocessor control units.

This chapter is organized as follows. First, we introduce the basics of finite state machines, with state diagrams and state tables. We show how to develop these from the specification of system behavior. Next, the two main types of finite state machines, Mealy and Moore machines, are described. We look at their specifications as state tables and diagrams, as well as typical implementations in hardware. We examine the process of converting the state diagram to its sequential circuit implementation. Finally, some practical issues, including how to handle unused states and machine conversion, are discussed.

2.1 STATE DIAGRAMS AND STATE TABLES

State diagrams and **state tables** are convenient mechanisms for specifying the behavior of sequential systems. The state table is similar to the truth tables presented in Chapter 1. Unlike a truth table, however, a state table does not consider the system clock in specifying its transitions. It is implicit that transitions occur only when allowed by

HISTORICAL PERSPECTIVE: FINITE STATE MACHINES AND MICROPROCESSORS

A microprocessor contains three main sections: a register section; an arithmetic/logic unit; and a control unit (see Chapter 4). The control unit inputs the instructions to be executed, as well as other information, such as the values in flag registers. (See Chapter 3.) The control unit outputs the control signals to load and modify the contents of registers, to perform arithmetic and logical functions, and to access memory and input/output devices. It outputs these signals in the proper sequence so that the microprocessor fetches, decodes, and executes each instruction correctly.

The control unit of a microprocessor is, in essence, a finite state machine. The instruction and flag values are the machine inputs, and the control signals are the machine outputs. Each state represents one step in the process of fetching or executing an instruction. By going through the states in the correct sequence, the control unit outputs the control signals in the order needed to process each instruction.

the clock, typically on the leading edge of the clock. In this chapter, all circuits change their states and outputs only on the leading edge of the clock.

To illustrate state tables, consider the alarm clock example at the beginning of this chapter. A state table is set up as shown in Table 2.1(a). The present state and all inputs occupy the left half of the table. The next state and outputs to be generated by each combination of the present state and inputs are listed on the right half of the table.

Table 2.1(b) shows the state table for this example. The state table has two inputs, *Alarm* and *Weekday*, and one output, *Turn off alarm*. When you are asleep and the alarm is on, you go from being asleep to being awake in bed. You also turn off the alarm. These are represented by the first row of Table 2.1(b). The other two rows encode your actions after turning off the alarm; either you get up or go back to sleep.

This table covers everything you would do but does not indicate what you would not do. For instance, if you are asleep at 4 a.m. and the alarm does not go off, you remain asleep. Table 2.1(c) shows the state table for this example, including these conditions of inaction. The first row of the table covers the situation in which you are asleep, the alarm does not go off, and you remain asleep. The last row indicates that, once you are awake and up, you stay awake and up. (This doesn't mean you never sleep again; this model only covers waking up with the alarm.) The other new row is the third row; it is included for completeness. You should never be awake in bed while the alarm is going off; you should shut it off as you wake up. However, the alarm could malfunction or, in a semiconscious state, you might miss the off switch. This extra line in the table accounts for these conditions. With

Table 2.1

State tables: (a) generic, (b) for the alarm clock problem, and (c) for the alarm clock problem with inaction states

Present state	Inputs	Next state	Outputs	Present state	Alarm	Weekday	Next state	Turn off alarm
				Asleep	On	X	Awake in bed	Yes
				Awake in bed	Off	Yes	Awake and up	No
				Awake in bed	Off	No	Asleep	No

(a) (b)

Present State	Alarm	Weekday	Next state	Turn off alarm
Asleep	Off	X	Asleep	No
Asleep	On	X	Awake in bed	Yes
Awake in bed	On	X	Awake in bed	Yes
Awake in bed	Off	Yes	Awake and up	No
Awake in bed	Off	No	Asleep	No
Awake and up	X	X	Awake and up	No

(c)

this line included, the table specifies actions for every possible set of input values for every state.

A state table may be represented graphically by a state diagram. To derive this state diagram, each state is represented by a circled **vertex**, and each row of the state table is shown as a directed arc from the present state vertex to the next state vertex. Figure 2.2 shows the state diagram for the alarm clock example twice, each with a different representation of the outputs. In the first state diagram, the outputs are associated with the states. In Figure 2.2(b), the outputs are associated with the arcs; an output of 1 signifies that "Turn off alarm" is Yes. By convention, inputs which are don't care and inactive outputs are not shown.

As another example, consider the state table for the JK flip-flop shown in Table 2.2(a). The flip-flop has two states, Y and Z, and two inputs, J and K. Each row of the table gives a present state, values of the J and K inputs, the next state to occur under these conditions, and the output value, Q. For example, when the flip-flop is in state Z, and the inputs are J = 1 and K = 1, the next state is state Y.

State transitions may be combined to present a more compact state table. The state table shown in Table 2.2(b) is an equivalent representation of the state table of Table 2.2(a). The first row, with Present State = Y, J = 0, K = X (don't care) and Next State = Y, is a

Figure 2.2

State diagrams for the alarm clock example

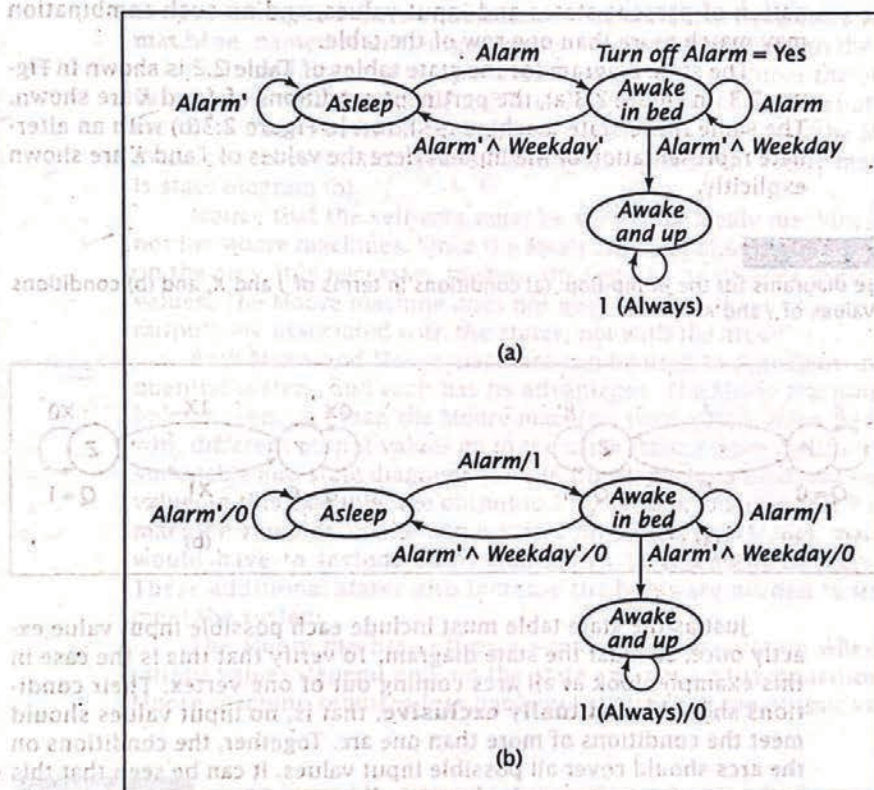


Table 2.2

State tables for the JK flip-flop

Present State	J	K	Next State	Q	Present State	J	K	Next State	Q
Y	0	0	Y	0	Y	0	X	Y	0
Y	0	1	Y	0	Y	1	X	Z	1
Y	1	0	Z	1	Z	X	0	Z	1
Y	1	1	Z	1	Z	X	1	Y	0
Z	0	0	Z	1					
Z	0	1	Y	0					
Z	1	0	Z	1					
Z	1	1	Y	0					

(a)

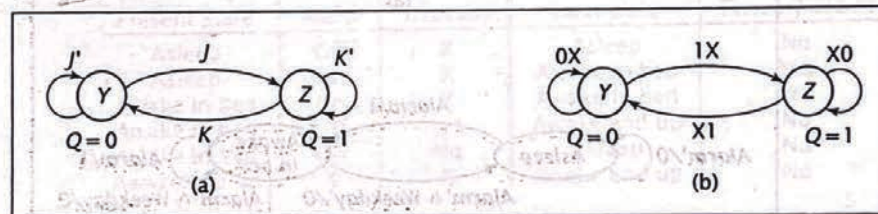
(b)

combination of the first two rows of the previous table. The other rows are derived in a similar manner. Regardless of which representation is used, a complete state table must include each possible combination of present states and input values, and no such combination may match more than one row of the table.

The state diagram for the state tables of Table 2.2 is shown in Figure 2.3. In Figure 2.3(a), the pertinent conditions of J and K are shown. The same finite state machine is shown in Figure 2.3(b) with an alternate representation of the inputs. Here the values of J and K are shown explicitly.

FIGURE 2.3

State diagrams for the JK flip-flop: (a) conditions in terms of J and K , and (b) conditions as values of J and K



Just as the state table must include each possible input value exactly once, so must the state diagram. To verify that this is the case in this example, look at all arcs coming out of one vertex. Their conditions should be **mutually exclusive**, that is, no input values should meet the conditions of more than one arc. Together, the conditions on the arcs should cover all possible input values. It can be seen that this is the case for each state in the state diagram of Figure 2.3.

By convention, **self-arcs** are sometimes not shown. It is implicit that all conditions not covered by an arc cause the machine to remain in the same state. In Figure 2.3, the arcs that go from state Y back to itself, and from state Z back to itself, could be removed. (As will be seen in the next section, this can only be done for certain types of state diagrams.)

2.2 MEALY AND MOORE MACHINES

The whole purpose of a finite state machine, or just about any digital circuit, is to generate output data. This data can be the result of computations, or signals to control other circuitry. A system must generate outputs or it can serve no useful purpose.

A finite state machine can represent outputs in one of two ways. A **Moore machine**, named after Edward Moore, associates its outputs with the states. The outputs are represented either within the vertex corresponding to a state or adjacent to the vertex. In contrast, a **Mealy machine**, named after George Mealy, associates outputs with the transitions. In addition to the input values, each arc also shows the output values generated during the transition. The format of the label of each arc is INPUTS/OUTPUTS. In Figure 2.2, state diagram (a) is the Moore machine representation of the alarm clock system; the Mealy machine is state diagram (b).

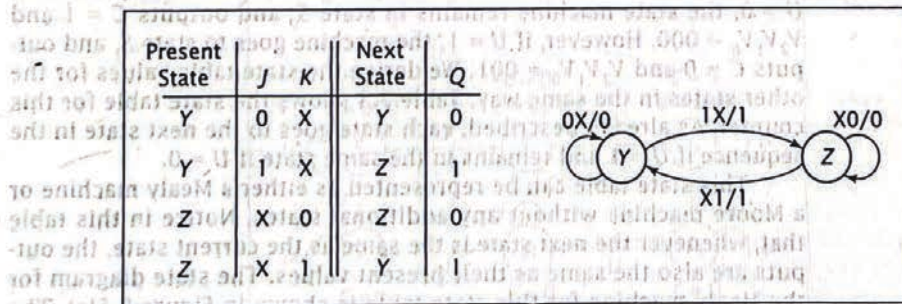
Notice that the self-arcs must be shown for Mealy machines, but not for Moore machines. Since the Mealy machine shows output values on the arcs, it is necessary to show the self-arcs to provide the output values. The Moore machine does not need to show these arcs since its outputs are associated with the states, not with the arcs.

Both Mealy and Moore machines can be used to represent any sequential system, and each has its advantages. The Mealy machine can be more compact than the Moore machine, particularly when two arcs with different output values go to the same state. Figure 2.4 shows the state table and state diagrams for a JK flip-flop with a modified output value. In this example, the output is 1 if the state changes and 0 if the machine remains in the same state. An equivalent Moore machine would have to include additional states to represent this system. These additional states also increase the hardware needed to implement the system.

The Moore machine offers a simpler implementation when the output values depend only on the state and not on the transitions. A Moore machine requires less hardware to produce the output values

Figure 2.4

JK flip-flop with modified outputs: (a) state table and (b) state diagram



than does a Mealy machine, since its outputs depend only on its state; the outputs of a Mealy machine depend on both its state and its input values. The Moore machine is well suited for representing the control units of microprocessors and central processing units. This is examined more closely in Chapters 6 and 7.

2.3 DESIGNING STATE DIAGRAMS

To illustrate the process of developing state tables and state diagrams from system specifications, this section presents three examples of varying complexity: a counter, a string checker, and a toll booth controller.

2.3.1 Modulo 6 COUNTER

A modulo 6 counter is a 3-bit counter that counts through the sequence

000 → 001 → 010 → 011 → 100 → 101 → 000 ... (0 → 1 → 2 → 3 → 4 → 5 → 0 → ...)

Unlike a regular 3-bit counter, it does not use the values 110 (6) nor 111 (7). Its input U controls the counter. When $U = 1$, the counter increments its value on the rising edge of the clock. When $U = 0$, it retains its current value regardless of the value of the clock. The value of the count is represented as the 3-bit output $V_2V_1V_0$. There is an additional output, C , which is 1 when going from 5 to 0, and 0 otherwise. (To simplify this example, the C output remains at 1 until the counter goes from 0 to 1.)

The finite state machine for this counter must have six states, arbitrarily labeled S_0, S_1, S_2, S_3, S_4 , and S_5 . State S_i corresponds to counter output i ; the states follow the sequence

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_0 \rightarrow \dots$

To derive the state table for this counter, look at each state individually; for each state, examine what happens for all possible values of the inputs. For example, in state S_0 , input U can be either 0 or 1. If $U = 0$, the state machine remains in state S_0 and outputs $C = 1$ and $V_2V_1V_0 = 000$. However, if $U = 1$, the machine goes to state S_1 and outputs $C = 0$ and $V_2V_1V_0 = 001$. We derive the state table values for the other states in the same way. Table 2.3 shows the state table for this counter. As already described, each state goes to the next state in the sequence if $U = 1$ and remains in the same state if $U = 0$.

This state table can be represented as either a Mealy machine or a Moore machine without any additional states. Notice in this table that, whenever the next state is the same as the current state, the outputs are also the same as their present values. The state diagram for the Mealy machine for this state table is shown in Figure 2.5(a). The outputs are ordered C, V_2, V_1, V_0 . Figure 2.5(b) shows the Moore machine state diagram for this system. Note that each arc in these state diagrams corresponds to one row of the state table.

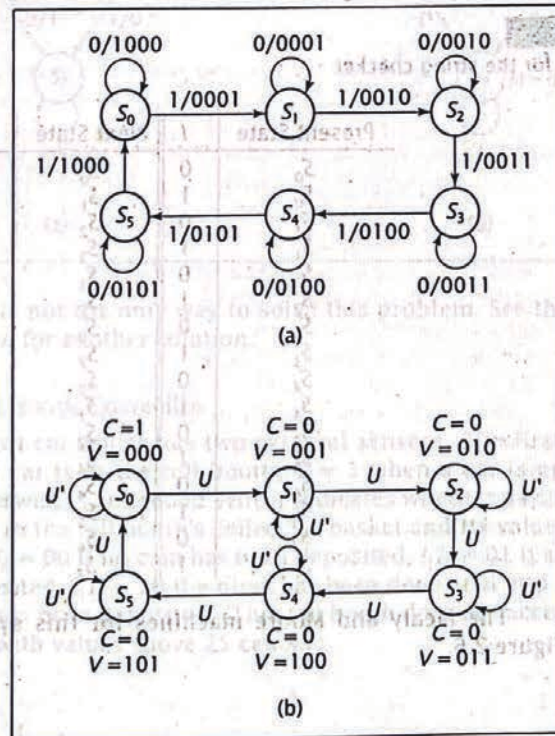
Table 2.3

State table for the modulo 6 counter

Present State	U	Next State	C	$V_2V_1V_0$
S_0	0	S_0	1	000
S_0	1	S_1	0	001
S_1	0	S_1	0	001
S_1	1	S_2	0	010
S_2	0	S_2	0	010
S_2	1	S_3	0	011
S_3	0	S_3	0	011
S_3	1	S_4	0	100
S_4	0	S_4	0	100
S_4	1	S_5	0	101
S_5	0	S_5	0	101
S_5	1	S_0	1	000

Figure 2.5

State diagrams for the modulo 6 counter: (a) Mealy and (b) Moore



2.3.2 String Checker

A string checker inputs a string of bits, one bit per clock cycle. When the previous three bits form the pattern 110, it sets output $M = 1$; otherwise $M = 0$. The pattern is checked continuously throughout the entire bit stream. The system does not check the first three bits, then the next three, and so on. Rather, it checks bits 1, 2, and 3, then 2, 3, and 4, then 3, 4, and 5, and so on forever.

One way to design this system is to have the last three bits received represent the state of the system. This would result in eight states, S_0 to S_7 . State S_i is active when the last three bits form the binary equivalent of i . For example, if the last three bits read in are 011, then S_3 is active. The last bit read in is the least significant bit. When it is read in, the other bits are shifted left in the binary representation of i ; the most significant bit is shifted out and lost.

As shown in the state table of Table 2.4, each state goes to one of two possible next states, depending on whether input bit I is 0 or 1. For example, state S_2 corresponds to the condition in which the last three input bits are 010. When bit I is input, these bits are shifted left. The leftmost zero is lost and the new binary value is 10 I , which corresponds to state S_4 (100) if $I = 0$ or S_5 (101) if $I = 1$.

Table 2.4

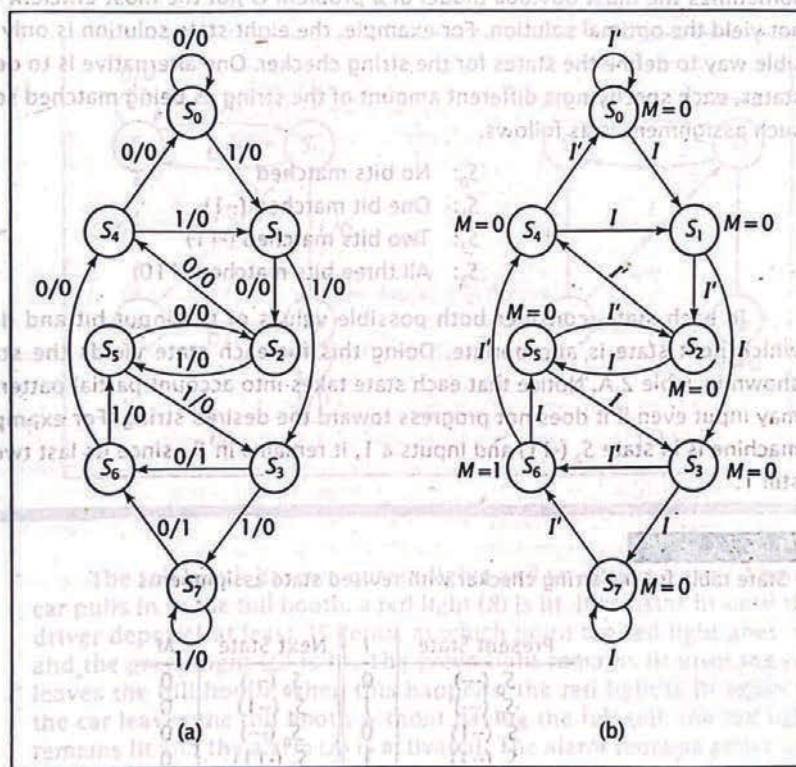
State table for the string checker

Present State	I	Next State	M
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_2	0
S_1	1	S_3	0
S_2	0	S_4	0
S_2	1	S_5	0
S_3	0	S_6	1
S_3	1	S_7	0
S_4	0	S_0	0
S_4	1	S_1	0
S_5	0	S_2	0
S_5	1	S_3	0
S_6	0	S_0	0
S_6	1	S_1	0
S_7	0	S_4	1
S_7	1	S_5	0

The Mealy and Moore machines for this system are shown in Figure 2.6.

Figure 2.6

State diagrams for the string checker: (a) Mealy and (b) Moore



This is not the only way to solve this problem. See the Practical Perspective for another solution.

2.3.3 Toll Booth Controller

A toll booth controller has two external sensors. The first indicates whether a car is at the toll booth; $C = 1$ when a car is present and $C = 0$ otherwise. The second sensor indicates whether a coin has been deposited in the toll booth's collection basket and its value. This sensor sets $I_1I_0 = 00$ if no coin has been deposited, $I_1I_0 = 01$ if a nickel has been deposited, $I_1I_0 = 10$ if a dime has been deposited and $I_1I_0 = 11$ if a quarter has been deposited. (This toll booth does not accept pennies nor coins with values above 25 cents.)

PRACTICAL PERSPECTIVE: DIFFERENT MODELS FOR THE SAME PROBLEM

Sometimes the most obvious model of a problem is not the most efficient and does not yield the optimal solution. For example, the eight-state solution is only one possible way to define the states for the string checker. One alternative is to define four states, each specifying a different amount of the string as being matched so far. One such assignment is as follows.

- S_0 : No bits matched
- S_1 : One bit matched (--1)
- S_2 : Two bits matched (-11)
- S_3 : All three bits matched (110)

In each state, consider both possible values of the input bit and determine which next state is appropriate. Doing this for each state yields the state table shown in Table 2.A. Notice that each state takes into account partial patterns that it may input even if it does not progress toward the desired string. For example, if the machine is in state S_2 (-11) and inputs a 1, it remains in S_2 , since its last two bits are still 1.

Table 2.A.

State table for the string checker with revised state assignments

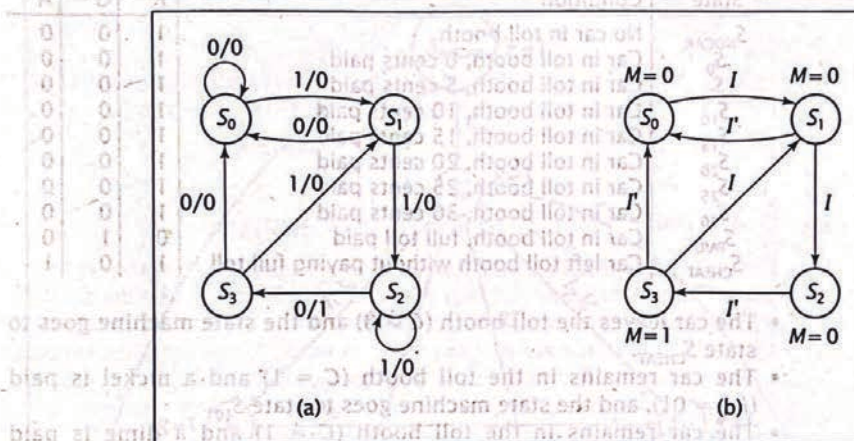
Present State	I	Next State	M
S_0 (---)	0	S_0 (---)	0
S_0 (---)	1	S_1 (--1)	0
S_1 (--1)	0	S_0 (---)	0
S_1 (--1)	1	S_2 (-11)	0
S_2 (-11)	0	S_3 (110)	1
S_2 (-11)	1	S_2 (-11)	0
S_3 (110)	0	S_0 (---)	0
S_3 (110)	1	S_1 (--1)	0

Figure 2.A shows the Mealy and Moore machines for this state table. As with the state tables, these machines are much simpler than those in the original design because they have only half the number of states. These state diagrams would require less complex hardware designs than the original state diagrams in Figure 2.6. Most important, they generate the same outputs for the same sequence of inputs as in the original design, thus meeting the system specification.

(continued)

FIGURE 2.A

State diagrams for the string checker with revised state assignments: (a) Mealy and (b) Moore



The toll booth has two output lights and an output alarm. When a car pulls in to the toll booth, a red light (R) is lit. It remains lit until the driver deposits at least 35 cents, at which point the red light goes off and the green light (G) is lit. The green light remains lit until the car leaves the toll booth; when this happens, the red light is lit again. If the car leaves the toll booth without paying the full toll, the red light remains lit and the alarm (A) is activated. The alarm remains active until another car enters the toll booth, when it is turned off.

This system requires several states, corresponding to the listed conditions. In addition, it requires several more states to keep track of how much money has been deposited. Table 2.5 shows the states that are sufficient to specify the finite state machine, along with their outputs.

To develop the state table, we again look at each individual state and determine all possible actions that can occur. In state S_{NOCAR} , for example, there is no car in the toll booth. The only two things that can happen are (1) a car enters the toll booth ($C = 1$) and the state machine goes to state S_0 , or (2) no car arrives ($C = 0$) and the state machine remains in state S_{NOCAR} . Other states are more complicated. For example, during state S_5 (car in toll booth, 5 cents paid), any of the following can occur.

Table 2.5

States for the toll booth controller

State	Condition	R	G	A
S_{NOCAR}	No car in toll booth	1	0	0
S_0	Car in toll booth, 0 cents paid	1	0	0
S_5	Car in toll booth, 5 cents paid	1	0	0
S_{10}	Car in toll booth, 10 cents paid	1	0	0
S_{15}	Car in toll booth, 15 cents paid	1	0	0
S_{20}	Car in toll booth, 20 cents paid	1	0	0
S_{25}	Car in toll booth, 25 cents paid	1	0	0
S_{30}	Car in toll booth, 30 cents paid	1	0	0
S_{PAID}	Car in toll booth, full toll paid	0	1	0
S_{CHEAT}	Car left toll booth without paying full toll	1	0	1

- The car leaves the toll booth ($C = 0$) and the state machine goes to state S_{CHEAT} .
- The car remains in the toll booth ($C = 1$) and a nickel is paid ($I_1I_0 = 01$), and the state machine goes to state S_{10} .
- The car remains in the toll booth ($C = 1$) and a dime is paid ($I_1I_0 = 10$), and the state machine goes to state S_{15} .
- The car remains in the toll booth ($C = 1$) and a quarter is paid ($I_1I_0 = 11$), and the state machine goes to state S_{30} .
- The car remains in the toll booth ($C = 1$) and nothing new is paid ($I_1I_0 = 00$), and the state machine remains in state S_5 .

Table 2.6

State table for the toll booth controller

Present State	C	I_1I_0	Next State	R	G	A	Present State	C	I_1I_0	Next State	R	G	A
S_{NOCAR}	1	XX	S_0	1	0	0	S_{15}	0	XX	S_{CHEAT}	1	0	1
S_{PAID}	0	XX	S_{NOCAR}	1	0	0	S_{15}	1	01	S_{20}	1	0	0
S_{CHEAT}	1	XX	S_0	1	0	0	S_{15}	1	10	S_{25}	1	0	0
S_0	0	XX	S_{CHEAT}	1	0	1	S_{15}	1	11	S_{PAID}	0	1	0
S_0	1	01	S_5	1	0	0	S_{15}	0	XX	S_{CHEAT}	1	0	1
S_0	1	10	S_{10}	1	0	0	S_{20}	1	01	S_{25}	1	0	0
S_0	1	11	S_{25}	1	0	0	S_{20}	1	10	S_{30}	1	0	0
S_5	0	XX	S_{CHEAT}	1	0	1	S_{20}	1	11	S_{PAID}	0	1	0
S_5	1	01	S_{10}	1	0	0	S_{25}	0	XX	S_{CHEAT}	1	0	1
S_5	1	10	S_{15}	1	0	0	S_{25}	1	01	S_{30}	1	0	0
S_5	1	11	S_{30}	1	0	0	S_{25}	1	10	S_{PAID}	0	1	0
S_{10}	0	XX	S_{CHEAT}	1	0	1	S_{25}	1	11	S_{PAID}	0	1	0
S_{10}	1	01	S_{15}	1	0	0	S_{30}	0	XX	S_{CHEAT}	1	0	1
S_{10}	1	10	S_{20}	1	0	0	S_{30}	1	01	S_{PAID}	0	1	0
S_{10}	1	11	S_{25}	1	0	0	S_{30}	1	10	S_{PAID}	0	1	0
			S_{PAID}	0	1	0	S_{30}	1	11	S_{PAID}	0	1	0

Figure 2.7

State diagram for the Moore machine for the toll booth controller

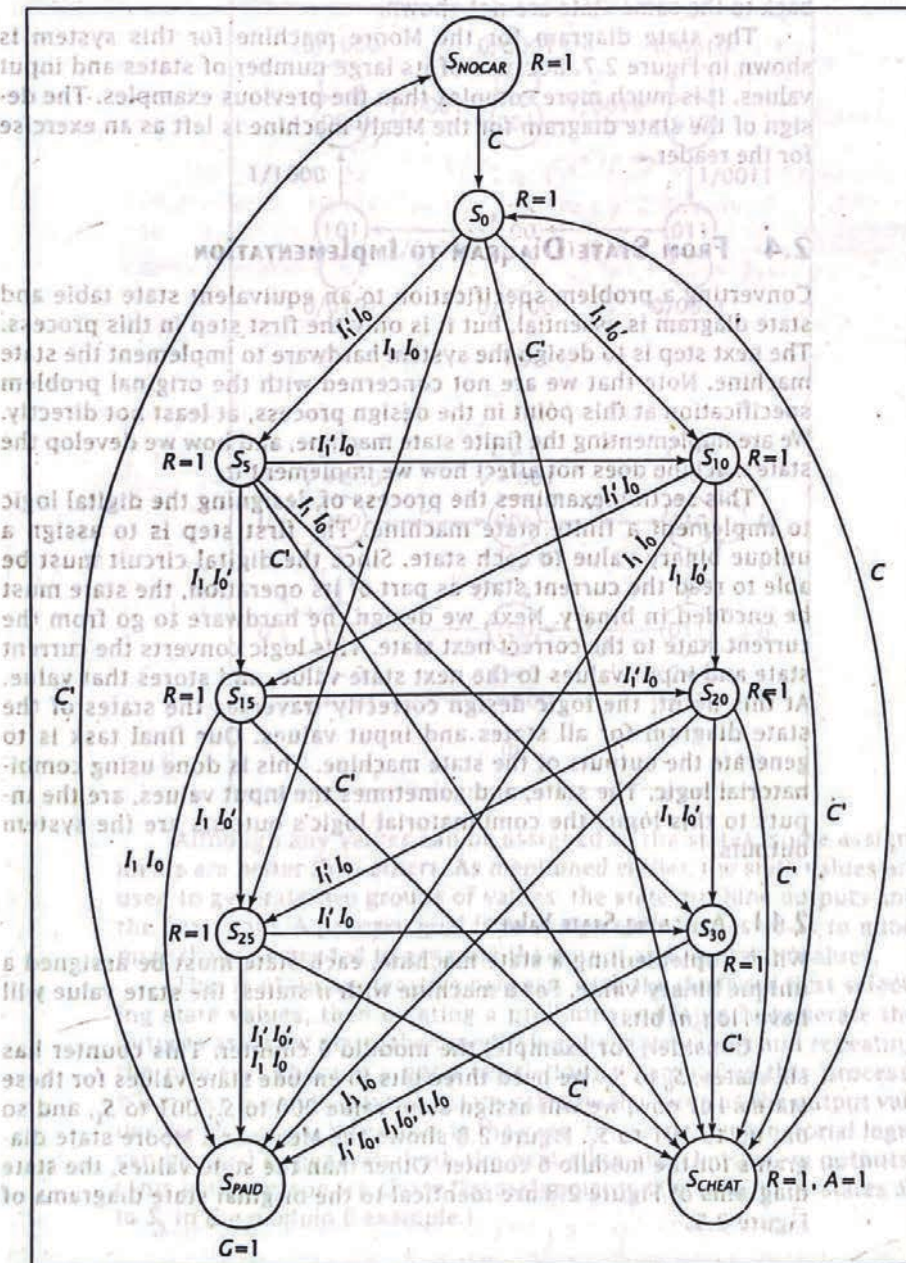


Table 2.6 summarizes the transitions that can occur between states. Impossible conditions, such as having money deposited while no car is in the toll booth, are not considered here. Also, transitions that lead back to the same state are not shown.

The state diagram for the Moore machine for this system is shown in Figure 2.7. Because of its large number of states and input values, it is much more complex than the previous examples. The design of the state diagram for the Mealy machine is left as an exercise for the reader.

2.4 FROM STATE DIAGRAM TO IMPLEMENTATION

Converting a problem specification to an equivalent state table and state diagram is essential, but it is only the first step in this process. The next step is to design the system hardware to implement the state machine. Note that we are not concerned with the original problem specification at this point in the design process, at least not directly. We are implementing the finite state machine, and how we develop the state machine does not affect how we implement it.

This section examines the process of designing the digital logic to implement a finite state machine. The first step is to assign a unique binary value to each state. Since the digital circuit must be able to read the current state as part of its operation, the state must be encoded in binary. Next, we design the hardware to go from the current state to the correct next state. This logic converts the current state and input values to the next state value, and stores that value. At this point, the logic design correctly traverses the states of the state diagram for all states and input values. Our final task is to generate the outputs of the state machine. This is done using combinatorial logic. The state, and sometimes the input values, are the inputs to this logic; the combinatorial logic's outputs are the system outputs.

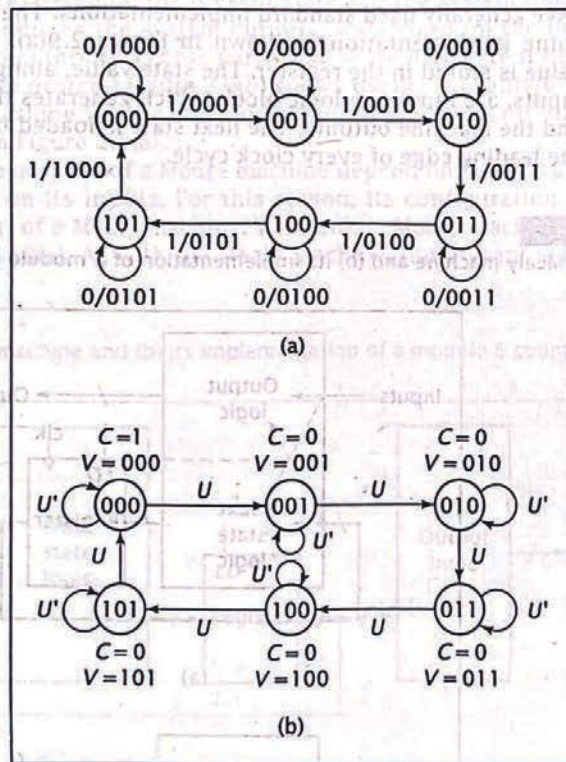
2.4.1 ASSIGNING STATE VALUES

When implementing a state machine, each state must be assigned a unique binary value. For a machine with n states, the state value will have $\lceil \log_2 n \rceil$ bits.

Consider, for example, the modulo 6 counter. This counter has six states, S_0 to S_5 . We need three bits to encode state values for these states. For now, we will assign state value 000 to S_0 , 001 to S_1 , and so on, up to 101 to S_5 . Figure 2.8 shows the Mealy and Moore state diagrams for the modulo 6 counter. Other than the state values, the state diagrams of Figure 2.8 are identical to the original state diagrams of Figure 2.5.

FIGURE 2.8

State diagrams for the modulo 6 counter with values shown: (a) Mealy and (b) Moore



Although any values can be assigned to the states, some assignments are better than others. As mentioned earlier, the state values are used to generate two groups of values: the state machine outputs and the next state. A primary goal is to assign state values so as to minimize the logic needed to generate the output and next state values.

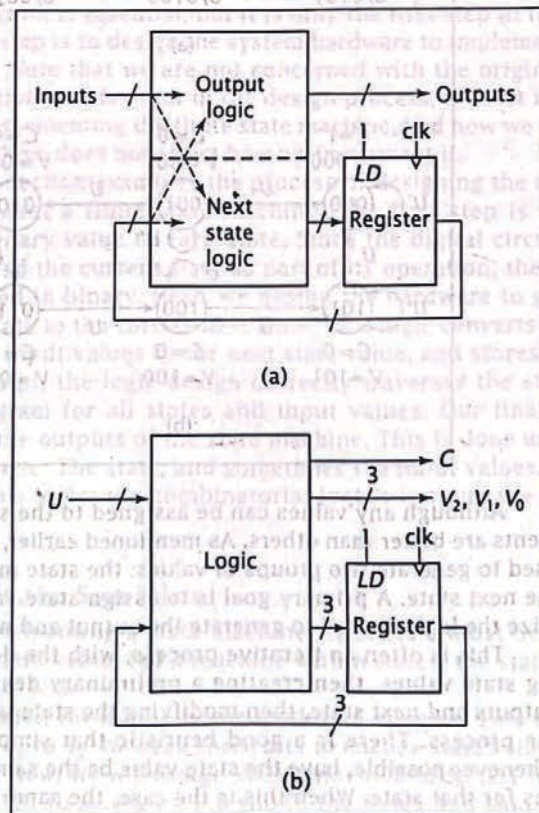
This is often an iterative process, with the designer first selecting state values, then creating a preliminary design to generate the outputs and next state, then modifying the state values and repeating the process. There is a good heuristic that simplifies this process. Whenever possible, have the state value be the same as the output values for that state. When this is the case, the same combinatorial logic can be used to generate both the next state and the system outputs. (This is the reason we chose the assignments 000 to 101 for states S_0 to S_5 in the modulo 6 example.)

2.4.2 MEALY AND MOORE MACHINE IMPLEMENTATIONS

Once the state values have been defined, we are ready to design the digital logic to implement the system. Mealy and Moore machines each have generally used standard implementations. The generic Mealy machine implementation is shown in Figure 2.9(a). The current state value is stored in the register. The state value, along with the machine inputs, are input to a logic block, which generates the next state value and the machine outputs. The next state is loaded into the register on the leading edge of every clock cycle.

FIGURE 2.9

(a) Generic Mealy machine and (b) its implementation of a modulo 6 counter



The logic block is designed to meet the digital specifications and will be different for every system. It may consist of combinational logic gates, multiplexers, lookup ROMs, and other logic components.

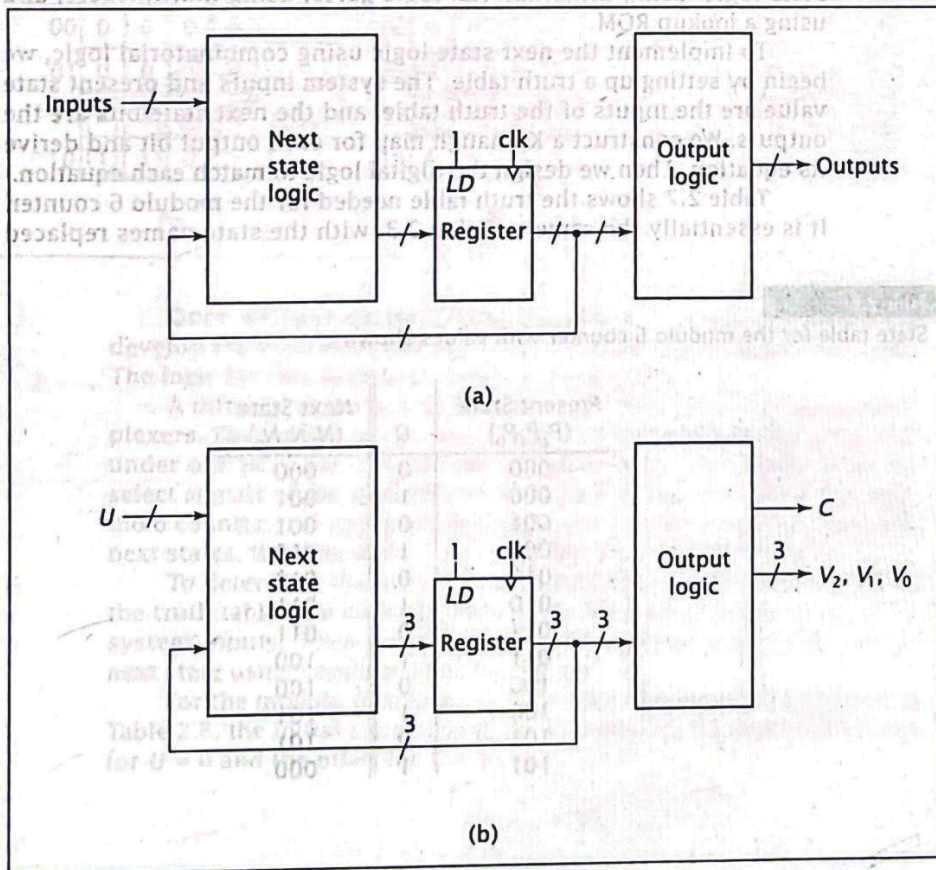
The logic block cannot include any sequential components, since it must generate its values within one clock cycle.

The logic block itself contains two parts. One part generates the next state, based on the present state and the system inputs, and the other produces the system outputs, also based on the present state and system inputs. (These parts may share hardware if doing so simplifies the design.) Figure 2.9(b) shows the logical configuration of the Mealy machine for the modulo 6 counter, which has the state diagram shown in Figure 2.8(a).

The outputs of a Moore machine depend only on its present state and not on its inputs. For this reason, its configuration is different from that of a Mealy machine. The generic Moore machine is shown in Figure 2.10(a). As with the Mealy machine, the present state is stored

Figure 2.10

(a) Generic Moore machine and (b) its implementation of a modulo 6 counter



in the register, and this value is fed back to a logic block. The logic block also receives the system inputs and generates the next state value. Since the system outputs depend only on the present state, only the state value is input to a logic block that generates the outputs. The two logic blocks are equivalent to the one combined block of the Mealy machine; in fact, the next state logic section is the same in both systems. However, the output logic block does not use the system inputs to generate the system outputs, so it may be simpler than its counterpart in the Mealy machine. The system to implement the Moore machine for the modulo 6 counter, which has the state diagram of Figure 2.8(b), is shown in Figure 2.10(b).

2.4.3 GENERATING THE NEXT STATE

Since both the Mealy machine and the Moore machine must traverse the same states under the same conditions, their next state logic is identical. This section presents three methods for designing the next state logic: using combinatorial logic gates, using multiplexers, and using a lookup ROM.

To implement the next state logic using combinatorial logic, we begin by setting up a truth table. The system inputs and present state value are the inputs of the truth table, and the next state bits are the outputs. We construct a Karnaugh map for each output bit and derive its equation. Then we design the digital logic to match each equation.

Table 2.7 shows the truth table needed for the modulo 6 counter. It is essentially the same as Table 2.3, with the state names replaced

Table 2.7

State table for the modulo 6 counter with values shown

Present State ($P_2P_1P_0$)	U	Next State ($N_2N_1N_0$)
000	0	000
000	1	001
001	0	001
001	1	010
010	0	010
010	1	011
011	0	011
011	1	100
100	0	100
100	1	101
101	0	101
101	1	000

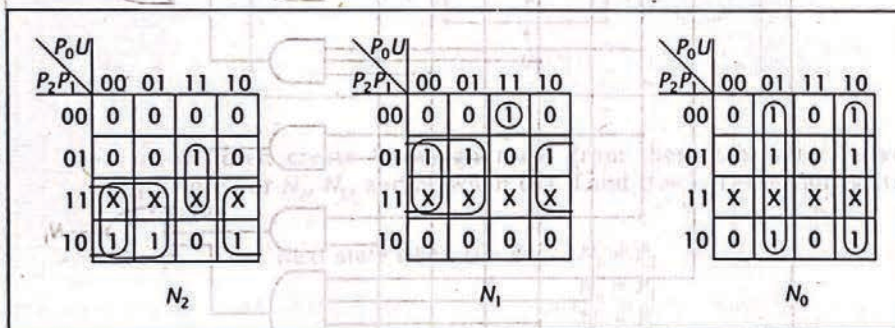
by their values and the system outputs eliminated. $P_2P_1P_0$ is the present state value and $N_2N_1N_0$ is the next state value.

Once the table has been created, we draw the Karnaugh maps and generate the equations for N_2 , N_1 , and N_0 as a function of the present state bits P_2 , P_1 , P_0 , and input U . The Karnaugh maps are shown in Figure 2.11. They yield the following equations.

$$\begin{aligned} N_2 &= P_2P_0' + P_2U' + P_1P_0U \\ N_1 &= P_1P_0' + P_1U' + P_2'P_1'P_0U \\ N_0 &= P_0'U + P_0U' \end{aligned}$$

FIGURE 2.11

Karnaugh maps for the next state of the modulo 6 counter



Once we have derived these equations, it is straightforward to develop the combinatorial logic that implements the next state logic. The logic for this system is shown in Figure 2.12.

A different approach to designing the next state logic uses multiplexers. Each input to the multiplexer corresponds to the next state under one possible value of the system inputs. The inputs drive the select signals of the multiplexer. As shown in Figure 2.13 for the modulo 6 counter, this approach uses input U to select one of two possible next states, the next state if $U = 0$, or the next state if $U = 1$.

To determine the inputs to the multiplexer, we begin by splitting the truth table into multiple tables, one for each possible value of the system inputs. Then we follow the procedure as used to derive the next state using combinatorial logic gates.

For the modulo 6 counter, there is only one input, U . As shown in Table 2.8, the initial state table is broken into two separate tables, one for $U = 0$ and the other for $U = 1$.

FIGURE 2.12

Implementation of the next state logic for the modulo 6 counter

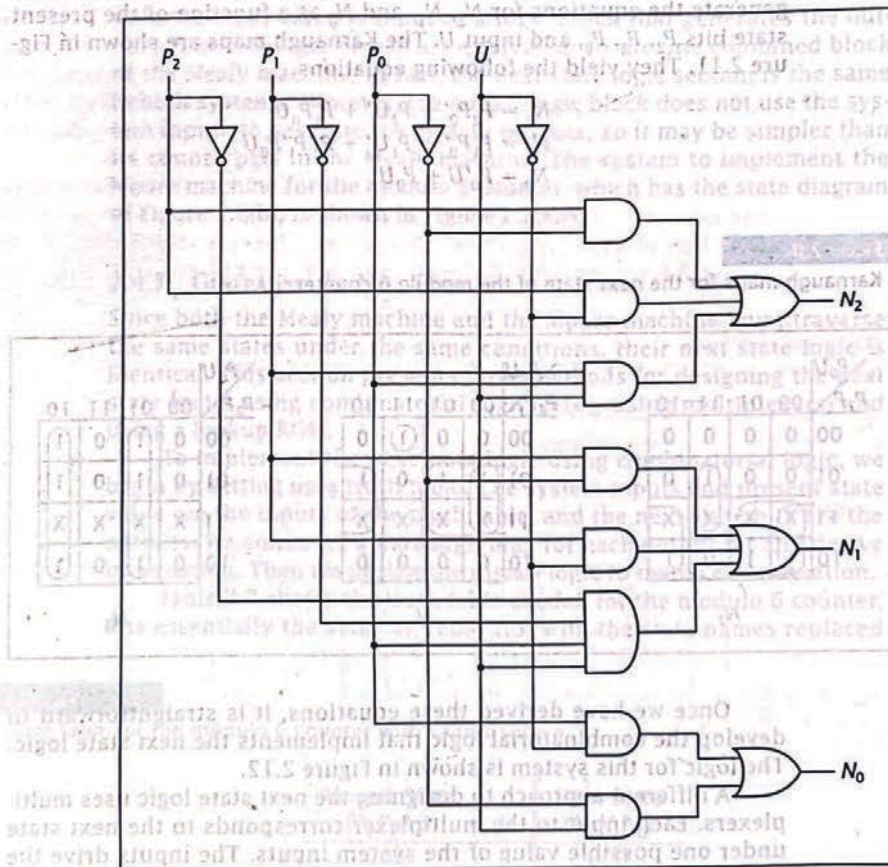


Table 2.8

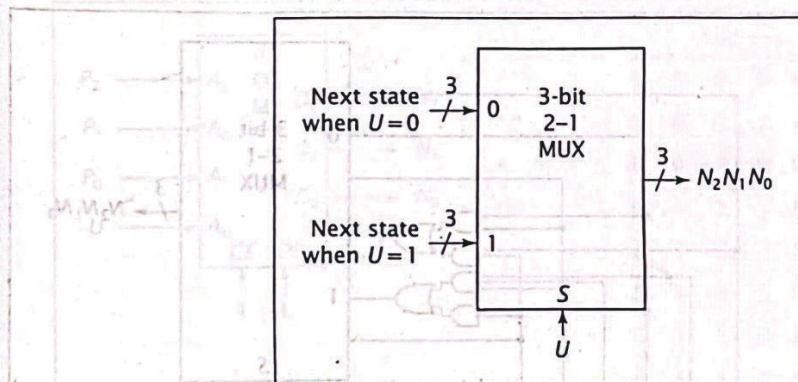
State tables for the modulo 6 counter with values shown

$P_2 P_1 P_0$	$N_2 N_1 N_0$	$P_2 P_1 P_0$	$N_2 N_1 N_0$
000	000	000	001
001	001	001	010
010	010	010	011
111	111	111	100
100	100	100	101
101	101	101	000

$U = 0$ $U = 1$

FIGURE 2.13

Preliminary implementation of the next state logic for the modulo 6 counter using a multiplexer



We then create Karnaugh maps from these tables to derive equations for N_2 , N_1 , and N_0 when $U = 0$ and $U = 1$. These values are as follows.

Next state when $U = 0$:

$$N_2 = P_2$$

$$N_1 = P_1$$

$$N_0 = P_0$$

Next state when $U = 1$:

$$N_2 = P_2P_0' + P_1P_0$$

$$N_1 = P_1P_0' + P_2'P_1'P_0$$

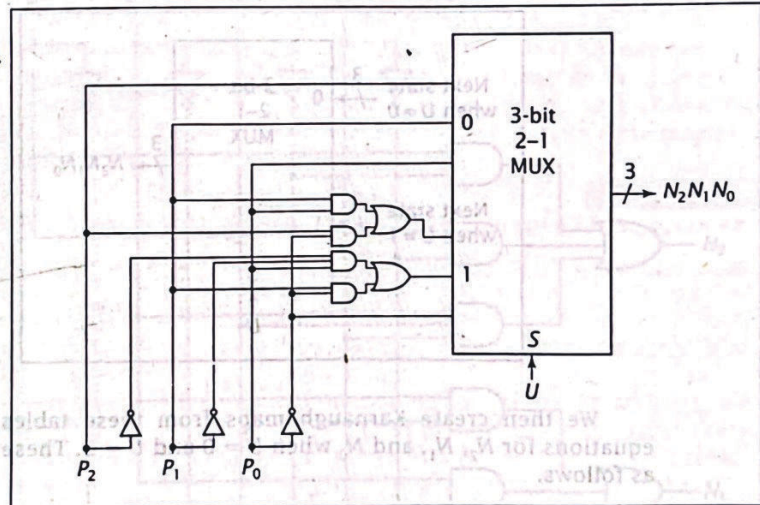
$$N_0 = P_0'$$

Finally, we implement these values using combinatorial logic, as shown in Figure 2.14.

A third approach to generating the next state of a finite state machine is to use a lookup ROM. In this method, the present state value and all system inputs are connected to the address inputs of the ROM; the next state is derived from the ROM outputs. The correct value must be stored in each location of the ROM to ensure proper operation. Figure 2.15(a) shows the next state generator for the modulo 6 counter using a lookup ROM. The three bits that encode the present state, P_2 , P_1 , and P_0 , are connected to the three high-order address inputs of the ROM. The one condition bit, U , drives the low-order address bit A_0 . The data in each location is the value of the next state for the present state and input values which generate its address. For example, when the current state is S_4 ($P_2P_1P_0 = 100$) and $U = 1$, the next state should be S_5 ($N_2N_1N_0 = 101$). These values of P_2 , P_1 , P_0 , and U generate address 9 (1001) to the lookup ROM; the ROM must have the value 5 (101) stored at this location. The values in the other locations

Figure 2.14

Final implementation of the next state logic for the modulo 6 counter using a multiplexer



are generated in the same way. Figure 2.15(b) shows the data stored in each location in the lookup ROM.

The order of the inputs to the lookup ROM shown in Figure 2.15 are not the only allowable configuration. The inputs could have been arranged in any order, as long as the correct values were stored in the proper locations of the ROM. In general, however, keeping the present state bits in order and assigning them to the high-order address inputs simplifies the design process. This assignment keeps all locations associated with a given state at adjacent locations, which makes it easier to design and debug the system.

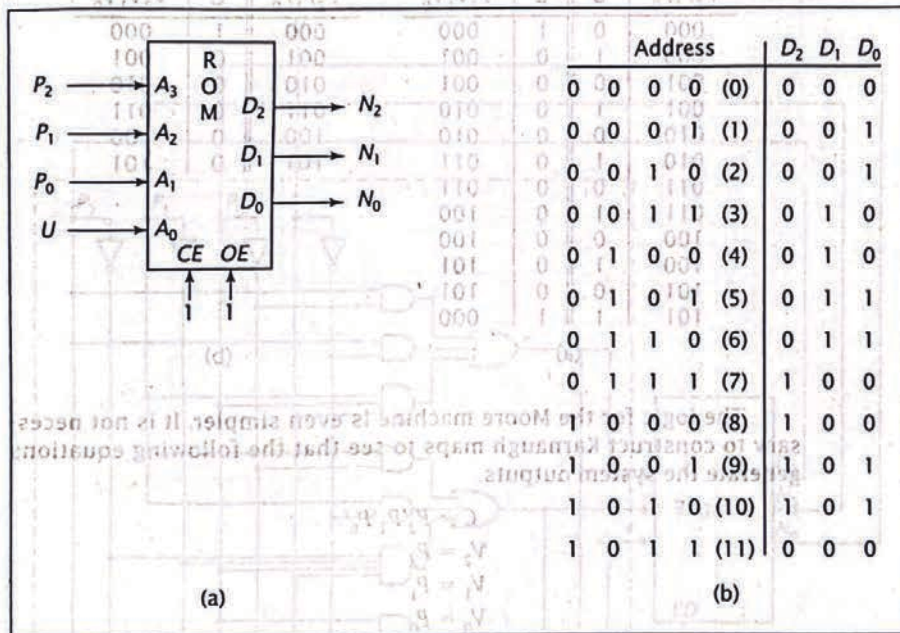
2.4.4 GENERATING SYSTEM OUTPUTS

Although Mealy and Moore machines use different values to generate system outputs, both follow the same design procedure to develop their output logic. Two approaches to output logic design, using combinational logic gates and using a lookup ROM, are similar to the techniques used to create the next state logic.

As before, the combinational logic technique begins by creating a truth table. For the Mealy machine, the table inputs are the present state and the system inputs; and the table outputs are the system outputs. For the Moore machine, only the present state bits are table in-

Figure 2.15

Implementation of the next state logic for the modulo 6 counter using a lookup ROM:
(a) logic diagram and (b) lookup ROM data



puts, since only these bits are used to generate the system outputs. The system outputs are still the table outputs. Table 2.9 shows the tables for the Mealy and Moore machines for the modulo 6 counter. The data in these tables is taken from Table 2.3 in the original problem specification. Recall that V_2, V_1 , and V_0 are the system outputs.

As with the next state logic, we construct Karnaugh maps from these tables and derive equations for the outputs. For the Mealy machine, these values are

$$\begin{aligned}
 C &= P_2'P_1'P_0'U' + P_2P_0U \\
 V_2 &= P_2P_0' + P_2U' + P_1P_0U \\
 V_1 &= P_1P_0' + P_1U' + P_2'P_1'P_0U \\
 V_0 &= P_0'U + P_0U'
 \end{aligned}$$

Note that the equations for V_2, V_1 , and V_0 are exactly the same as those for N_2, N_1 , and N_0 of the next stage logic. This is the result of optimally assigning state values to states S_0 through S_5 . In this case, we can use the same combinatorial logic to generate both sets of signals.

Table 2.9

Tables of outputs for the (a) Mealy and (b) Moore machines for the modulo 6 counter.

$P_2P_1P_0$	U	C	$V_2V_1V_0$	$P_2P_1P_0$	C	$V_2V_1V_0$
000	0	1	000	000	1	000
000	1	0	001	001	0	001
001	0	0	001	010	0	010
001	1	0	010	011	0	011
010	0	0	010	100	0	100
010	1	0	011	101	0	101
011	0	0	011			
011	1	0	100			
100	0	0	100			
100	1	0	101			
101	0	0	101			
101	1	1	000			

(a)

(b)

The logic for the Moore machine is even simpler. It is not necessary to construct Karnaugh maps to see that the following equations generate the system outputs.

$$C = P_2'P_1'P_0'$$

$$V_2 = P_2$$

$$V_1 = P_1$$

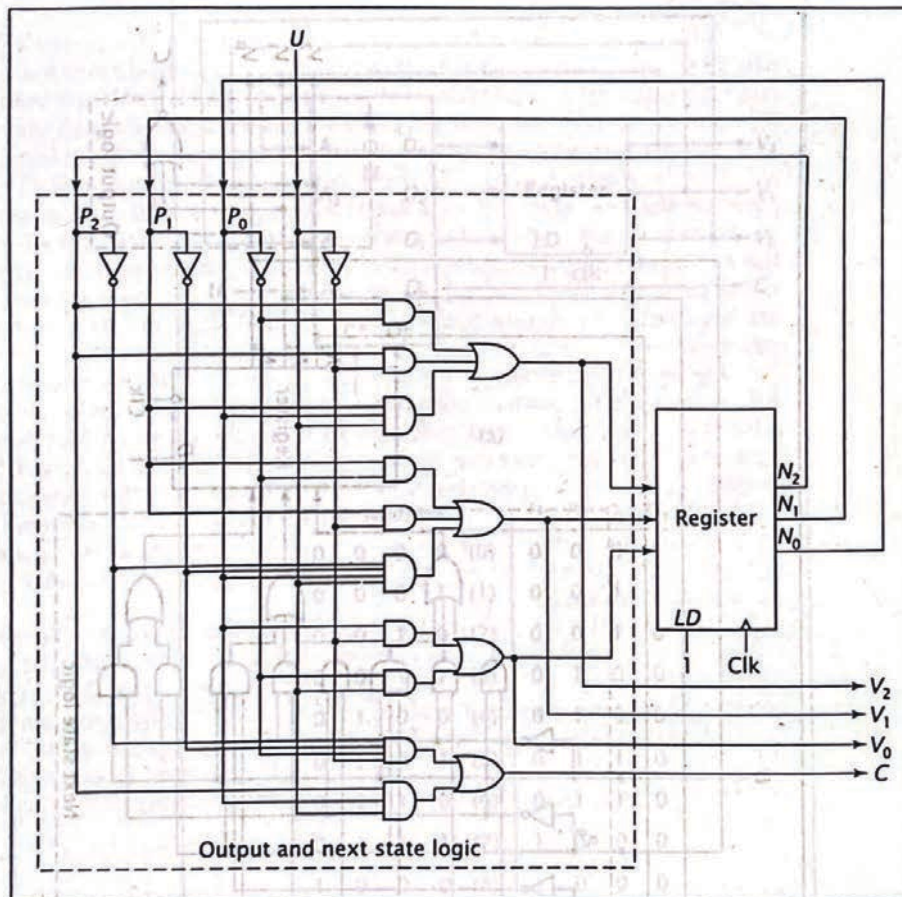
$$V_0 = P_0$$

In either machine, we would construct any additional logic necessary to generate the outputs. The complete designs for the Mealy and Moore machine implementations of the modulo 6 counter are shown in Figures 2.16 and 2.17, respectively.

It is also possible to generate system outputs using a lookup ROM. Just as was the case for the next state generator, the inputs of the lookup ROM are the present state and the system inputs. The outputs of the ROM are the system outputs. A significant advantage of this approach is that we can use the same ROM to generate both the next state and the system outputs. Figure 2.18 shows the lookup ROM implementation of the Moore machine of the modulo 6 counter. Since $V_2 = N_2$, $V_1 = N_1$, and $V_0 = N_0$, only one output is used for each pair. If the outputs were not the same as the next state, which is usually the case, separate output bits would be needed. (This is a direct result of optimally assigning values to the states.) Also note that, although it may appear otherwise, the system inputs are not used to generate the system outputs. Here, V_2 , V_1 , and V_0 are derived directly from the present state. For each present state, the outputs are the same regardless of the value of input U . Also, output C is calculated solely from the

Figure 2.16

Mealy machine implementation of the modulo 6 counter



current state and stored in the lookup ROM. We can see in the truth table of Figure 2.18 that, for a given state, C also has the same value for all values of U .

2.4.5 AN ALTERNATIVE DESIGN

There are other methods of implementing finite state machines. One method uses a counter to store the current state and a decoder to generate signals corresponding to each state. The counter can be

Figure 2.17

Moore machine implementation of the modulo 6 counter

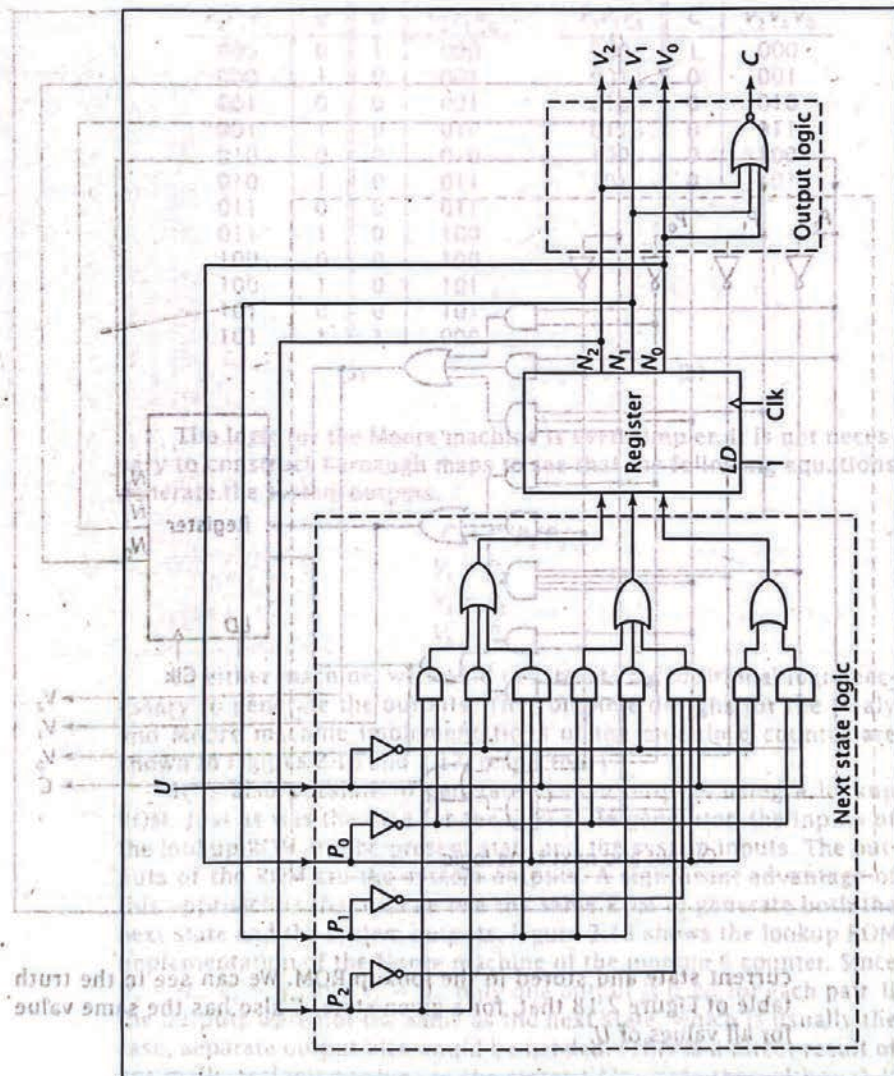
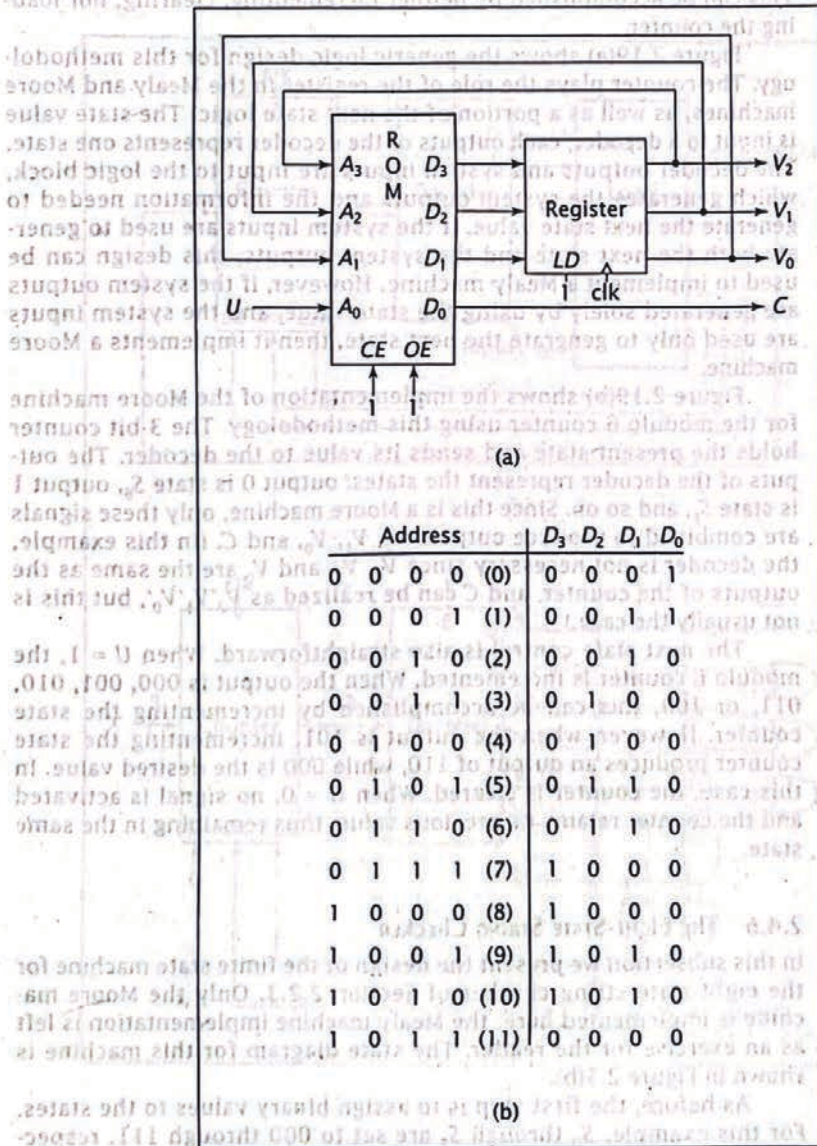


FIGURE 2.18

Moore machine implementation of the modulo 6 counter using a lookup ROM



incremented, cleared, or loaded with a value to go from one state to another. Unlike the previous implementations, it is not necessary to regenerate the same state value in order to remain in the same state. This can be accomplished by neither incrementing, clearing, nor loading the counter.

Figure 2.19(a) shows the generic logic design for this methodology. The counter plays the role of the register in the Mealy and Moore machines, as well as a portion of the next state logic. The state value is input to a decoder; each output of the decoder represents one state. The decoder outputs and system inputs are input to the logic block, which generates the system outputs and the information needed to generate the next state value. If the system inputs are used to generate both the next state and the system outputs, this design can be used to implement a Mealy machine. However, if the system outputs are generated solely by using the state value, and the system inputs are used only to generate the next state, then it implements a Moore machine.

Figure 2.19(b) shows the implementation of the Moore machine for the modulo 6 counter using this methodology. The 3-bit counter holds the present state and sends its value to the decoder. The outputs of the decoder represent the states: output 0 is state S_0 , output 1 is state S_1 , and so on. Since this is a Moore machine, only these signals are combined to produce outputs V_2 , V_1 , V_0 , and C . (In this example, the decoder is not necessary since V_2 , V_1 , and V_0 are the same as the outputs of the counter, and C can be realized as $V_2'V_1'V_0'$, but this is not usually the case.)

The next state control is also straightforward. When $U = 1$, the modulo 6 counter is incremented. When the output is 000, 001, 010, 011, or 100, this can be accomplished by incrementing the state counter. However, when the output is 101, incrementing the state counter produces an output of 110, while 000 is the desired value. In this case, the counter is cleared. When $U = 0$, no signal is activated and the counter retains its previous value, thus remaining in the same state.

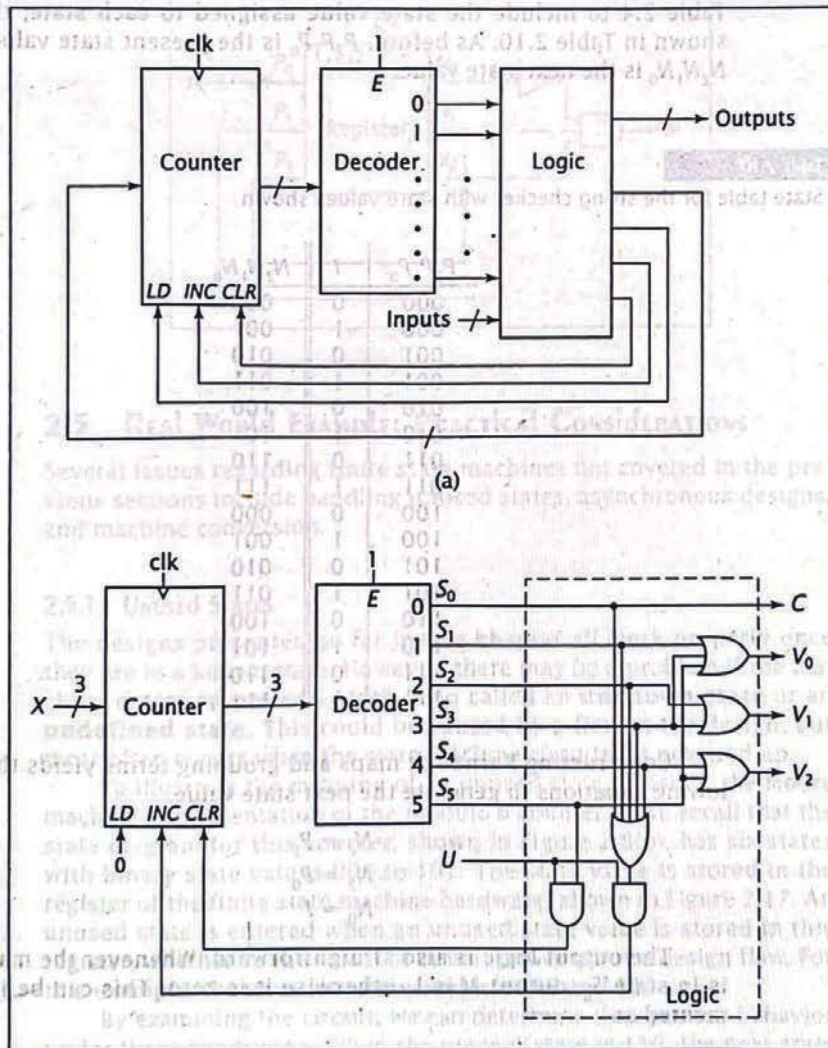
2.4.6 THE EIGHT-STATE STRING CHECKER

In this subsection we present the design of the finite state machine for the eight-state string checker of Section 2.2.3. Only the Moore machine is implemented here; the Mealy machine implementation is left as an exercise for the reader. The state diagram for this machine is shown in Figure 2.5(b).

As before, the first step is to assign binary values to the states. For this example, S_0 through S_7 are set to 000 through 111, respectively. These values can replace S_0 through S_7 in the state diagram.

Figure 2.19

Alternative state machine hardware design: (a) generic hardware and (b) Moore machine implementation of the modulo 6 counter



Next, the hardware to implement this state diagram is designed. Starting with the generic hardware shown in Figure 2.10(b), the primary task is to design the next state and output logic.

To create the next state logic, we first modify the state table of Table 2.4 to include the state value assigned to each state; this is shown in Table 2.10. As before, $P_2P_1P_0$ is the present state value and $N_2N_1N_0$ is the next state value.

Table 2.10

State table for the string checker with state values shown

$P_2P_1P_0$	I	$N_2N_1N_0$
000	0	000
000	1	001
001	0	010
001	1	011
010	0	100
010	1	101
011	0	110
011	1	111
100	0	000
100	1	001
101	0	010
101	1	011
110	0	100
110	1	101
111	0	110
111	1	111

Constructing Karnaugh maps and grouping terms yields the following equations to generate the next state value.

$$\begin{aligned} N_2 &= P_1 \\ N_1 &= P_0 \\ N_0 &= I \end{aligned}$$

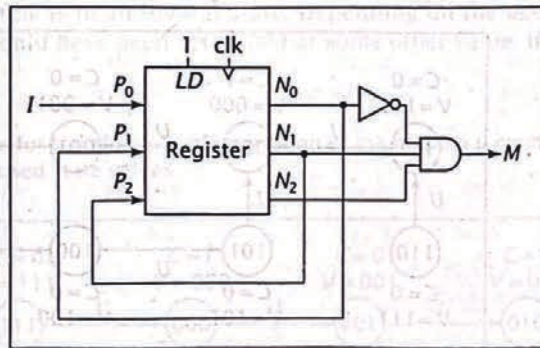
The output logic is also straightforward. Whenever the machine is in state S_6 , output M is 1; otherwise it is zero. This can be implemented as

$$M = P_2P_1P_0'$$

The complete logic design for this machine is shown in Figure 2.20.

Figure 2.20

Moore machine hardware implementation of the eight-state string checker



2.5 REAL WORLD EXAMPLE: PRACTICAL CONSIDERATIONS

Several issues regarding finite state machines not covered in the previous sections include handling unused states, asynchronous designs, and machine conversion.

2.5.1 UNUSED STATES

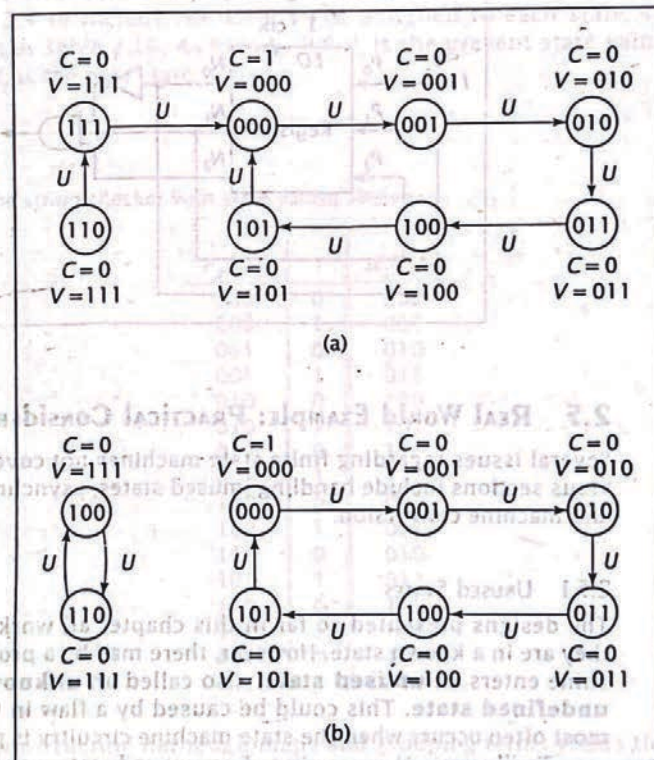
The designs presented so far in this chapter all work properly once they are in a known state. However, there may be a problem if the machine enters an **unused state**, also called an **unknown state** or an **undefined state**. This could be caused by a flaw in the design, but most often occurs when the state machine circuitry is powered up.

To illustrate the meaning of an unused state, consider the Moore machine implementation of the modulo 6 counter. First, recall that the state diagram for this counter, shown in Figure 2.8(b), has six states with binary state values 000 to 101. The state value is stored in the register of the finite state machine hardware, shown in Figure 2.17. An unused state is entered when an unused state value is stored in this register, whether as the result of circuit power up or a design flaw. For this example, the unused state values are 110 and 111.

By examining the circuit, we can determine the system's behavior under these conditions. When the present state is 110, the next state is 110 if $U = 0$ or 111 if $U = 1$. For present state value 111, the next state is 111 if $U = 0$ and 000 if $U = 1$. The revised state diagram, including the unused states, is shown in Figure 2.21(a).

Figure 2.21

State diagrams for (a) the Moore implementation of the modulo 6 counter including unused states and (b) an invalid implementation



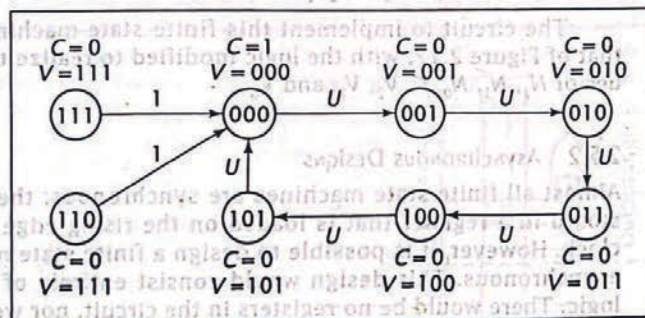
This circuit might still be considered acceptable. It is possible to reach a known state from any unknown state; however, the user must set $U=1$ for up to two clock cycles to guarantee that this happens. Whether or not this is acceptable is up to the designer. In contrast, Figure 2.21(b) presents an unacceptable implementation. If a circuit that realizes this state diagram powers up with a state value of 110 or 111, it can never reach a valid state.

A more common approach is to create dummy states for all unused state values. Each dummy state would go to a known state at the next clock cycle, usually to a reset state. In this example, we create two dummy states for the unused values 110 and 111. This guarantees that the state machine will be functional after at most one clock cycle. Figure 2.22 shows a revised state diagram for the modulo 6 counter. By convention, the value 1 on the arcs from states 110 and 111 to 000 in

indicate that the transfer is unconditional—that it is always taken. Also note that the outputs for these states are $C = 0$ and $V = 111$. This is a design decision; the value of V is set to 111 to show the user that the state machine is in an invalid state. Depending on the user's requirements, it could have been set to 000 or some other value, if desired.

FIGURE 2.22

Revised state diagram for the Moore implementation of the modulo 6 counter with dummy states for unused state values



Given the revised state diagram, we can design this finite state machine using the procedure we used for the original design. The revised state table, including outputs, is shown in Table 2.11.

Table 2.11

State table for the modulo 6 counter with unused states

$P_2P_1P_0$	U	$N_2N_1N_0$	C	$V_2V_1V_0$
000	0	000	1	000
000	1	001	1	000
001	0	001	0	001
001	1	010	0	001
010	0	010	0	010
010	1	011	0	010
011	0	011	0	011
011	1	100	0	011
100	0	100	0	100
100	1	101	0	100
101	0	101	0	101
101	1	000	0	101
110	0	000	0	111
110	1	000	0	111
111	0	000	0	111
111	1	000	0	111

We use this table to construct Karnaugh maps, which yield the following values for the next state and output signals.

$$\begin{aligned}N_2 &= P_2P_1'P_0' + P_2P_1'U' + P_1P_0U \\N_1 &= P_2'P_1P_0' + P_2'P_1U' + P_2'P_1'P_0U \\N_0 &= P_2'P_0'U + P_1'P_0'U + P_2'P_0U' + P_1'P_0U' \\C &= P_2'P_1'P_0' \\V_2 &= P_2 \\V_1 &= P_1 \\V_0 &= P_0 + P_2P_1\end{aligned}$$

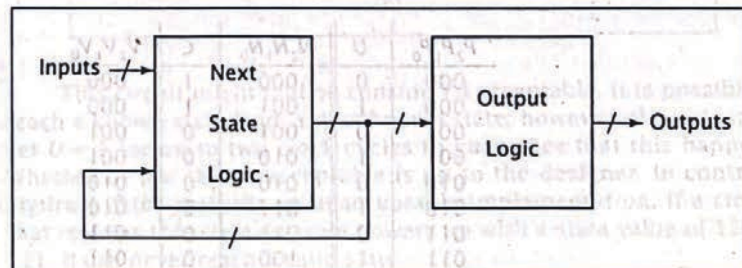
The circuit to implement this finite state machine is similar to that of Figure 2.17, with the logic modified to realize the revised values of N_2 , N_1 , N_0 , C , V_2 , V_1 , and V_0 .

2.5.2 ASYNCHRONOUS DESIGNS

Almost all finite state machines are synchronous; the state value is stored in a register that is loaded on the rising edge of the system clock. However, it is possible to design a finite state machine that is asynchronous. This design would consist entirely of combinatorial logic. There would be no registers in the circuit, nor would there be a system clock. As a result, asynchronous designs are usually faster than their synchronous counterparts. A generic design for an asynchronous Moore machine is shown in Figure 2.23.

Figure 2.23

Generic asynchronous Moore machine

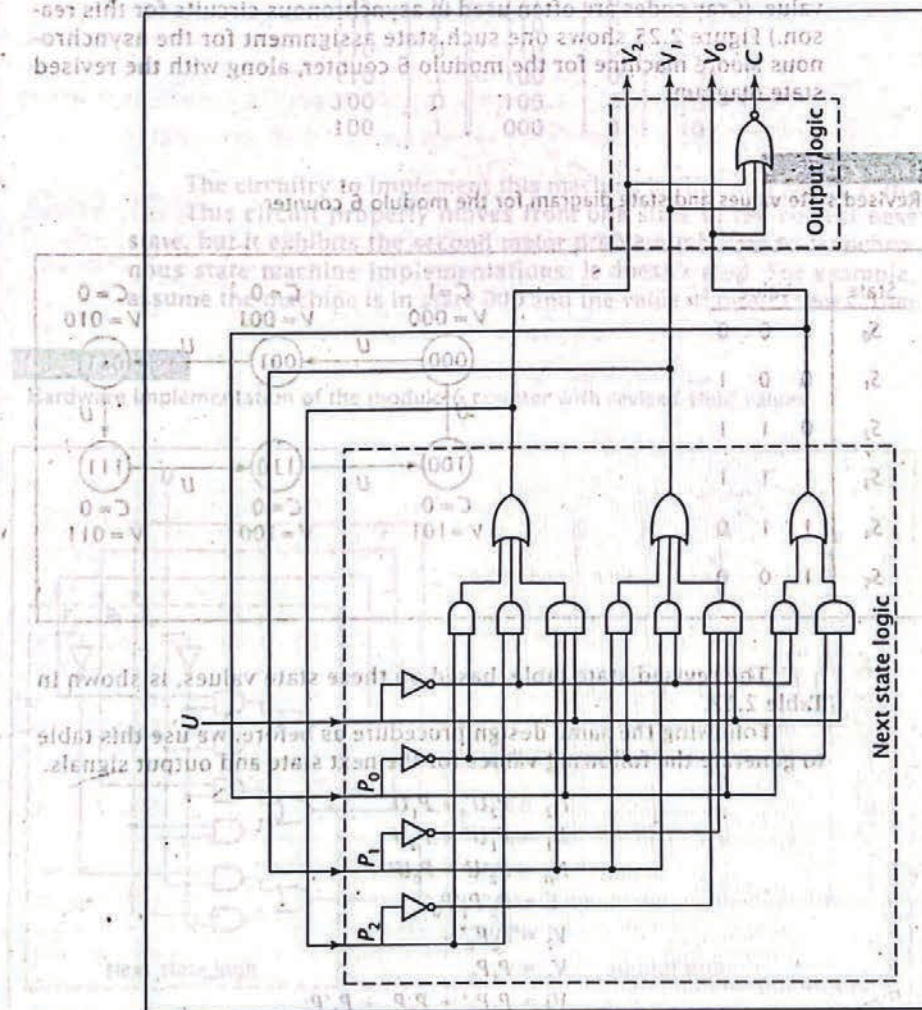


To illustrate this design method, again consider the Moore machine for the original modulo 6 counter. Since this is a methodology for implementing the machine, the state diagram shown in Figure 2.8(b) is used without modification. The next state and output logic are also the same since the machine traverses the same state sequence. A preliminary asynchronous design for this circuit is shown

in Figure 2.24. This circuit does not implement the finite state machine properly. There are two major problems with asynchronous circuit implementations, and this circuit has both of them. The first is the non-uniformity of gate delays. Consider the case when the machine is in state 011 and $U=0$. When U becomes 1, the machine should generate a next state value of 100. If the propagation delays of the portions of

Figure 2.24

Preliminary asynchronous Moore machine implementation of the modulo 6 counter



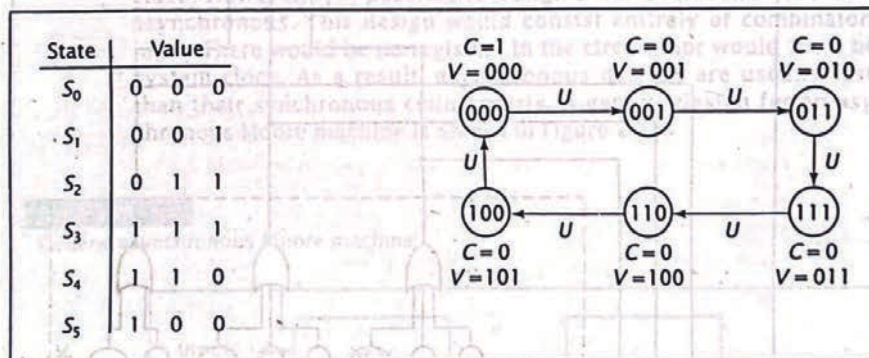
the circuit which generate P_2 , P_1 , and P_0 are always exactly the same, this would work. However, this is almost never the case.

Assume that P_0 is calculated slightly faster than P_1 and P_2 . When P_0 changes from 1 to 0, the machine enters state 010 (since P_1 and P_2 have not changed yet). The new value of P_0 is fed back as an input to the portions of the circuit which generate P_2 and P_1 (and P_0). Clearly this will not yield the desired state value. If the circuitry to set P_1 is fastest, the next state will be 001. If P_2 is fastest, the machine would produce an invalid next state value of 111.

This problem can be alleviated by assigning state values so that each state transition results in a change to only one bit of the state value. (Gray codes are often used in asynchronous circuits for this reason.) Figure 2.25 shows one such state assignment for the asynchronous Moore machine for the modulo 6 counter, along with the revised state diagram.

Figure 2.25

Revised state values and state diagram for the modulo 6 counter



The revised state table, based on these state values, is shown in Table 2.12.

Following the same design procedure as before, we use this table to generate the following values for the next state and output signals.

$$N_2 = P_2 U' + P_1 U$$

$$N_1 = P_1 U' + P_0 U$$

$$N_0 = P_2' U + P_0 U'$$

$$C = P_2' P_1' P_0'$$

$$V_2 = P_2 P_0'$$

$$V_1 = P_1 P_0$$

$$V_0 = P_2 P_1' + P_2 P_0 + P_1' P_0$$

Table 2.12

State table for the modulo 6 counter with revised state values

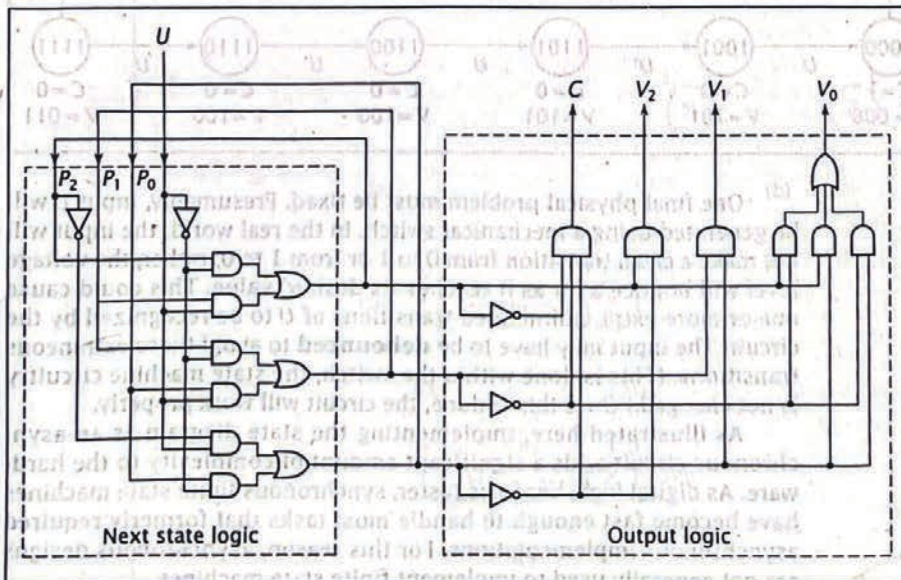
$P_2 P_1 P_0$	U	$N_2 N_1 N_0$	C	$V_2 V_1 V_0$
000	0	000	1	000
000	1	001	1	000
001	0	001	0	001
001	1	011	0	001
011	0	011	0	010
011	1	111	0	010
111	0	111	0	011
111	1	110	0	011
110	0	110	0	100
110	1	100	0	100
100	0	100	0	101
100	1	000	0	101

The circuitry to implement this machine is shown in Figure 2.26.

This circuit properly moves from one state to the correct next state, but it exhibits the second major problem inherent to asynchronous state machine implementations: It doesn't stop. For example, assume the machine is in state 000 and the value of input U goes from

Figure 2.26

Hardware implementation of the modulo 6 counter with revised state values

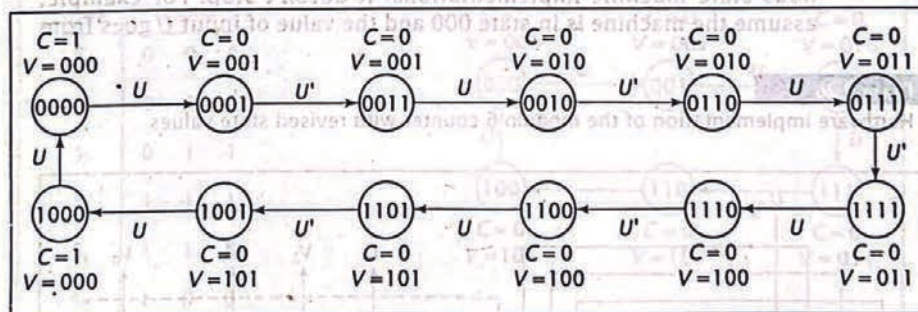


0 to 1. The machine generates a next state value of 001 and the correct system outputs in a few nanoseconds. This new value is fed back into the circuit and, if U is still set to 1, proceeds to generate the next state value of 011, continuing until U is set to 0. In its present form, the circuit acts more like a random number generator than a modulo 6 counter.

This problem can also be resolved, but at the expense of increased system complexity. The state diagram must be modified so that each state is split in two. The first of each state pair is entered when $U = 1$. The machine stays in this state until U becomes 0; then it enters the second state of the pair. This holds the machine in the same state until the value of U changes, thus causing the state transitions to occur on the rising and falling edges of U rather than whenever $U = 1$. Figure 2.27 shows the state diagram incorporating the extra states. The design of the hardware to implement this state machine is left as an exercise for the student.

Figure 2.27

Revised state diagram for the modulo 6 counter with additional states



One final physical problem must be fixed. Presumably, input U will be generated using a mechanical switch. In the real world, the input will not make a clean transition from 0 to 1 or from 1 to 0; rather, the voltage level will bounce a bit as it reaches its desired value. This could cause one or more extra, unintended transitions of U to be recognized by the circuit. The input may have to be **debounced** to avoid these extraneous transitions. (This is done within the switch; the state machine circuitry is not changed.) Once this is done, the circuit will work properly.

As illustrated here, implementing the state diagram as an asynchronous circuit adds a significant amount of complexity to the hardware. As digital logic becomes faster, synchronous finite state machines have become fast enough to handle most tasks that formerly required asynchronous implementations. For this reason, asynchronous designs are not generally used to implement finite state machines.

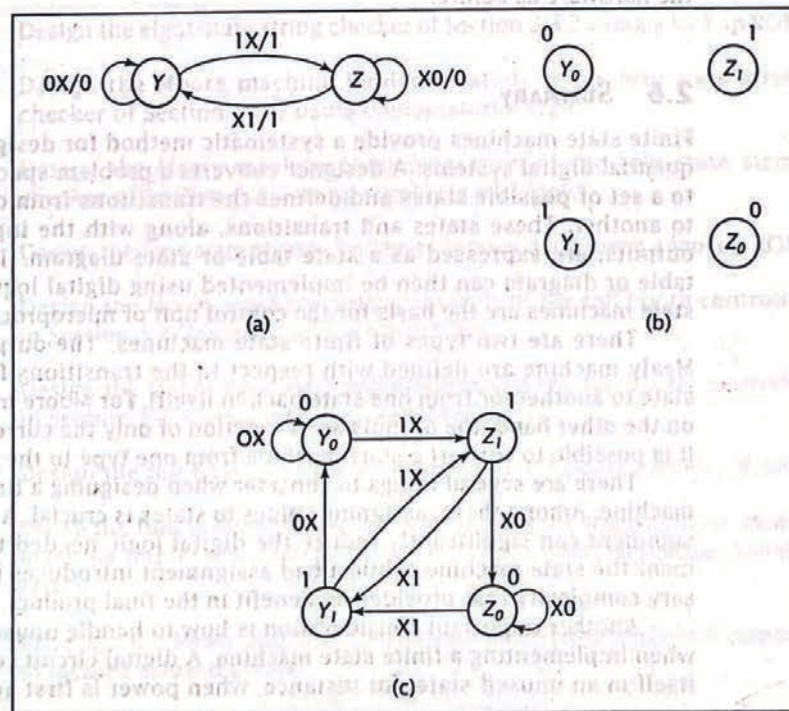
2.5.3 MACHINE CONVERSION

There are standard methods for converting between Mealy and Moore state diagrams. This may become necessary when the desired implementation is changed during the design process, or to compare equivalent implementations before choosing a final design.

The process of converting a Mealy state diagram to a Moore state diagram is relatively straightforward. First, we replace each state with a set of m states, where all of the arcs which go into the original state have a total of m different output values. We associate one of the m output patterns with each new state. To illustrate this, consider the state diagram of Figure 2.4, the JK flip-flop with modified outputs; it is redrawn as Figure 2.28(a). Since state Y has two different output values on arcs which are input to it, 0 and 1, we replace it with two states, Y_0 and Y_1 . Output value 0 is associated with Y_0 and output value 1 is assigned to Y_1 . Similarly, state Z is split into two states. These states and their outputs are shown in Figure 2.28(b).

FIGURE 2.28

Converting a Mealy state diagram to a Moore state diagram: (a) original Mealy state diagram, (b) new states and output values, and (c) final Moore state diagram



The final task is to convert the arcs from the Mealy representation to the Moore representation. Each arc from a state in the Mealy state diagram is replaced by one arc from each state in the Moore state diagram that was derived from that state. In this case, each arc from state Y is replaced by two arcs, one from Y_0 and the other from Y_1 . The destination of each arc is the state corresponding to the destination state of the arc in the Mealy machine which has the desired output value. Since outputs are now associated with the states, they are not included in the arcs. For example, the arc from Y to Z in the Mealy state diagram is replaced by arcs labeled "1X" from Y_0 to Z_1 and from Y_1 to Z_1 . Note that these arcs go to Z_1 rather than to Z_0 because Z_1 has the desired output value of 1. Repeating this process yields the Moore state diagram of Figure 2.28(c). The arcs from states Y_0 and Z_1 to themselves are shown for clarity; they may be excluded by convention.

Once the state diagram has been converted, the state machine hardware can be designed using the techniques presented earlier in this chapter.

By comparison, the process of converting from a Moore state diagram to a Mealy state diagram is trivial. We simply add the output of a state to all arcs that go to that state and remove the output values from the states themselves. Once the conversion is done, we design the hardware as before.

2.6 SUMMARY

Finite state machines provide a systematic method for designing sequential digital systems. A designer converts a problem specification to a set of possible states and defines the transitions from one state to another. These states and transitions, along with the inputs and outputs, are expressed as a state table or state diagram. The state table or diagram can then be implemented using digital logic. Finite state machines are the basis for the control unit of microprocessors.

There are two types of finite state machines. The outputs of a Mealy machine are defined with respect to the transitions from one state to another (or from one state back to itself). For Moore machines, on the other hand, the outputs are a function of only the current state. It is possible to convert a state machine from one type to the other.

There are several things to consider when designing a finite state machine. Among these, assigning values to states is crucial. A good assignment can significantly reduce the digital logic needed to implement the state machine, while a bad assignment introduces unnecessary complexity that provides no benefit in the final product.

Another important consideration is how to handle unused states when implementing a finite state machine. A digital circuit could find itself in an unused state, for instance, when power is first applied to

the circuit. A well-designed state machine must recover from such a situation and return to normal functioning if it is to be useful.

Problems

- 1 Create the state table and draw the state diagram for
 - a) a D flip-flop
 - b) a T flip-flop
- 2 Create the state table and draw the state diagram for the SR flip-flop. Include a third state, U , for the state in which the output is undefined.
- 3 Modify the state table and state diagram for the toll booth controller example in Section 2.3.3 so that all states, even those that cannot physically occur, are accounted for in the design.
- 4 Develop the state diagram for the Mealy machine implementation of the toll booth controller example in Section 2.3.3.
- 5 Design the Mealy machine implementation of the eight-state string checker of Section 2.3.2 using combinatorial logic.
- 6 Design the eight-state string checker of Section 2.3.2 using a lookup ROM.
- 7 Design the Moore machine implementation of the four-state string checker of Section 2.3.2 using combinatorial logic.
- 8 Design the Mealy machine implementation of the four-state string checker of Section 2.3.2 using combinatorial logic.
- 9 Design the four-state string checker of Section 2.3.2 using a lookup ROM.
- 10 Design the Moore machine implementation of the toll booth controller in Section 2.3.3 using combinatorial logic.
- 11 Design the Mealy machine implementation of the toll booth controller in Section 2.3.3 using combinatorial logic.
- 12 Design the toll booth controller of Section 2.3.3 using a lookup ROM.
- 13 Verify the functions for the next state and output values for the Moore machine implementation of the modulo 6 counter including unused states.
- 14 Redesign the Mealy machine implementation of the modulo 6 counter to include unused states.

15 Why isn't it necessary to include unused states in the designs of the four- and eight-state string checkers?

16 Redesign the Moore machine implementation of the toll booth controller to include unused states.

17 Design the asynchronous circuit that corresponds to the state diagram of Figure 2.28.

18 In a classic problem, a farmer, a fox, a chicken, and a sack of corn are on the left bank of a river. There is a small boat that will hold the farmer and at most one of the other three. Using as many trips as needed, the farmer must transfer the other three and himself to the right bank of the river. However, if he leaves the fox and chicken on one bank of the river while he is on the other bank, the fox will eat the chicken. Likewise, if he is on one bank and the chicken and corn are on the other bank, the chicken will eat the corn.

Design a Moore finite state machine using combinatorial logic to simulate this situation. Each state value has four bits, A , B , C , D , representing the farmer, fox, chicken, and corn, respectively. Each bit is 0 if its item is on the left bank and 1 if it is on the right bank. Two input bits, YZ , encode the action. If $YZ = 00$, the farmer crosses the river alone. For $YZ = 01$, 10, or 11, the farmer crosses the river with the fox, chicken, or corn, respectively. There are two outputs. Output L is set to 1 whenever the fox eats the chicken or the chicken eats the corn. Output W is set to 1 when all four reach the right bank of the river. Otherwise, no output is active. When either $L = 1$ or $W = 1$, the machine resets to state 0000 on the next clock cycle.

Include all unused states in your design.

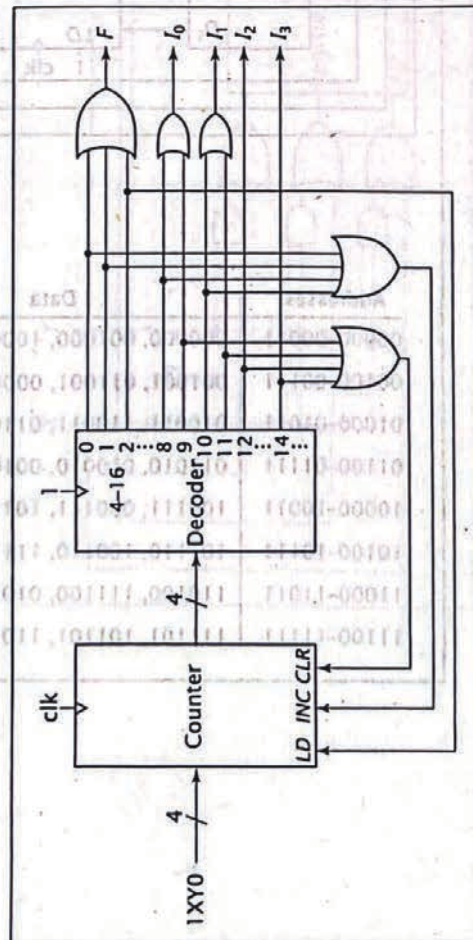
19 Repeat Problem 18, but design a Mealy machine instead of a Moore machine.

20 A certain television game show has three contestants, A , B , and C . Each player has a signaling button which goes to a circuit that outputs YZ . The circuit sets YZ to 00 when no player has pressed his or her button, and sets $YZ = 01$, 10, or 11 when the first player to press the signaling button is player A , B , or C , respectively. A clue is displayed and one of the players presses the signaling button; this locks out the other players from being recognized, and activates the player's output light, A_o , B_o , or C_o . The player then gives an answer. If the answer is correct, the host presses the R (for right/reset) button, enabling all contestants' buttons. If not, the host presses the W (for wrong) button. This disables that player's button and turns off his or her output light, but doesn't affect the other player's buttons. They can ring in and are processed in the same manner. (Thus, if a second player presses the button but

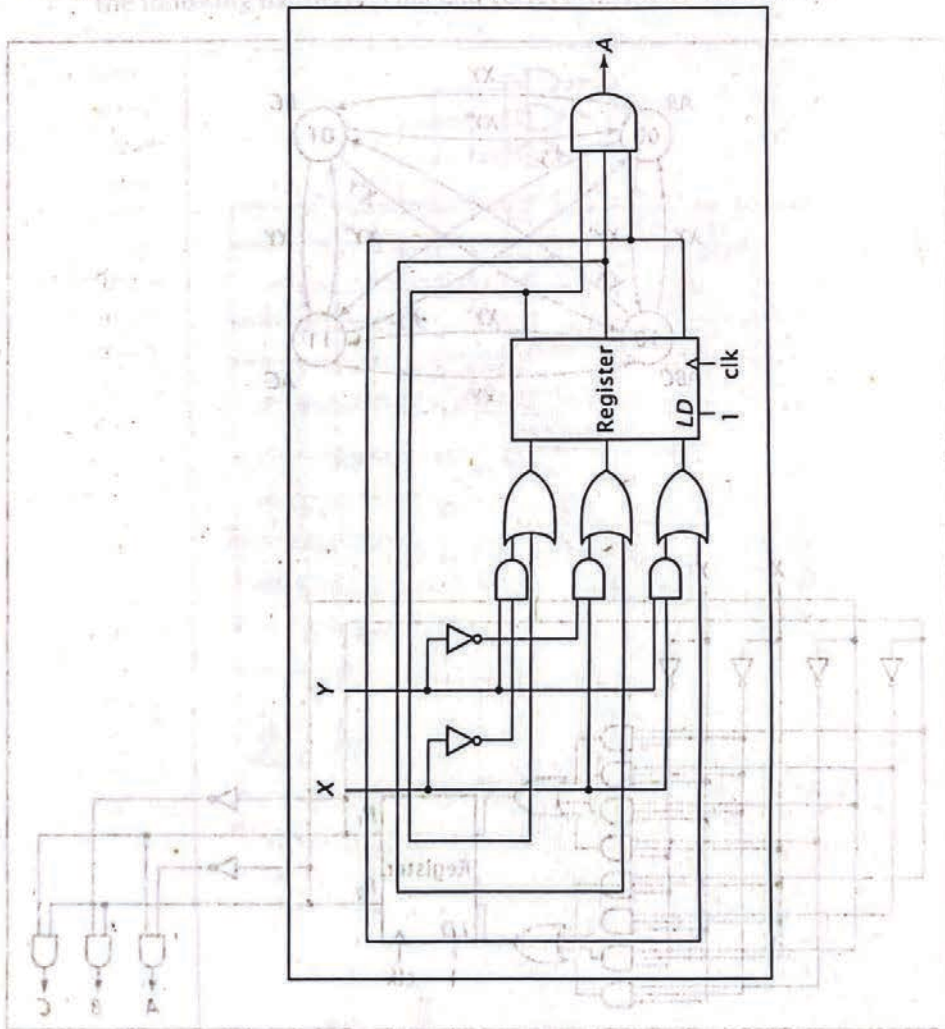
answers incorrectly, only the third player's button is active.) At any time that no player is answering a question, the host can reset the game by pressing the R button, as when, for example, time expires.

Design the Moore machine signal button controller for this game show. Implement the finite state machine using a lookup ROM. Show the contents of the ROM.

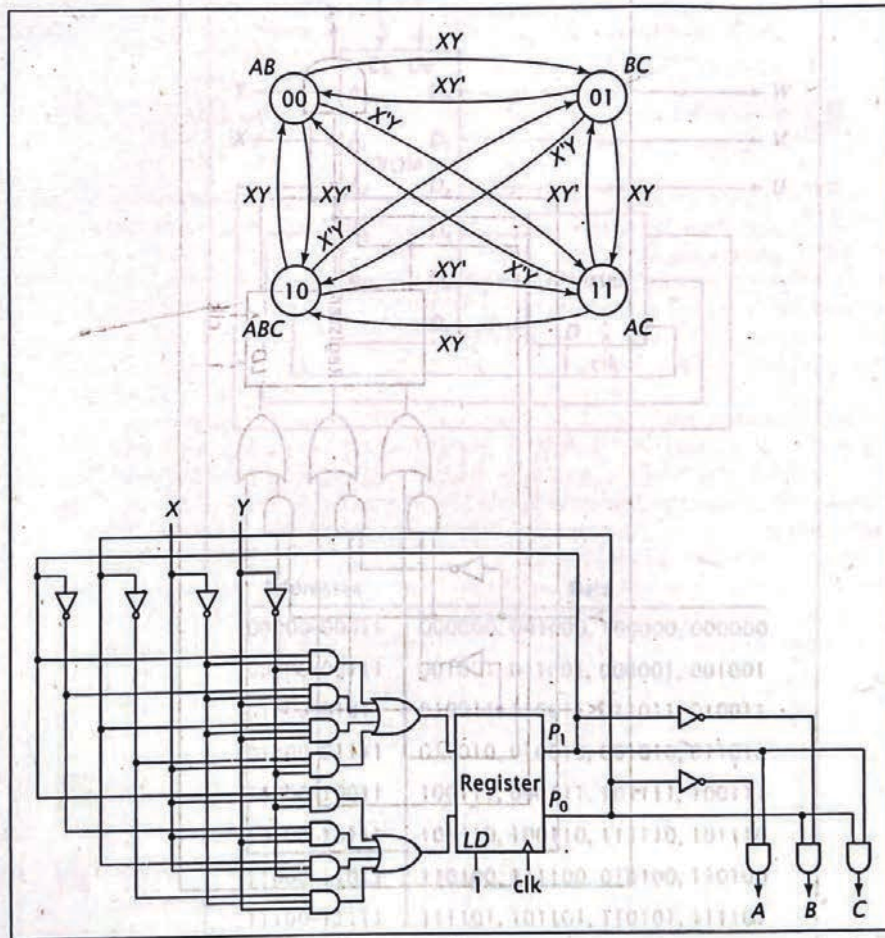
- 21 Repeat Problem 20, but design a Mealy machine instead of a Moore machine.
- 22 Show the state diagram that corresponds to the following hardware. X and Y are system inputs. Ignore states for which decoder outputs are not shown.



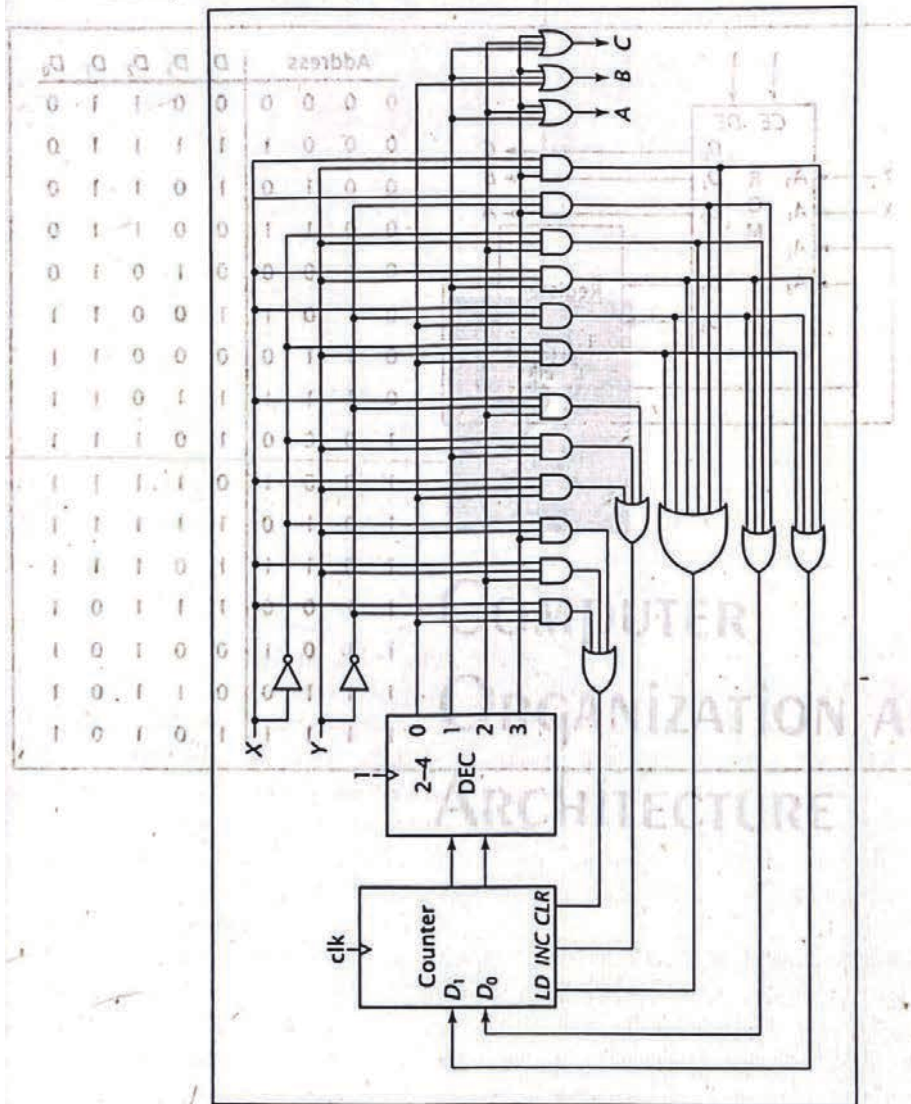
24



- 25 The following state diagram is supposed to be implemented by the circuit shown below. Find and correct the errors in the hardware.



26 The state diagram of problem 25 is supposed to be implemented by the following hardware. Find and correct the errors in the hardware.



- 27 The state diagram of problem 25 is supposed to be implemented by the following circuit with the ROM data shown beside it. Find and correct the errors in the data; do not modify the circuit hardware.

