

APPENDIX B: REDUCTION OF DIGITAL LOGIC

B.1 Reduction of Combinational Logic and Sequential Logic

In Appendix A, we focused primarily on the functional correctness of digital logic circuits. Only a small amount of consideration was given to the possibility that there may be more than one way to design a circuit, with some designs being better than others in terms of component count (that is, the numbers and sizes of the logic gates.)

In this appendix, we take a systematic approach to reducing the numbers of components in a design. We first look at reducing the sizes of combinational logic expressions, which loosely correspond to the numbers and sizes of the logic gates in an implementation of a digital circuit. We then look at reducing the numbers of states in finite state machines (FSMs), and explore a few areas of FSM design that impact the numbers and sizes of logic gates in implementations of FSMs.

B.2 Reduction of Two-Level Expressions

In many cases the canonical **sum-of-products** (SOP) or **product-of-sums** (POS) forms are not minimal in terms of their number and size. Since a smaller Boolean equation translates to a lower gate input count in the target circuit, reduction of the equation is an important consideration when circuit complexity is an issue.

Three methods of reducing Boolean equations are described in the sections that follow: **algebraic reduction**, **Karnaugh map (K-Map) reduction**, and **tabular reduction**. The algebraic method forms the basis for the other two methods. It is also the most abstract method, relying as it does on only the theorems of Boolean algebra.

The K-map and tabular methods are in fact pencil-and-paper implementations

of the algebraic method. We discuss them because they allow the student to visualize the reduction process, and to thus have a better intuition for how the process works. These manual processes can be used effectively to minimize functions that have (about) six or fewer variables. For larger functions, a computer-aided design (CAD) approach is generally more effective.

B.2.1 THE ALGEBRAIC METHOD

The algebraic method applies the properties of Boolean algebra that were introduced in Section A.5 in a systematic manner to reduce expression size. Consider the Boolean equation for the majority function, which is repeated below from Appendix A:

$$F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \quad (\text{B.1})$$

The properties of Boolean algebra can be applied to reduce the equation to a simpler form as shown in equations B.2 – B.4:

$$F = \bar{A}BC + A\bar{B}C + AB(\bar{C} + C) \quad \text{Distributive property} \quad (\text{B.2})$$

$$F = \bar{A}BC + A\bar{B}C + AB(1) \quad \text{Complement property} \quad (\text{B.3})$$

$$F = \bar{A}BC + A\bar{B}C + AB \quad \text{Identity property} \quad (\text{B.4})$$

The corresponding circuit for Equation B.4 is shown in Figure B-1. In comparison with the majority circuit shown in Figure A-16, the gate count is reduced from 8 to 6 and the gate input count is reduced from 19 to 13.

We can reduce Equation B.4 further. By applying the property of idempotence, we obtain Equation B.5, in which we have reintroduced the minterm ABC .

$$F = \bar{A}BC + A\bar{B}C + AB + ABC \quad \text{Idempotence property} \quad (\text{B.5})$$

We can then apply the distributive, complement, and identity properties again and obtain a simpler equation as shown below:

$$F = \bar{A}BC + AC(\bar{B} + B) + AB \quad \text{Distributive property} \quad (\text{B.6})$$

$$F = \bar{A}BC + AC(1) + AB \quad \text{Complement property} \quad (\text{B.7})$$

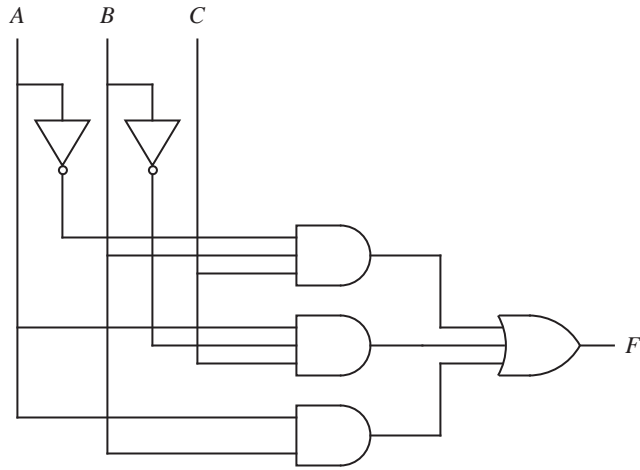


Figure B-1 Reduced circuit for the majority function.

$$F = \bar{A}BC + AC + AB \quad \text{Identity property} \quad (\text{B.8})$$

Equation B.8 has a smaller gate input count of 11. We iterate this method one more time and reduce the equation further as shown below:

$$F = \bar{A}BC + AC + AB + ABC \quad \text{Idempotence property} \quad (\text{B.9})$$

$$F = BC(\bar{A} + A) + AC + AB \quad \text{Distributive property} \quad (\text{B.10})$$

$$F = BC(1) + AC + AB \quad \text{Complement property} \quad (\text{B.11})$$

$$F = BC + AC + AB \quad \text{Identity property} \quad (\text{B.12})$$

Equation B.12 is now in its minimal two-level form, and can be reduced no further.

B.2.2 THE K-MAP METHOD

The K-map method is, in effect, a graphical technique that can be used to visualize the minterms in a function along with variables that are common to them. Variables that are common to more than one minterm are candidates for elimination, as discussed above. The basis of the K-map is the *Venn diagram*, which was originally devised to visualize concepts in set theory.

The Venn diagram for binary variables consists of a rectangle that represents the binary universe in SOP form. A Venn diagram for three variables A , B , and C is shown in Figure B-2. Within the universe is a circle for each variable. Within its

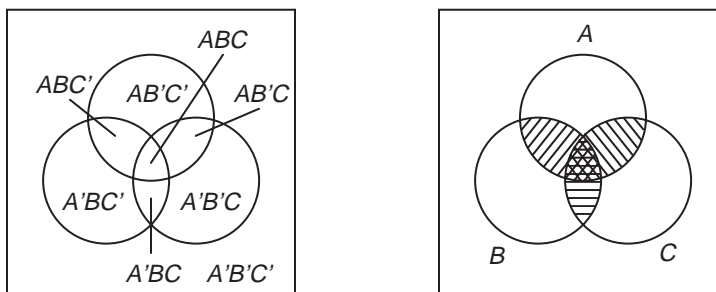


Figure B-2 A Venn diagram representation for 3 binary variables (left) and for the majority function (right).

circle a variable has the value 1, and outside of its circle a variable has the value 0. Intersections represent the minterms, as shown in the figure.

Adjacent shaded regions are candidates for reduction since they vary in exactly one variable. In the figure, region ABC can be combined with each of the three adjacent regions to produce a reduced form of the majority function. The K-map is just a topological, or relationship-preserving transformation of the Venn diagram. As in the Venn diagram, in the K-map, minterms that differ in exactly one variable are placed next to each other.

A K-map for the majority function is shown in Figure B-3. Each cell in the

		AB			
		00	01	11	10
C	0			1	
	1		1	1	1

Figure B-3 A K-map for the majority function.

K-map corresponds to an entry in the truth table for the function, and since there are eight entries in the truth table, there are eight cells in the corresponding K-map. A 1 is placed in each cell that corresponds to a true entry. A 0 is entered in each remaining cell, but can be omitted from the K-map for clarity as it is here. The labeling along the top and left sides is arranged in a **Gray code**, in which exactly one variable changes between adjacent cells along each dimension.

Adjacent 1's in the K-map satisfy the condition needed to apply the complement property of Boolean algebra. Since there are adjacent 1's in the K-map shown in Figure B-3, a reduction is possible. Groupings of adjacent cells are made into rectangles in sizes that correspond to powers of 2, such as 1, 2, 4 and 8. These groups are referred to as **prime implicants**. As groups increase in size above a 1-group (a group with one member), more variables are eliminated from a Boolean expression, and so the largest groups that can be obtained are used. In order to maintain the adjacency property, the shapes of groups must always be rectangular, and each group must contain a number of cells that corresponds to an integral power of two.

We start the reduction process by creating groups for 1's *that can be contained in no larger group*, and progress to larger groups until all cells with a 1 are covered at least once. The adjacency criterion is crucial, since we are looking for groups of minterms that differ in such a way that a reduction can be applied by using the complement and identity properties of Boolean algebra, as in Equation B.13:

$$ABC + AB\bar{C} = AB(C + \bar{C}) = AB(1) = AB \quad (\text{B.13})$$

For the majority function, three groups of size two are made as shown in Figure

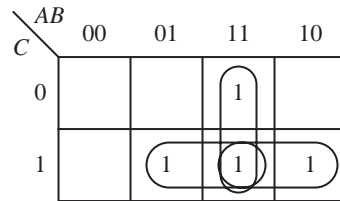


Figure B-4 Adjacency groupings for the majority function.

B-4. Every cell with a 1 has at least one neighboring cell with a 1, and so there are no 1-groups. We look next at 2-groups, and find that all of the 1-cells are covered by 2-groups. One of the cells is included in all three groups, which is allowed in the reduction process by the property of idempotence. The complement property eliminates the variable that differs between cells, and the resulting minimized equation is obtained (Equation B.14):

$$M = BC + AC + AB \quad (\text{B.14})$$

The BC term is derived from the 2-group $(ABC + \bar{A}BC)$, which reduces to $BC(A + \bar{A})$ and then to BC . The AC term is similarly derived from the 2-group $(ABC + A\bar{B}C)$, and the AB term is similarly derived from the 2-group

$(ABC + AB\bar{C})$. The corresponding circuit is shown in Figure B-5. The gate

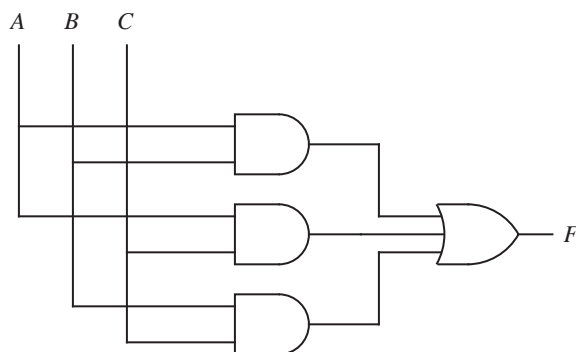


Figure B-5 Minimized AND-OR circuit for the majority function.

count is reduced from 8 to 4 as compared with the circuit shown in Figure A-16, and the gate input count is reduced from 19 to 9.

Looking more closely at the method of starting with 1-cells that can be included in no larger subgroups, consider what would happen if we started with the largest groups first. Figure B-6 shows both approaches applied to the same K-map. The

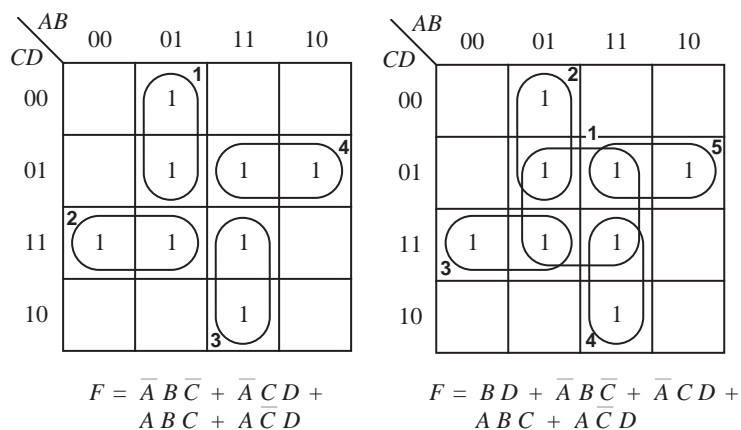


Figure B-6 Minimal K-map grouping (left) and K-map grouping that is not minimal (right) of a K-map.

reduction on the left is obtained by working with 1's that can be included in no larger subgroup, which is the method we have been using. Groupings are made in the order indicated by the numbers. A total of four groups are obtained, each of size two. The reduction on the right is obtained by starting with the largest groups first. Five groups are thus obtained, one of size four and four of size two.

Thus, the minimal equation is not obtained if we start with the largest groups first. Both equations shown in Figure B-6 describe the same function, and a logically correct circuit will be obtained in either case, however, one circuit will not be produced from a minimized equation.

As another example, consider the K-map shown in Figure B-7. The edges of the

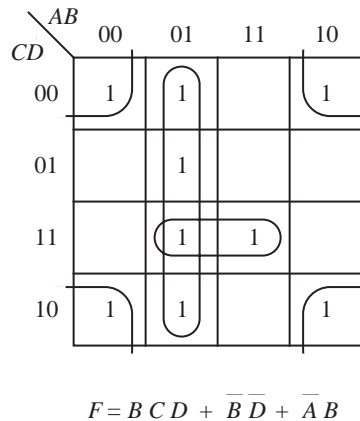


Figure B-7 The corners of a K-map are logically adjacent.

K-map wrap around horizontally and vertically, and the four corners are logically adjacent. The corresponding minimized equation is shown in the figure.

Don't cares

Now consider the K-maps shown in Figure B-8. The *d* entries denote *don't cares*, which can be treated as 0's or as 1's, at our convenience. A don't care represents a condition that cannot arise during operation. For example, if $X=1$ represents the condition in which an elevator is on the ground floor, and $Y=1$ represents the condition in which the elevator is on the top floor, then X and Y will not both be 1 at the same time, although they may both be 0 at the same time. Thus, a truth table entry for an elevator function that corresponds to $X = Y = 1$ would be marked as a don't care.

In Figure B-8, a more complex function is shown in which two different results are obtained from applying the same minimization process. The K-map on the left treats the top right don't care as a 1 and the bottom left don't care as a 0. The K-map on the right treats the top right don't care as a 0 and the bottom left don't care as a 1. Both K-maps result in minimized Boolean equations of the same size,

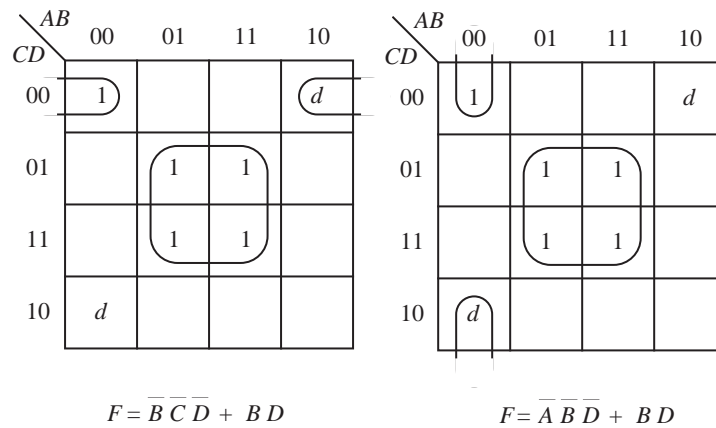


Figure B-8 Two different minimized equations are produced from the same K-map.

and so it is possible to have more than one minimal expression for a Boolean function. In practice, one equation may be preferred over another, possibly in order to reduce the fan-out for one of the variables, or to take advantage of sharing minterms with other functions.

Higher Dimensional Maps

Figure B-9 shows a K-map in five variables. Each cell is adjacent to five others,

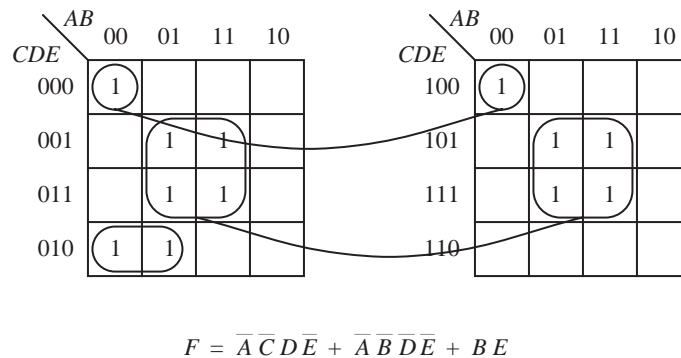


Figure B-9 A K-map in five variables.

and in order to maintain the inverse property for adjacent cells, the map on the left overlays the map on the right, creating a three-dimensional structure. Groupings are now made in three dimensions as shown in the figure. Since the

three-dimensional structure is mapped onto a two-dimensional page, some visualization is required on the part of the reader.

A six-variable K-map is shown in Figure B-10, in which the maps are overlaid

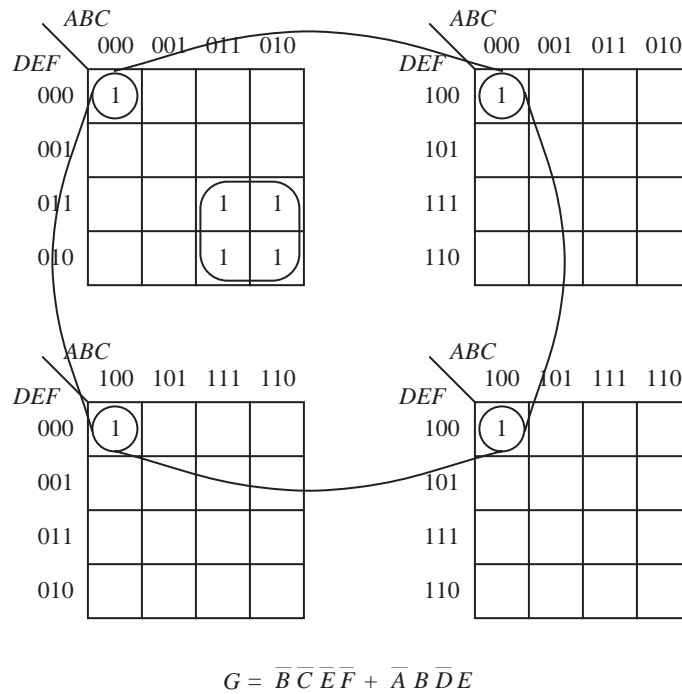


Figure B-10 A K-map in six variables.

four deep, in the order: top-left, top-right, bottom-right, and bottom left. K-maps can be extended to higher dimensions for seven or more variables, but the visualization and the tedium tend to dominate the process. An algorithmic approach for more than four variables that lends itself to a simple implementation on a computer is described in Section B.2.3.

Multilevel circuits

It should be emphasized that a K-map reduces the size of a two-level expression, as measured by the number and sizes of the terms. This process does not necessarily produce a minimal form for multilevel circuits. For example, Equation B.14 is in its minimal two-level form, since only two levels of logic are used in its representation: three ANDed collections of variables (product terms) that are

Ored together. The corresponding logic diagram that is shown in Figure B-5 has a gate-input count of 9. A three-level form can be created by factoring out one of the variables algebraically, such as A, as shown in Equation B.15.

$$M = BC + A(B + C) \quad (\text{B.15})$$

The corresponding logic diagram that is shown in Figure B-11 has a gate input

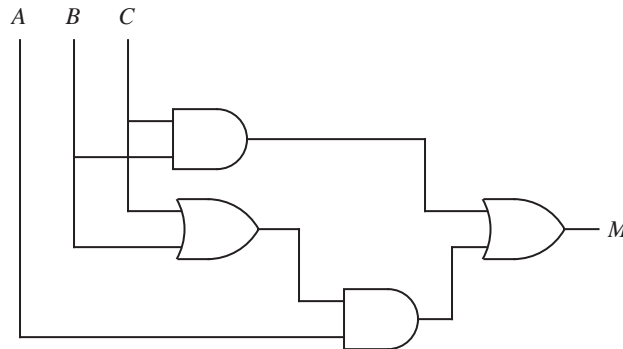


Figure B-11 A three-level circuit implements the majority function with a gate-input count of 8.

count of 8, and thus a less complex circuit is realized by using a multilevel approach. There is now a greater delay between the inputs and the outputs, however, and so we might create another measure of circuit complexity: the **gate delay**. A two-level circuit has a gate delay of two because there are two logic gates on the longest path from an input to an output. The circuit shown in Figure B-11 has a gate delay of three.

Although there are techniques that aid the circuit designer in discovering trade-offs between circuit depth and gate input count, the development of algorithms that cover the space of possible alternatives in reasonable time is only a partially solved problem.

Map-Entered Variables

A simplified form for representing a function on a K-map is possible by allowing variables to be entered in the cells. For example, consider the four-variable K-map shown in Figure B-12. Only eight cells are used even though there are four variables, which would normally require $2^4 = 16$ cells. The **map-entered variable** D is treated as a 1 for the purpose of grouping, which for this case results in a one-group since there are no adjacent 1's to the D cell. The resulting

$\begin{array}{c} AB \\ \diagdown \\ C \end{array}$		00	01	11	10
		0	1	1	0
0	D				
1			1	1	

$$F = BC + \overline{A}\overline{B}\overline{C}D$$

Figure B-12 An example of a K-map with a map-entered variable D .

reduced equation is shown in the figure. Notice that the variable D appears in the minterm $\overline{A}\overline{B}\overline{C}D$, since D can assume a value of 0 or 1 even though we treated D as a 1 for the purpose of forming the one-group.

The general procedure for producing a reduced expression from a K-map with map-entered variables is to first obtain an expression for the 1-cells while treating the map-entered variables as 0's. Minterms are then added for each variable while treating 1's as don't cares since the 1's have already been covered. The process continues until all variables are covered.

Consider the map shown in Figure B-13, in which D , E , and \overline{E} are map-entered

$\begin{array}{c} AB \\ \diagdown \\ C \end{array}$		00	01	11	10
		0	1	1	0
0	D	d	E	\overline{E}	
1			1	1	

Figure B-13 A K-map with two map-entered variables D and E .

variables, and d represents a don't care. The 1's are considered first, which produces the term BC . Variable D is considered next, which produces the term $\overline{A}\overline{C}D$. Variable E is considered next, which produces the term BE . Finally, variable \overline{E} is considered, which produces the term $A\overline{B}\overline{C}\overline{E}$. Notice that a map-entered variable and its complement are considered separately, as for E in this example. Equation B.16 shows the reduced form:

$$F = BC + \overline{A}\overline{C}D + BE + A\overline{B}\overline{C}\overline{E} \quad (\text{B.16})$$

B.2.3 THE TABULAR METHOD

An automated approach to reducing Boolean expressions is commonly used for single and multiple output functions. The tabular method, also known as the **Quine-McCluskey** method, successively forms Boolean cross products among groups of terms that differ in one variable, and then uses the smallest set of reduced terms to cover the functions. This process is easier than the map method to implement on a computer, and an extension of the method allows terms to be shared among functions.

Reduction of Single Functions

The truth table shown in Figure B-14 describes a function F in four variables A ,

A	B	C	D	F
0	0	0	0	d
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	d
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	d

Figure B-14 A truth table representation of a function with don't cares.

B , C , and D , and includes three don't cares. The tabular reduction process begins by grouping minterms for which F is nonzero according to the number of 1's in each minterm. Don't care conditions are considered to be nonzero for this process. Minterm 0000 contains no 1's and is in its own group, as shown in Figure B-15a. Minterms 0001, 0010, 0100, and 1000 all contain a single 1, but only minterm 0001 has a nonzero entry and so it forms another group.

The next group has two 1's in each minterm, and there are six possible minterms

Initial setup	After first reduction	After second reduction																																																																																																																
<table> <tr> <th>A</th> <th>B</th> <th>C</th> <th>D</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	A	B	C	D	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	<table> <tr> <th>A</th> <th>B</th> <th>C</th> <th>D</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>—</td> </tr> <tr> <td>0</td> <td>0</td> <td>—</td> <td>1</td> </tr> <tr> <td>0</td> <td>—</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>—</td> <td>1</td> <td>1</td> </tr> <tr> <td>—</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>—</td> <td>1</td> </tr> <tr> <td>—</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>—</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>—</td> </tr> <tr> <td>—</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>—</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>—</td> <td>1</td> </tr> </table>	A	B	C	D	0	0	0	—	0	0	—	1	0	—	0	1	0	—	1	1	—	0	1	1	0	1	—	1	—	1	0	1	0	1	1	—	1	0	1	—	—	1	1	1	1	—	1	1	1	1	—	1	<table> <tr> <th>A</th> <th>B</th> <th>C</th> <th>D</th> </tr> <tr> <td>0</td> <td>—</td> <td>—</td> <td>1</td> </tr> <tr> <td>—</td> <td>—</td> <td>1</td> <td>1</td> </tr> <tr> <td>—</td> <td>1</td> <td>—</td> <td>1</td> </tr> </table>	A	B	C	D	0	—	—	1	—	—	1	1	—	1	—	1
A	B	C	D																																																																																																															
0	0	0	0																																																																																																															
0	0	0	1																																																																																																															
0	0	1	1																																																																																																															
0	1	0	1																																																																																																															
0	1	1	0																																																																																																															
1	0	1	0																																																																																																															
0	1	1	1																																																																																																															
1	0	1	1																																																																																																															
1	1	0	1																																																																																																															
1	1	1	1																																																																																																															
A	B	C	D																																																																																																															
0	0	0	—																																																																																																															
0	0	—	1																																																																																																															
0	—	0	1																																																																																																															
0	—	1	1																																																																																																															
—	0	1	1																																																																																																															
0	1	—	1																																																																																																															
—	1	0	1																																																																																																															
0	1	1	—																																																																																																															
1	0	1	—																																																																																																															
—	1	1	1																																																																																																															
1	—	1	1																																																																																																															
1	1	—	1																																																																																																															
A	B	C	D																																																																																																															
0	—	—	1																																																																																																															
—	—	1	1																																																																																																															
—	1	—	1																																																																																																															
(a)	(b)	(c)																																																																																																																

Figure B-15 The tabular reduction process.

that can belong to this group. Only minterms 0011, 0101, 0110, and 1010 have nonzero entries, and so they comprise this group. There are three nonzero entries in the next group, which has three 1's in each minterm. The nonzero minterms are 0111, 1011, and 1110. Finally, there is one nonzero entry that contains four 1's, and the corresponding minterm makes up the last group. For larger truth tables, the process continues until all nonzero entries are covered. The groups are organized so that adjacent groups differ in the number of 1's by one, as shown in Figure B-15a.

The next step in the reduction process is to form a consensus (the logical form of a cross product) between each pair of adjacent groups for all terms that differ in only one variable. The general form of the consensus theorem is restated from Appendix A below:

$$XY + \bar{X}Z + YZ = XY + \bar{X}Z \quad (\text{B.17})$$

The term YZ is redundant, since it is covered by the remaining terms, and so it

can be eliminated. Algebraically, we can prove the theorem as shown below:

$$\begin{aligned}
 XY + \bar{X}Z + YZ &= XY + \bar{X}Z + YZ(X + \bar{X}) \\
 &= XY + \bar{X}Z + XYZ + \bar{X}YZ \\
 &= XY + XYZ + \bar{X}Z + \bar{X}YZ \\
 &= XY(1 + Z) + \bar{X}Z(1 + Y) \\
 &= XY + \bar{X}Z
 \end{aligned}$$

The consensus theorem also has a dual form:

$$(X + Y)(\bar{X} + Z)(Y + Z) = (X + Y)(\bar{X} + Z) \quad (\text{B.18})$$

The idea of applying consensus in tabular reduction is to take advantage of the inverse property of Boolean algebra, similar to the way we did for K-maps in the previous section. For example, 0000 and 0001 differ in variable D , so 000_ is listed at the top of the reduced table shown in Figure B-15b. The underscore marks the position of the variable that has been eliminated, which is D for this case. Minterms 0000 and 0001 in Figure B-15a are marked with checks to indicate that they are now covered in the reduced table.

After every term in the first group is crossed with every term in the second group, we then move on to form a consensus between terms in the second and third groups. Note that it is possible that some terms cannot be combined into a smaller term because they differ in more than one variable. For example, terms 0001 and 0011 combine into the smaller term 00_1 as shown in the top of the second group in Figure B-15b, but terms 0001 and 0110 cannot be combined because they differ in three variables.

Once a term is marked with a check, it can still be used in the reduction process by the property of idempotence. The objective in this step of the process is to discover all of the possible reduced terms, so that we can find the smallest set of terms that covers the function in a later step.

The process continues for the remaining groups. Any term that is not covered after all consensus groupings are made is marked with an asterisk to indicate that it is a prime implicant. After the first reduction is made for this example, all of the minterms shown in Figure B-15a are covered so there are no prime implicants at this point.

Now that the first reduction is made, we can start on the next reduction. In order for two reduced terms to be combined, they must again differ in exactly one variable. The underscores must line up, and only one of the remaining variables can differ. The first entry shown in Figure B-15b has an underscore in the rightmost field, which does not coincide with any term in the next group, so an asterisk is placed next to it indicating that it can be reduced no further and is therefore a prime implicant. We continue by moving on to the second and third groups of Figure B-15b. Terms 00_1 and 01_1 combine to form reduced term 0__1 in the table shown in Figure B-15c. The process continues until the second reduction is completed, which is shown in Figure B-15c.

In constructing the reduced table shown in Figure B-15c, the prime implicants from the previously constructed table (Figure B-15b) are not included. The process continues for additional reductions until only prime implicants remain. For this example, the process stops after the second reduction when the three terms become prime implicants as shown in Figure B-15c.

Taken as a whole, the prime implicants form a set that completely covers the function, although not necessarily minimally. In order to obtain a minimal covering set, a **table of choice** is constructed as shown in Figure B-16. Each prime

Prime Implicants	Minterms						
	0001	0011	0101	0110	0111	1010	1101
0 0 0 _	√						
* 0 1 1 _				√	√		
* 1 0 1 _						√	
0 __ 1	√	√	√		√		
_ _ 1 1		√			√		
* _ 1 _ 1			√		√		√

Figure B-16 Table of choice.

implicant has a row in the table of choice. The columns represent minterms in the original function that must be covered. Don't care conditions do not need to be covered, and are not listed.

A check is placed in each box that corresponds to a prime implicant that covers a

minterm. For example, prime implicant 000_ covers minterm 0001, so a check is placed in the corresponding box. Some prime implicants cover several minterms, as for 0__1 which covers four minterms. After all boxes are considered, columns that contain a single check are identified. A single check in a column means that only one prime implicant covers the minterm, and the corresponding prime implicant that covers the minterm is marked with an asterisk to indicate that it is **essential**.

Essential prime implicants cannot be eliminated, and they must be included in the reduced equation for the function. For this example, prime implicants 011_, 101_, and _1_1 are essential. An essential prime implicant may cover more than one minterm, and so a **reduced table of choice** is created in which the essential prime implicants and the minterms they cover are removed, as shown in Figure B-17. The reduced table of choice may also have essential prime implicants, in

Eligible Set	Minterms		Set 1	Set 2
	0001	0011	0 0 0 _	0 _ _ 1
X 0 0 0 _	✓		_ _ 1 1	
Y 0 _ _ 1	✓	✓		
Z _ _ 1 1		✓		

Figure B-17 Reduced table of choice.

which case a second reduced table of choice is created, and the process continues until the final reduced table of choice has only nonessential prime implicants.

The prime implicants that remain in the reduced table of choice form the **eligible set**, from which a minimal subset is obtained that covers the remaining minterms. As shown in Figure B-17, there are two sets of prime implicants that cover the two remaining minterms. Since Set 2 has the fewest terms, we choose that set and obtain a minimized equation for F which is made up of essential prime implicants and the eligible prime implicants in Set 2:

$$F = \bar{A}BC + A\bar{B}C + BD + \bar{A}D \quad (\text{B.19})$$

Instead of using visual inspection to obtain a covering set from the eligible set, the process can be carried out algorithmically. The process starts by assigning a variable to each of the prime implicants in the eligible set as shown in Figure

B-17. A logical expression is written for each column in the reduced table of choice as shown below:

Column	Logical Sums
0001	$(X + Y)$
0011	$(Y + Z)$

In order to find a set that completely covers the function, prime implicants are grouped so that there is at least one check in each column. This means that the following relation must hold, in which G represents the terms in the reduced table of choice:

$$G = (X + Y)(Y + Z)$$

Applying the properties of Boolean algebra yields:

$$G = (X + Y)(Y + Z) = XY + XZ + Y + YZ = XZ + Y$$

Each of the product terms in this equation represents a set of prime implicants that covers the terms in the reduced table of choice. The smallest product term (Y) represents the smallest set of prime implicants ($0 _ _ 1$) that covers the remaining terms. The same final equation is produced as before:

$$F = \bar{A}BC + A\bar{B}C + BD + \bar{A}D \quad (\text{B.20})$$

Reduction of Multiple Functions

The tabular reduction method reduces a single Boolean function. When there is more than one function that use the same variables, then it may be possible to share terms, resulting in a smaller collective size of the equations. The method described here forms an intersection among all possible combinations of shared terms, and then selects the smallest set that covers all of the functions.

As an example, consider the truth table shown in Figure B-18 that represents three functions in three variables. The notation m_i denotes minterms according to the indexing shown in the table.

The canonical (unreduced) form of the Boolean equations is:

Minterm	A	B	C	F_0	F_1	F_2
m_0	0	0	0	1	0	0
m_1	0	0	1	0	1	0
m_2	0	1	0	0	0	1
m_3	0	1	1	1	1	1
m_4	1	0	0	0	1	0
m_5	1	0	1	0	0	0
m_6	1	1	0	0	1	1
m_7	1	1	1	1	1	1

Figure B-18 A truth table for three functions in three variables.

$$F_0(A,B,C) = m_0 + m_3 + m_7$$

$$F_1(A,B,C) = m_1 + m_3 + m_4 + m_6 + m_7$$

$$F_2(A,B,C) = m_2 + m_3 + m_6 + m_7$$

An intersection is made for every combination of functions as shown below:

$$F_{0,1}(A,B,C) = m_3 + m_7$$

$$F_{0,2}(A,B,C) = m_3 + m_7$$

$$F_{1,2}(A,B,C) = m_3 + m_6 + m_7$$

$$F_{0,1,2}(A,B,C) = m_3 + m_7$$

Using the tabular reduction method described in the previous section, the following prime implicants are obtained:

Function	Prime Implicant
F_0	000, _11
F_1	0_1, 1_0, _11, 11_
F_2	_1_
$F_{0,1}$	_11
$F_{0,2}$	_11
$F_{1,2}$	_11, 11_

$$F_{0,1,2} \quad \quad \quad _11$$

The list of prime implicants is reduced by eliminating those prime implicants in functions that are covered by higher order functions. For example, $_11$ appears in $F_{0,1,2}$, thus it does not need to be included in the remaining functions. Similarly, $11_$ appears in $F_{1,2}$, and does not need to appear in F_1 or in F_2 (for this case, it does not appear as a prime implicant in F_2 anyway.) Continuing in this manner, a reduced set of prime implicants is obtained:

Function	Prime Implicant
F_0	000
F_1	0_1, 1_0
F_2	_1_
$F_{0,1}$	none
$F_{0,2}$	none
$F_{1,2}$	11_
$F_{0,1,2}$	_11

A multiple output table of choice is then constructed as shown in Figure B-19.

Prime Implicants \ Min-terms	$F_0(A,B,C)$			$F_1(A,B,C)$					$F_2(A,B,C)$			
	m_0	m_3	m_7	m_1	m_3	m_4	m_6	m_7	m_2	m_3	m_6	m_7
F_0 * 0 0 0	√											
F_1 * 0 _ 1				√	√							
F_1 * 1 _ 0						√	√					
F_2 * _ 1 _									√	√	√	√
$F_{1,2}$ 1 1 _							√	√			√	√
$F_{0,1,2}$ * _ 1 1		√	√		√			√		√		√

Figure B-19 A multiple output table of choice.

The rows correspond to the prime implicants, and the columns correspond to the minterms that must be covered for each function. Portions of rows are blocked out where prime implicants from one function cannot be used to cover another. For example, prime implicant 000 was obtained from function F_0 , and therefore cannot be used to cover a minterm in F_1 or F_2 , and so these regions are

blocked out. If, in fact, a prime implicant in F_0 can be used to cover a minterm in one of the remaining functions, then it will appear in a higher order function such as $F_{0,1}$ or $F_{0,1,2}$.

The minimal form for the output equations is obtained in a manner similar to the tabular reduction process. We start by finding all of the essential prime implicants. For example, minterm m_0 in function F_0 is covered only by prime implicant 000, and thus 000 is essential. The row containing 000 is then removed from the table and all columns that contain a check mark in the row are also deleted. The process continues until either all functions are covered or until only nonessential prime implicants remain, in which case the smallest set of nonessential prime implicants that are needed to cover the remaining functions is obtained using the method described in the previous section.

The essential prime implicants are marked with asterisks in Figure B-19. For this case, only one nonessential prime implicant (11_) remains, but since all minterms are covered by the essential prime implicants, there is no need to construct a reduced table. The corresponding reduced equations are:

$$F_0(A,B,C) = \bar{A}\bar{B}\bar{C} + BC$$

$$F_1(A,B,C) = \bar{A}C + A\bar{C} + BC$$

$$F_2(A,B,C) = B$$

B.2.4 LOGIC REDUCTION: EFFECT ON SPEED AND PERFORMANCE

Up to this point, we have largely ignored physical characteristics that affect performance, and have focused entirely on organizational issues such as circuit depth and gate count. In this section, we explore a few practical considerations of digital logic.

Switching speed: The propagation delay (latency) between the inputs and output of a logic gate is a continuous effect, even though we considered propagation delay to be negligible in the early part of Appendix A. A change at an input to a logic gate is also a continuous effect. In Figure B-20, an input to a NOT gate has a finite transition time, which is measured as the time between the 10% and 90% points on the waveform. This is referred to as the *rise time* for a rising signal and the *fall time* for a falling signal.

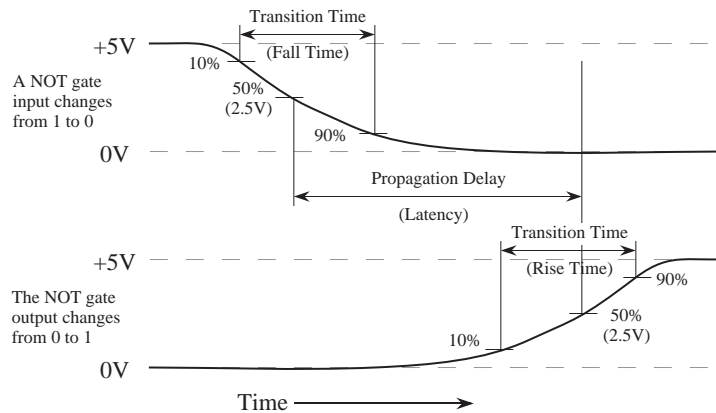


Figure B-20 Propagation delay for a NOT gate (adapted from [Hamacher *et al.*, 1990]).

The propagation delay is the time between the 50% transitions on the input and output waveforms. The propagation delay is influenced by a number of parameters, and power is one parameter over which we have a good deal of control. As power consumption increases, propagation delay decreases, up to a limit. A rule of thumb is that the product of power consumption and the propagation delay for a logic gate stays roughly the same. Although we generally want fast logic, we do not want to operate with a high power dissipation because the consumed power manifests itself as heat that must be removed to maintain a safe and reliable operating condition.

In the complementary metal-oxide semiconductor (CMOS) logic family, power dissipation scales with speed. At a switching rate of 1 MHz, the power dissipation of a CMOS gate is about 1 mW. At this rate of power dissipation, 10,000 CMOS logic gates dissipate $10,000 \text{ gates} \times 1 \text{ mW/gate} = 10 \text{ W}$, which is at the limit of heat removal for a single integrated circuit using conventional approaches (for a 1 cm^2 chip).

Single CMOS chips can have on the order of 10^7 logic gates, however, and operate at rates up to several hundred MHz. This gate count and speed are achieved partially by increasing the chip size, although this accounts for little more than a factor of 10. The key to achieving such a high component count and switching speed while managing power dissipation is to switch only a fraction of the logic gates at any time, which luckily is the most typical operating mode for an integrated circuit.

Circuit depth: The latency between the inputs and outputs of a circuit is governed

by the number of logic gates on the longest path from any input to any output. This is known as **circuit depth**. In general, a circuit with a small circuit depth operates more quickly than a circuit with a large circuit depth. There are a number of ways to reduce circuit depth that involve increasing the complexity of some other parameter. We look at one way of making this trade-off here.

In Appendix A, we used a MUX to implement the majority function. Now consider using a four variable MUX to implement Equation B.21 shown below. The equation is in two-level form, because only two levels of logic are used in its representation: six AND terms that are ORed together. A single MUX can implement this function, which is shown in the left side of Figure B-21. The

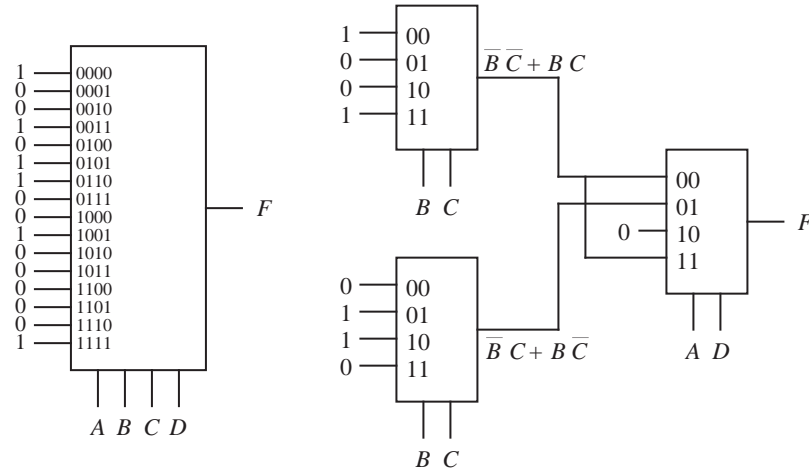


Figure B-21 A four-variable function implemented with a 16-to-1 MUX (left) and with 4-to-1 MUXes (right).

corresponding circuit depth is two (that is, the gate-level configuration of the inside of the MUX has two gate delays). If we factor out A and B then we obtain the four-level Equation B.22, and the corresponding four-level circuit shown in the right side of Figure B-21.

$$F(A, B, C, D) = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + A\bar{B}\bar{C}D + ABCD \quad (\text{B.21})$$

$$F(A, B, C, D) = \bar{A}\bar{B}(\bar{C}\bar{D} + CD) + \bar{A}B(\bar{C}D + C\bar{D}) + A\bar{B}(\bar{C}D) + AB(CD) \quad (\text{B.22})$$

The gate input count of a 4-to-1 MUX is 18 as taken from Figure A-23 (including inverters), so the gate input count of the decomposed MUX circuit is $3 \times 18 = 54$. A single 16-to-1 MUX has a gate input count of 100. The 4-to-1 MUX

implementation has a circuit depth of four (not including inverters) while the 16-to-1 MUX implementation has a circuit depth of two. We have thus reduced the overall circuit complexity at the expense of an increase in the circuit depth.

Although there are techniques that aid the circuit designer in discovering trade-offs between circuit complexity and circuit depth, the development of algorithms that cover the space of possible alternatives in reasonable time is only a partially solved problem.

Fan-in vs. circuit depth: Suppose that we need a four-input OR gate as used in Figure A-23, but only two-input OR gates are available. What should we do? This is a common practical problem that is encountered in a variety of design situations. The associative property of Boolean algebra can be used to decompose the OR gate that has a fan-in of four into a configuration of OR gates that each have a fan-in of two as shown in Figure B-22. In general, the decomposition of

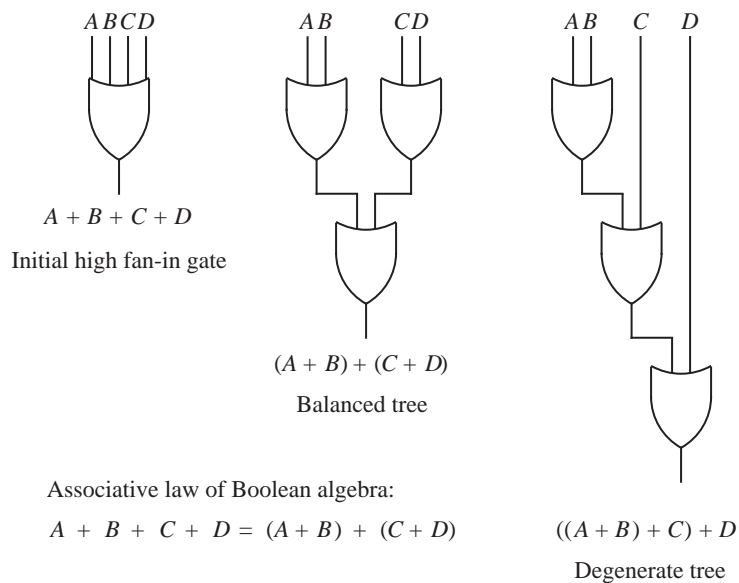


Figure B-22 A logic gate with a fan-in of four is decomposed into logically equivalent configurations of logic gates with fan-ins of two.

the four-input OR gate should be performed in balanced tree fashion in order to reduce circuit depth. A degenerate tree can also be used as shown in Figure B-22, which produces a functionally equivalent circuit with the same number of logic gates as the balanced tree, but results in a maximum circuit depth.

Although it is important to reduce circuit depth in order to decrease the latency between the inputs and the outputs, one reason for preferring the degenerate tree to the balanced tree is that the degenerate tree has a minimum cross sectional diameter at each stage, which makes it easy to split the tree into pieces that are spread over a number of separate circuits. This mirrors a practical situation encountered in packaging digital circuits. The depth of the balanced tree is $\lceil \log_F(N) \rceil$ logic gates for an N -input gate mapped to logic gates with a fan-in of F and the depth of the degenerate tree is $\lceil \frac{N-1}{F-1} \rceil$ logic gates for an N -input gate mapped to logic gates with a fan-in of F .

In theory, any binary function can be realized in two levels of logic gates given an arbitrarily large stage of AND gates followed by an arbitrarily large stage of OR gates, both having arbitrarily large fan-in and fan-out. For example, an entire computer program can be compiled in just two gate levels if it is presented in parallel to a Boolean circuit that has an AND stage followed by an OR stage that is designed to implement this function. Such a circuit would be prohibitively large, however, since every possible combination of inputs must be considered.

Fan-outs larger than about 10 are too costly to implement in many logic families due to the sacrifice in performance, as it is similar to filling 10 or more leaky buckets from a single faucet. Boolean algebra for two-level expressions is still used to describe complex digital circuits with high fan-outs, however, and then the two-level Boolean expressions are transformed into multilevel expressions that conform to the fan-in and fan-out limitations of the technology. Optimal fan-in and fan-out are argued to be $e \cong 2.7$ (Mead and Conway, 1980) in terms of transistor stepping size for bringing a signal from an integrated circuit (IC) to a pin of the IC package. The derivation of that result is based on capacitance of bonding pads, signal rise times, and other considerations. The result cannot be applied to all aspects of computing since it does not take into account overall performance, which may create local variations that violate the e rule dramatically. Electronic digital circuits typically use fan-ins and fan-outs of between 2 and 10.

B.3 State Reduction

In Appendix A, we explored a method of designing an FSM without considering that there may exist a functionally equivalent machine with fewer states. In this section, we focus on reducing the number of states. We begin with a description of an FSM that has some number of states, and then we hypothesize that a functionally equivalent machine exists that contains a single state. We apply all com-

binations of inputs to the hypothesized machine, and observe the outputs. If the FSM produces a different output for the same input combination at different times, then there are at least two states that are distinguishable, which are therefore not equivalent. The distinguishable states are placed in separate groups, and the process continues until no further distinctions can be made. If any remaining groups have more than one state, then those states are equivalent and a smaller equivalent machine can be constructed in which each group is collapsed into a single state.

As an example, consider state machine M_0 described by the state table shown in Figure B-23. We begin the reduction process by hypothesizing that all five states

Present state \ Input	X	
	0	1
A	C/0	E/1
B	D/0	E/1
C	C/1	B/0
D	C/1	A/0
E	A/0	C/1

Figure B-23 Description of state machine M_0 to be reduced.

can be reduced to a single state, obtaining partition P_0 for a new machine M_1 :

$$P_0 = (ABCDE)$$

We then apply a single input to the original machine M_0 and observe the outputs. When M_0 is in state A , and an input of 0 is applied, then the output is 0. When the machine is in state A and an input of 1 is applied, then the output is 1. States B and E behave similarly, but states C and D produce outputs of 1 and 0 for inputs of 0 and 1, respectively. Thus, we know that states A , B , and E can be distinguished from states C and D , and we obtain a new partition P_1 :

$$P_1 = (ABE) (CD)$$

After a single input is applied to M_0 , we know that the machine will be in either the ABE group or the CD group. We now need to observe the behavior of the machine from its new state. One way to do this is to enumerate the set of possible next states in a tree as shown in Figure B-24. The process of constructing the tree begins by listing all of the states in the same partition. For machine M_0 , the

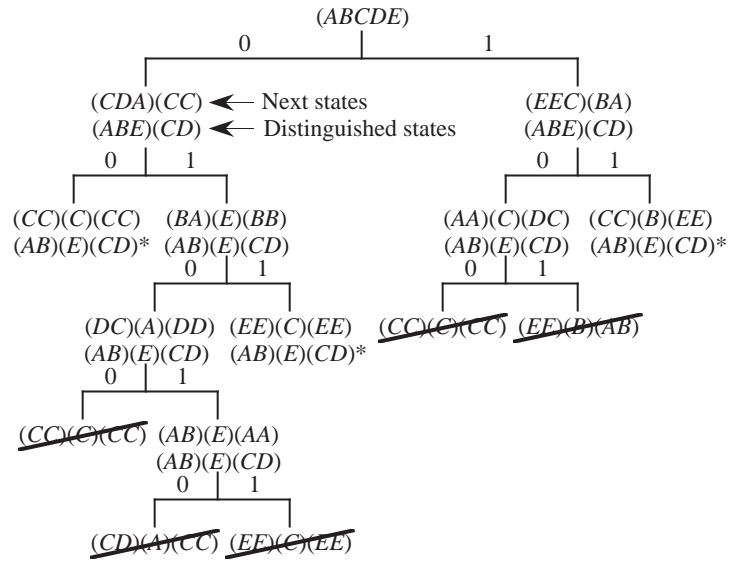


Figure B-24 A next state tree for M_0 .

initial partition $(ABCDE)$ is shown at the root of the tree. After a 0 is applied at the input to M_0 , the next state will be one of C, D, C, C , or A for an initial state of A, B, C, D , or E , respectively. This is shown as the $(CDA)(CC)$ partition in the 0 side of the tree, down one **ply** (one level) from the root. The output produced by the (CDA) group is different from the output produced by the (CC) group, and so their corresponding initial states are distinguishable. The corresponding states that are distinguished are the groups (ABE) and (CD) , which form the partition $(ABE)(CD)$ as shown.

Similarly, after a 1 is applied at the input to M_0 , the next state will be one of E, E, B, A , or C for an initial state of A, B, C, D , or E , respectively. This is shown on the right side of the tree. To form the next ply, we look at the (CDA) and (CC) groups separately. When a 0 is applied at the input when M_0 is in any of states C, D , or A , then the outputs will be the same for states C and D (the output is a 1, and the next states are C and C) but will be different for state A (the output is a 0, and the next state is C). This is shown as $(CC)(C)$ on the 0,0 path from the root.

Similarly, when a 0 is applied at the input when M_0 is in either of states C or D , then the outputs are the same, and the set of target states are $(CC)(C)(CC)$ on the 0,0 path from the root as shown, which corresponds to a partition on the initial

states of $(AB)(E)(CD)$ if we trace back to the root. Thus, at this point, A is indistinguishable from B , and C is indistinguishable from D , but each parenthesized group can be distinguished from each other if we apply the sequence 0,0 to M_0 and observe the outputs, regardless of the initial state.

Continuing in this manner, the tree is expanded until no finer partitions can be created. For example, when a partition contains a group of states that can no longer be distinguished, as for $(CC)(C)(CC)$, then an asterisk is placed adjacent to the partition for the corresponding initial states and the tree is not expanded further from that point. The tree shown in Figure B-24 is expanded beyond this point only to illustrate various situations that can arise.

If a partition is created that is visited elsewhere in the tree, then a slash is drawn through it and the tree is not expanded from that point. For purposes of comparing similar partitions, $(CD)(A)(CC)$ is considered to be the same as $(CD)(A)(C)$, and $(AA)(C)(DC)$ is considered to be the same as $(A)(C)(DC)$, which is the same as $(DC)(A)(C)$ and $(CD)(A)(C)$. Thus the $(CD)(A)(CC)$ and $(AA)(C)(DC)$ partitions are considered to be the same. After the tree is constructed, the partitions with asterisks expose the indistinguishable states. Each group of parentheses in an asterisk partition identifies a group of indistinguishable states. For machine M_0 , states A and B are indistinguishable, and states C and D are indistinguishable. Thus, we can construct a functionally equivalent machine to M_0 that contains only three states, in which A and B are combined into a single state and C and D are combined into a single state.

The process of constructing the next state tree is laborious because of its potential size, but we use it here in order to understand a simpler method. Rather than construct the entire tree, we can simply observe that once we have the first partition P_1 , the next partition can be constructed by looking at the next states for each group and noting that if two states within a group have next states that are in different groups, then they are distinguishable since the resulting outputs will eventually differ. This can be shown by constructing the corresponding distinguishing tree. Starting with P_1 for M_0 , we observe that states A and B have next states C and D for an input of 0, and have a next state of E for an input of 1, and so A and B are grouped together in the next partition. State E , however, has next states of A and C for inputs of 0 and 1, respectively, which differ from the next states for A and B , and thus state E is distinguishable from states A and B . Continuing for the (CD) group of P_1 , the next partition is obtained as shown below:

$$P_2 = (AB) (CD) (E)$$

After applying the method for another iteration, the partition repeats, which is a condition for stopping the process:

$$P_3 = (AB) (CD) (E) \checkmark$$

No further distinctions can be made at this point, and the resulting machine M_1 has three states in its reduced form. If we make the assignment $A' = AB$, $B' = CD$, and $C' = E$, in which the prime symbols mark the states for machine M_1 , then a reduced state table can be created as shown in Figure B-25.

Input Current state	X	
	0	1
$AB: A'$	$B'/0$	$C'/1$
$CD: B'$	$B'/1$	$A'/0$
$E: C'$	$A'/0$	$B'/1$

Figure B-25 A reduced state table for machine M_1 .

B.3.1 THE STATE ASSIGNMENT PROBLEM

It may be the case that different state assignments for the same machine result in different implementations. For example, consider machine M_2 shown in the left side of Figure B-26. Two different state assignments are shown. State assignment

Input P.S.	X	
	0	1
A	B/1 A/1	
B	C/0 D/1	
C	C/0 D/0	
D	B/1 A/0	

Machine M_2

Input S_0S_1	X	
	0	1
A: 00	01/1 00/1	
B: 01	10/0 11/1	
C: 10	10/0 11/0	
D: 11	01/1 00/0	

State assignment SA_0

Input S_0S_1	X	
	0	1
A: 00	01/1 00/1	
B: 01	11/0 10/1	
C: 11	11/0 10/0	
D: 10	01/1 00/0	

State assignment SA_1

Figure B-26 Two state assignments for machine M_2 .

SA_0 is a simple numerical ordering of $A \rightarrow 00$, $B \rightarrow 01$, $C \rightarrow 10$, and $D \rightarrow 11$. State assignment SA_1 is the same as SA_0 except that the assignments for C and D are interchanged. We consider an implementation of M_2 using state assignment SA_0 with AND, OR, and NOT gates, and apply K-map reduction to reduce the sizes of the equations. Figure B-27 shows the results of reducing the next state func-

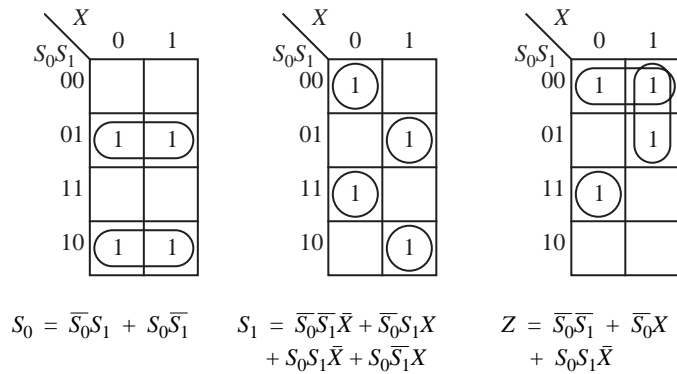


Figure B-27 Boolean equations for machine M_2 using state assignment SA_0 .

tions S_0 and S_1 and the output function Z . A corresponding circuit will have a gate input count of 29 as measured by counting the number of variables and the number of terms in the equations (note that one of the terms is shared, and is counted just once). If we use state assignment SA_1 instead, then we will obtain a gate input count of 6 as shown in Figure B-28. (s_0 and s_1 do not contribute to the

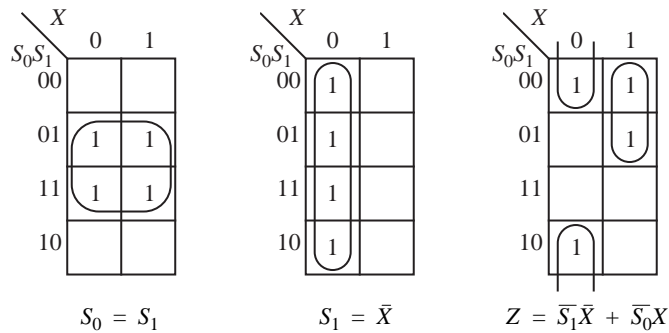


Figure B-28 Boolean equations for machine M_2 using state assignment SA_1 .

gate input count because they do not feed into logic gates.)

State assignment SA_1 is clearly better than SA_0 in terms of gate input count, but may not be better with regard to other criteria. For example, if an implementation is made with 8-to-1 MUXes, then the gate input count will be the same for SA_0 and SA_1 . A further consideration is that it is not an easy process to find the best assignment for any one criterion. In fact, better gate input counts may be possible by increasing the number of state bits for some cases.

REDUCTION EXAMPLE: A SEQUENCE DETECTOR

In this section, we tie together the reduction methods described in the previous sections. The machine we would like to design outputs a 1 when exactly two of the last three inputs are 1 (this machine appeared in an example in Appendix A). An input sequence of 011011100 produces an output sequence of 001111010. There is one serial input line, and we can assume that initially no inputs have been seen.

We start by constructing a state transition diagram, as shown in Figure B-29.

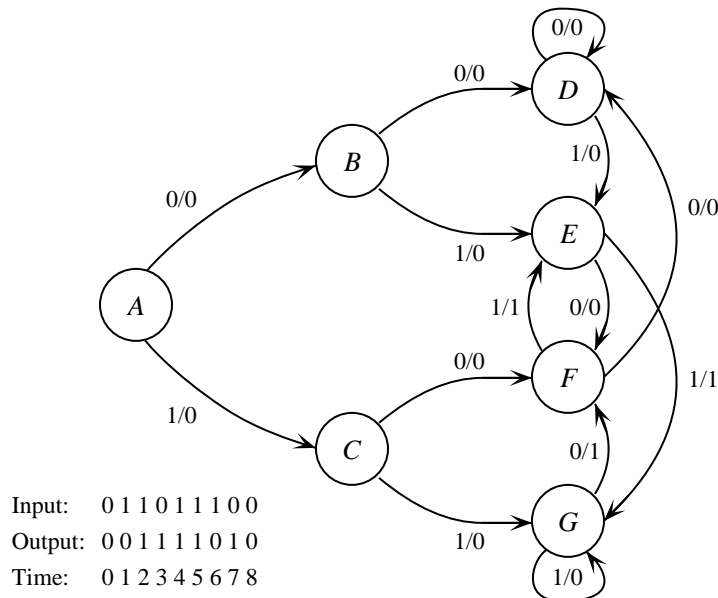


Figure B-29 State transition diagram for sequence detector.

There are eight possible three-bit sequences that our machine will observe: 000, 001, 010, 011, 100, 101, 110, and 111. State A is the initial state, in which we assume that no inputs have yet been seen. In states B and C, we have seen only one input, so we cannot yet output a 1. In states D, E, F, and G we have only seen two inputs, so we cannot yet output a 1, even though we have seen two 1's at the input when we enter state G. The machine makes all subsequent transitions among states D, E, F, and G. State D is visited when the last two inputs are 00. States E, F, and G are visited when the last two inputs are 01, 10, or 11,

respectively.

The next step is to create a state table and reduce the number of states. The state table shown in Figure B-30 is taken directly from the state transition diagram.

Present state \ Input	X	
	0	1
A	B/0	C/0
B	D/0	E/0
C	F/0	G/0
D	D/0	E/0
E	F/0	G/1
F	D/0	E/1
G	F/1	G/0

Figure B-30 State table for sequence detector.

We then apply the state reduction technique by hypothesizing that all states are equivalent, and then refining our hypothesis. The process is shown below:

$$P_0 = (ABCDEFGG)$$

$$P_1 = (ABCD) (EF) (G)$$

$$P_2 = (A) (BD) (C) (E) (F) (G)$$

$$P_3 = (A) (BD) (C) (E) (F) (G) \checkmark$$

States *B* and *D* along the 0,0,0 path in the state transition diagram are equivalent. We create a reduced table, using primed letters to denote the new states as shown in Figure B-31.

Next, we make an arbitrary state assignment as shown in Figure B-32. We then use the state assignment to create K-maps for the next state and output functions as shown in Figure B-33. Notice that there are four don't care conditions that arise because the 110 and 111 state assignments are unused. Finally, we create the gate-level circuit, which is shown in Figure B-34. ■

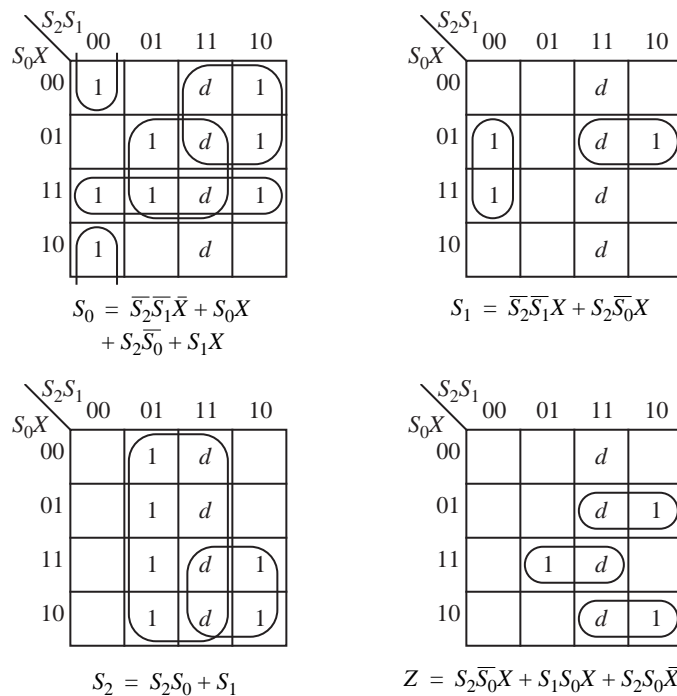


Figure B-33 K-map reduction of next state and output functions for sequence detector.

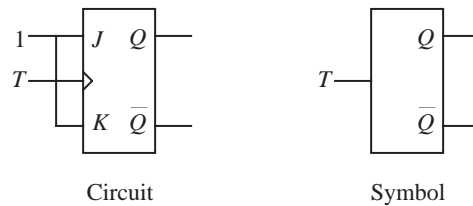


Figure B-36 Logic diagram and symbol for a T flip-flop.

All of the flip-flops discussed up to this point as well as several variations are available as separate components, and in the past designers would choose one form or the other depending on characteristics such as cost, performance, availability, *etc.* These days, with the development of VLSI technology, the D flip-flop is typically used throughout a circuit. High speed circuits making use of low density logic, however, such as gallium arsenide (GaAs), may still find an application for the various forms. For situations such as this, we consider the problem of choosing a flip-flop that minimizes the total number of components in a circuit, in which a flip-flop is considered to be a single component.

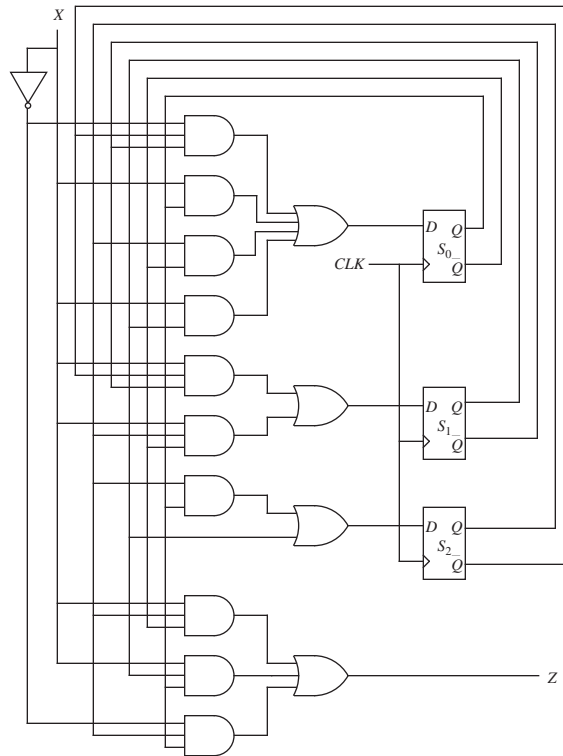


Figure B-34 Gate-level implementation of sequence detector.

The four flip-flops discussed up to this point can be described by **excitation tables**, as shown in Figure B-37. Each table shows the settings that must be applied at the inputs at time t in order to change the output at time $t+1$.

As an example of how excitation tables are used in the design of a finite state machine, consider using a J-K flip-flop in the design of a serial adder. We start with the serial adder shown in Figure B-38. State *A* represents the case in which there is no carry from the previous time step, and state *B* represents the case in which there is a carry from the previous time step.

We then create a truth table for the appropriate flip-flop. Figure B-39 shows a truth table that specifies functions for D, S-R, T, and J-K flip-flops as well as the output Z. We will only make use of the functions for the J-K flip-flop and the Z output here.

The way we construct the truth table is by first observing what the current state

	Q_t	Q_{t+1}	S	R
$S\text{-}R$ flip-flop	0	0	0	0
	0	1	1	0
	1	0	0	1
	1	1	0	0

	Q_t	Q_{t+1}	D
D flip-flop	0	0	0
	0	1	1
	1	0	0
	1	1	1

	Q_t	Q_{t+1}	J	K
$J\text{-}K$ flip-flop	0	0	0	d
	0	1	1	d
	1	0	d	1
	1	1	d	0

	Q_t	Q_{t+1}	T
T flip-flop	0	0	0
	0	1	1
	1	0	1
	1	1	0

Figure B-37 Excitation tables for four flip-flops.

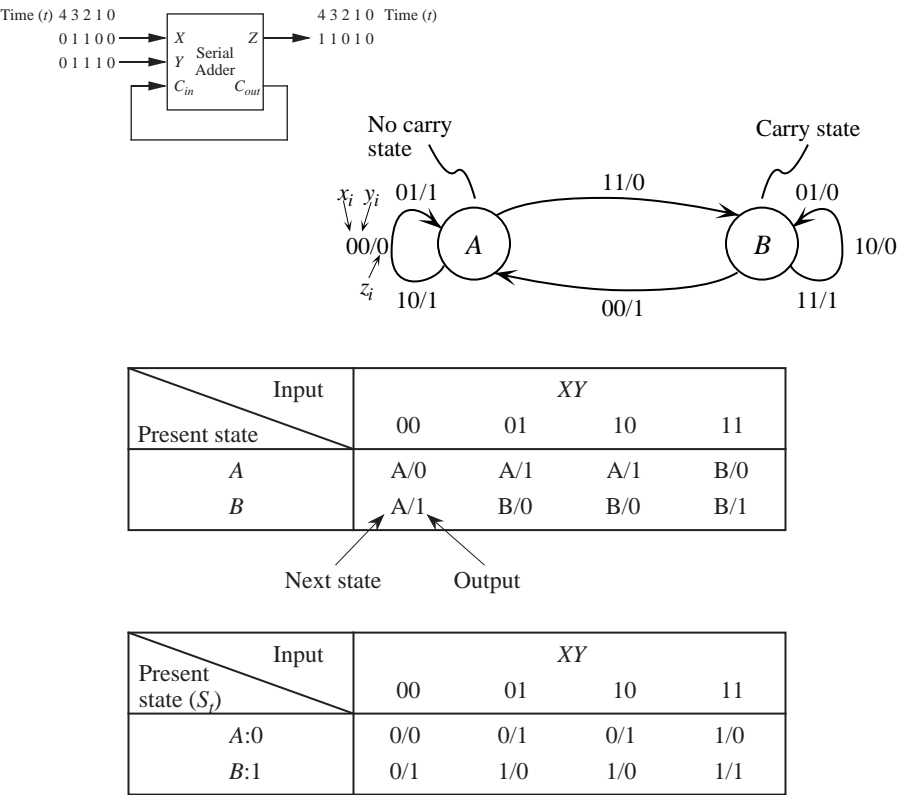


Figure B-38 State transition diagram, state table, and state assignment for a serial adder.

Present State			(Set) (Reset)						
X	Y	S_t	D	S	R	T	J	K	Z
0	0	0	0	0	0	0	0	d	0
0	0	1	0	0	1	1	d	1	1
0	1	0	0	0	0	0	0	d	1
0	1	1	1	0	0	0	d	0	0
1	0	0	0	0	0	0	0	d	1
1	0	1	1	0	0	0	d	0	0
1	1	0	1	1	0	1	1	d	0
1	1	1	1	0	0	0	d	0	1

Figure B-39 Truth table showing next-state functions for a serial adder for D, S-R, T, and J-K flip-flops. Shaded functions are used in the example.

S_t is, and then comparing it to what we want the next state to be. We then use the excitation tables to set the flip-flop inputs accordingly. For example, in the first line of the truth table shown in Figure B-39, when $X=Y=0$ and the current state is 0, then the next state must be 0 as read from the state table of Figure B-38. In order to achieve this for a J-K flip-flop, the J and K inputs must be 0 and d , respectively, as read from the J-K excitation table in Figure B-39. Continuing in this manner, the truth table is completed and the reduced Boolean equations are obtained as shown below:

$$\begin{aligned}
 J &= XY \\
 K &= \bar{X}\bar{Y} \\
 Z &= \bar{X}\bar{Y}S + \bar{X}Y\bar{S} + XYS + X\bar{Y}\bar{S}
 \end{aligned}$$

The corresponding circuit is shown in Figure B-40. Notice that the design has a small gate input count (20), as compared with a gate input count of 25 for the logically equivalent circuit shown in Figure B-41 which uses a D flip-flop. Flip-flops are not included in the gate input count, although they do contribute to circuit complexity.

EXCITATION TABLE EXAMPLE: A MAJORITY CHECKER

For this example, we would like to design a circuit using T flip-flops and 8-to-1 MUXes that computes the majority function (see Figure A-15) for three inputs that are presented to an FSM in serial fashion. The circuit outputs a 0 until the

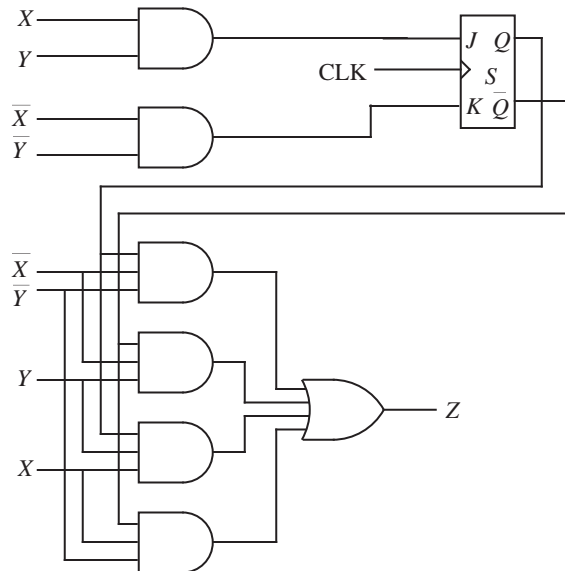


Figure B-40 Logic design for a serial adder using a J-K flip-flop.

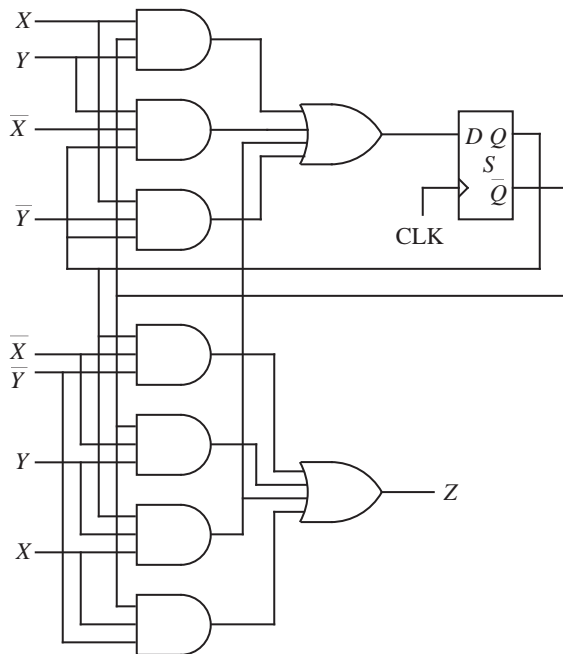


Figure B-41 Logic design for a serial adder using a D flip-flop.

third input is seen, at which point a 0 or a 1 is produced at the output according to whether there are more 0's or 1's at the input, respectively. For example, an input of 011100101 produces an output of 001000001.

We start by creating a state transition diagram that enumerates all possible states of the FSM. In Figure B-42, a state transition diagram is shown in which the states are organized according to the number of inputs that have been seen. State *A* is the initial state in which no inputs have yet been seen. The three inputs are symbolized with the notation: $_ _ _$. After the first input is seen, the FSM makes a transition to state *B* or state *C* for an input of 0 or 1, respectively. The input history is symbolized with the notation: $0_ _$ and $1_ _$ for states *B* and *C*, respectively. States *D*, *E*, *F* and *G* enumerate all possible histories for two inputs, as symbolized by the notation: $00_$, $01_$, $10_$, and $11_$, respectively.

The FSM outputs a 0 when making transitions to states *B* through *G*. On the third input, the FSM returns to state *A* and outputs a 0 or a 1 according to the majority function. A total of eight states are used in the FSM of Figure B-42,

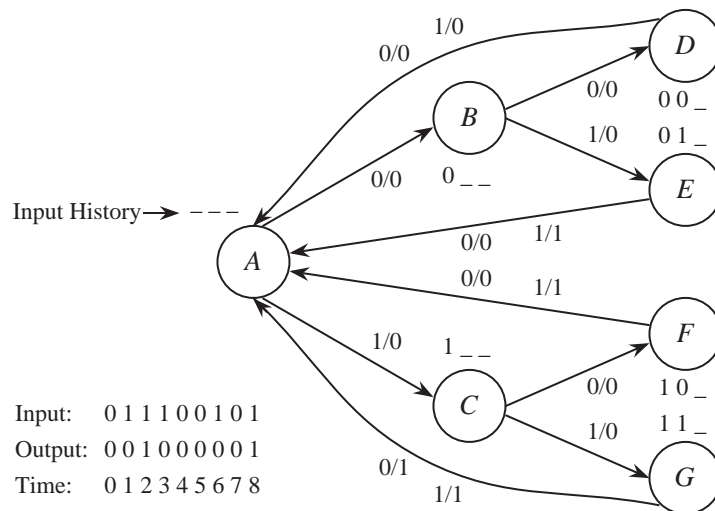


Figure B-42 State transition diagram for a majority FSM.

which are summarized in the state table shown in Figure B-43a.

The eight state FSM can be reduced to a seven state FSM. The reduction process is shown in Figure B-43b. States *E* and *F* can be combined, as shown in the reduced table of Figure B-43c. We use the reduced state table in creating a state

Input		X	
P.S.		0	1
A	B/0	C/0	
B	D/0	E/0	
C	F/0	G/0	
D	A/0	A/0	
E	A/0	A/1	
F	A/0	A/1	
G	A/1	A/1	

$P_0 = (ABCDEFGG)$
 $P_1 = (ABCD)(EF)(G)$
 $P_2 = (AD)(B)(C)(EF)(G)$
 $P_3 = (A)(B)(C)(D)(EF)(G)$
 $P_4 = (A)(B)(C)(D)(EF)(G) \checkmark$

Input		X	
P.S.		0	1
A: A'	B'/0	C'/0	
B: B'	D'/0	E'/0	
C: C'	E'/0	F'/0	
D: D'	A'/0	A'/0	
EF: E'	A'/0	A'/1	
G: F'	A'/1	A'/1	

(a)
(b)
(c)

Figure B-43 (a) State table for majority FSM; (b) partitioning; (c) reduced state table.

assignment, which is shown in Figure B-44a for D flip-flops. We want to use T flip-flops, and keeping the same state assignment, obtain the state table shown in Figure B-44b. The T flip-flop version is obtained by comparing the current state

Input		X	
P.S.		0	1
$S_2S_1S_0$	$S_2S_1S_0Z$	$S_2S_1S_0Z$	
A': 000	001/0	010/0	
B': 001	011/0	100/0	
C': 010	100/0	101/0	
D': 011	000/0	000/0	
E': 100	000/0	000/1	
F': 101	000/1	000/1	

$S_2S_1S_0$
A': 000
B': 001
C': 010
D': 011
E': 100
F': 101

Input		X	
P.S.		0	1
$T_2T_1T_0Z$	$T_2T_1T_0Z$	$T_2T_1T_0Z$	
A': 000	001/0	010/0	
B': 001	000/0	010/0	
C': 010	110/0	111/0	
D': 011	011/0	011/0	
E': 100	100/0	100/1	
F': 101	101/1	101/1	

(a)
(b)

Figure B-44 (a) State assignment for reduced majority FSM using D flip-flops; and (b) using T

with the next state, and following the excitation mapping for a T flip-flop shown in Figure B-37. The T flip-flop version has a 0 for the next state when the current and next states are the same in the D flip-flop version, and has a 1 if the current and next states differ in the D flip-flop version. There are three bits used for the binary coding of each state, and so there are three next state functions (s_0 , s_1 , and s_2) and an output function Z . The corresponding circuit using T flip-flops is shown in Figure B-45. Zeros are used for don't care states 110 and 111. ■

■ SUMMARY

Circuits that are generated from unreduced expressions may become very large,

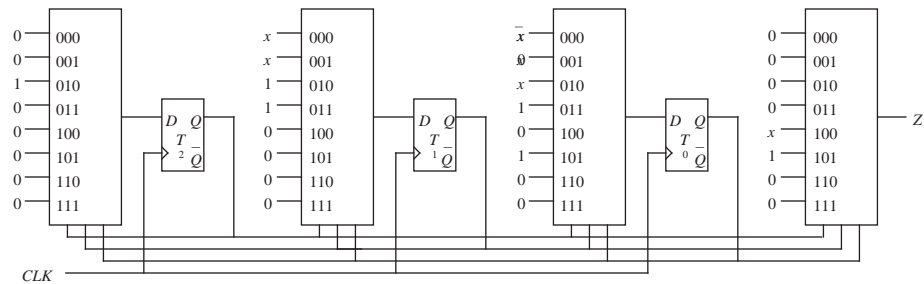


Figure B-45 Logic circuit for majority FSM.

and so the expressions are reduced when possible into logically equivalent smaller expressions. One method of reducing expressions is to perform algebraic manipulation using the properties of Boolean algebra. This approach is powerful but involves trial and error, and is tedious to carry out by hand. A simpler method is to apply K-map minimization. This is a more visual technique, but becomes difficult to carry out for more than about six variables. The tabular method lends itself to automation, and allows terms to be shared among functions.

An FSM can be in only one of a finite number of states at any time, but there are infinitely many FSMs that have the same external behavior. The number of flip-flops that are needed for an FSM may be reduced through the process of state reduction, and the complexity of the combinational logic in the FSM may be reduced by choosing an appropriate state assignment. The choice of flip-flop types also influences the complexity of the resulting circuit. The D flip-flop is commonly used for FSMs, but other flip-flops can be used such as the S-R, J-K, and T flip-flops.

■ FURTHER READING

(Booth, 1984) gives a good explanation of the Quine-McCluskey reduction process. (Kohavi, 1978) provides a thorough treatment of combinational logic reduction and state reduction. (Agrawal and Cheng, 1990) cover design for testability based on state assignments.

Agrawal, V.D. and K. T. Cheng, "Finite State Machine Synthesis with Embedded Test Function," *Journal of Electronic Testing: Theory and Applications*, vol. 1, pp. 221–228, (1990).

Booth, T. L., *Introduction to Computer Engineering: Hardware and Software*

Design, 3/e, John Wiley & Sons, New York, (1984).

Kohavi, Z., *Switching and Finite Automata Theory*, 2/e, McGraw-Hill, New York, (1978).

■ PROBLEMS

B.1 Given the following functions, construct K-maps and find minimal sum-of-products expressions for f and g .

$f(A, B, C, D) = 1$ when two or more inputs are 1, otherwise $f(A, B, C, D) = 0$.

$g(A, B, C, D) = 1$ when the number of inputs that are 1 is even (including the case when no inputs are 1), otherwise $g(A, B, C, D) = \overline{f(A, B, C, D)}$.

B.2 Use K-maps to simplify function f and its don't care condition below. Perform the reduction for (a) the sum-of-products form and (b) the product-of-sums form.

$$f(A, B, C, D) = \sum (2, 8, 10, 11) + \sum_d (0, 9)$$

B.3 Given a logic circuit, is it possible to generate a truth table that contains don't cares? Explain your answer.

B.4 The following K-map is formed incorrectly. Show the reduced equation that is produced by the incorrect map, and then form the K-map correctly and derive the reduced equation from the correct map. Note that both K-maps will produce functionally correct equations, but only the properly formed K-map will produce a minimized two-level equation.

		ABC							
		000	001	011	010	110	111	101	100
D	0	1			1	1			1
	1	1			1	1			1

B.5 A 4-to-1 multiplexer can be represented by the truth table shown below.

Use map-entered variables to produce a reduced SOP Boolean equation.

A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

B.6 Use the tabular method to reduce the function:

$$f(A, B, C, D) = \sum (3, 5, 7, 10, 13, 15) + \sum_d (2, 6)$$

B.7 Use the tabular method to reduce the following multiple output truth table:

Minterm	A	B	C	D	F_0	F_1	F_2
m_0	0	0	0	0	0	0	1
m_1	0	0	0	1	0	0	0
m_2	0	0	1	0	0	0	0
m_3	0	0	1	1	1	0	0
m_4	0	1	0	0	0	0	1
m_5	0	1	0	1	1	1	0
m_6	0	1	1	0	0	0	0
m_7	0	1	1	1	1	1	0
m_8	1	0	0	0	0	0	0
m_9	1	0	0	1	0	0	0
m_{10}	1	0	1	0	0	1	1
m_{11}	1	0	1	1	0	0	0
m_{12}	1	1	0	0	0	0	0
m_{13}	1	1	0	1	1	1	0
m_{14}	1	1	1	0	1	1	1
m_{15}	1	1	1	1	1	1	1

B.8 Reduce the equation for F shown below to its minimal two-level form, and implement the function using a three-input, one-output PLA.

$$F(A, B, C) = ABC + \bar{A}BC + A\bar{B}\bar{C} + \bar{A}\bar{B}C$$

B.9 Use function decomposition to implement function f below with two

4-to-1 MUXes. Parenthesize the equation so that C and D are in the inner-most level as in Equation B.22. Make sure that every input to each MUX is assigned to a value (0 or 1), to a variable, or to a function.

$$f(A, B, C, D) = ABCD + AB\bar{C}D + ABC\bar{D} + \bar{A}B$$

B.10 Reduce the following state table:

Input Present state	X	
	0	1
A	$D/0$	$G/1$
B	$C/0$	$G/0$
C	$A/0$	$D/1$
D	$B/0$	$C/1$
E	$A/1$	$E/0$
F	$C/1$	$F/0$
G	$E/1$	$G/1$

B.11 Reduce the following state table:

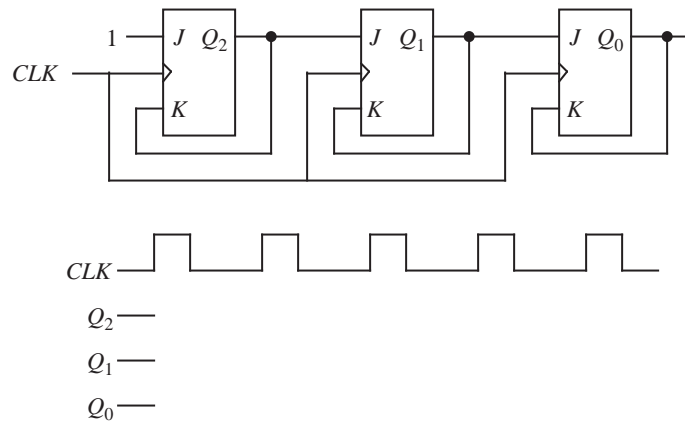
Input Present state	XY			
	00	01	10	11
A	$A/0$	$B/0$	$C/0$	$D/0$
B	$A/0$	$B/1$	$D/0$	$D/1$
C	$E/1$	$B/0$	$B/0$	$E/1$
D	$A/0$	$D/1$	$D/0$	$B/1$
E	$C/1$	$D/0$	$D/0$	$E/1$

B.12 The following ternary state table may or may not reduce. Show the reduc-

tion process (the partitions) and the reduced state table.

Present state \ Input	x		
	0	1	2
A	$B/0$	$E/2$	$G/1$
B	$D/2$	$A/1$	$D/0$
C	$D/2$	$G/1$	$B/0$
D	$B/2$	$F/1$	$C/0$
E	$A/0$	$E/2$	$C/1$
F	$C/0$	$E/2$	$F/1$
G	$D/0$	$E/2$	$A/1$

B.13 The following circuit has three master-slave J-K flip-flops. There is a single input CLK and three outputs Q_2 , Q_1 , and Q_0 that initially have the value 0. Complete the timing diagram by showing the values for Q_2 , Q_1 , and Q_0 . Assume that there are no delays through the flip-flops.



B.14 Use a T flip-flop to design a serial adder, using the approach described in Section B.3.2.

B.15 In the following reduced state table, the state assignments have already been made. Design the machine using D flip-flops, AND and OR gates. Use K-maps to reduce the expressions for the next state and output functions. Be careful to construct the K-maps correctly, since there are only three rows in

the state table.

Present state \ Input	X	
	0	1
YZ		
A: 00	00/0	01/1
B: 01	10/1	00/1
C: 10	01/1	10/0

B.16 Draw a logic diagram that shows a J-K flip-flop can be created using a D flip-flop.

