Asynchronous Circuit Design. Chris J. Myers
Copyright © 2001 by John Wiley & Sons, Inc.
ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

5

Huffman Circuits

Never do today what you can put off till tomorrow. Delay may give clearer light as to what is best to be done.

—Aaron Burr

Delay is preferable to error.

—Thomas Jefferson

Feedback is your friend.

—Professor Doyle at my Caltech Frosh Camp

In this chapter we introduce the Huffman school of thought to the synthesis of asynchronous circuits. *Huffman circuits* are designed using a traditional asynchronous state machine approach. As depicted in Figure 5.1, an asynchronous state machine has primary inputs, primary outputs, and fed-back state variables. The state is stored in the feedback loops and thus may need delay elements along the feedback path to prevent state changes from occurring too rapidly. The design of Huffman circuits begins with a specification given in a flow table which may have been derived from an AFSM, BM, or XBM machine. The goal of the synthesis procedure is to derive a correct circuit netlist which has been optimized according to design criteria such as area, speed, or power. The approach taken for the synthesis of synchronous state machines is to divide the synthesis problem into three steps. The first step is *state minimization*, in which compatible states are merged to produce a simpler flow table. The second step is *state assignment* in which a binary encoding is assigned to each state. The third step is *logic minimization* in

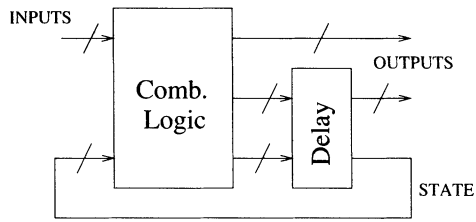


Fig. 5.1 Huffman circuit model.

which an optimized netlist is derived from an encoded flow table. The design of Huffman circuits can follow the same three-step process, but each step must be modified to produce correct circuits under an asynchronous timing model.

Huffman circuits are typically designed under the *bounded gate and wire delay* model. Under this model, circuits are guaranteed to work regardless of gate and wire delays as long as a bound on the delays is known. In order to design correct Huffman circuits, it is also necessary to put some constraints on the behavior of the environment, namely when inputs are allowed to change. There are a number of different restrictions on inputs that have been proposed, each resulting in variations in the synthesis procedure. The first is *single-input change* (SIC), which states that only one input is allowed to change at a time. In other words, each input change must be separated by a minimum time interval. If this minimum time interval is set to be the maximum delay for the circuit to stabilize, the restriction is called single-input change *fundamental mode.* This is quite restrictive, though, so another approach is to allow *multiple-input changes* (MIC). Again, if input changes are allowed only after the circuit stabilizes, this mode of operation is called multiple-input change fundamental mode. We first introduce each of the synthesis steps under the single-input change fundamental mode restriction, and later the synthesis methods are extended to support a limited form of multiple-input changes: extended burst mode.

5.1 SOLVING COVERING PROBLEMS

The last step of state minimization, state assignment, and logic synthesis is to solve a *covering problem.* For this reason, we begin this chapter by describing an algorithm for solving covering problems. Informally, a covering problem exists whenever you must select a set of choices with minimum cost which satisfy a set of constraints. The classic example is when deriving a minimum two-level sum-of-products cover of a logic function, one must choose the minimum number of prime implicants that cover all the minterms of a given function.

More formally, the set of choices is represented using a Boolean vector, $\mathbf{x} = (x_1, \ldots, x_n)$. If a Boolean variable, x_i, is set to 1, this indicates that this choice has been included in the solution. When $x_i = 0$, this indicates that this choice has not been included in the solution. A covering problem can now be expressed as a product-of-sums f where each product (or *clause*) represents a constraint that must be satisfied. Each clause is the sum of those choices that would satisfy the constraint. We are also given a cost function:

$$\text{cost}(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i \tag{5.1}$$

where n is the number of choices and w_i is the cost of each choice. The goal now is to solve the covering problem by finding the assignment of the x_i's which results in the minimum cost.

Example 5.1.1 An example product-of-sums formulation of a covering problem is shown below. Note that any solution must include x_1, must not include x_2, must either not include x_3 or include x_4, etc.

$$f = x_1\overline{x_2}(\overline{x_3}+x_4)(\overline{x_3}+x_4+x_5+x_6)(\overline{x_1}+x_4+x_5+x_6)(\overline{x_4}+x_1+x_6)(\overline{x_5}+x_6)$$

When choices appear only in their positive form (i.e., uncomplemented), it is a *unate covering problem.* When any choice appears in both its positive and negative form (i.e., complemented), it is a *binate covering problem.* In this section we consider the more general case of the binate covering problem, but the solution clearly applies to both.

The function f is represented using a *constraint matrix*, $\mathbf{A}$, which includes a column for each x_i variable and a row for every clause. Each entry of the matrix a_{ij} is "–" if the variable x_i does not appear in the clause, "0" if it appears complemented, and "1" if it appears uncomplemented. The ith row of $\mathbf{A}$ is denoted a_i while the jth column is denoted by A_j.

Example 5.1.2 The constraint matrix for the covering problem from Example 5.1.1 is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \begin{bmatrix} 1 & - & - & - & - & - \\ - & 0 & - & - & - & - \\ - & - & 0 & 1 & - & - \\ - & - & 0 & 1 & 1 & 1 \\ 0 & - & - & 1 & 1 & 1 \\ 1 & - & - & 0 & - & 1 \\ - & - & - & - & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix}$$

The binate covering problem is to find an assignment to the Boolean variables, $\mathbf{x}$, of minimum cost such that for every row a_i either

1. $\exists j \;.\; (a_{ij} = 1) \wedge (x_j = 1)$; or
2. $\exists j \;.\; (a_{ij} = 0) \wedge (x_j = 0)$.

```
bcp(A, x, b) {
  (A, x) = reduce(A, x);    /* Find essentials / apply dominance.*/
  L = lower_bound(A, x);  /* Compute lower bound for this matrix.*/
  if (L ≥ cost(b)) then return(b);        /* Check against best.*/
  if (terminalCase(A)) then { /* Check if solved or infeasible.*/
    if (A has no rows) return(x);    /* Matrix is fully reduced.*/
    else return(b);          /* Matrix is infeasible return best.*/
  }
  c = choose_column(A);              /* Select column to branch on.*/
  x_c = 1;                  /* Include selected column in solution.*/
  A^1 = select_column(A, c);                  /* Use c in solution.*/
  x^1 = bcp(A^1, x, b)                  /* Recurse with c in solution.*/
  if (cost(x^1)<cost(b)) then {           /* If better record it.*/
    b = x^1;
    if (cost(b)= L) return(b);         /* If lower bound, return.*/
  }
  x_c = 0;            /* Exclude selected column from the solution.*/
  A^0 = remove_column(A, c);          /* Do not use c in solution.*/
  x^0 = bcp(A^0, x, b)              /* Recurse with c not in solution.*/
  if (cost(x^0)<cost(b)) then b = x^0;    /* If better, record it.*/
  return(b);
}
```

Fig. 5.2 Branch-and-bound algorithm for binate covering.

A branch-and-bound algorithm for solving the binate covering problem is shown in Figure 5.2. It takes as input a constraint matrix, $\mathbf{A}$, the current partial solution vector $\mathbf{x}$, and the best solution found so far, $\mathbf{b}$. The *bcp* algorithm is invoked with $\mathbf{x} = \mathbf{0}$ and $\mathbf{b} = no\_solution$. The cost of *no_solution* is defined to be infinite. The *bcp* algorithm returns a vector representing a minimal cover. In the rest of this section we describe this algorithm in detail.

5.1.1 Matrix Reduction Techniques

The first step of the *bcp* algorithm is to reduce the matrix using a number of reduction techniques, shown in Figure 5.3. The first deals with *essential rows*. A row a_i of $\mathbf{A}$ is essential when there exists exactly one j such that a_{ij} is not equal to "–". This corresponds to a clause which consists of only a single literal. If the literal is x_j (i.e., $a_{ij} = 1$), the variable is *essential*, and the solution $\mathbf{x}$ is updated such that $x_j = 1$. The matrix $\mathbf{A}$ is reduced by removing the column corresponding to the essential literal, A_j, and all rows a_k which are solved by selecting this column (i.e., $a_{kj} = 1$). If the literal is $\overline{x_j}$ (i.e., $a_{ij} = 0$), the variable is *unacceptable*. The matrix $\mathbf{A}$ is reduced by removing the column corresponding to the unacceptable literal, A_j, and all rows a_k which are solved by excluding this column (i.e., $a_{kj} = 0$). In other

```
reduce(A,x) {
  do {
    A' = A;
    (A,x) = find_essential_rows(A,x);
    A = delete_dominating_rows(A);
    (A,x) = delete_dominated_columns(A,x);
  } while (A ≠ ∅ and A ≠ A');  /* Repeat until no improvement.*/
  return(A,x);
}
```

Fig. 5.3 Reduce algorithm.

words, the variable is set to the value of the literal, the column corresponding to the variable is removed, and any row where the variable has the same value is removed.

Example 5.1.3 The constraint matrix shown in Example 5.1.2 has two essential rows, rows 1 and 2. Row 1 implies that x_1 is an essential variable and must be set to 1. Setting x_1 to 1 allows us to remove rows 1 and 6. The new constraint matrix after x_1 is selected is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_2 & x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \left[\begin{array}{ccccc} 0 & - & - & - & - \\ - & 0 & 1 & - & - \\ - & 0 & 1 & 1 & 1 \\ - & - & 1 & 1 & 1 \\ - & - & - & 0 & 1 \end{array}\right] \end{array} \begin{matrix} \\ 2 \\ 3 \\ 4 \\ 5 \\ 7 \end{matrix}$$

Row 2 implies that x_2 is an unacceptable variable and must be set to 0. The new constraint matrix after x_2 is excluded is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \left[\begin{array}{cccc} 0 & 1 & - & - \\ 0 & 1 & 1 & 1 \\ - & 1 & 1 & 1 \\ - & - & 0 & 1 \end{array}\right] \end{array} \begin{matrix} \\ 3 \\ 4 \\ 5 \\ 7 \end{matrix}$$

The second reduction is *row dominance.* A row a_k dominates another row a_i if it has all the 1's and 0's of a_i. In other words, a row a_k dominates another row a_i if for each column A_j of $\mathbf{A}$, one of the following is true:

- $a_{ij} = -$
- $a_{ij} = a_{kj}$

Dominating rows can be removed without affecting the set of solutions.

Example 5.1.4 In the matrix from Example 5.1.3, row 4 dominates row 3, so row 4 can be removed. The resulting constraint matrix is:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \begin{bmatrix} 0 & 1 & - & - \\ - & 1 & 1 & 1 \\ - & - & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} \\ 3 \\ 5 \\ 7 \end{matrix}$$

The third reduction is *column dominance*. A column A_j dominates another column A_k if for each clause a_i of A, one of the following is true:

- $a_{ij} = 1$.
- $a_{ij} = -$ and $a_{ik} \neq 1$.
- $a_{ij} = 0$ and $a_{ik} = 0$.

Dominated columns can be removed without affecting the existence of a solution. Note that when you remove a column, the associated variable is set to 0. This means that if any rows include that column with a 0 entry, they can be removed as well.

Example 5.1.5 In the matrix from Example 5.1.4, column x_6 dominates columns x_3 and x_5. Note that by eliminating the column associated with x_3 and x_5, rows 3 and 7 are also removed. The new matrix is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_4 & x_6 \end{matrix} \\ \begin{bmatrix} 1 & 1 \end{bmatrix} \end{array} \begin{matrix} \\ 5 \end{matrix}$$

At this point, either x_4 or x_6 can be selected to solve the covering problem. Therefore, there are two possible minimal solutions: either $x_1 = x_4 = 1$ with $x_2 = x_3 = x_5 = x_6 = 0$ or $x_1 = x_6 = 1$ with $x_2 = x_3 = x_4 = x_5 = 0$. If either assignment is plugged into the initial product-of-sums given above, it is clear that they yield a 1 for f.

Assuming that an assignment of 1 to any variable has weight 1, then either solution found in our example is a minimum-cost solution. If the weights are not equal, it is necessary to also check the weights of the columns before removing dominated columns. Namely, if the weight of the dominating column, w_j, is greater than the weight of the dominated column, w_k, then x_k should not be removed.

Example 5.1.6 Consider the constraint matrix below with $w_1 = 3$, $w_2 = 1$, and $w_3 = 1$.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 \end{matrix} \\ \begin{bmatrix} 1 & 1 & - \\ - & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} \\ 1 \\ 2 \end{matrix}$$

In this example, column x_1 dominates x_2, and selecting x_1 solves this matrix. However, x_1 has a higher weight than x_2, so we do not remove x_2. In fact, the lowest-cost solution is $x_2 = x_3 = 1$ and $x_1 = 0$, which has cost 2, while $x_1 = 1$ and $x_2 = x_3 = 0$ would have cost 3.

5.1.2 Bounding

After the reduction steps described above, the matrix may or may not be solved. If it is solved, the cost of the solution can be determined by Equation 5.1. The reduced matrix may have a *cyclic core* and thus may not be completely solved. At this point, it is worthwhile to test whether or not a good solution can be derived from the partial solution found up to this point. This is accomplished by determining a lower bound on the final cost, starting with the current partial solution. If the cost of the current solution or lower bound on the partial solution is greater than or equal to the cost of the best solution found so far, the previous best solution is returned.

Finding an exact lower bound is as difficult as solving the covering problem, so we need to use a heuristic algorithm. A satisfactory heuristic method to determine the lower bound is to find a *maximal independent set* (MIS) of rows. Two rows are independent when it is not possible to satisfy both by setting a single variable to 1. By this definition, any row which contains a complemented variable is dependent on any other clause, so we must ignore these rows. A heuristic algorithm to find a maximal independent set to compute a lower bound is shown in Figure 5.4.

Example 5.1.7 The length of the shortest row in the cyclic constraint matrix below is 2, so the algorithm chooses row 1 to be part of our maximal independent set. Row 1 intersects rows 2, 7, and 10. Row 3 is also length 2, so it is added to the independent set. Row 3 intersects rows 4, 5, 8, and 9. This leaves only row 6. Therefore, the maximal independent set is $\{1, 3, 6\}$ and the lower bound is 3. This constraint matrix can be solved by selecting three columns: x_2, x_3, and x_4.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \end{matrix} \\ \left[\begin{array}{ccccccccc} 1 & 1 & - & - & - & - & - & - & - \\ 1 & - & 1 & - & - & - & - & - & - \\ - & - & - & 1 & 1 & - & - & - & - \\ - & - & - & 1 & - & 1 & - & - & - \\ - & - & 1 & - & 1 & 1 & - & - & - \\ - & - & 1 & - & - & - & 1 & - & - \\ - & 1 & - & - & - & - & 1 & - & - \\ - & - & - & 1 & - & - & - & 1 & - \\ - & - & - & 1 & - & - & - & - & 1 \\ - & 1 & - & - & - & - & - & 1 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}$$

5.1.3 Termination

The next step determines if $\mathbf{A}$ has been reduced to a terminal case. If $\mathbf{A}$ has no more rows, all the constraints have been satisfied by the current solution, $\mathbf{x}$, and the algorithm has reached a terminal case. Another possible terminating case is when there does not exist any solution for a given constraint matrix.

```
lower_bound(A,x) {
  MIS = ∅
  A = delete_rows_with_complemented_variables(A);
  do {
    i =choose_shortest_row(A);
    MIS = MIS ∪ {i};      /* Add row to maximal independent set.*/
    A =delete_intersecting_rows(A,i);
  } while (A ≠ ∅);                /* Repeat until matrix empty.*/
  return(|MIS| + cost(x)); /* Size of set plus partial solution.*/
}
```

Fig. 5.4 Lower-bound algorithm.

Example 5.1.8 Consider the following formula:

$$f = (x_1 + x_2)(\overline{x_1} + x_2)(x_1 + \overline{x_2})(\overline{x_1} + \overline{x_2})$$

The constraint matrix is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 \end{matrix} \\ \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

There is clearly no assignment to x_1 or x_2 that makes this formula true. In this case, the function *terminalCase* detects this, and the previous best solution is returned.

5.1.4 Branching

If **A** has not been reduced to a terminal case, the matrix is said to be *cyclic*. To find an exact minimal solution, it is necessary to consider different possible alternatives at this point. The first step is to determine a column to branch on. A column which intersects many short rows should be preferred. This is based on the assumption that shorter rows have a lower chance to be covered. This can be accomplished by assigning a weight to each row that is inversely proportional to the *row length* (i.e., the number of 1's in the row) and by summing the weights of all the rows covered by a column in order to determine the value of that column. The column with the highest value is chosen for case splitting.

Example 5.1.9 Consider the constraint matrix from Example 5.1.7. The row weights are shown in Table 5.1. The weights associated with the columns for this example are shown in Table 5.2. Clearly, the best choice is column x_4.

Next, the variable x_c associated with the column selected for branching is set to 1, the constraint matrix is reduced, and *bcp* is called recursively.

Table 5.1 Row weights for Example 5.1.7.

Row	Weight
1	1/2
2	1/2
3	1/2
4	1/2
5	1/3
6	1/2
7	1/2
8	1/2
9	1/2
10	1/3

Table 5.2 Column weights for Example 5.1.7.

Column	Weight
x_1	1.00
x_2	1.33
x_3	1.33
x_4	2.00
x_5	0.83
x_6	0.83
x_7	1.00
x_8	0.83
x_9	0.83

Example 5.1.10 The result after selecting x_4 is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_5 & x_6 & x_7 & x_8 & x_9 \end{matrix} \\ \left[\begin{array}{cccccccc} 1 & 1 & - & - & - & - & - & - \\ 1 & - & 1 & - & - & - & - & - \\ - & - & 1 & 1 & 1 & - & - & - \\ - & - & 1 & - & - & 1 & - & - \\ - & 1 & - & - & - & 1 & - & - \\ - & 1 & - & - & - & - & 1 & 1 \end{array}\right] \end{array} \begin{matrix} 1 \\ 2 \\ 5 \\ 6 \\ 7 \\ 10 \end{matrix}$$

The result returned, $\mathbf{x}^1$, is then checked to see if it is better than the best solution found so far. If so, it becomes the new best solution. If it also meets the lower bound, L, we have found a minimal solution, and it can be returned. If not, we remove x_c from consideration by setting it to 0, reduce the constraint matrix, and call *bcp* again.

Example 5.1.11 The result after removing x_4 is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{array}{cccccccc} x_1 & x_2 & x_3 & x_5 & x_6 & x_7 & x_8 & x_9 \end{array} \\ \left[\begin{array}{cccccccc} 1 & 1 & - & - & - & - & - & - \\ 1 & - & 1 & - & - & - & - & - \\ - & - & - & 1 & - & - & - & - \\ - & - & - & - & 1 & - & - & - \\ - & - & 1 & 1 & 1 & - & - & - \\ - & - & 1 & - & - & 1 & - & - \\ - & 1 & - & - & - & 1 & - & - \\ - & - & - & - & - & - & 1 & - \\ - & - & - & - & - & - & - & 1 \\ - & 1 & - & - & - & - & 1 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}$$

The result $\mathbf{x}^0$ is then compared with the previous best, and if it is lower cost, it becomes the new best solution. Finally, the best solution is returned.

5.2 STATE MINIMIZATION

A Huffman flow table is used to describe the sequence of outputs that should be provided for any possible sequence of inputs. There may be many such flow tables to represent the same behavior. The original flow table that is given as a specification may often contain more rows, or states, than is necessary to represent this behavior. To build a Huffman circuit for a given flow table, it is necessary to encode each state using state variables. If the number of states to be encoded is s and the number of state variables used is n, then n must be greater than or equal to $\lceil \log_2 s \rceil$. If we can reduce the number of states, we can also reduce the number of state variables needed to encode the states. Therefore, the first step in the synthesis process for Huffman circuits is to minimize the number of states in the flow table by finding the flow table with the smallest number of rows which produces the same desired behavior.

In this section we describe a procedure to find a flow table with the minimum number of rows to represent a desired behavior. The procedure first identifies all *compatible pairs* of states. These are states which can potentially be merged. Next, it finds all *maximal compatibles*. These are the largest sets of states which are all pairwise compatible. As illustrated later, it may not be possible to find a minimal solution using only maximal compatibles. Therefore, the procedure extends this list to a set of *prime compatibles*. These sets of states are potentially smaller than the maximal compatibles. A minimal solution can always be found using the prime compatibles. Finally, a covering problem is setup where the prime compatibles are the potential solutions and the states are what needs to be covered. In this section we describe state minimization for single-input change fundamental mode, which is the same as for synchronous state machines. Modifications needed for XBM machines are described later.

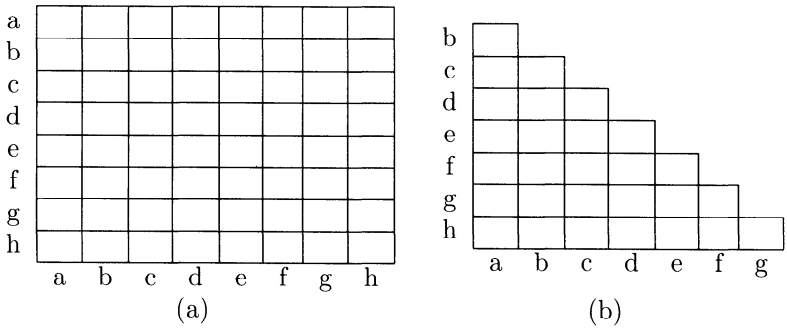


Fig. 5.5 Example pair chart.

5.2.1 Finding the Compatible Pairs

The first step in state minimization is to determine all compatible pairs. To do this, we make use of a *pair chart* (also known as a *compatibility table*), shown in Figure 5.5. Since compatibility is a reflexive and symmetric relation, the top of the chart is redundant and does not need to be considered. The reduced pair chart is depicted in Figure 5.5(b).

The first step to fill in the pair chart is to go through each pair of states, (u,v), in turn, and check if they are *unconditionally compatible*. Two states u and v are unconditionally compatible when they are *output compatible* and for each input produce the same next state when they are both specified. Two states are output compatible when for each input in which they are both specified to produce an output, they produce the same output. When two states u and v are unconditionally compatible, the corresponding (u,v) entry is marked with the symbol $\sim$.

> **Example 5.2.1** Consider the flow table shown in Figure 5.6. States a and b are output compatible since with input x_1, they both output a 0, and for all other inputs the output is unspecified in either a or b. They are not, however, unconditionally compatible because on input x_3 they produce different next states, namely d and a, respectively. Rows b and c, however, are both output compatible and unconditionally compatible. Rows a and g are also unconditionally compatible even though they have different next states on input x_6 because the next states are a and g (exactly those being considered). The pair chart after the first step is shown in Figure 5.7.

The second step is to mark each pair of states which are *incompatible*. When two states u and v are not output compatible, the states are incompatible. When two states u and v are incompatible, the (u,v) entry is marked with the symbol $\times$.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
a	a,0	–	d,0	e,1	b,0	a,–	–
b	b,0	d,1	a,–	–	a,–	a,1	–
c	b,0	d,1	a,1	–	–	–	g,0
d	–	e,–	–	b,–	b,0	–	a,–
e	b,–	e,–	a,–	–	b,–	e,–	a,1
f	b,0	c,–	–,1	h,1	f,1	g,0	–
g	–	c,1	–	e,1	–	g,0	f,0
h	a,1	e,0	d,1	b,0	b,–	e,–	a,1

Fig. 5.6 Huffman flow table used to illustrate state minimization.

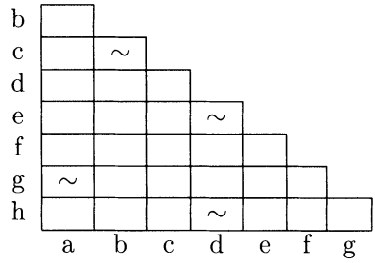


Fig. 5.7 Pair chart after marking unconditional compatibles.

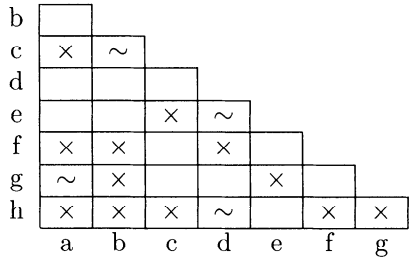


Fig. 5.8 Pair chart after marking output incompatible states.

Example 5.2.2 States a and c are incompatible since for input x_3, a outputs a 0 and c outputs a 1. The pair chart after the second step is shown in Figure 5.8.

The third step is to fill in the remaining entries with the pairs of next states which differ. These state pairs are only *conditionally compatible.* If their differing next states are merged into a new state during state minimization, these states become compatible. When two states u and v are compatible only when states s and t are merged, the (u,v) entry is marked with s,t.

	a	b	c	d	e	f	g
b	a,d						
c	×	~					
d	b,e	a,b d,e	d,e a,g				
e	a,b a,d	d,e a,b a,e	×	~			
f	×	×	c,d	×	c,e b,f e,g		
g	~	×	c,d f,g	c,e b,e a,f	×	e,h	
h	×	×	×	~	a,b a,d	×	×

Fig. 5.9 Pair chart after marking conditional compatibles.

	a	b	c	d	e	f	g
b	a,d						
c	×	~					
d	b,e	a,b d,e	d,e a,g				
e	a,b a,d	d,e a,b a,e	×	~			
f	×	×	c,d	×	×		
g	~	×	c,d f,g	×	×	e,h	
h	×	×	×	~	a,b a,d	×	×

Fig. 5.10 Final pair chart.

Example 5.2.3 States a and b go to states d and a on input x_3, so they are only compatible if d and a are compatible. The pair chart after the third step is shown in Figure 5.9.

The final step is to check each pair of conditional compatibles, and if any pair of next states are known to be incompatible, the pair of states are are also incompatible. Again, in this case, the (u,v) entry is marked with the symbol $\times$.

Example 5.2.4 For states d and g to be compatible, it must be the case that states c and e are merged. However, we know that states c and e are incompatible. Therefore, we also know that states d and g are incompatible. The pair chart after the fourth and final step is shown in Figure 5.10.

5.2.2 Finding the Maximal Compatibles

After finding the compatible pairs, the next step is to find larger sets of states that can be covered by a single state of some table. Note that if S is a compatible, any subset of S is also a compatible. A *maximal compatible* is a compatible that is not a subset of any larger compatible. From the set of maximal compatibles, it is possible to determine all other compatibles. In this subsection we present two approaches to finding the maximal compatibles.

The first approach uses the compatible pairs found in the preceding subsection. Begin by initializing a *compatible list* (c-list) with the compatible pairs in the rightmost column having at least one non-$\times$ entry in the compatibility table. Examine the columns from right to left, and perform the following steps:

1. Set S_i to the set of states in column i which do not contain an $\times$.
2. Intersect S_i with each member of the current c-list.
3. If the intersection has more than one member, add to the c-list an entry composed of the intersection unioned with i.
4. Before moving to the next column, remove duplicated entries and those that are a subset of other entries.
5. Finally, add pairs which consist of i and any members of S_i that did not appear in any of the intersections.
6. Repeat until the left of the pair chart has been reached.

The final c-list plus any individual states not contained in any other member make up the maximal compatibles.

Example 5.2.5 The procedure applied to the compatible pairs from the pair chart in Figure 5.10 is shown below.

First step:	$c = \{fg\}$
$S_e = h$:	$c = \{fg, eh\}$
$S_d = eh$:	$c = \{fg, deh\}$
$S_c = dfg$:	$c = \{cfg, deh, cd\}$
$S_b = cde$:	$c = \{cfg, deh, bde, bcd\}$
$S_a = bdeg$:	$c = \{cfg, deh, bcd, abde, ag\}$

As an alternative approach, we can use the incompatible pairs, which can readily be found from the compatibility table. If s_i and s_j have been found to be incompatible, we know that no maximal compatible can include both s_i and s_j. In general, every group of states that does not include a pair of incompatible states is in fact a compatible set of states. We can now write a Boolean formula that gives the conditions for a set of states to be compatible. For each state s_i, $x_i = 1$ means that s_i is in the set. Therefore, given that the states s_i and s_j are incompatible, the clause $(\overline{x_i} + \overline{x_j})$ can be used to express that a valid compatible set cannot include both states. If we form a conjunction of clauses for each incompatible pair, we can express the conditions necessary for a set of states to be compatible. Next, we convert this product-of-sums into a sum-of-products by multiplying it out and eliminating absorbed terms (i.e., terms which contain a subset of the literals of other terms). Each term of the resulting sum-of-products defines a maximal compatible set. The states which correspond to variables that do not occur in the term make up the maximal compatible.

Example 5.2.6 The function derived for the incompatible pairs from the pair chart in Figure 5.10 is shown below.

$$(\overline{a}+\overline{c})(\overline{a}+\overline{f})(\overline{a}+\overline{h})(\overline{b}+\overline{f})(\overline{b}+\overline{g})(\overline{b}+\overline{h})(\overline{c}+\overline{e})$$
$$(\overline{c}+\overline{h})(\overline{d}+\overline{f})(\overline{d}+\overline{g})(\overline{e}+\overline{f})(\overline{e}+\overline{g})(\overline{f}+\overline{h})(\overline{g}+\overline{h})$$

After multiplying out the function above, we get the following sum-of-products:

$$\overline{a}\,\overline{b}\,\overline{d}\,\overline{e}\,\overline{h}+\overline{a}\,\overline{b}\,\overline{c}\,\overline{f}\,\overline{g}+\overline{a}\,\overline{e}\,\overline{f}\,\overline{g}\,\overline{h}+\overline{c}\,\overline{f}\,\overline{g}\,\overline{h}+\overline{b}\,\overline{c}\,\overline{d}\,\overline{e}\,\overline{f}\,\overline{h}$$

The set of maximal compatibles implied by this function are

$$cfg, deh, bcd, abde, ag$$

which match those found by the first method.

5.2.3 Finding the Prime Compatibles

Recall that some pairs of states are compatible only if other pairs are merged into a single state. In other words, the selection of one compatible may imply that another must be selected. The set of compatibles implied by each compatible is called its *class set.* The implied compatibles must be selected to guarantee *closure* of the solution. Assume that C_1 and C_2 are two compatibles and Γ_1 and Γ_2 are their respective class sets. If $C_1 \subset C_2$, it may appear that C_2 is clearly better, but if $\Gamma_1 \subset \Gamma_2$, C_1 may actually be the better choice. The selection of C_2 may actually imply extra compatibles that make it difficult to use. Therefore, the best set of compatibles may include ones that are not maximal. To address this problem, we introduce the notion of *prime compatibles.* A compatible C_1 is prime if and only if there does not exist $C_2 \supset C_1$ such that $\Gamma_2 \subseteq \Gamma_1$. The following theorem states that an optimum solution can always be found using only prime compatibles.

Theorem 5.1 (Grasselli, 1965) *The members of at least one minimal covering are prime compatibility classes.*

An algorithm to find the prime compatibles is shown in Figure 5.11. This algorithm takes as input the compatibility table and the list of maximal compatibles, and it returns the set of prime compatibles. The algorithm begins by initializing *done*, a variable to keep track of compatibles that have been processed, to the empty set. It then loops through the list of compatibles beginning with the largest. The loop finds all compatibles of size k and adds them to the list of prime compatibles, P. It then considers each prime of size k. If it has a nonempty class set, the algorithm attempts to find a smaller compatible which has a smaller class set. Any such compatible found is added to the set of prime compatibles. This process continues until all prime compatibles have been considered in turn.

```
prime_compatibles(C, M){
  done = ∅;                          /* Initialize already computed set.*/
  for (k = |largest(M)|; k ≥ 1; k − −) { /* Loop largest to smallest.*/
    foreach (q ∈ M; |q| = k) enqueue(P, q); /* Queue all of size k.*/
    foreach (p ∈ P; |p| = k) {     /* Consider all primes of size k.*/
      if (class_set(C, p) = ∅) then continue;    /* If empty, skip.*/
      foreach (s ∈ max_subsets(p)) { /* Check all maximal subsets.*/
        if (s ∈ done) then continue;        /* If computed, skip.*/
        Γ_s = class_set(C, s);            /* Find subset's class set.*/
        prime = true;                    /* Initialize prime as true.*/
        foreach (q ∈ P; |q| ≥ k) {       /* Check all larger primes.*/
          if (s ⊂ q) then {   /* If contained in prime, check it.*/
            Γ_q = class_set(C, q);             /* Compute class set.*/
            if (Γ_q ⊆ Γ_s) then {          /* If smaller, not prime.*/
              prime = false;
              break;
            }
          }
        }
        if (prime = 1) then enqueue(P, s);  /* If prime, queue it.*/
        done = done ∪ {s};                    /* Mark as computed.*/
      }
    }
  }
  return(P);                          /* Return prime compatibles.*/
}
```

Fig. 5.11 Algorithm to find prime compatibles.

Example 5.2.7 For our running example, the size of the largest maximal compatible is 4, and it is *abde*. It is the only maximal compatible of size 4, so $P = \{abde\}$. Next, each prime p of size 4 is considered. The class set of the prime p, Γ_p, is found by examining the compatibility table. If the class set is empty, this means that the compatible is unconditionally compatible, so no further computation on this prime is necessary. In this case, p is set equal to $\{abde\}$, and its class set is determined by considering each pair. First, for (a,b) to be compatible, (a,d) must be merged, but this pair is included in *abde*. Next, for (a,d) to be compatible, (b,e) must be merged, which is again included in *abde*. Similarly, (a,e) requires (a,b) and (a,d), which are included. The pair (b,d) requires (a,b) and (d,e) which are included. Finally, (d,e) is unconditionally compatible. This means that the class set of *abde* is the empty set. Since this is the only prime of size 4, we move to primes of size 3.

There are three maximal compatibles of size 3, *bcd*, cfg, and *deh*. Let us consider *bcd* first. In this case, (b,c) are unconditionally compatible, but (b,d) requires (a,b) and (d,e) to be merged. Also, (c,d) requires (d,e)

Table 5.3 Prime compatibles and their class sets.

	Prime compatibles	Class set
1	$abde$	$\emptyset$
2	bcd	$\{(a,b),(d,e),(a,g)\}$
3	cfg	$\{(c,d),(e,h)\}$
4	deh	$\{(a,b),(a,d)\}$
5	bc	$\emptyset$
6	cd	$\{(d,e),(a,g)\}$
7	cf	$\{(c,d)\}$
8	cg	$\{(c,d),(f,g)\}$
9	fg	$\{(e,h)\}$
10	dh	$\emptyset$
11	ag	$\emptyset$
12	f	$\emptyset$

and (a,g) to be merged. Therefore, the class set for bcd is $\{(a,b), (d,e), (a,g)\}$. Since the class set is nonempty, the subsets of size 2, $\{bc, bd, cd\}$, must each be checked as a potential prime.

First, *done* is checked to see if it has already been processed. In this case, none of them have. Next, the class set is found for each potential prime compatible. The class set for bc is empty. Next, each prime q which is bigger than this compatible is checked. The only prime found is bcd. In this case, bc has a smaller class set than bcd, so bc is added to the list of primes and to *done*. Next, bd is checked, which has a class set of $\{(a,b),(d,e)\}$. The prime $abde$ found earlier is a superset of bd, and it has an empty class set. Therefore, bd is not a prime compatible, so it is only added to *done*. The last potential prime to be considered is cd, which has a class set of $\{(d,e), (a,g)\}$. This one is only a subset of bcd, so cd is a prime compatible.

Next, cfg is considered, which has a class set of $\{(c,d), (e,h)\}$. In this case, all three subsets of size 2 are prime compatibles. Finally, deh is examined, which has a class set of $\{(a,b),(a,d)\}$. In this case, de is discarded even though it has an empty class set because it is a subset of $abde$, which also has an empty class set. The compatible eh is also discarded because its class set is the same as the one for deh. Finally, the compatible dh is prime, since its class set is empty.

Next, each of the primes of size 2 is considered. The first two, ag and bc, are skipped because they have empty class sets. For the prime cd, c is not prime because it is part of bc, which has an empty class set. Similarly, d is not prime because it is part of $abde$, which also has an empty class set. For the prime cf, f is prime, since it is not part of any other prime with an empty class set. At this point, all primes have been found, and they are tabulated in Table 5.3.

5.2.4 Setting Up the Covering Problem

A collection of prime compatibles forms a valid solution when it is a *closed cover*. A collection of compatibles is a *cover* when all states are contained in some compatible in the set. A collection is *closed* when all compatibles implied by some compatible in the collection are contained in some other compatible in the collection. The variable $c_i = 1$ when the ith prime compatible is a member of the solution. Using the c_i variables, it is possible to write a Boolean formula that represents the conditions for a solution to be a closed cover. The formula is a product-of-sums where each product is a *covering* or *closure constraint*.

There is one covering constraint for each state. The product is simply a disjunction of the prime compatibles that include the state. In other words, for the covering constraint to yield 1, one of the primes that includes the state must be in the solution. There is a closure constraint for each implied compatible for each prime compatible. In other words, for the closure constraint to yield 1, either the prime compatible is not included in the solution or some other prime compatible is included which satisfies the implied compatible. The closure constraints imply that the covering problem is binate.

Example 5.2.8 The covering constraint for state a is

$$(c_1 + c_{11})$$

Similarly, the entire set of covering constraints can be constructed to produce the following product-of-sums:

$$(c_1 + c_{11})(c_1 + c_2 + c_5)(c_2 + c_3 + c_5 + c_6 + c_7 + c_8)$$
$$(c_1 + c_2 + c_4 + c_6 + c_{10})(c_1 + c_4)(c_3 + c_7 + c_9 + c_{12})$$
$$(c_3 + c_8 + c_9 + c_{11})(c_4 + c_{10})$$

The prime *bcd* requires the following states to be merged: (a,b), (a,g), (d,e). Therefore, if we include *bcd* in the cover (i.e., c_2), we must also select compatibles which will merge these other state pairs. For instance, *abde* is the only prime compatible that merges a and b. Therefore, we have a closure constraint of the form

$$c_2 \Rightarrow c_1$$

The prime ag is the only one that merges states a and g, so we also need a closure constraint of the form

$$c_2 \Rightarrow c_{11}$$

Finally, primes *abde* and *deh* both merge states d and e, so the resulting closure constraint is

$$c_2 \Rightarrow (c_1 + c_4)$$

Converting the implication into disjunctions, we can express the complete set of closure constraints for *bcd* as follows:

$$(\overline{c_2} + c_1)(\overline{c_2} + c_{11})(\overline{c_2} + c_1 + c_4)$$

After finding all covering and closure constraints, we obtain the following formula:

$$\begin{aligned}
&(c_1 + c_{11})(c_1 + c_2 + c_5)(c_2 + c_3 + c_5 + c_6 + c_7 + c_8) \\
&\cdot(c_1 + c_2 + c_4 + c_6 + c_{10})(c_1 + c_4)(c_3 + c_7 + c_9 + c_{12}) \\
&\cdot(c_3 + c_8 + c_9 + c_{11})(c_4 + c_{10})(\overline{c_2} + c_1)(\overline{c_2} + c_{11})(\overline{c_2} + c_1 + c_4) \\
&\cdot(\overline{c_3} + c_2 + c_6)(\overline{c_3} + c_4)(\overline{c_4} + c_1)(\overline{c_4} + c_1)(\overline{c_6} + c_{11})(\overline{c_6} + c_1 + c_4) \\
&\cdot(\overline{c_7} + c_2 + c_6)(\overline{c_8} + c_2 + c_6)(\overline{c_8} + c_3 + c_9)(\overline{c_9} + c_4) \quad = \quad 1
\end{aligned}$$

To find a minimal number of prime compatibles that include all the states has now been formulated into a binate covering problem. We can rewrite the equation above as a constraint matrix:

$$\mathbf{A} = \begin{array}{c}
\begin{array}{cccccccccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{11} & c_{12} \end{array} \\
\left[\begin{array}{cccccccccccc}
1 & - & - & - & - & - & - & - & - & - & 1 & - \\
1 & 1 & - & - & 1 & - & - & - & - & - & - & - \\
- & 1 & 1 & - & 1 & 1 & 1 & 1 & - & - & - & - \\
1 & 1 & - & 1 & - & 1 & - & - & - & 1 & - & - \\
1 & - & - & 1 & - & - & - & - & - & - & - & - \\
- & - & 1 & - & - & - & 1 & - & 1 & - & - & 1 \\
- & - & 1 & - & - & - & - & 1 & 1 & - & 1 & - \\
- & - & - & 1 & - & - & - & - & - & 1 & - & - \\
1 & 0 & - & - & - & - & - & - & - & - & - & - \\
- & 0 & - & - & - & - & - & - & - & - & 1 & - \\
1 & 0 & - & 1 & - & - & - & - & - & - & - & - \\
- & 1 & 0 & - & - & 1 & - & - & - & - & - & - \\
- & - & 0 & 1 & - & - & - & - & - & - & - & - \\
1 & - & - & 0 & - & - & - & - & - & - & - & - \\
1 & - & - & 0 & - & - & - & - & - & - & - & - \\
- & - & - & - & - & 0 & - & - & - & - & 1 & - \\
1 & - & - & 1 & - & 0 & - & - & - & - & - & - \\
- & 1 & - & - & - & 1 & 0 & - & - & - & - & - \\
- & 1 & - & - & - & 1 & - & 0 & - & - & - & - \\
- & - & 1 & - & - & - & - & 0 & 1 & - & - & - \\
- & - & - & 1 & - & - & - & - & 0 & - & - & -
\end{array}\right]
\end{array}
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \\ 20 \\ 21 \end{array}$$

Rows 4, 11, and 17 dominate row 5, so rows 4, 11, and 17 can be removed. Also, row 14 dominates row 15 (they are identical), so row 14 can be removed. The resulting constraint matrix is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{11} & c_{12} \end{matrix} \\ \left[\begin{array}{cccccccccccc} 1 & - & - & - & - & - & - & - & - & - & 1 & - \\ 1 & 1 & - & - & 1 & - & - & - & - & - & - & - \\ - & 1 & 1 & - & 1 & 1 & 1 & 1 & - & - & - & - \\ 1 & - & - & 1 & - & - & - & - & - & - & - & - \\ - & - & 1 & - & - & - & 1 & - & 1 & - & - & 1 \\ - & - & 1 & - & - & - & - & 1 & 1 & - & 1 & - \\ - & - & - & 1 & - & - & - & - & - & 1 & - & - \\ 1 & 0 & - & - & - & - & - & - & - & - & - & - \\ - & 0 & - & - & - & - & - & - & - & - & 1 & - \\ - & 1 & 0 & - & - & 1 & - & - & - & - & - & - \\ - & - & 0 & 1 & - & - & - & - & - & - & - & - \\ 1 & - & - & 0 & - & - & - & - & - & - & - & - \\ - & - & - & - & - & 0 & - & - & - & - & 1 & - \\ - & 1 & - & - & - & 1 & 0 & - & - & - & - & - \\ - & 1 & - & - & - & 1 & - & 0 & - & - & - & - \\ - & - & 1 & - & - & - & - & 0 & 1 & - & - & - \\ - & - & - & 1 & - & - & - & - & 0 & - & - & - \end{array}\right] \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 12 \\ 13 \\ 15 \\ 16 \\ 18 \\ 19 \\ 20 \\ 21 \end{array}$$

Next, we compute a lower bound. In doing so, we ignore all rows which include a 0 entry (i.e., rows 9 – 21). The length of the shortest remaining row is 2, so we choose row 1 to be part of our maximal independent set. Row 1 intersects with rows 2, 5, and 7. The shortest remaining row is row 8, which we add to our independent set. This row does not intersect any remaining row. The next row chosen is row 6. This row intersects row 3, leaving no rows left. Therefore, our maximal independent set is $\{1, 6, 8\}$, which gives a lower bound of 3. This is clearly not a terminal case, so we must select a variable to branch on. The column for variable c_1 has a weight of 1.33, which is best, so we set c_1 to 1 and remove all rows that it intersects to get the table below, and we recursively call the *bcp* algorithm.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{11} & c_{12} \end{matrix} \\ \left[\begin{array}{ccccccccccc} 1 & 1 & - & 1 & 1 & 1 & 1 & - & - & - & - \\ - & 1 & - & - & - & 1 & - & 1 & - & - & 1 \\ - & 1 & - & - & - & - & 1 & 1 & - & 1 & - \\ - & - & 1 & - & - & - & - & - & 1 & - & - \\ 0 & - & - & - & - & - & - & - & - & 1 & - \\ 1 & 0 & - & - & 1 & - & - & - & - & - & - \\ - & 0 & 1 & - & - & - & - & - & - & - & - \\ - & - & - & - & 0 & - & - & - & - & 1 & - \\ 1 & - & - & - & 1 & 0 & - & - & - & - & - \\ 1 & - & - & - & 1 & - & 0 & - & - & - & - \\ - & 1 & - & - & - & - & 0 & 1 & - & - & - \\ - & - & 1 & - & - & - & - & 0 & - & - & - \end{array}\right] \end{array} \begin{array}{c} \\ 3 \\ 6 \\ 7 \\ 8 \\ 10 \\ 12 \\ 13 \\ 16 \\ 18 \\ 19 \\ 20 \\ 21 \end{array}$$

This table can be reduced a bit. Column c_4 dominates column c_{10}, so c_{10} can be removed. This makes c_4 an essential variable, since it

is now the only possible solution for row 8. Therefore, we set c_4 to 1 and remove intersecting rows. Now, c_9 dominates c_{12}, so c_{12} can be removed. The resulting cyclic matrix is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_3 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{11} \end{matrix} \\ \left[\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & - & - \\ - & 1 & - & - & 1 & - & 1 & - \\ - & 1 & - & - & - & 1 & 1 & 1 \\ 0 & - & - & - & - & - & - & 1 \\ 1 & 0 & - & 1 & - & - & - & - \\ - & - & - & 0 & - & - & - & 1 \\ 1 & - & - & 1 & 0 & - & - & - \\ 1 & - & - & 1 & - & 0 & - & - \\ - & 1 & - & - & - & 0 & 1 & - \end{matrix}\right] \end{array} \begin{matrix} \\ 3 \\ 6 \\ 7 \\ 10 \\ 12 \\ 16 \\ 18 \\ 19 \\ 20 \end{matrix}$$

Next, we calculate the lower bound for this matrix. We ignore rows 10, 12, 16, 18, 19, and 20 because they all contain a 0 entry. This leaves rows 3, 6, and 7, which are not independent, since they all intersect in column c_3. Therefore, the lower bound of this matrix is 1. When we add that to the partial solution (i.e., $\{c_1, c_4\}$), we get a lower bound of 3. Column c_3 has a value of 0.75, which is the highest, so it is selected to branch on. After setting c_3 to 1 and removing intersecting rows, we get the following matrix:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{11} \end{matrix} \\ \left[\begin{matrix} 0 & - & - & - & - & - & 1 \\ 1 & - & 1 & - & - & - & - \\ - & - & 0 & - & - & - & 1 \\ 1 & - & 1 & 0 & - & - & - \\ 1 & - & 1 & - & 0 & - & - \end{matrix}\right] \end{array} \begin{matrix} \\ 10 \\ 12 \\ 16 \\ 18 \\ 19 \end{matrix}$$

In this matrix, rows 18 and 19 dominate row 12, so rows 18 and 19 can be removed. Column c_{11} dominates columns c_5, c_7, c_8, and c_9, so they can be removed. The resulting cyclic matrix is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_6 & c_{11} \end{matrix} \\ \left[\begin{matrix} 0 & - & 1 \\ 1 & 1 & - \\ - & 0 & 1 \end{matrix}\right] \end{array} \begin{matrix} \\ 10 \\ 12 \\ 16 \end{matrix}$$

The lower bound of this matrix is 1, which when added to the length of the current partial solution, $\{c_1, c_3, c_4\}$, we get 4. We select c_2, which results in the following matrix:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_6 & c_{11} \end{matrix} \\ \left[\begin{matrix} - & 1 \\ 0 & 1 \end{matrix}\right] \end{array} \begin{matrix} \\ 10 \\ 16 \end{matrix}$$

In this matrix, c_{11} is essential and selecting it solves the matrix. The final solution is $\{c_1, c_2, c_3, c_4, c_{11}\}$, which becomes our new best solution.

It is, however, larger than our lower bound, 4, so we try again removing c_2 to produce the following matrix:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_6 & c_{11} \end{matrix} \\ \left[\begin{array}{cc} 1 & - \\ 0 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ \begin{matrix} 12 \\ 16 \end{matrix} \end{array}$$

To solve this matrix, we must select both c_6 and c_{11}, so we find another solution $\{c_1, c_3, c_4, c_6, c_{11}\}$ of size 5. We then back up further, to the point where we selected c_3 and consider not selecting it. The resulting constraint matrix is shown below:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{11} \end{matrix} \\ \left[\begin{array}{ccccccc} 1 & 1 & 1 & 1 & 1 & - & - \\ - & - & - & 1 & - & 1 & - \\ - & - & - & - & 1 & 1 & 1 \\ 0 & - & - & - & - & - & 1 \\ - & - & 0 & - & - & - & 1 \\ 1 & - & 1 & 0 & - & - & - \\ 1 & - & 1 & - & 0 & - & - \\ - & - & - & - & 0 & 1 & - \end{array}\right] \end{array} \begin{array}{c} \\ \begin{matrix} 3 \\ 6 \\ 7 \\ 10 \\ 16 \\ 18 \\ 19 \\ 20 \end{matrix} \end{array}$$

This matrix is cyclic, and we select to branch on c_9. The matrix after selecting c_9 is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_5 & c_6 & c_7 & c_8 & c_{11} \end{matrix} \\ \left[\begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & - \\ 0 & - & - & - & - & 1 \\ - & - & 0 & - & - & 1 \\ 1 & - & 1 & 0 & - & - \\ 1 & - & 1 & - & 0 & - \end{array}\right] \end{array} \begin{array}{c} \\ \begin{matrix} 3 \\ 10 \\ 16 \\ 18 \\ 19 \end{matrix} \end{array}$$

In this matrix c_5 dominates c_7 and c_8, so the columns corresponding to c_7 and c_8 as well as rows 18 and 19 can be removed. Now, c_5 dominates columns c_2 and c_6, which results in those columns being removed along with rows 10 and 16. Finally, c_5 has become essential to cover row 3, so it must be selected solving the matrix. The solution found is $\{c_1, c_4, c_5, c_9\}$ which with size 4 is better than the previous best, which is size 5. At this point, the algorithm goes back and removes c_9 from the solutions, and we obtain another solution of size 5: $\{c_1, c_2, c_4, c_7, c_{11}\}$. At this point, we would recurse back and consider removing c_1 from the solution, resulting in the following matrix:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{11} & c_{12} \end{matrix} \\ \left[\begin{array}{ccccccccccc} - & - & - & - & - & - & - & - & - & 1 & - \\ 1 & - & - & 1 & - & - & - & - & - & - & - \\ 1 & 1 & - & 1 & 1 & 1 & 1 & - & - & - & - \\ - & - & 1 & - & - & - & - & - & - & - & - \\ - & 1 & - & - & - & 1 & - & 1 & - & - & 1 \\ - & 1 & - & - & - & - & 1 & 1 & - & 1 & - \\ - & - & 1 & - & - & - & - & - & 1 & - & - \\ 0 & - & - & - & - & - & - & - & - & - & - \\ 0 & - & - & - & - & - & - & - & - & 1 & - \\ 1 & 0 & - & - & 1 & - & - & - & - & - & - \\ - & 0 & 1 & - & - & - & - & - & - & - & - \\ - & - & 0 & - & - & - & - & - & - & - & - \\ - & - & - & - & 0 & - & - & - & - & 1 & - \\ 1 & - & - & - & 1 & 0 & - & - & - & - & - \\ 1 & - & - & - & 1 & - & 0 & - & - & - & - \\ - & 1 & - & - & - & - & 0 & 1 & - & - & - \\ - & - & 1 & - & - & - & - & 0 & - & - & - \end{array}\right] \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 12 \\ 13 \\ 15 \\ 16 \\ 18 \\ 19 \\ 20 \\ 21 \end{array}$$

In this matrix, c_{11} is essential and c_2 and c_4 are unacceptable. After setting c_{11} to 1 and c_2 and c_4 to 0, we get

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_3 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{12} \end{matrix} \\ \left[\begin{array}{cccccccc} - & 1 & - & - & - & - & - & - \\ 1 & 1 & 1 & 1 & 1 & - & - & - \\ - & - & - & - & - & - & - & - \\ 1 & - & - & 1 & - & 1 & - & 1 \\ - & - & - & - & - & - & 1 & - \\ 0 & - & 1 & - & - & - & - & - \\ 0 & - & - & - & - & - & - & - \\ - & - & 1 & 0 & - & - & - & - \\ - & - & 1 & - & 0 & - & - & - \\ 1 & - & - & - & 0 & 1 & - & - \\ - & - & - & - & - & 0 & - & - \end{array}\right] \end{array} \begin{array}{c} \\ 2 \\ 3 \\ 5 \\ 6 \\ 8 \\ 12 \\ 13 \\ 18 \\ 19 \\ 20 \\ 21 \end{array}$$

In this matrix, row 5 is dominated by all others, so all rows but row 5 are removed. All remaining columns mutually dominate, so the resulting terminal case is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_3 \end{matrix} \\ \left[\begin{array}{c} - \end{array}\right] \end{array} \begin{array}{c} \\ 5 \end{array}$$

There is no solution to this matrix, so this implies that c_1 must be part of any valid solution. At this point, the *bcp* algorithm terminates, returning the best solution found: $\{c_1, c_4, c_5, c_9\}$.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	1,0	{1,4},1	1,0	1,1	1,0	1,1	1,1
4	1,1	{1,4},0	1,1	{1,5},0	{1,5},0	{1,4},–	1,1
5	{1,5},0	{1,4},1	1,1	–	1,–	1,1	9,0
9	{1,5},0	5,1	–,1	4,1	9,1	9,0	9,0

Fig. 5.12 Reduced Huffman flow table.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	1,0	1,1	1,0	1,1	1,0	1,1	1,1
4	1,1	1,0	1,1	1,0	1,0	1,–	1,1
5	1,0	1,1	1,1	–	1,–	1,1	9,0
9	1,0	5,1	–,1	4,1	9,1	9,0	9,0

Fig. 5.13 Final reduced Huffman flow table.

5.2.5 Forming the Reduced Flow Table

After finding a minimal solution, it is necessary to derive the reduced flow table. There is a row in the new flow table for each compatible selected. The entries in this row are found by combining the entries from all states contained in the compatible. If in any state the value of the output is specified, the value of the output in the merged state takes that value (note that by construction of the compatibles there can be no conflict in specified output values). For the next states, the reduced machine must go to a merged state, which includes all the next states for all the states contained in this compatible.

Example 5.2.9 For compatible 1, *abde*, under input x_1, we find that a and b are the only possible next states. The only compatible which contains both of these states is compatible 1. Next, consider input x_2 for compatible 1. The only possible next states are d and e, which appear in both compatible 1, *abde*, and compatible 4, *deh*. Therefore, we have a choice of which next state to use. The reduced flow table reflecting these choices is shown in Figure 5.12. One possible final reduced flow table is shown in Figure 5.13.

5.3 STATE ASSIGNMENT

After finding a minimum row flow table representation of an asynchronous finite state machine, it is now necessary to encode each row using a unique binary code. In synchronous design, a correct encoding can be assigned arbitrarily using n bits for a flow table with 2^n rows or less. In asynchronous design, more care must be taken to ensure that a circuit can be built that is independent of signal delays.

	x_1	x_2	x_3	x_4
a	a	b	d	c
b	c	b	b	b
c	c	d	b	c
d	a	d	d	b

(a)

	y_1y_2	$y_1y_2y_3$
a	00	000
b	01	011
c	10	110
d	11	101

(b)

Fig. 5.14 (a) Simple Huffman flow table. (b) Two potential state assignments.

We first define a few terms. When the codes representing the present state and next state are the same, the circuit is *stable.* When the codes differ in a single bit location, the circuit is *in transition* from the present state to the next state. When the codes differ in multiple bit positions, the circuit is *racing* from the present state to the next state. If the circuit is racing and there exists a possibility where differences in delays can cause it to reach a different stable state than the one intended, the race is *critical.* All other races are *noncritical.* A transition from state s_i to state s_j is *direct* (denoted $[s_i,s_j]$) when all state variables are excited to change at the same time. When all state transitions are direct, the state assignment is called a *minimum-transition-time state assignment.* A flow table in which each unstable state leads directly to a stable state is called a *normal flow table.* A direct transition $[s_i,s_j]$ races critically with another direct transition $[s_k,s_l]$ when unequal delays can cause these two transitions to pass through a common state. A state assignment for a Huffman circuit is correct when it is free of critical races.

Example 5.3.1 A simple flow table is shown in Figure 5.14(a) and two potential state assignments for this machine are shown in Figure 5.14(b). Consider the machine in state b under input x_2. When the input changes to x_1, the machine is excited to enter state c. Using the first code, the machine is going from state 01 to state 10, which means that y_1 and y_2 are racing. Unless the two bits switch at about the same time, it is likely that the machine will momentarily pass through state 00 or state 11. In either case, the machine would then become excited to go to state a and the machine would end up in the wrong state. On the other hand, using the second code, the machine would be excited to go from state 011 to 110. Even though y_1 and y_3 are racing, y_2 is stable 1, keeping the machine from intersecting state 000 or 101.

5.3.1 Partition Theory and State Assignment

In this subsection we introduce *partition theory* and a theorem that relates it to critical race free state assignment. First, a partition π on a set S is a set of subsets of S such that their pairwise intersection is empty. The disjoint subsets of π are called *blocks.* A partition is *completely specified* if the union of

the subsets is S. Otherwise, the partition is said to be *incompletely specified.* Elements of S which do not appear in π are said to be *unspecified.*

A state assignment composed of n state variables $y_1, \ldots, y_n$ is composed of the *τ-partitions* $\tau_1, \ldots, \tau_n$ induced by their respective state variable. In other words, each state coded with a 0 in bit position y_1 is in one block of the partition τ_1, while those coded with a 1 are in the other block. Since each partition is created by a single variable, each partition can be composed of only one or two blocks. The order in which the blocks appear, or which one is assigned a 0 or 1, does not matter. This means that once we find one critical race free state assignment, any formed by complementing a state variable in each state or reordering the state variables is also a valid assignment.

Example 5.3.2 The partitions induced by the two state codes from Figure 5.14(b) are shown below.

First state code:

$$\begin{aligned} \tau_1 &= \{ab; cd\} \\ \tau_2 &= \{ac; bd\} \end{aligned}$$

Second state code:

$$\begin{aligned} \tau_1 &= \{ab; cd\} \\ \tau_2 &= \{ad; bc\} \\ \tau_3 &= \{ac; bd\} \end{aligned}$$

A partition π_2 is less than or equal to another partition π_1 (denoted $\pi_2 \leq \pi_1$) if and only if all elements specified in π_2 are also specified in π_1 and each block of π_2 appears in a unique block of π_1. A *partition list* is a collection of partitions of the form $\{s_p, s_q;\ s_r, s_s\}$ or $\{s_p, s_q;\ s_t\}$, where $[s_p,s_q]$ and $[s_r,s_s]$ are transitions in the same column and s_t is a stable state also in the same column. Since these partitions are composed of exactly two blocks, they are also called *dichotomies.* A state assignment for a normal flow table is a minimum transition time assignment free of critical races if and only if each partition in the partition list is $\leq$ some τ_i. These conditions are expressed more formally in the following theorem.

Theorem 5.2 (Tracey, 1966) *A row assignment allotting one y-state per row can be used for direct transition realization of normal flow tables without critical races if, and only if, for every transition $[s_i, s_j]$:*

1. *If $[s_m, s_n]$ is another transition in the same column, then at least one y-variable partitions the pair $\{s_i, s_j\}$ and the pair $\{s_m, s_n\}$ into separate blocks.*

2. *If s_k is a stable state in the same column then at least one y-variable partitions the pair $\{s_i, s_j\}$ and the state s_k into separate blocks.*

3. *For $i \neq j$, s_i and s_j are in separate blocks of at least one y-variable partition.*

Example 5.3.3 The partition list for our example in Figure 5.14(a) is shown below.

$$\begin{aligned} \pi_1 &= \{ad; bc\} \\ \pi_2 &= \{ab; cd\} \\ \pi_3 &= \{ad; bc\} \\ \pi_4 &= \{ac; bd\} \end{aligned}$$

It is clear that the partition induced by the first code is not sufficient since it does not address partitions π_1 and π_3. The second state code includes a partition that is less than or equal to each of these, so it is a valid code.

5.3.2 Matrix Reduction Method

The partition list can be converted into a *Boolean matrix*. Each partition in the partition list forms a row in this matrix and each column represents a state. The entries are annotated with a 0 if the corresponding state is a member of the first block of the partition, with a 1 if the state is a member of the second block, and a – if the state does not appear in the partition.

Example 5.3.4 Consider the flow table shown in Figure 5.15. The partition list is given below.

$$\begin{aligned} \pi_1 &= \{ab; cf\} \\ \pi_2 &= \{ae; cf\} \\ \pi_3 &= \{ac; de\} \\ \pi_4 &= \{ac; bf\} \\ \pi_5 &= \{bf; de\} \\ \pi_6 &= \{ad; bc\} \\ \pi_7 &= \{ad; ce\} \\ \pi_8 &= \{ac; bd\} \\ \pi_9 &= \{ac; ef\} \\ \pi_{10} &= \{bd; ef\} \end{aligned}$$

This partition list is converted into the Boolean matrix shown in Figure 5.16.

The state assignment problem has now been reduced to finding a Boolean matrix C with a minimum number of rows such that each row in the original Boolean matrix constructed from the partition list is covered by some row of C. The rows of this reduced matrix represent the two-block τ-partitions. The columns of this matrix represent one possible state assignment. The number of rows therefore is the same as the number of state variables needed in the assignment.

	x_1	x_2	x_3	x_4
a	a,0	c,1	d,0	c,1
b	a,0	f,1	c,1	b,0
c	f,1	c,1	c,1	c,1
d	–,–	d,0	d,0	b,0
e	a,0	d,0	c,1	e,1
f	f,1	f,1	–,–	e,1

Fig. 5.15 More complex Huffman flow table.

	a	b	c	d	e	f
π_1	0	0	1	–	–	1
π_2	0	–	1	–	0	1
π_3	0	–	0	1	1	–
π_4	0	1	0	–	–	1
π_5	–	0	–	1	1	0
π_6	0	1	1	0	–	–
π_7	0	–	1	0	1	–
π_8	0	1	0	1	–	–
π_9	0	–	0	–	1	1
π_{10}	–	0	–	0	1	1

Fig. 5.16 Boolean matrix.

5.3.3 Finding the Maximal Intersectibles

One approach to minimizing the size of the Boolean matrix would be to find two rows that intersect, and replace those two rows with their *intersection.* Two rows of a Boolean matrix, R_i and R_j, have an intersection if R_i and R_j agree wherever both R_i and R_j are specified. The intersection is formed by creating a row which has specified values taken from either R_i or R_j. Entries where neither R_i or R_j are specified are left unspecified. A row, R_i, *includes* another row, R_j, when R_j agrees with R_i wherever R_i is specified. A row, R_i, *covers* another row, R_j, if R_i includes R_j or R_i includes the complement of R_j (denoted $\overline{R_j}$).

Example 5.3.5 Consider the Boolean matrix shown in Figure 5.16. Rows 1 and 2 intersect, and their intersection is 001 – 01. Also, rows 3, 4, 8, and 9 intersect to form the row 010111. The complement of row 5 and and row 6 intersect. Recall that the assignment of 0 to the left partition and 1 to the right is arbitrary. Finding the complement of the row effectively reverses this decision. The intersection of these two rows is 011001. Finally, rows 7 and 10 intersect to form the new row 001011. The results are summarized in Figure 5.17(a). There are no remaining rows that can be intersected, and the state assignment

	a	b	c	d	e	f
(π_1,π_2)	0	0	1	–	0	1
$(\pi_3,\pi_4,\pi_8,\pi_9)$	0	1	0	1	1	1
$(\overline{\pi_5},\pi_6)$	0	1	1	0	0	1
(π_7,π_{10})	0	0	1	0	1	1

(a)

	$y_1y_2y_3y_4$
a	0000
b	0110
c	1011
d	-100
e	0101
f	1111

(b)

Fig. 5.17 (a) Reduced Boolean matrix. (b) Corresponding state assignment.

	a	b	c	d	e	f
(π_1,π_7,π_{10})	0	0	1	0	1	1
$(\pi_2,\overline{\pi_5},\pi_6)$	0	1	1	0	0	1
$(\pi_3,\pi_4,\pi_8,\pi_9)$	0	1	0	1	1	1

(a)

	$y_1y_2y_3$
a	000
b	011
c	110
d	001
e	101
f	111

(b)

Fig. 5.18 (a) Minimal Boolean matrix. (b) Corresponding state assignment.

that it implies requires four state variables (one for each row), as shown in Figure 5.17(b). However, the matrix shown in Figure 5.18(a) also covers the partition while requiring only three state variables, as shown in Figure 5.18(b).

The problem is that the order in which we combine the intersections can affect the final result. In order to find a minimum row reduced matrix, it is necessary to find all possible row intersections. If a set of partitions, π_i, π_j, ..., π_k, has an intersection, it is called *intersectible*. An intersectible may be enlarged by adding a partition π_l if and only if π_l has an intersection with every element in the set. An intersectible which cannot be enlarged further is called a *maximal intersectible*.

The first step in finding the reduced matrix is to find all pairwise intersectibles. For each pair of rows R_i and R_j, one should check whether R_i and R_j have an intersection. It is also necessary to check whether R_i and $\overline{R_j}$ have an intersection since the resulting row would also cover R_i and R_j. If there are n partitions to cover, this implies the need to consider $2n$ *ordered partitions*. The following theorem can be used to reduce the number of ordered partitions considered.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	$\overline{\pi_5}$
π_2	~										
π_3	×	×									
π_4	×	×	~								
π_5	×	×	~	×							
π_6	×	~	×	×	×						
π_7	~	×	×	×	×	~					
π_8	×	×	~	~	×	×	×				
π_9	×	×	~	~	×	×	×	~			
π_{10}	~	×	×	×	×	×	~	×	~		
$\overline{\pi_5}$	×	~	×	~	×	~	×	×	×	×	
$\overline{\pi_{10}}$	×	×	×	×	×	×	×	~	×	×	×

Fig. 5.19 Pair chart for state assignment example.

Theorem 5.3 (Unger, 1969) *Let D be a set of ordered partitions derived from some set of unordered partitions. For some state s, label as p_1, p_2, etc. the members of D having s in their left sets, and label as q_1, q_2, etc. the members of D that do not contain s in either set. Then a minimal set of maximal intersectibles covering each member of D or its complement can be found by considering only the ordered partitions labeled as p's or q's. (The complements of the p's can be ignored.)*

Example 5.3.6 Consider the example shown in Figure 5.16. If we select $s = a$, then we only need to consider $\pi_1, \ldots, \pi_{10}$ and $\overline{\pi_5}$ and $\overline{\pi_{10}}$.

To find the pairwise intersectibles, we can now construct a pair chart for the remaining ordered partitions.

Example 5.3.7 The pairwise intersectibles from Figure 5.16 are found using the pair chart shown in Figure 5.19, and they are listed below.

$$(\pi_1, \pi_2)(\pi_1, \pi_7)(\pi_1, \pi_{10})(\pi_2, \pi_6)(\pi_2, \overline{\pi_5})(\pi_3, \pi_4)(\pi_3, \pi_5)(\pi_3, \pi_8)(\pi_3, \pi_9)$$
$$(\pi_4, \pi_8)(\pi_4, \pi_9)(\pi_4, \overline{\pi_5})(\pi_6, \pi_7)(\pi_6, \overline{\pi_5})(\pi_7, \pi_{10})(\pi_8, \pi_9)(\pi_8, \overline{\pi_{10}})(\pi_9, \pi_{10})$$

The second step is to use the set of pairwise intersectibles to derive all maximal intersectibles. The approach taken is the same as the one described earlier to find maximal compatibles.

Table 5.4 Maximal intersectibles for Example 5.3.4.

x_1	(π_1, π_2)
x_2	(π_1, π_7, π_{10})
x_3	$(\pi_2, \pi_6, \overline{\pi_5})$
x_4	$(\pi_3, \pi_4, \pi_8, \pi_9)$
x_5	(π_3, π_5)
x_6	$(\pi_4, \overline{\pi_5})$
x_7	(π_6, π_7)
x_8	$(\pi_8, \overline{\pi_{10}})$
x_9	(π_9, π_{10})

Example 5.3.8 Here is the derivation of the maximal intersectibles from the intersectibles from Example 5.3.7:

First step: $c = \{(\pi_9, \pi_{10})\}$

$S_{\pi_8} = \pi_9, \overline{\pi_{10}}$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \pi_9), (\pi_8, \overline{\pi_{10}})\}$

$S_{\pi_7} = \pi_{10}$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \pi_9), (\pi_8, \overline{\pi_{10}}), (\pi_7, \pi_{10})\}$

$S_{\pi_6} = \pi_7, \overline{\pi_5}$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \pi_9), (\pi_8, \overline{\pi_{10}}), (\pi_7, \pi_{10}), (\pi_6, \pi_7), (\pi_6, \overline{\pi_5})\}$

$S_{\pi_4} = \pi_8, \pi_9, \overline{\pi_5}$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \overline{\pi_{10}}), (\pi_7, \pi_{10}), (\pi_6, \pi_7), (\pi_6, \overline{\pi_5}), (\pi_4, \pi_8, \pi_9), (\pi_4, \overline{\pi_5})\}$

$S_{\pi_3} = \pi_4, \pi_5, \pi_8, \pi_9$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \overline{\pi_{10}}), (\pi_7, \pi_{10}), (\pi_6, \pi_7), (\pi_6, \overline{\pi_5}), (\pi_4, \overline{\pi_5}), (\pi_3, \pi_4, \pi_8, \pi_9), (\pi_3, \pi_5)\}$

$S_{\pi_2} = \pi_6, \overline{\pi_5}$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \overline{\pi_{10}}), (\pi_7, \pi_{10}), (\pi_6, \pi_7), (\pi_4, \overline{\pi_5}), (\pi_3, \pi_4, \pi_8, \pi_9), (\pi_3, \pi_5), (\pi_2, \pi_6, \overline{\pi_5})\}$

$S_{\pi_1} = \pi_2, \pi_7, \pi_{10}$: $c = \{(\pi_9, \pi_{10}), (\pi_8, \overline{\pi_{10}}), (\pi_6, \pi_7), (\pi_4, \overline{\pi_5}), (\pi_3, \pi_4, \pi_8, \pi_9), (\pi_3, \pi_5), (\pi_2, \pi_6, \overline{\pi_5}), (\pi_1, \pi_7, \pi_{10}), (\pi_1, \pi_2)\}$

The final set of maximal intersectibles is shown in Table 5.4.

5.3.4 Setting Up the Covering Problem

The last step is to select a minimum number of maximal intersectibles such that each row of the partition matrix is covered by one intersectible. This again results in a covering problem, which can be solved as above. In this covering problem, the clauses are now the partitions, while the literals are the maximal intersectibles. Note there are no implications in this problem, so the covering problem is unate.

	x_1	x_2	x_3	x_4
000	000,0	011,1	100,0	011,1
110	000,0	111,1	011,1	110,0
011	111,1	011,1	011,1	011,1
100	–,–	100,0	100,0	110,0
101	000,0	100,0	011,1	101,1
111	111,1	111,1	–,–	111,1

Fig. 5.20 Huffman flow table from Figure 5.15 after state assignment.

Example 5.3.9 The constraint matrix for the maximal intersectibles from Table 5.4 is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 \end{matrix} \\ \begin{bmatrix} 1 & 1 & - & - & - & - & - & - & - \\ 1 & - & 1 & - & - & - & - & - & - \\ - & - & - & 1 & 1 & - & - & - & - \\ - & - & - & 1 & - & 1 & - & - & - \\ - & - & 1 & - & 1 & 1 & - & - & - \\ - & - & 1 & - & - & - & 1 & - & - \\ - & 1 & - & - & - & - & 1 & - & - \\ - & - & - & 1 & - & - & - & 1 & - \\ - & - & - & 1 & - & - & - & - & 1 \\ - & 1 & - & - & - & - & - & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} \pi_1 \\ \pi_2 \\ \pi_3 \\ \pi_4 \\ \pi_5 \\ \pi_6 \\ \pi_7 \\ \pi_8 \\ \pi_9 \\ \pi_{10} \end{matrix}$$

This matrix cannot be reduced, so we find a maximal independent set of rows, { 1, 3, 6 }, which shows that the lower bound is 3. We then select column x_4 to branch on, reduce the matrix, and initiate recursion on the following matrix:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_5 & x_6 & x_7 & x_8 & x_9 \end{matrix} \\ \begin{bmatrix} 1 & 1 & - & - & - & - & - & - \\ 1 & - & 1 & - & - & - & - & - \\ - & - & 1 & 1 & 1 & - & - & - \\ - & - & 1 & - & - & 1 & - & - \\ - & 1 & - & - & - & 1 & - & - \\ - & 1 & - & - & - & - & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} \pi_1 \\ \pi_2 \\ \pi_5 \\ \pi_6 \\ \pi_7 \\ \pi_{10} \end{matrix}$$

In this matrix, column x_3 dominates columns x_5 and x_6. Column x_2 dominates columns x_8 and x_9. Now, π_5 and π_{10} are essential rows, implying the need to include x_3 and x_2 in the solution. This solves the matrix. Therefore, the solution $(\pi_3, \pi_4, \pi_8, \pi_9)$, $(\pi_2, \pi_6, \overline{\pi_5})$, and (π_1, π_7, π_{10}) has been found. Since its size, 3, matches the lower bound we found earlier, we are done. The resulting state code is the transpose of the Boolean matrix shown in Figure 5.18. The Huffman flow table after state assignment is shown in Figure 5.20.

	a	b	c	d	e	f
π_2	0		1		0	1
π_3	0	–	0	1	1	–
π_4	0	1	0	–	–	1
π_5	–	0	–	1	1	0
π_6	0	1	1	0	–	–
π_8	0	1	0	1	–	–
π_9	0	–	0	–	1	1

Fig. 5.21 Boolean matrix.

5.3.5 Fed-Back Outputs as State Variables

Previously, we ignored the values of the outputs during state assignment. It may be possible, however, to feed back the outputs and use them as state variables. To do this, we determine in each state under each input the value of each output upon entry. This information can be used to satisfy some of the necessary partitions. By reducing the number of partitions that must be satisfied, we can reduce the number of explicit state variables that are needed. We will illustrate this through an example.

Example 5.3.10 Consider again the flow table from Figure 5.15. The value of the output is always 0 upon entering states a, b, and d, and it is always 1 upon entering states c, e, and f. Previously, we found the following partition: $\{ab; cf\}$. If we consider the fed-back output as a state variable, it satisfies this partition since it is 0 in states a and b and 1 in states c and f. Using the output as a state variable reduces the partition list to the one given below which is converted into the Boolean matrix shown in Figure 5.21.

$$\begin{aligned}
\pi_2 &= \{ae, cf\} \\
\pi_3 &= \{ac, de\} \\
\pi_4 &= \{ac, bf\} \\
\pi_5 &= \{bf, de\} \\
\pi_6 &= \{ad, bc\} \\
\pi_8 &= \{ac, bd\} \\
\pi_9 &= \{ac, cf\}
\end{aligned}$$

Using Theorem 5.3 and state a, we determine that the only complement that needs to be considered is $\overline{\pi_5}$. The pairwise intersectibles from Figure 5.21 are found using the pair chart shown in Figure 5.22 and are listed below.

$$(\pi_2, \pi_6)(\pi_2, \overline{\pi_5})(\pi_3, \pi_4)(\pi_3, \pi_5)(\pi_3, \pi_8)(\pi_3, \pi_9)(\pi_4, \pi_8)$$
$$(\pi_4, \pi_9)(\pi_4, \overline{\pi_5})(\pi_6, \overline{\pi_5})(\pi_8, \pi_9)$$

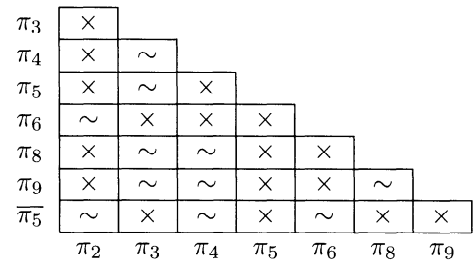


Fig. 5.22 Pair chart for output state assignment example.

Table 5.5 Maximal intersectibles for Example 5.3.4 using outputs as state variables.

x_1	$(\pi_2, \pi_6, \overline{\pi_5})$
x_2	$(\pi_3, \pi_4, \pi_8, \pi_9)$
x_3	(π_3, π_5)
x_4	$(\pi_4, \overline{\pi_5})$

Here is the derivation of the maximal intersectibles:

First step: $c = \{(\pi_8, \pi_9)\}$

$S_{\pi_6} = \overline{\pi_5}$: $c = \{(\pi_8, \pi_9), (\pi_6, \overline{\pi_5}), \}$

$S_{\pi_4} = \pi_8, \pi_9, \overline{\pi_5}$: $c = \{(\pi_4, \pi_8, \pi_9), (\pi_4, \overline{\pi_5}), (\pi_6, \overline{\pi_5})\}$

$S_{\pi_3} = \pi_4, \pi_5, \pi_8, \pi_9$: $c = \{(\pi_3, \pi_4, \pi_8, \pi_9), (\pi_3, \pi_5), (\pi_4, \overline{\pi_5}), (\pi_6, \overline{\pi_5})\}$

$S_{\pi_2} = \pi_6, \overline{\pi_5}$: $c = \{(\pi_3, \pi_4, \pi_8, \pi_9), (\pi_3, \pi_5), (\pi_4, \overline{\pi_5}), (\pi_2, \pi_6, \overline{\pi_5})\}$

The final set of maximal intersectibles is shown in Table 5.5.

The constraint matrix for the maximal intersectibles from Table 5.5 is shown below.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 \end{matrix} \\ \begin{bmatrix} 1 & - & - & - \\ - & 1 & 1 & - \\ - & 1 & - & 1 \\ 1 & - & 1 & 1 \\ 1 & - & - & - \\ - & 1 & - & - \\ - & 1 & - & - \end{bmatrix} \end{array} \begin{matrix} \pi_2 \\ \pi_3 \\ \pi_4 \\ \pi_5 \\ \pi_6 \\ \pi_8 \\ \pi_9 \end{matrix}$$

Row π_2 is essential, so x_1 must be part of the solution. Row π_8 is also essential, so x_2 must be part of the solution. This solves the entire matrix and the final solution is $(\pi_2, \pi_6, \overline{\pi_5})$ and $(\pi_3, \pi_4, \pi_8, \pi_9)$.

The new flow table after state assignment is shown in Figure 5.23. Note that only two state variables are needed. When only these two variables are considered, it appears that states b and f have the same code. Also, states d and e appear to have the same state code. The fed-back output, however, serves to disambiguate these states.

	x_1	x_2	x_3	x_4
00	00,0	01,1	10,0	01,1
11	00,0	11,1	01,1	11,0
01	11,1	01,1	01,1	01,1
10	–,–	10,0	10,0	11,0
10	00,0	10,0	01,1	10,1
11	11,1	11,1	–,–	11,1

Fig. 5.23 Huffman flow table from Figure 5.15 after state assignment using outputs as fed-back state variables.

5.4 HAZARD-FREE TWO-LEVEL LOGIC SYNTHESIS

After finding a critical race free state assignment, the next step of the design process is to synthesize the logic for each next state and output signal. The traditional synchronous approach to logic synthesis for FSMs would be to derive a *sum-of-products* (SOP) implementation for each next state and output signal. For asynchronous FSMs, care must be taken to avoid *hazards* in the SOP implementation. A circuit has a hazard when there exists an assignment of delays such that a *glitch*, an unwanted signal transition, can occur. Hazards must be avoided in asynchronous design since they may be misinterpreted by other parts of the circuit as a valid signal transition causing erroneous behavior. In this section we describe a method for hazard-free two-level logic synthesis under the SIC fundamental-mode assumption.

5.4.1 Two-Level Logic Minimization

An *incompletely specified Boolean function* f of n variables $x_1, x_2, \ldots, x_n$ is a mapping: $f : \{0,1\}^n \rightarrow \{0,1,-\}$. Each element m of $\{0,1\}^n$ is called a *minterm*. The value of a variable x_i in a minterm m is given by $m(i)$. The *ON-set* of f is the set of minterms which return 1. The *OFF-set* of f is the set of minterms which return 0. The *don't care (DC)-set* of f is the set of minterms which return $-$.

A *literal* is either the variable, x_i, or its complement, $\overline{x_i}$. The literal x_i evaluates to 1 in the minterm m when $m(i) = 1$. The literal $\overline{x_i}$ evaluates to 1 when $m(i) = 0$. A *product* is a conjunction (AND) of literals. A product evaluates to 1 for a given minterm if each literal evaluates to 1 in the minterm, and the product is said to *contain* the minterm. A set of minterms which can be represented with a product is called a *cube*. A product Y contains another product X (i.e., $X \subseteq Y$) if the minterms contained in X are a subset of those in Y. The *intersection* of two products is the set of minterms contained in both products. A *sum-of-products* (SOP) is a set of products that are disjunctively combined. In other words, a SOP contains a minterm when one of the products in the SOP contains the minterm.

yz \ wx	00	01	11	10
00	1	1	1	1
01	0	1	1	–
11	0	1	1	0
10	0	–	0	0

(a)

yz \ wx	00	01	11	10
00	1	1	1	1
01	0	1	1	–
11	0	1	1	0
10	0	–	0	0

(b)

Fig. 5.24 (a) Karnaugh map for small two-level logic minimization example. (b) Minimal two-level SOP cover.

An *implicant* of a function is a product that contains no minterms in the OFF-set of the function. A *prime implicant* is an implicant which is contained by no other implicant. A *cover* of a function is a SOP which contains the entire ON-set and none of the OFF-set. A cover may optionally include part of the DC-set. The two-level logic minimization problem is to find a minimum-cost cover of the function. Ignoring hazards, a minimal cover is always composed only of prime implicants, as stated in the following theorem.

Theorem 5.4 (Quine, 1952) *A minimal SOP must always consist of a sum of prime implicants if any definition of cost is used in which the addition of a single literal to any formula increases the cost of the formula.*

Example 5.4.1 Consider a function of four variables, w, x, y, and z, depicted in the Karnaugh map in Figure 5.24(a). The minterms are divided as follows:

$$\begin{aligned}
\text{ON-set} &= \{\overline{w}\,\overline{x}\,\overline{y}\,\overline{z}, \overline{w}x\overline{y}\,\overline{z}, wx\overline{y}\,\overline{z}, w\overline{x}\,\overline{y}\,\overline{z}, \overline{w}\overline{x}y\overline{z}, wx\overline{y}z, \overline{w}xyz, wxyz\} \\
\text{OFF-set} &= \{\overline{w}\,\overline{x}\,\overline{y}z, \overline{w}\,\overline{x}yz, w\overline{x}yz, \overline{w}\,\overline{x}y\overline{z}, wxy\overline{z}, w\overline{x}y\overline{z}\} \\
\text{DC-set} &= \{w\overline{x}\,\overline{y}z, \overline{w}xy\overline{z}\}
\end{aligned}$$

The product $\overline{y}z$ is not an implicant since it contains the OFF-set minterm $\overline{w}\,\overline{x}\,\overline{y}z$. The product $w\overline{y}z$ is an implicant since it does not intersect the OFF-set. It is not, however, a prime implicant, since it is contained in the implicant $w\overline{y}$, which is prime. The minimal SOP cover for this function is

$$f = \overline{y}\,\overline{z} + xz$$

which is depicted in Figure 5.24(b).

5.4.2 Prime Implicant Generation

For functions of fewer than four variables, prime implicants can easily be found using a Karnaugh map. For functions of more variables, the Karnaugh map method quickly becomes too tedious. A better approach is Quine's tabular

method, but this method requires that all minterms be listed explicitly at the beginning. In this subsection we briefly explain a recursive procedure based on *consensus* and *complete sums*.

The consensus theorem states that $xy + \overline{x}z = xy + \overline{x}z + yz$. The product yz is called the consensus for xy and $\overline{x}z$. A complete sum is defined to be a SOP formula composed of all the prime implicants. The following results are very useful for finding complete sums.

Theorem 5.5 (Blake, 1937) *A SOP is a complete sum if and only if:*

1. *No term includes any other term.*
2. *The consensus of any two terms of the formula either does not exist or is contained in some term of the formula.*

Theorem 5.6 (Blake, 1937) *If we have two complete sums f_1 and f_2, we can obtain the complete sum for $f_1 \cdot f_2$ using the following two steps:*

1. *Multiply out f_1 and f_2 using the following properties:*
   - $x \cdot x = x$ (idempotent)
   - $x \cdot (y + z) = xy + xz$ (distributive)
   - $x \cdot \overline{x} = 0$ (complement)
2. *Eliminate all terms* absorbed *by some other term (i.e., $a + ab = a$).*

Based upon this result and Boole's expansion theorem, we can define a recursive procedure for finding the complete sum for a function f:

$$\mathrm{cs}(f) = \mathrm{abs}([x_1 + \mathrm{cs}(f(0, x_2, \ldots, x_n))] \cdot [\overline{x_1} + \mathrm{cs}(f(1, x_2, \ldots, x_n))])$$

where $abs(f)$ removes absorbed terms from f.

Example 5.4.2 The ON-set and DC-set of the function depicted in Figure 5.24(a) can be expressed using the following formula:

$$f(w, x, y, z) = \overline{y}\,\overline{z} + xz + w\overline{x}\,\overline{y}z + \overline{w}xy\overline{z}$$

which is clearly not a complete sum. A complete sum can be found using the recursive procedure as follows:

$$\begin{aligned}
f(w, x, y, z) &= \overline{y}\,\overline{z} + xz + w\overline{x}\,\overline{y}z + \overline{w}xy\overline{z} \\
f(w, x, y, 0) &= \overline{y} + \overline{w}xy \\
f(w, x, 0, 0) &= 1 \\
f(w, x, 1, 0) &= \overline{w}x \\
\mathrm{cs}(f(w, x, y, 0)) &= \mathrm{abs}((y + 1)(\overline{y} + \overline{w}x)) = \overline{y} + \overline{w}x \\
f(w, x, y, 1) &= x + w\overline{x}\,\overline{y} \\
f(w, 0, y, 1) &= w\overline{y}
\end{aligned}$$

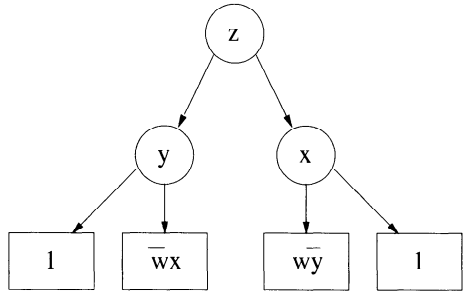


Fig. 5.25 Recursion tree for prime generation example.

$$
\begin{aligned}
f(w,1,y,1) &= 1 \\
\mathrm{cs}(f(w,x,y,1)) &= \mathrm{abs}((x+w\overline{y})(\overline{x}+1)) = x + w\overline{y} \\
\mathrm{cs}(f(w,x,y,z)) &= \mathrm{abs}((z+\overline{y}+\overline{w}x)(\overline{z}+x+w\overline{y})) \\
&= \mathrm{abs}(xz + w\overline{y}z + \overline{y}\,\overline{z} + x\overline{y} + w\overline{y} + \overline{w}x\overline{z} + \overline{w}x) \\
&= xz + \overline{y}\,\overline{z} + x\overline{y} + w\overline{y} + \overline{w}x
\end{aligned}
$$

The recursion tree for this example is shown in Figure 5.25.

5.4.3 Prime Implicant Selection

In order to select the minimal number of prime implicants necessary to cover the function, we need to solve a covering problem. We can create a constraint matrix where the rows are the minterms in the function and the columns are the prime implicants.

Example 5.4.3 The constraint matrix for the example shown in Figure 5.24(a) is given below.

	xz	$\overline{y}\,\overline{z}$	$x\overline{y}$	$w\overline{y}$	$\overline{w}x$
$\overline{w}\,\overline{x}\,\overline{y}\,\overline{z}$	–	1	–	–	–
$\overline{w}x\overline{y}\,\overline{z}$	–	1	1	–	1
$wx\overline{y}\,\overline{z}$	–	1	1	1	–
$w\overline{x}\,\overline{y}\,\overline{z}$	–	1	–	1	–
$\overline{w}x\overline{y}z$	1	–	1	–	1
$wx\overline{y}z$	1	–	1	1	–
$\overline{w}xyz$	1	–	–	–	1
$wxyz$	1	–	–	–	–

The rows corresponding to the minterms $\overline{w}\,\overline{x}\,\overline{y}\,\overline{z}$ and $wxyz$ are essential. This implies that $\overline{y}\,\overline{z}$ and xz are *essential primes*. Selecting these two primes solves the entire covering problem, so they make up our minimal SOP cover.

5.4.4 Combinational Hazards

For asynchronous design, the two-level logic minimization problem is complicated by the fact that there can be no hazards in the SOP implementation. Let us consider the design of a function f to implement either an output or next-state variable. Under the SIC model, when an input changes, the circuit moves from one minterm m_1 to another minterm m_2, where the two minterms differ in value in exactly one variable, x_i. During this transition, there are four possible transitions that f can make.

1. If $f(m_1) = 0$ and $f(m_2) = 0$, then f is making a *static* $0 \rightarrow 0$ *transition.*
2. If $f(m_1) = 1$ and $f(m_2) = 1$, then f is making a *static* $1 \rightarrow 1$ *transition.*
3. If $f(m_1) = 0$ and $f(m_2) = 1$, then f is making a *dynamic* $0 \rightarrow 1$ *transition.*
4. If $f(m_1) = 1$ and $f(m_2) = 0$, then f is making a *dynamic* $1 \rightarrow 0$ *transition.*

In order to design a hazard-free SOP cover, we must consider each of these cases in turn. First, if during a static $0 \rightarrow 0$ transition, the cover of f due to differences in delays momentarily evaluate to 1, we say that there exists a *static 0-hazard.* In a SOP cover of a function, no product term is allowed to include either m_1 or m_2 since they are members of the OFF-set. Therefore, the only way this can occur is if some product includes both x_i and $\overline{x_i}$. Clearly, such a product is not useful since it contains no minterms. If we exclude such product terms from the cover, the SOP cover can never produce a static 0-hazard. This result is summarized in the following theorem.

Theorem 5.7 (McCluskey, 1965) *A circuit has a static 0-hazard between the adjacent minterms* m_1 *and* m_2 *that differ only in* x_j *iff* $f(m_1) = f(m_2) = 0$*, there is a product term,* p_i*, in the circuit that includes* x_j *and* $\overline{x_j}$*, and all other literals in* p_i *have value* 1 *in* m_1 *and* m_2*.*

If during a static $1 \rightarrow 1$ transition, the cover of f can momentarily evaluate to 0, we say that there exists a *static 1-hazard.* In a SOP cover, consider the case where there is one product p_1 which contains m_1 but not m_2 and another product p_2 which contains m_2 but not m_1. The cover includes both m_1 and m_2, but there can be a static 1-hazard. If p_1 is implemented with a faster gate than p_2, the gate for p_1 can turn off faster than the gate for p_2 turns on, which can lead to the cover momentarily evaluating to a 0. In order to eliminate all static 1-hazards, for each possible transition $m_1 \rightarrow m_2$, there must exist a product in the cover that includes both m_1 and m_2. This result is summarized in the following theorem.

Theorem 5.8 (McCluskey, 1965) *A circuit has a static* 1*-hazard between adjacent minterms* m_1 *and* m_2 *where* $f(m_1) = f(m_2) = 1$ *iff there is no product term that has the value* 1 *in both* m_1 *and* m_2*.*

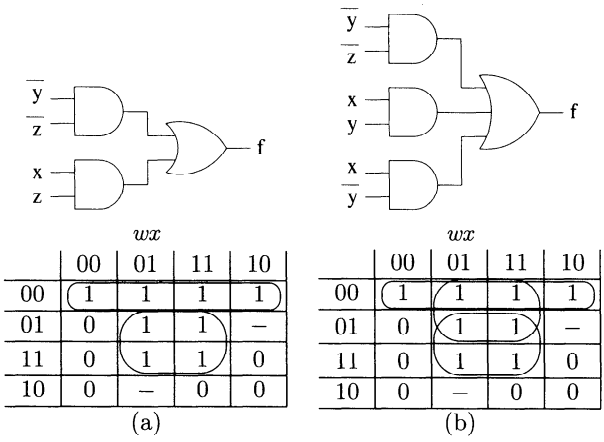


Fig. 5.26 (a) Circuit with static 1-hazard. (b) Circuit without static 1-hazard.

Example 5.4.4 Consider the transition from $\overline{w}x\overline{y}\,\overline{z}$ to $\overline{w}x\overline{y}z$. The function should maintain a constant value of 1 during this transition, so it is a static $1 \rightarrow 1$ transition. Consider the implementation shown in Figure 5.26(a). If the gate implementing $\overline{y}\,\overline{z}$ changes to 0 faster than xz changes to 1, then it is possible that the output of the function can momentarily yield a 0. The result is a static 1-hazard. If we include the prime $x\overline{y}$ in the cover as shown in Figure 5.26(b), this hazard is eliminated since this product yields a 1 during the transition, holding the output at a constant 1.

The next type of transition is the dynamic $0 \rightarrow 1$ transition. If during a $0 \rightarrow 1$ transition, the cover can change from 0 to 1 back to 0 and finally stabilize at 1, we say that the cover has a *dynamic* $0 \rightarrow 1$ *hazard*. Again, assuming no useless product terms (ones that include both x_i and $\overline{x_i}$), this is impossible under the SIC assumption. No product in the cover is allowed to include m_1 since it is in the OFF-set. Any product in the cover that includes m_2 will turn on monotonically. Similarly, there are no *dynamic* $1 \rightarrow 0$ *hazards*. This result is summarized in the following theorem.

Theorem 5.9 (McCluskey, 1965) *A SOP circuit has a dynamic hazard between adjacent minterms m_1 and m_2 that differ only in x_j iff $f(m_1) \neq f(m_2)$, the circuit has a product term p_i that contains x_j and $\overline{x_j}$, and all other literals of p_i have value 1 in m_1 and m_2.*

A simple, inefficient approach to produce a hazard-free SOP cover under SIC operation is to include all prime implicants in the cover. This eliminates static 1-hazards for input transitions, since the two minterms m_1 and m_2 are distance 1 apart; they must be included together in some prime. This means that an implicant exists which is made up of all literals that are equal in both m_1 and m_2. This implicant must be part of some prime implicant.

Example 5.4.5 For the example shown in Figure 5.24(a), the following cover is guaranteed to be hazard-free under the SIC assumption:

$$f = xz + \overline{y}\,\overline{z} + x\overline{y} + w\overline{y} + \overline{w}x$$

This approach is clearly inefficient, so a better approach is to reformulate the covering problem. The traditional SOP covering problem is to find the minimum number of prime implicants that cover all minterms in the ON-set. Instead, let us form an implicant out of each pair of states m_1 and m_2 involved in a static $1 \rightarrow 1$ transition which includes each literal that is the same value in both m_1 and m_2. The covering problem is now to find the minimum number of prime implicants that cover each of these *transition cubes.*

Example 5.4.6 For the example shown in Figure 5.24(a), let's assume that there exists a static $1 \rightarrow 1$ transition between each pair of minterms that are *distance 1 apart* (i.e., differ in a single literal). The resulting constraint matrix would be

	xz	$\overline{y}\,\overline{z}$	$x\overline{y}$	$w\overline{y}$	$\overline{w}x$
$\overline{w}\,\overline{y}\,\overline{z}$	–	1	–	–	–
$\overline{x}\,\overline{y}\,\overline{z}$	–	1	–	–	–
$\overline{w}x\overline{y}$	–	–	1	–	1
$wx\overline{y}$	–	–	1	1	–
$w\overline{y}\overline{z}$	–	1	–	1	–
$x\overline{y}z$	1	–	1	–	–
$\overline{w}xz$	1	–	–	–	1
wxz	1	–	–	–	–

Again, xz and $\overline{y}\,\overline{z}$ are essential, and they must be included in the cover, leading to the following reduced constraint matrix:

	$x\overline{y}$	$w\overline{y}$	$\overline{w}x$
$\overline{w}x\overline{y}$	1	–	1
$wx\overline{y}$	1	1	–

The prime $x\overline{y}$ dominates the others, so the final hazard-free cover is

$$f = xz + \overline{y}\,\overline{z} + x\overline{y}$$

5.5 EXTENSIONS FOR MIC OPERATION

In Section 5.4, we restricted the class of circuits to those which only allowed a single input to change at a time. In other words, if these machines are specified as XBM machines, each input burst is allowed to include only a single transition. In this section we extend the synthesis method to allow multiple input changes. This allows us to synthesize any XBM machine which satisfies the maximal set property.

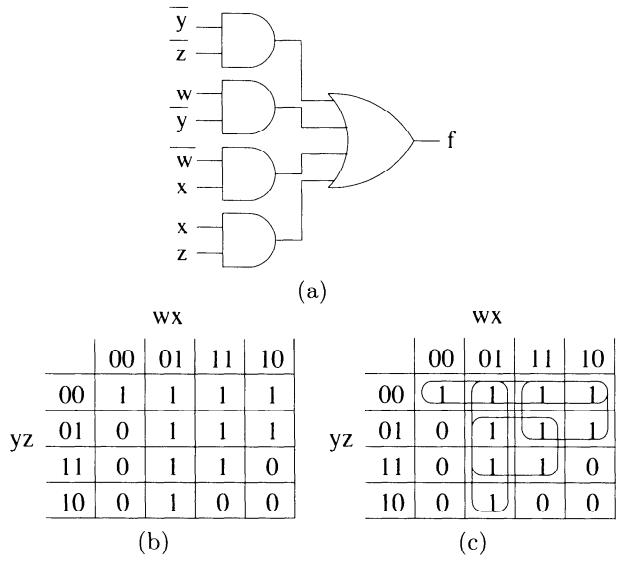


Fig. 5.27 (a) Hazardous circuit implementation. (b) Karnaugh map for small two-level logic minimization example. (c) Minimal two-level SOP cover.

5.5.1 Transition Cubes

Transitions in the MIC case begin in one minterm m_1 and end in another m_2, where the values of multiple variables may change during the transition. The minterm m_1 is called the *start point* and m_2 is called the *end point* of the transition. The smallest cube that contains both m_1 and m_2 is called the *transition cube*, and it is denoted $[m_1, m_2]$. This cube includes all possible minterms that a machine may pass through starting in m_1 and ending in m_2. The transition cube can also be represented with a product which contains a literal for each variable x_i in which $m_1(i) = m_2(i)$. An *open transition cube* $[m_1, m_2)$ includes all minterms in $[m_1, m_2]$ except those in m_2. An open transition cube usually must be represented using a set of products.

Example 5.5.1 Consider the transition $[\overline{w}\,x\,\overline{y}\,\overline{z},\ w\,x\,\overline{y}\,z]$ in the Karnaugh map in Figure 5.27(b). The transition cube for this transition is $x\,\overline{y}$, and it includes the four minterms that may be passed through during this transition. As another example, consider the open transition cube $[\overline{w}\,x\,y\,z,\ w\,x\,y\,\overline{z})$. This transition is represented by two products $\overline{w}\,x\,y$ and $x\,y\,z$.

5.5.2 Function Hazards

If a function f does not change monotonically during a multiple-input change, f has a *function hazard* for that transition. A function f contains a function

hazard during a transition from m_1 to m_2 if there exists an m_3 and m_4 such that:

1. $m_3 \neq m_1$ and $m_4 \neq m_2$.
2. $m_3 \in [m_1, m_2]$ and $m_4 \in [m_3, m_2]$.
3. $f(m_1) \neq f(m_3)$ and $f(m_4) \neq f(m_2)$.

If $f(m_1) = f(m_2)$, it is a *static function hazard*, and if $f(m_1) \neq f(m_2)$, it is a *dynamic function hazard*.

> **Example 5.5.2** Consider the transition $[\overline{w}\,\overline{x}\,\overline{y}\,\overline{z},\ \overline{w}\,x\,\overline{y}\,z]$ in the Karnaugh map in Figure 5.27(b), which has a transition cube $\overline{w}\,\overline{y}$ with $m_3 = m_4 = \overline{w}\,\overline{x}\,\overline{y}\,z$. This transition has a static function hazard since $f(\overline{w}\,\overline{x}\,\overline{y}\,\overline{z}) = f(\overline{w}\,x\,\overline{y}\,z) = 1$ and $f(\overline{w}\,\overline{x}\,\overline{y}\,z) = 0$. The transition $[\overline{w}\,x\,y\,\overline{z},\ w\,\overline{x}\,y\,z]$ has a dynamic function hazard where $m_3 = w\,x\,y\,\overline{z}$ and $m_4 = w\,x\,y\,z$, since $f(m_1) = 1$, $f(m_3) = 0$, $f(m_4) = 1$, and $f(m_2) = 0$.

The following theorem states that if a transition has a function hazard, there does not exist an implementation of the function which avoids the hazard during this transition.

Theorem 5.10 (Eichelberger, 1965) *If a Boolean function, f, contains a function hazard for the input change m_1 to m_2, it is impossible to construct a logic gate network realizing f such that the possibility of a hazard pulse occurring for this transition is eliminated.*

Fortunately, as explained later, the synthesis method for XBM machines never produces a design with a transition that has a function hazard.

5.5.3 Combinational Hazards

Allowing multiple inputs to change besides introducing the potential of function hazards also complicates the elimination of combinational hazards. Even in the SIC case, if we use a minimum transition time state assignment, we must deal with static combinational hazards. Since after the input(s) change and the output and next state logic are allowed to stabilize (under the fundamental mode assumption), multiple changing next-state variables may be fed back to the input of the FSM. Again, the circuit moves from one minterm m_1 to another minterm m_2, but this time, multiple state variables may be changing concurrently. If we assume that we have only *normal flow tables* (all unstable states lead directly to stable states), and outputs are changed only in unstable states, then the only possible transitions are static ones. In other words, we restrict the set of allowable flow tables such that fed-back state variable changes cannot produce further output or next-state variable changes. Therefore, for state variable changes, we only need to consider static hazards.

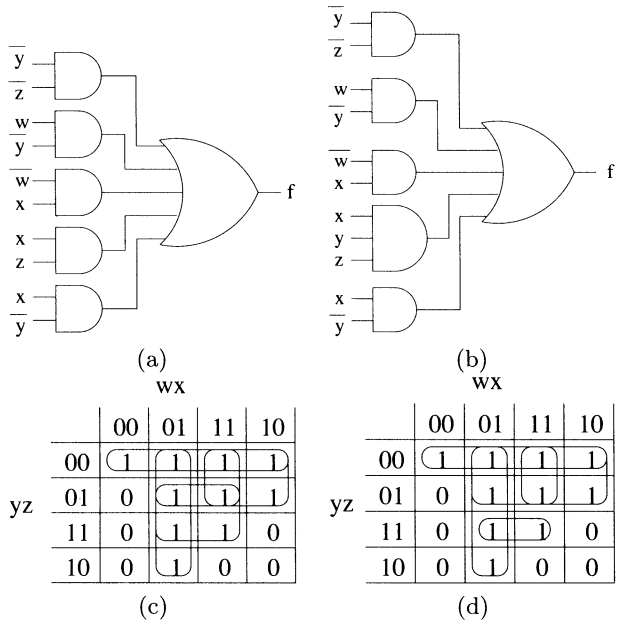


Fig. 5.28 (a) Static hazard-free circuit. (b) Dynamic hazard-free circuit. (c) Static hazard-free cover. (d) Dynamic hazard-free cover.

The results for static hazards in the MIC case are similar to those for the SIC case. Again, there can be no static 0-hazards. Static 1-hazards are a little more interesting. Since multiple variables may be changing concurrently, the cover may pass through other minterms along the way between m_1 and m_2. To be free of static 1-hazards, it is necessary that a single product in the cover include all these minterms. In other words, each transition cube, $[m_1, m_2]$, where $f(m_1) = f(m_2) = 1$, must be contained in some product in the cover to eliminate static 1-hazards. These transition cubes, therefore, are called *required cubes.*

> **Example 5.5.3** The circuit shown in Figure 5.27(a) is free of static 1-hazards under the SIC model. However, for the transition $[\overline{w}\,x\,\overline{y}\,\overline{z}, w\,x\,\overline{y}\,z]$ there does not exist a product in the cover that includes this entire transition. In order to eliminate the static 1-hazard for this transition, it is necessary to add the product $x\,\overline{y}$. Now, all prime implicants are included in the cover. The resulting circuit is shown in Figure 5.28(a).

The following theorem states that a cover composed of all the prime implicants is guaranteed to be static hazard-free for every function hazard-free transition.

Theorem 5.11 (Eichelberger, 1965) *A SOP realization of f (assuming no product terms with complementary literals) will be free of all static logic hazards* iff *the realization contains all the prime implicants of f.*

The results for dynamic hazards are a bit more complicated. For each dynamic $1 \rightarrow 0$ transition $[m_1, m_2]$, if a product in the SOP cover intersects $[m_1, m_2]$ (i.e., it includes a minterm from the transition), it must also include the start point, m_1. If it does not include the start point, then if this product term is slow, it may turn on after the other product terms have turned off, causing a glitch during the $1 \rightarrow 0$ transition. For each dynamic $0 \rightarrow 1$ transition $[m_1, m_2]$, if a product in the SOP cover intersects $[m_1, m_2]$, it must also include the end point, m_2. In this case, the illegally intersecting product term may turn on and off quickly before the other product terms hold the function on. The result would be a glitch on the output. Since the transition cubes for dynamic $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions must be carefully intersected, they are called *privileged cubes.* These results are summarized in the following theorem.

Theorem 5.12 (Bredeson, 1972) *Let f be a function which contains no dynamic function hazard for the transition $[m_1, m_2]$ where $f(m_1) = 0$ and $f(m_2) = 1$, and let $m_3 \in [m_1, m_2]$ where $f(m_3) = 1$. A SOP realization of f (assuming no product terms with complementary literals) will contain no dynamic logic hazard for transition $[m_1, m_2]$ (or $[m_2, m_1]$)* iff *for any m_3 which is covered by an implicant α in the SOP cover, it is also true that α covers m_2.*

The end point of the transition cube for a dynamic $0 \rightarrow 1$ transition is also a required cube. The *transition subcubes* for each dynamic $1 \rightarrow 0$ transition are required cubes. The transition subcubes for $1 \rightarrow 0$ transition $[m_1, m_2]$ are all cubes of the form $[m_1, m_3]$ such that $f(m_3) = 1$. Note that as an optimization you can eliminate any subcube contained in another.

> **Example 5.5.4** The transition $[\overline{w}\,x\,\overline{y}\,\overline{z},\ \overline{w}\,\overline{x}\,\overline{y}\,z]$ is a dynamic $1 \rightarrow 0$ transition from the Karnaugh map in Figure 5.28(c). The cubes required for this transition are $\overline{w}\,x\,\overline{y}$ and $\overline{w}\,\overline{y}\,\overline{z}$. For this transition, the product xz illegally intersects this transition, since it does not contain the start point. On the other hand, the cube that we added to eliminate the static 1-hazard, $x\,\overline{y}$, includes the start point, so it does not illegally intersect this transition. The result is that we must reduce the prime xz to xyz to eliminate this problem. Therefore, to eliminate dynamic hazards, it may be necessary to include nonprime implicants in the cover. In fact, if we also have a static transition $[\overline{w}\,x\,\overline{y}\,z,\ wxyz]$, then there would be no solution since xz is the only product that could include this transition, and it would cause a dynamic hazard. Fortunately, this situation can always be avoided for XBM machines.

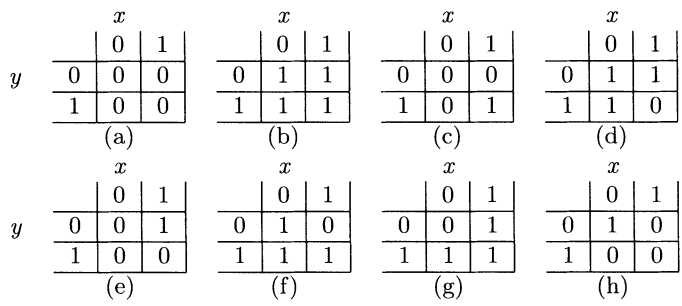


Fig. 5.29 In each example, consider the transition cube $[\overline{x}\,\overline{y}, x\,y]$: (a) legal BM $0 \rightarrow 0$ transition, (b) legal BM $1 \rightarrow 1$ transition, (c) legal BM $0 \rightarrow 1$ transition, (d) legal BM $1 \rightarrow 0$ transition, (e) illegal BM $0 \rightarrow 0$ transition, (f) illegal BM $1 \rightarrow 1$ transition, (g) illegal BM $0 \rightarrow 1$ transition, (h) illegal BM $1 \rightarrow 0$ transition.

5.5.4 Burst-Mode Transitions

If we begin with a legal BM machine specification, the types of transitions possible are restricted. Namely, a function may change value only after every transition in the input burst has occurred. A transition $[m_1, m_2]$ for a function f is a *burst-mode transition* if for every minterm $m_i \in [m_1, m_2)$, $f(m_i) = f(m_1)$.

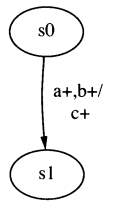
> **Example 5.5.5** Consider the Karnaugh maps shown in Figure 5.29 with the transition $[\overline{x}\,\overline{y},\ x\,y]$. The transitions in Figure 5.29(a), (b), (c), and (d) are legal burst-mode transitions. Those in Figure 5.29(e), (f), (g), and (h) are illegal since they do not maintain a constant value throughout $[\overline{x}\,\overline{y},\ x\,y)$.

If a function f has only burst-mode transitions, it is free of function hazards. Also, BM machines are free of dynamic $0 \rightarrow 1$ hazards. Finally, for any legal BM machine, there exists a hazard-free cover for each output and next-state variable before state minimization. These results are summarized in the following three theorems.

Theorem 5.13 (Nowick, 1993) *If f has a BM transition $[m_1, m_2]$, then f is free of function hazards for that transition.*

Theorem 5.14 (Nowick, 1993) *If f has a $0 \rightarrow 1$ BM transition in $[m_1, m_2]$, then a SOP implementation is free of logic hazards for this transition.*

Theorem 5.15 (Nowick, 1993) *Let G be any BM specification, let z be any output variable of G, let F be an unminimized flow table synthesized from G using an arbitrary state assignment, and let f_z be the output function for z in table F. Then the set of required cubes for f_z is a hazard-free cover.*



(a)

	ab 00	01	11	10	xy
s0	s0,0	s0,0	s1,1	s0,0	01
s1			s1,1		10

(b)

Fig. 5.30 (a) Simple burst-mode transition. (b) Corresponding flow table.

Example 5.5.6 Consider the simple burst-mode transition and corresponding flow table shown in Figure 5.30. For this burst-mode transition, the transition cube $[\overline{a}\,\overline{b}\,\overline{x}\,y,\ a\,b\,\overline{x}\,y]$ is a dynamic $0 \rightarrow 1$ transition for output c and next-state variable X and a dynamic $1 \rightarrow 0$ transition for next-state variable Y. This implies that $a\,b\,\overline{x}\,y$ is a required cube for c and X, while $a\,\overline{b}\,\overline{x}\,y$ and $\overline{a}\,b\,\overline{x}\,y$ are required cubes for Y. The cube $\overline{x}\,y$ is a priveledged cube for c, X, and Y. The transition cube $[a\,b\,\overline{x}\,y,\ a\,b\,x\,\overline{y}]$ is a static $1 \rightarrow 1$ transition for output c and next-state variable X and a static $0 \rightarrow 0$ transition for next-state variable Y. This implies that $a\,b$ is a required cube for c and X.

5.5.5 Extended Burst-Mode Transitions

When inputs are allowed to change nonmonotonically during multiple-input changes as in an XBM machine, we need to generalize the notion of transition cubes to allow the start and end points to be cubes rather than simply minterms. In the *generalized transition cube* $[c_1, c_2]$, the cube c_1 is called the *start cube* and c_2 is called the *end cube*. The *open generalized transition cube*, $[c_1, c_2)$, is defined to be all minterms in $[c_1, c_2]$ excluding those in c_2 (i.e., $[c_1, c_2) = [c_1, c_2] - c_2$).

In an XBM machine, some signals are rising, some are falling, and others are levels which can change nonmonotonically. Rising and falling signals change monotonically (i.e., at most once in a legal transition cube). Level signals must hold the same value in c_1 and c_2, where the value is either a constant (0 or 1) or a don't care (−). Level signals, if they are don't care, may change nonmonotonically. In an XBM machine, the types of transitions are again restricted such that each function may change value only after the completion of an input burst. A generalized transition $[c_1, c_2]$ for a function f is an *extended burst-mode transition* if for every minterm $m_i \in [c_1, c_2)$, $f(m_i) = f(c_1)$ and for every minterm $m_i \in c_2$, $f(m_i) = f(c_2)$. The following theorem states that if a function has only extended burst-mode transitions, then it is function hazard-free.

yz \ xl	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	1	1	0	0
10	1	1	0	0

(a)

yz \ xl	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	0	1	0
10	0	0	0	0

(b)

Fig. 5.31 (a) Example extended burst-mode transition. (b) Example illustrating dynamic $0 \rightarrow 1$ hazard issues.

Theorem 5.16 (Yun, 1999) *Every extended burst-mode transition is function hazard-free.*

Example 5.5.7 Consider the Karnaugh map shown in Figure 5.31(a), where x, y, and z are transition signals and l is a level signal. During the generalized transition $[\overline{x} - \overline{y}-, x - y-]$, x and y will rise, z is a directed don't-care (assume that it is rising), and l is a level signal which may change value arbitrarily. This transition is an extended burst-mode transition, since f is always 1 in $[\overline{x} - \overline{y}-, x - y-)$ and f is always 0 while in the cube $x - y-$.

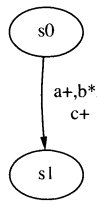
Example 5.5.8 Consider the generalized transition $[\overline{x} - \overline{y}-, x - yz]$ shown in Figure 5.31(a). In this transition, x, y, and z will rise. This transition, however, is not an extended burst-mode transition, since in $[\overline{x} - \overline{y}-, x - yz)$, f can be either 1 or 0.

The *start subcube*, c_1', is a maximal subcube of c_1 such that each signal undergoing a directed don't-care transition is set to its initial value (i.e., 0 for a rising transition and 1 for a falling transition). The *end subcube*, c_2', is a maximal subcube of c_2 such that each signal undergoing a directed don't-care transition is set to its final value (i.e., 1 for a rising transition and 0 for a falling transition).

Example 5.5.9 Consider again the transition $[\overline{x} - \overline{y}-,x - y-]$ in Figure 5.31(a), where z is a rising directed don't care. The start subcube is $\overline{x} - \overline{y}\,\overline{z}$ and the end subcube is $x - yz$.

If we consider the transition cube $[c_1', c_2']$, the hazard considerations are the same as before. In other words, if this is a static $1 \rightarrow 1$ transition [i.e., $f(c_1') = f(c_2') = 1$], the entire transition cube must be included in some product term in the cover. If it is a dynamic $1 \rightarrow 0$ transition, then any product that intersects this transition cube must contain the start subcube, c_1'. Unlike burst mode, dynamic $0 \rightarrow 1$ transitions must be considered. Namely, any product that intersects the transition cube for a dynamic $0 \rightarrow 1$ transition must contain the end subcube, c_2'. These results are summarized in the following theorems.

Theorem 5.17 (Yun, 1999) *The output of a SOP implementation is hazard-free during a* $1 \rightarrow 0$ *extended burst-mode transition iff every product term intersecting the transition cube* $[c_1, c_2]$ *also contains the start subcube* c_1'.



(a)

	ab 00	01	11	10	xy
s0	s0,0	s0,0	s1,1	s1,1	01
s1			s1,1	s1,1	10

(b)

Fig. 5.32 (a) Simple extended burst-mode transition. (b) Corresponding flow table.

Theorem 5.18 (Yun, 1999) *The output of a SOP implementation is hazard-free during a* $0 \rightarrow 1$ *extended burst-mode transition iff every product term intersecting the transition cube* $[c_1, c_2]$ *also contains the end subcube* c_2'.

Example 5.5.10 Consider the Karnaugh map shown in Figure 5.31(b) and the dynamic $0 \rightarrow 1$ transitions $[\overline{x} - \overline{y}\,\overline{z}, x - \overline{y}z]$ and $[\overline{x}l\overline{y}\overline{z}, xlyz]$. If we implement these two transitions with the logic $f = x\overline{y}z + xlz$, there is a dynamic $0 \rightarrow 1$ hazard. The problem is that the cube xlz illegally intersects the transition $[\overline{x} - \overline{y}\,\overline{z}, x - \overline{y}z]$ since it does not contain the entire end subcube. To eliminate this hazard, it is necessary to reduce xlz to $xlyz$.

Example 5.5.11 Consider the simple extended burst-mode transition and corresponding flow table shown in Figure 5.32. For this extended burst-mode transition, the transition cube $[a\,\overline{x}\,y,\ a\,\overline{x}\,y]$ is a dynamic $0 \rightarrow 1$ transition for output c and next-state variable X and a dynamic $1 \rightarrow 0$ transition for next-state variable Y. This implies that $a\,\overline{x}\,y$ is a required cube for c and X, while $\overline{a}\,\overline{x}\,y$ is a required cube for Y. Assuming that b is a rising directed don't care, the start subcube for this transition is $\overline{a}\,\overline{b}\,\overline{x}\,y$ and the end subcube is $a\,b\,\overline{x}\,y$. The cube $\overline{x}\,y$ is a priveledged cube for c, X, and Y. Therefore, no cube in the cover for c or X can intersect this cube unless it includes $a\,b\,\overline{x}\,y$. No cube in the cover for Y can intersect this cube unless it includes $\overline{a}\,\overline{b}\,\overline{x}\,y$. The transition cube $[a\,\overline{x}\,y,\ a\,x\,\overline{y}]$ is a static $1 \rightarrow 1$ transition for output c and next-state variable X and a static $0 \rightarrow 0$ transition for next-state variable Y. This implies that $a\,b$ is a required cube for c and X.

Unfortunately, not every XBM transition can be implemented free of dynamic hazards. Consider the following example.

Example 5.5.12 The circuit shown in Figure 5.33(a) latches a conditional signal and converts it to a dual-rail signal. It can be described using the XBM machine in Figure 5.33(b). A circuit implementation for signal x derived from this XBM machine is shown in Figure 5.33(c) and its corresponding Karnaugh map is shown in Figure 5.33(d). In the circuit shown in Figure 5.33(c), if when c goes high, d is high, then x is set to high. This signal is fed-back as a state variable X. This feedback creates a static $1 \rightarrow 1$ transition $[dc\overline{X}\,\overline{Y}, dcX\overline{Y}]$, so the product $dc\overline{Y}$ must be completely covered by some product in the cover to

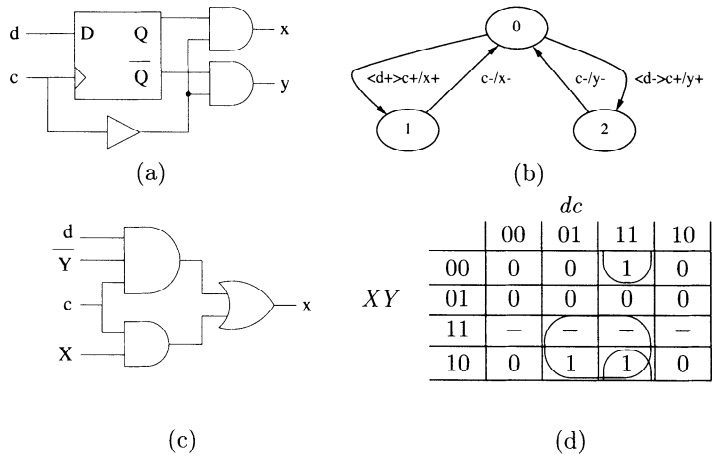


Fig. 5.33 (a) dff Circuit that latches a conditional signal and converts it to a dual-rail signal. (b) XBM machine to describe circuit from (a). (c) Asynchronous implementation with a dynamic hazard. (d) K-map showing the dynamic $1 \rightarrow 0$ hazard.

prevent a static $1 \rightarrow 1$ hazard. A hold time after x goes high, the conditional signal d is allowed to change. At some point in the future with d now at either value, c goes low. This starts a dynamic $1 \rightarrow 0$ transition $[-cX\overline{Y}, -\overline{c}X\overline{Y}]$ which can be represented by the product $X\overline{Y}$. This product must not be intersected unless the intersecting product term also includes the start subcube (i.e., $cX\overline{Y}$). However, the product $dc\overline{Y}$ must intersect this product, but it cannot contain the entire start subcube. The result is that there is no hazard-free solution.

To address this problem, we must modify the state assignment as well as the machine operation in these situations. This solution is discussed in Section 5.5.7 where we address state assignment.

5.5.6 State Minimization

Using the state minimization procedure described in Section 5.2, it is possible that no hazard-free cover exists for some variable in the design. This procedure will need to be modified for the MIC case. We illustrate this through an example.

Example 5.5.13 Consider the flow table fragment in Figure 5.34(a). Under our original definition of compatibility, states A and D can be merged to form the new state shown in Figure 5.34(b). There is a static $1 \rightarrow 1$ transition from input $a\,\overline{b}\,\overline{c}$ to $a\,\overline{b}\,c$ which has transition cube $a\,\overline{b}$. Recall that to be free of static 1-hazards, the product $a\,\overline{b}$ must be included in some product in the final cover. There is also a dynamic $1 \rightarrow 0$ transition from input $\overline{a}\,\overline{b}\,\overline{c}$ to $a\,b\,\overline{c}$ which has start point $\overline{a}\,\overline{b}\,\overline{c}$ and tran-

Inputs a b c

State	000	001	011	010	110	111	101	100
A	A,1	C,0	–	A,1	B,0	–	–	A,1
...								
D	–	–	–	–	–	–	E,1	D,1

(a)

Inputs a b c

State	000	001	011	010	110	111	101	100
AD	A,1	C,0	–	A,1	B,0	–	E,1	A,1

(b)

Fig. 5.34 (a) Fragment of a flow table. (b) Illegal state merging.

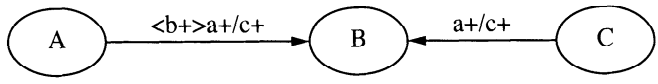
sition cube $\bar{c}$. Note that the product $a\bar{b}$ intersects this transition cube, but it does not include the start point $\bar{a}\bar{b}\bar{c}$. Therefore, this intersection is illegal and leads to a dynamic $1 \rightarrow 0$ hazard. However, if we restrict our product to $a\bar{b}c$ so that it no longer illegally intersects the dynamic transition, it no longer covers the entire static transition, so we have a static 1-hazard.

The use of conditional signals in XBM machines further complicates state minimization, which we again illustrate with an example.

Example 5.5.14 Consider the fragment of an XBM machine shown in Figure 5.35(a) and corresponding flow table shown in Figure 5.35(b). State A is not output compatible with either state B or C, due to the entry for input 10. States B and C are compatible, and combining them results in the new flow table shown in Figure 5.35(c). For input 11 in state A, the machine is transitioning to state B and there is a static $1 \rightarrow 1$ transition on output c which must be included completely by some product in the cover for it to be hazard-free. While in state BC, there is a dynamic $0 \rightarrow 1$ transition on output c, $[0-, 1-]$. Both the end cube and end subcube for this transition are $1-$, which must not be intersected without being included completely. However, in the minimized machine the required cube for the static transition during the state change cannot be expanded to include the entire end subcube, nor can it be reduced so as not to intersect the end cube. Therefore, there is no hazard-free solution to the flow table shown in Figure 5.35(c). Therefore, states B and C should not be combined.

To solve these problems, we must restrict the conditions under which two states are compatible. Under these new restrictions on state minimization, it can be proven that a hazard-free cover can always be found. Two states s_1 and s_2 are *dhf-compatible* (or dynamic hazard-free compatible) when they are compatible and for each output z and transition $[c_1, c_2]$ of s_1 and for each transition $[c_3, c_4]$ of s_2:

1. If z has a $1 \rightarrow 0$ transition in $[c_1, c_2]$ and a $1 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$ or $c_1' \in [c_3, c_4]$.



(a)

Inputs a b

State	00	01	11	10
A	A,0	A,0	B,1	D,0
B	–	–	B,1	B,1
C	C,0	C,0	B,1	B,1

(b)

Inputs a b

State	00	01	11	10
A	A,0	A,0	B,1	D,0
BC	B,0	B,0	B,1	B,1

(c)

Fig. 5.35 (a) XBM machine fragment showing potential for dynamic hazard. (b) Initial flow table. (c) Illegal state merging.

2. If z has a $1 \rightarrow 0$ transition in $[c_1, c_2]$ and a $1 \rightarrow 0$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$, $c_1 = c_3$, $[c_1, c_2] \subseteq [c_3, c_4]$, or $[c_3, c_4] \subseteq [c_1, c_2]$.

3. If z has a $0 \rightarrow 1$ transition in $[c_1, c_2]$ and a $1 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$ or $c_2' \in [c_3, c_4]$.

4. If z has a $0 \rightarrow 1$ transition in $[c_1, c_2]$ and a $0 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$, $c_2 = c_4$, $[c_1, c_2] \subseteq [c_3, c_4]$, or $[c_3, c_4] \subseteq [c_1, c_2]$.

The states s_1 and s_2 must also satisfy the following further restriction for each s_3, which can transition to s_1 in $[c_3, c_4]$ and another transition $[c_1, c_2]$ of s_2:

1. If z has a $1 \rightarrow 0$ transition in $[c_1, c_2]$ and a $1 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$ or $c_1' \in [c_3, m_4]$.

2. If z has a $0 \rightarrow 1$ transition in $[c_1, c_2]$ and a $1 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$ or $c_2' \in [c_3, c_4]$.

Similarly, for each s_3 which can transition to s_2 in $[c_3, c_4]$ and another transition $[c_1, c_2]$ of s_1:

1. If z has a $1 \rightarrow 0$ transition in $[c_1, c_2]$ and a $1 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$ or $c_1' \in [c_3, m_4]$.

2. If z has a $0 \rightarrow 1$ transition in $[c_1, c_2]$ and a $1 \rightarrow 1$ transition in $[c_3, c_4]$, then $[c_1, c_2] \cap [c_3, c_4] = \emptyset$ or $c_2' \in [c_3, c_4]$.

pxy \ dc	00	01	11	10
000	0	0	0	0
001	–	–	–	–
011	–	–	–	–
010	–	–	–	–
110	0	1	1	0
111	–	–	–	–
101	–	–	–	–
100	0	1	1	0

(a)

pxy \ dc	00	01	11	10
000	0	0	1	0
001	–	–	–	–
011	–	–	–	–
010	–	–	–	–
110	0	1	1	0
111	–	–	–	–
101	–	–	–	–
100	0	1	1	0

(b)

Fig. 5.36 Partial Karnaugh maps for (a) X and (b) P.

5.5.7 State Assignment

The state assignment method described earlier can be used directly to find a critical race free state assignment under BM operation. Since state changes happen only after all inputs have changed, the state change can be thought to occur within the individual column, and the number of input changes does not matter.

For XBM machines, it may be necessary to add additional state variables to eliminate dynamic hazards when there are conditional signals. Consider again the example from Figure 5.33. In this example, there is an unavoidable dynamic hazard. To solve this problem, a new state variable is added for each conditional input burst in which the next input burst is unconditional and enables an output to fall. This state variable is set to high after the conditional input burst but before the output burst is allowed to begin. Intuitively, this state variable is storing the value of the conditional signal. It can be set low in the next output burst.

Example 5.5.15 Consider again the example from Figure 5.33. Partial Karnaugh maps (q is omitted) after adding a new state variable p are shown in Figure 5.36(a) for output X and Figure 5.36(b) for state variable P. The privileged cube is $px\overline{y}$, which cannot be intersected without including the start subcube, which is $cpx\overline{y}$. Now, the implementation for x is simply cp, which legally intersects this privileged cube. The new state variable, P, can be implemented with the function $cp + dc\overline{x}\,\overline{q}$ (note that q is not shown in the figure), which again does not illegally intersect the dynamic transition, so the resulting circuit shown in Figure 5.37 is now hazard-free.

5.5.8 Hazard-Free Two-Level Logic Synthesis

In this section we describe two-level logic synthesis in the presence of multiple input changes. In this case, the union of the required cubes forms the ON-

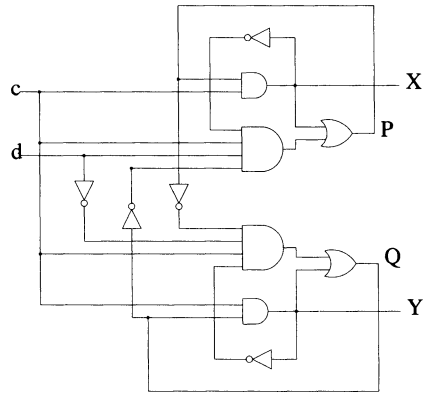


Fig. 5.37 Hazard-free dff circuit.

set for the function. Each of the required cubes must be contained in some product of the cover to ensure hazard freedom.

Example 5.5.16 Consider the Karnaugh map shown in Figure 5.38, which has the following transition cubes:

$$\begin{aligned} t_1 &= [a\,\overline{b}\,\overline{c}\,d, a\,b\,\overline{c}\,\overline{d}] \\ t_2 &= [a\,\overline{b}\,c\,\overline{d}, a\,\overline{b}\,c\,d] \\ t_3 &= [\overline{a}\,b\,\overline{c}\,\overline{d}, \overline{a}\,\overline{b}\,\overline{c}\,d] \\ t_4 &= [\overline{a}\,b\,c\,d, a\,\overline{b}\,c\,\overline{d}] \end{aligned}$$

Transition t_1 is a $1 \rightarrow 1$ transition, so it produces a required cube $a\,\overline{c}$. Transition t_2 is a $0 \rightarrow 0$ transition, so it does not produce any required cube. Transition t_3 is a $1 \rightarrow 0$ transition, so it produces a required cube for each transition subcube. First, the transition $[\overline{a}\,b\,\overline{c}\,\overline{d}, \overline{a}\,\overline{b}\,\overline{c}\,\overline{d}]$ produces the required cube $\overline{a}\,\overline{c}\,\overline{d}$. Second, the transition $[\overline{a}\,b\,\overline{c}\,\overline{d}, \overline{a}\,b\,\overline{c}\,d]$ produces the required cube $\overline{a}\,b\,\overline{c}$. Transition t_4 is also a $1 \rightarrow 0$ transition, so again we produce a required cube for each transition subcube. The result are two additional required cubes bcd and $\overline{a}\,c$ (note that the required cubes $\overline{a}\,c\,d$ and $\overline{a}\,b\,c$ are both contained in $\overline{a}\,c$, so they can be discarded). This gives us a final required cube set of

$$\text{req-set} = \{a\overline{c}, \overline{a}\,\overline{c}\,\overline{d}, \overline{a}\,b\,\overline{c}, b\,c\,d, \overline{a}\,c\}$$

The transition cubes for each dynamic $1 \rightarrow 0$ transition are privileged cubes since they cannot be intersected unless the intersecting product includes its start subcube. Similarly, transition cubes for each dynamic $0 \rightarrow 1$ transition are also privileged cubes since they cannot be intersected unless the intersecting product includes its end subcube. If a cover includes a product that intersects a privileged cube without including its corresponding start subcube or end subcube, the cover is not hazard-free.

		ab			
		00	01	11	10
cd	00	1	1	1	1
	01	0	1	1	1
	11	1	1	1	0
	10	1	1	0	0

Fig. 5.38 Karnaugh map for small two-level logic minimization example.

Example 5.5.17 Transitions t_3 and t_4 from Example 5.5.16 are dynamic $1 \rightarrow 0$ transitions, so their corresponding transition cubes are privileged cubes. This gives us the following set of privileged cubes:

$$\text{priv-set} \quad = \quad \{\overline{a}\,\overline{c}, c\}$$

Recall that we may not be able to produce a SOP cover that is free of dynamic hazards using only prime implicants, so we introduce the notion of a *dynamic-hazard-free implicant* (or *dhf-implicant*). A dhf-implicant is an implicant which does not illegally intersect any privileged cube. A *dhf-prime implicant* is a dhf-implicant that is contained in no other dhf-implicant. Note that a dhf-prime implicant may actually not be a prime implicant. A minimal hazard-free cover includes only dhf-prime implicants.

The first step to find all dhf-prime implicants is to find the ordinary prime implicants using the recursive procedure described earlier. We can start this procedure using a SOP that contains the required cubes and a set of implicants that represents the DC-set. After finding the prime implicants, we next check each prime to see if it illegally intersects a privileged cube. If it does, we attempt to shrink the cube to make it dhf-prime.

Example 5.5.18 The recursion to find the primes proceeds as follows:

$$\begin{aligned}
f(a,b,c,d) &= a\,\overline{c} + \overline{a}\,\overline{c}\,\overline{d} + \overline{a}\,b\,\overline{c} + b\,c\,d + \overline{a}\,c \\
f(a,b,0,d) &= a + \overline{a}\,\overline{d} + \overline{a}\,b \\
f(0,b,0,d) &= \overline{d} + b \\
f(1,b,0,d) &= 1 \\
\text{cs}(f(a,b,c,0)) &= \text{abs}((a + \overline{d} + b)(\overline{a} + 1)) = a + \overline{d} + b \\
f(a,b,1,d) &= b\,d + \overline{a} \\
f(0,b,1,d) &= 1 \\
f(1,b,1,d) &= b\,d \\
\text{cs}(f(a,b,1,d)) &= \text{abs}((a + 1)(\overline{a} + b\,d)) = \overline{a} + b\,d \\
\text{cs}(f(a,b,c,d)) &= \text{abs}((c + a + \overline{d} + b)(\overline{c} + \overline{a} + b\,d)) \\
&= \text{abs}(\overline{a}\,c + b\,c\,d + a\,\overline{c} + a\,b\,d + \overline{c}\,\overline{d} + \overline{a}\,\overline{d} + b\,\overline{c} + \\
&\quad \overline{a}\,b + b\,d) \\
&= \overline{a}\,c + a\,\overline{c} + \overline{c}\,\overline{d} + \overline{a}\,\overline{d} + b\,\overline{c} + \overline{a}\,b + b\,d
\end{aligned}$$

The next step is to determine which primes illegally intersect a privileged cube. First, the prime $a\,\overline{c}$ does not intersect any privileged cube, so it is a dhf-prime. The primes $\overline{c}\,\overline{d}$ and $b\,\overline{c}$ intersect the privileged cube $\overline{a}\,\overline{c}$, but they include the start subcube $\overline{a}\,b\,\overline{c}\,\overline{d}$. Since they do not intersect the other privileged cube, c, they are also dhf-primes. The prime $\overline{a}\,c$ intersects the privileged cube c, but it also includes the start subcube $\overline{a}\,b\,c\,d$. Therefore, since it does not intersect the other privileged cube $\overline{a}\,\overline{c}$, it is also a dhf-prime. The prime $\overline{a}\,b$ intersects both privileged cubes, but it also includes both start subcubes. The last two primes, $\overline{a}\,\overline{d}$ and $b\,d$, also intersect both privileged cubes. However, the prime $\overline{a}\,\overline{d}$ does not include the start subcube of c (i.e., $\overline{a}\,b\,c\,d$), and the prime $b\,d$ does not include the start subcube of $\overline{a}\,\overline{c}$ (i.e., $\overline{a}\,b\,\overline{c}\,\overline{d}$). Therefore, these last two primes are not dhf-prime implicants.

For these two remaining primes, we must shrink them until they no longer illegally intersect any privileged cubes. Let us consider first $\overline{a}\,\overline{d}$. We must add a literal that makes this prime disjoint from the privileged cube c. The only choice is to add $\overline{c}$ to make the new implicant, $\overline{a}\,\overline{c}\,\overline{d}$. This implicant no longer intersects the privileged cube c and it legally intersects the privileged cube $\overline{a}\,\overline{c}$ (i.e., it includes its start subcube). Note, though, that $\overline{a}\,\overline{c}\,\overline{d}$ is contained in the dhf-prime $\overline{c}\,\overline{d}$, so it is not a dhf-prime and can be discarded. Next, consider the prime $b\,d$. We must make it disjoint with the privileged cube $\overline{a}\,\overline{c}$. There are two possible choices, $a\,b\,d$ or $b\,c\,d$, but the implicant $a\,b\,d$ intersects the privileged cube c illegally, so it must be reduced further to $a\,b\,\overline{c}\,d$. This implicant, however, is contained in the dhf-prime $a\,\overline{c}$, so it can be discarded. The implicant $b\,c\,d$ does not illegally intersect any privileged cubes and is not contained in any other dhf-prime, so it is a dhf-prime. Our final set of dhf-prime implicants is

$$\text{dhf-prime} \quad = \quad \{\overline{a}\,c, a\,\overline{c}, \overline{c}\,\overline{d}, b\,\overline{c}, \overline{a}\,b, b\,c\,d\}$$

The two-level hazard-free logic minimization problem for XBM operation is to find a minimum-cost cover which covers every required cube using only dhf-prime implicants.

Example 5.5.19 The constraint matrix is shown below.

	$\overline{a}\,c$	$a\,\overline{c}$	$\overline{c}\,\overline{d}$	$b\,\overline{c}$	$\overline{a}\,b$	$b\,c\,d$
$a\,\overline{c}$	–	1	–	–	–	–
$\overline{a}\,\overline{c}\,\overline{d}$	–	–	1	–	–	–
$\overline{a}\,b\,\overline{c}$	–	–	–	1	1	–
$b\,c\,d$	–	–	–	–	–	1
$\overline{a}\,c$	1	–	–	–	–	–

There are four essential rows which make $\overline{a}\,c$, $a\,\overline{c}$, $\overline{c}\,\overline{d}$, and $b\,c\,d$ essential. To solve the remaining row, either the prime $b\,\overline{c}$ or $\overline{a}\,b$ can be selected. Therefore, there are two possible minimal hazard-free SOP covers which each include five product terms. Note that both covers require a non-

		abc						
		000	001	011	010	110	111	101	100
de	00	0	0	0	0	0	0	0	0
	01	0	0	0	0	0	0	0	0
	10	0	0	1	1	1	1	1	1
	11	0	0	1	1	0	0	1	1

Fig. 5.39 Karnaugh map for Example 5.5.20.

prime implicant, bcd. A minimal hazardous cover would only require four product terms, $a\bar{c}$, bd, $\bar{a}c$, and $\bar{c}\bar{d}$.

Our next example illustrates the hazard issues that must be addressed in the presence of directed don't cares and conditional signals.

Example 5.5.20 Consider the Karnaugh map shown in Figure 5.39 with the following generalized transition cubes:

$$\begin{aligned} t_1 &= [\bar{a}\,\bar{b}\,\bar{d}, \bar{a}\,b\,d] \\ t_2 &= [a\,b\,\bar{c}\,d\,\bar{e}, a\,b\,c\,d\,\bar{e}] \\ t_3 &= [a\,\bar{b}\,d, a\,\bar{b}\,\bar{d}] \end{aligned}$$

The first transition is a dynamic $0 \rightarrow 1$ transition which is triggered by b and d going high. During this transition, c is a falling directed don't care and e is an unspecified conditional signal. This transition contributes the required cube $\bar{a}\,b\,d$ and the privileged cube $\bar{a}$. This privileged cube must not be intersected except by a product which includes the entire end subcube, which is $\bar{a}\,b\,\bar{c}\,d$. The second transition is a static $1 \rightarrow 1$ transition which contributes the required cube $a\,b\,d\,\bar{e}$. Finally, the third transition is a dynamic $1 \rightarrow 0$ transition which is triggered by d going low. During this transition, c is a rising directed don't care and e is an unspecified conditional signal. The required cube for this transition is $a\,\bar{b}\,d$. This transition also makes $a\,\bar{b}$ a privileged cube which can only be intersected by products that include its start subcube, $a\,\bar{b}\,\bar{c}\,d$.

The primes for this example are:

$$\bar{a}\,b\,d, a\,\bar{b}\,d, a\,d\,\bar{e}, b\,d\,\bar{e}$$

The prime $a\,d\,\bar{e}$ intersects the privileged cube $a\,\bar{b}$, but it does not include its start subcube, $a\,\bar{b}\,\bar{c}\,d$. Therefore, this prime must be reduced to $a\,b\,d\,\bar{e}$, which does not intersect the privileged cube. Similarly, the prime $b\,d\,\bar{e}$ intersects the privileged cube $\bar{a}$ and does not contain its end subcube $\bar{a}\,b\,\bar{c}\,d$. Again, this prime is reduced to $a\,b\,d\,\bar{e}$ to eliminate the problem. Therefore, the final cover of the set of required cubes is

$$f = \bar{a}\,b\,d + a\,\bar{b}\,d + a\,b\,d\,\bar{e}$$

Again, a nonprime implicant is needed to be hazard-free.

5.6 MULTILEVEL LOGIC SYNTHESIS

Typically, two-level SOP implementations cannot be realized directly for most technologies. The reason is that the AND or OR stages of the gate could have arbitrarily large fan-in (i.e., numbers of inputs). In CMOS, for example, gates with more than three or four inputs are considered to be too slow. Therefore, two-level SOP implementations must be decomposed into multilevel implementations using laws of Boolean algebra. Again, however, care must be taken not to introduce hazards. In this section we present a number of *hazard-preserving transformations*. Therefore, if we begin with a hazard-free SOP implementation and apply only hazard-preserving transformations, the resulting multilevel implementation is also hazard-free. The following theorem gives the laws of Boolean algebra which are hazard-preserving.

Theorem 5.19 (Unger, 1969) *Given any expression f_1, if we transform it into another expression, f_2, using the following laws:*

- $A + (B + C) \Leftrightarrow A + B + C$ *(associative law)*
- $A(BC) \Leftrightarrow ABC$ *(associative law)*
- $\overline{(A + B)} \Leftrightarrow \overline{A}\,\overline{B}$ *(DeMorgan's theorem)*
- $\overline{(AB)} \Leftrightarrow \overline{A} + \overline{B}$ *(DeMorgan's theorem)*
- $AB + AC \Rightarrow A(B + C)$ *(distributive law)*
- $A + AB \Rightarrow A$ *(absorptive law)*
- $A + \overline{A}B \Rightarrow A + B$

then a circuit corresponding to f_2 will have no combinational hazards not present in circuits corresponding to f_1.

In other words, if we transform a circuit into a new circuit using the laws listed above the two circuits have equivalent hazards. Therefore, if the original circuit is hazard-free, so is the new circuit. Note that the last three laws can be applied only in one direction. For example, the distributive law in the reverse direction [i.e., $A(B + C) \Rightarrow AB + AC$] preserves static hazards, but it may introduce new dynamic hazards. For example, the function $f = A(\overline{A} + B)$ is free of dynamic hazards. If we multiply it out, we get $A\overline{A} + AB$, which has a dynamic hazard when $B = 1$ and A changes.

We can also create new functions from others, with the following effects:

- Insertion or deletion of inverters at the output of a circuit only interchanges 0 and 1-hazards.
- Insertion or deletion of inverters at the inputs only relocates hazards to duals of the original transition.

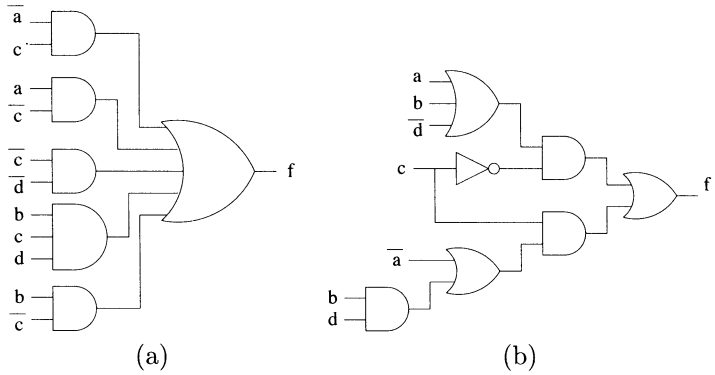


Fig. 5.40 (a) SOP implementation for small example. (b) Multilevel implementation for small example.

- The dual of a circuit (exchange AND and OR gates) produces a dual function with dual hazards.

Furthermore, there are many other hazard-preserving transformations not mentioned here.

In order to derive a hazard-free multilevel implementation, we first find a hazard-free SOP implementation. If we then convert it to a multilevel implementation using only the associative law, DeMorgan's theorem, factoring (not multiplying out), $A + AB \rightarrow A$, and $A + \overline{A}B \rightarrow A + B$, then it is also hazard-free. Similarly, to check a multilevel implementation for hazards, convert it to a SOP implementation using associative, distributive, and DeMorgan's laws and check for hazards (be sure not to perform any reductions using $A\overline{A} = 0$).

Example 5.6.1 Consider the two-level hazard-free implementation derived in Section 5.5.8, which has 11 literals as shown in Figure 5.40(a):

$$f = \overline{a}\,c + a\,\overline{c} + \overline{c}\,\overline{d} + b\,c\,d + b\,\overline{c}$$

Using factoring, we can obtain the following equation with eight literals, which is shown in Figure 5.40(b):

$$f = \overline{c}\,(a + b + \overline{d}) + c\,(\overline{a} + b\,d)$$

This is hazard-free since factoring is a hazard-preserving operation.

5.7 TECHNOLOGY MAPPING

After deriving optimized two-level or multilevel logic, the next step is to map these logic equations to some given gate library. This *technology mapping* step takes as input a set of technology-independent logic equations and a library

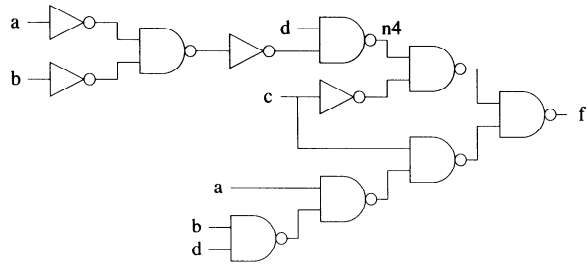


Fig. 5.41 NAND tree decomposition of the function $f = \overline{c}\,(a + b + \overline{d}) + c\,(\overline{a} + b\,d)$.

of cells implemented in some gate-array or standard-cell technology, and it produces a netlist of cells implementing the logic in the given technology. The technology mapping process is traditionally broken up into three major steps: *decomposition*, *partitioning*, and *matching/covering*. In this section we describe how these steps can be performed without introducing hazards.

The decomposition step transforms the network of logic equations into an equivalent network using only two-input/one-output *base functions*. A typical choice of base function is two-input NAND gates. Decomposition can be performed using recursive applications of DeMorgan's theorem and the associative law. As described in Section 5.6, these operations are hazard-preserving. This means that if you begin with a set of hazard-free logic equations, the equivalent network using only base functions is also hazard-free. Some technology mappers may perform simplification during the decomposition step. This process may remove redundant logic that has been added to eliminate hazards, so this simplification must be avoided.

Example 5.7.1 Consider the multilevel circuit from Section 5.6:

$$
\begin{aligned}
f &= \overline{c}\,(a + b + \overline{d}) + c\,(\overline{a} + b\,d) \\
f &= \overline{c}\,((a + b) + \overline{d}) + c\,(\overline{a} + b\,d) \text{ (associative law)} \\
f &= \overline{c}\,(\overline{(\overline{a}\,\overline{b})} + \overline{d}) + c\,(\overline{a} + b) \text{ (DeMorgan's theorem)} \\
f &= \overline{c\,(\overline{(\overline{a}\,\overline{b})}\,d)} + \overline{c\,(a\,\overline{(b\,d)})} \text{ (DeMorgan's theorem)} \\
f &= \overline{\overline{(\overline{c}\,(\overline{(\overline{a}\,\overline{b})}\,d))}\;\overline{(c\,(a\,\overline{(b\,d)}))}} \text{ (DeMorgan's theorem)}
\end{aligned}
$$

The NAND decomposition is depicted in Figure 5.41.

The partitioning step breaks up the decomposed network at points of multiple fanout into single output cones of logic which are to be individually mapped. Since the partitioning step does not change the topology of the network, it does not affect the hazard behavior of the network.

The matching and covering step examines each individual cone of logic and finds cells in the gate library to implement subnetworks within the cone. This matching step can be implemented either using *structural pattern-matching*

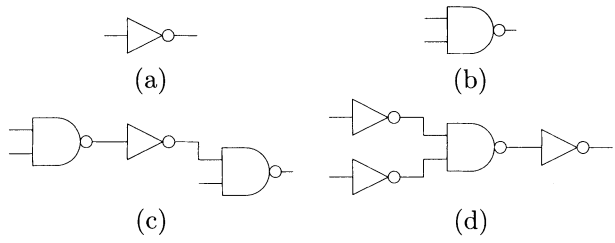


Fig. 5.42 (a) Inverter. (b) Two-input NAND gate. (c) Three-input NAND gate. (d) Two-input NOR gate.

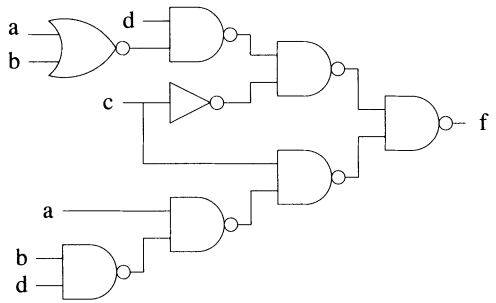


Fig. 5.43 Final mapped circuit for $f = \bar{c}(a + b + \bar{d}) + c(\bar{a} + b\,d)$.

techniques or *Boolean matching* techniques. In the structural techniques, each library element is also decomposed into base functions. Library elements are then compared against portions of the network to be mapped using pattern matching. Assuming that the decomposed logic and library gates are hazard-free, the resulting mapped logic is also hazard-free.

> **Example 5.7.2** Structural matching is applied to the network shown in Figure 5.41 with the library of gates shown in Figure 5.42. Assume that the inverter has cost 1, the two-input NAND gate has cost 3, the three-input NAND gate has cost 5, and the two-input NOR gate has cost 2. Consider the covering of the subtree starting at node $n4$. This subtree can either be covered with a three-input NAND gate and two inverters at a total cost of 7, or it can be covered with a two-input NAND gate and a two-input NOR gate at a cost of 5. The second choice would be made. The final mapped circuit is shown in Figure 5.43. Note that since the original logic and the library elements are hazard-free, the final mapped circuit is also hazard-free.

Boolean matching techniques attempt to find gates in the library which have equivalent Boolean functions. These techniques can perform better than structural methods, but they can also introduce hazards.

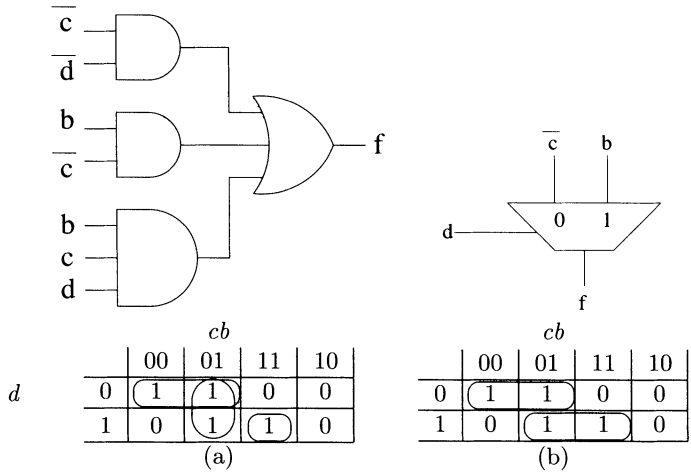


Fig. 5.44 (a) Original function mapped without hazards. (b) MUX implementation with a dynamic hazard.

Example 5.7.3 Consider part of the function from our earlier example shown in Figure 5.44(a). An equivalent function that may be found by a Boolean matching technique is the multiplexor shown in Figure 5.44(b). Karnaugh maps for each implementation are shown below the circuit diagrams. Recall that there is a dynamic $1 \rightarrow 0$ transition $[\overline{a}\,b\,\overline{c}\,\overline{d},\ \overline{a}\,\overline{b}\,\overline{c}\,d]$. In the first implementation, there are no hazards since all products that intersect this transition contain the start cube. On the other hand, the multiplexor implementation includes a product term that illegally intersects the privileged cube. The result is the multiplexor has a dynamic $1 \rightarrow 0$ hazard.

To allow the use of Boolean matching and library gates which are not hazard-free in the structural methods, a different approach is needed. First, we must characterize the hazards found in the library gates. Next, during the technology mapping step, we check that the library gate being chosen has a subset of the hazards in the logic being replaced. If this is the case, this logic gate can be used safely.

Example 5.7.4 The problem with using the multiplexor implementation in Example 5.7.3 is the existence of the dynamic $1 \rightarrow 0$ transition $[\overline{a}\,b\,\overline{c}\,\overline{d},\ \overline{a}\,\overline{b}\,\overline{c}\,d]$. If the original implementation had been $f = \overline{c}\,\overline{d} + b\,d$, this implementation would have a hazard for this dynamic transition also. Since the original implementation is derived to be hazard-free for all transitions of interest, this transition must not occur. Therefore, the multiplexor could be used in this situation.

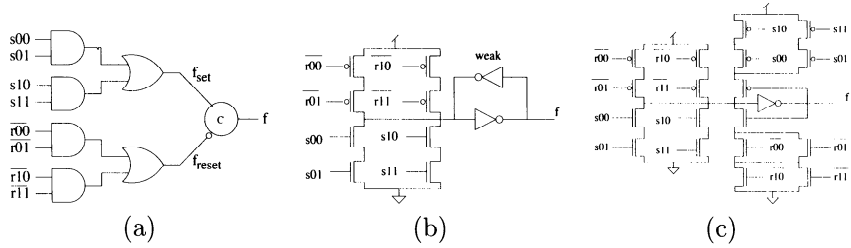


Fig. 5.45 (a) Generalized C-element logic structure with (b) weak-feedback and (c) fully static CMOS implementations.

5.8 GENERALIZED C-ELEMENT IMPLEMENTATION

Another interesting technology is to use *generalized C-elements* (gC) as the basic building blocks. In this technique, the implementation of the set and reset of a signal are decoupled. The basic structure is depicted in Figure 5.45(a), in which the upper sum-of-products represents the logic for the set, f_{set}, the lower sum-of-products represents the logic for the reset, f_{reset}, and the result is merged with a C-element. This can be implemented directly in CMOS as a single compact gate with weak feedback, as shown in Figure 5.45(b), or as a fully static gate, as shown in Figure 5.45(c).

A gC implementation reduces the potential for hazards. For example, static hazards cannot manifest on the output of a gC gate. Care has to be taken, though, during subsequent technology mapping to avoid introducing prolonged short-circuit current during decomposition. Prolonged short circuits are undesirable since they increase power dissipation as well as circuit switching time and should be avoided. Interestingly, avoiding such short circuits corresponds exactly to avoiding dynamic hazards caused by decomposing an N-stack (P-stack) in which its corresponding cube intersects a $1 \rightarrow 0$ ($0 \rightarrow 1$) transition without containing the start subcube (end subcube). By avoiding short circuits (i.e., not allowing decomposition of trigger signals which during a transition both enable and disable a P and N stack), the hazard constraints can be relaxed to no longer require that a product term intersecting a dynamic transition must include the start subcube. The problems with conditionals and dynamic hazards are also not present in gC implementations.

Given the hazard requirements discussed above, the hazard-free cover requirements for the set function, f_{set}, in an extended burst-mode gC become:

1. Each set cube of f_{set} must not include OFF-set minterms.
2. For every dynamic $0 \rightarrow 1$ transition $[c_1,c_2]$ in f_{set}, the end cube, c_2, must be completely covered by some product term.
3. Any product term of f_{set} intersecting the cube c_2 of a dynamic $0 \rightarrow 1$ transition $[c_1,c_2]$ must also contain the end subcube c_2'.

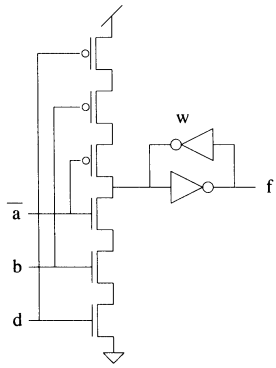


Fig. 5.46 Generalized C-element implementation for Example 5.5.20.

The second requirement describes the product terms that are required for the cover to turn on when it is supposed to. The first and third requirements describe the constraints that the required product terms must satisfy for the cover to be hazard-free. Hazard-freedom requirements for f_{reset} are analogous to f_{set}.

Example 5.8.1 Consider implementing the logic for Example 5.5.20 using a gC. For a gC, we need to consider only the two dynamic transitions. The logic derived for f_{set} and f_{reset} would be as follows:

$$\begin{aligned} f_{\text{set}} &= \overline{a}\,b\,d \\ f_{\text{reset}} &= a\,\overline{b}\,\overline{d} \end{aligned}$$

The resulting circuit implementation is shown in Figure 5.46.

5.9 SEQUENTIAL HAZARDS

The correctness of Huffman circuits designed using the methods described in this chapter rely on the assumption that outputs and state variables stabilize before either new inputs or fed-back state variables arrive at the input to the logic. A violation of this assumption can result in a *sequential hazard.* The presence of a sequential hazard is dependent on the timing of the environment, circuit, and feedback delays.

To illustrate why delay is needed in the feedback, consider the partial flow table shown in Figure 5.47, starting in state 1 with x at 0 and changing x to 1. The result should be that the machine ends up in state 2 with the output staying at 0. Let us assume that part of the logic perceives the state change before it perceives the input change. The result is that the machine may appear to be in state 2 with input x still at 0. In this state, the next state is

	x	
	0	1
1	①,0	2,0
2	3,1	②,0
3	③,1	k,1

Fig. 5.47 Example illustrating the need for feedback delay.

3, so the machine may become excited to go to state 3. Let us assume that after reaching state 3 the logic now detects that x is 1. The result is that the machine now gets excited to go to state k. In other words, if the state change is fed back too quickly, it is possible that the final state is state k when it should be state 2. Another problem is that even if the input is perceived before the machine ends up in state 3, while it thinks it is in state 2 with input 0, it may start to set the output to 1. This means that the output may glitch. Regardless of the state assignment and the values of the outputs, there is no correct circuit realization for this flow table without delay in the feedback. A flow table with a configuration like this is said to contain an *essential hazard.* In general, a flow table has an essential hazard if after three changes of some input variable x, the resulting state is different than the one reached after a single change. In order to eliminate essential hazards, there is a *feedback delay requirement* which can be set conservatively to

$$D_f \geq d_{\max} - d_{\min}$$

where D_f is the feedback delay, $d_{\max}$ is the maximum delay in the combinational logic, and $d_{\min}$ is the minimum delay through the combinational logic.

Sequential hazards can also result if the environment reacts too quickly. Recall that Huffman circuits in this chapter are designed using the fundamental-mode environmental constraint, which says that inputs are not allowed to change until the circuit stabilizes. To satisfy this constraint, a conservative separation time needed between inputs can be expressed as follows:

$$d_i \geq 2d_{\max} + D_f$$

where d_i is the separation time needed between input bursts. This separation needs a $2d_{\max}$ term since the circuit must respond to the input change followed by the subsequent state change.

Finally, XBM machines require a *setup time* and *hold time* for conditional signals. In other words, conditional signals must stabilize a setup time before the compulsory signal transition which samples them, and it must remain stable a hold time after the output and state changes complete. Outside this window of time, the conditional signals are free to change arbitrarily.

5.10 SOURCES

The Huffman school of thought on the design of asynchronous circuits originated with his seminal paper [170] (later republished in [172]). This paper introduced flow tables and describes a complete methodology that includes all the topics addressed in this chapter: state minimization, state assignment, and hazard-free logic synthesis.

There are a substantial number of possible modes of operation for Huffman circuits [382]. Much of the early work is restricted to SIC operation, and the approaches to design are quite similar to those presented in Sections 5.2, 5.3, and 5.4. Quite different SIC synthesis methods have been proposed for transition (rather than level)-sensitive design styles [86, 358]. David proposed a direct synthesis method that uses a single universal cell [98] which avoids the need for state assignment and hazards. The original definition of MIC operation is that all inputs must change within some interval of time, and they are considered to have changed simultaneously [129]. Also, no further input changes are allowed for another period of time. Several methods for MIC design have been developed that either encode the inputs into a one-hot code with a spacer [129] or delay the inputs or outputs [129, 241]. Chuang and Das developed an MIC approach which uses edge-triggered flip-flops and a local clock signal to solve the hazard issues [85]. Other local clock approaches were developed by Rey and Vaucher [322], Unger [384], Hayes [160, 161], and most recently by Nowick [301]. A mixed approach that uses local clocking only when critical races exist was proposed by Yenersoy [411]. Stevens [365] extended the MIC model to allow multiple inputs to change at any time as long as the input changes are grouped together in bursts, and for this reason this mode of operation is often called *burst mode*. In the most general mode of operation called *unrestricted input change* (UIC), any input may change at any time as long as no input changes twice in a given time period. A UIC design method is described in [383] which relies on the use of inertial delay elements. Finally, when the period of time between input changes is set to the time it takes the circuit to stabilize, the circuit is said to be operating in fundamental mode [262].

Huffman circuit synthesis approaches that are quite different from those found in this chapter are found in [96, 97, 313, 395, 407]. These methods do not follow the same breakdown into the steps of state minimization, state assignment, and logic synthesis. For example, Vingron presents a method which through the construction of a coherency tree essentially performs state minimization and state assignment together.

The binate covering problem and its solution were first described by Grasselli and Luccio [155, 156]. In addition to the matrix reduction techniques found in Section 5.1, numerous others have been proposed [140, 155, 156, 179]. Recently, there have been some interesting new approaches proposed for solving binate covering problems [94, 314].

Techniques for state minimization of completely specified state machines were first described by Huffman [170]. The first work on incompletely specified state machines is due to Ginsburg [145, 146] and Paull and Unger [306]. The approach described in Section 5.2 is due to Grasselli and Luccio [155], and the running example is taken from their paper. Numerous other heuristic techniques for state minimization are described in [382]. Recent work in state minimization has been performed by Rho et al. [323] for synchronous design and Fuhrer and Nowick [132] for asynchronous design.

Again, the original work in critical race free state assignment originated with Huffman [170]. The initial work on state assignment allowed for state changes to take place through a series of steps, often changing only one state bit at a time. Such assignments have been called *shared row assignments.* Huffman determined that a universal state assignment (one that works for any arbitrary flow table) existed and requires $2S_0 - 1$ state variables, where $S_0 = \lceil \log_2 r \rceil$, where r is the number of rows in the flow table [172]. Saucier later showed that, in some cases, this bound could be broken [337]. Systematic methods for finding shared row assignments were developed by Maki and Tracey [242] and Saucier [338]. More recent work capable of handling large state machines automatically has been performed by Fisher and Wu [124] and Kang et al. [188].

To improve performance, Liu introduced the unicode single transition time (USTT) state assignment in which state transitions are accomplished in a single step [238]. Liu determined a bound on the number of state variables needed to be $2^{S_0} - 1$. Tracey developed an efficient procedure to find a minimal USTT state assignment [380]. This procedure is the one described in Section 5.3. Friedman et al. determined tighter bounds on the number of state variables needed for a universal USTT state assignment [128]. In particular, he showed that a universal USTT state assignment could be accomplished using no more than either $21S_0 - 15$ or $(S_0^3 + 5S_0)/6$ variables. A small error in one of their other tighter bounds was found by Nanya and Tohma [289]. While previous work concentrated on minimizing the number of state variables, Tan showed that this did not necessarily result in reduced logic [374]. Tan developed a state assignment technique which attempts to minimize literals instead of state variables [374]. Sawin and Maki present a variant of Tan's procedure which is capable of detecting faults [340]. Another approach which can yield simpler realizations was proposed by Mukai and Tohma [276]. Hollaar presents a design methodology based on the *one-hot* row assignment, which provides for a fairly direct circuit realization [168]. An exact method for minimizing output logic during state assignment has been developed by Fuhrer et al. [131]. An improved method has recently been proposed by Rutten and Berkelaar [333]. For large state machines, exact state assignment techniques can fail to produce a result. For this reason, Smith developed heuristic techniques to address large state machines [359]. To speed up the entire design process, Maki et al. developed methods to find the logic equations directly during state assignment [243].

While in USTT assignments a state is assigned a single state code, in multicode STT assignments a state may be assigned multiple codes. Kuhl and Reddy developed a multicode STT assignment technique [214] and Nanya and Tohma devised a universal multicode STT assignment [290]. A more recent multicode approach has been developed by Kantabutra and Andreou [189].

The recursive prime generation procedure described in Section 5.4.2 is derived from Blake's early work on Boolean reasoning [42]. An excellent description of this work is given in [49]. Setting up the prime implicant selection procedure as a covering problem as described in Section 5.4.3 was first done by McCluskey [261]. The original basic theory of combinational hazards described in Section 5.4.4 is due to Huffman [171]. The conditions to ensure hazard freedom under SIC are from McCluskey [263].

Static function and logic hazards under MIC operation were first considered by Eichelberger [122]. He demonstrated that all static logic hazards could be eliminated by using a cover that includes all the prime implicants. He also described a method using a ternary logic to detect logic hazards. Brzozowski et al. developed similar approaches to detecting races and hazards in asynchronous logic [59, 60, 55, 56]. Seger developed a ternary simulation method based on the almost-equal-delay model [342]. Methods for eliminating dynamic function and logic hazards were developed by Unger [382], Bredeson and Hulina [48], Bredeson [47], Beister [30], and Frackowiak [126]. Nowick [301] developed the restrictions on state minimization for BM machines which guarantee a hazard-free logic implementation, and Yun and Dill [420, 421] generalized these restrictions to handle XBM machines, as described in Section 5.5.6. Nowick and Dill developed the first complete algorithm and tool for two-level hazard-free logic minimization [298, 301]. The method described in Section 5.5 follows that used by Nowick and Dill [298, 301]. Recently, Theobald and Nowick developed implicit and heuristic methods to solve the two-level minimization problem [378]. The extensions needed to support extended burst-mode specifications were developed by Yun and Dill [418, 420, 421, 424]. Some recent efficient hazard-free logic minimizers have been developed by Rutten et al. [334, 332] and Jacobson et al. [180, 281].

The concept of hazard-preserving transformations for multilevel logic synthesis described in Section 5.6 is taken from Unger [382]. Kung developed further the idea of hazard-preserving transformations for logic optimization [215]. Lin and Devadas presented a hazard-free multilevel synthesis approach in which the structure of the circuit is based on its representation as a *binary decision diagram* (BDD) [234]. The technology mapping procedure described in Section 5.7 was developed by Siegel [352, 353]. Recently, Chou et al. developed technology mapping techniques that optimized for average-case performance [81]. Modifications of the design procedure to support generalized C-element implementations are due to Yun and Dill [420, 421]. A technology mapping procedure for gC circuits which optimizes for average-case performance appears in [181]. A combinational complex-gate approach was proposed by Kudva et al. [213].

Sequential hazards and the need for feedback delay were presented originally by Unger [382]. Langdon extended the definition of essential hazards to MIC operation [219]. Armstrong et al. showed that if all delay is concentrated at the gate outputs, it is possible to design circuits without feedback delay [11]. Langdon presented a similar result [220]. Brzozowski and Singh showed that in some cases the circuit could be redesigned without using any feedback at all [58]. Magó presented several alternative locations in which to put delay to eliminate sequential hazards and evaluated the amount of delay needed and its affect on the state assignment [241]. The setup and hold-time restriction on conditional signals in XBM machines is due to Yun and Dill [420]. Chakraborty et al. developed a timing analysis technique to check the timing assumptions needed to eliminate sequential hazards [71, 72].

Two excellent references on synthesis algorithms for synchronous design are the books by De Micheli [271] and Hachtel and Somenzi [158]. They were used as a reference for the material in this chapter on binate covering, state minimization, and logic minimization

Problems

5.1 Binate Covering

Solve the following constraint matrix using the *bcp* algorithm.

$$\mathbf{A} = \begin{array}{c} \begin{matrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{11} \end{matrix} \\ \begin{bmatrix} - & - & - & - & - & - & - & - & - & 1 & - \\ 1 & - & - & 1 & - & - & - & - & - & - & - \\ 1 & 1 & - & 1 & 1 & 1 & 1 & - & - & - & - \\ 1 & - & 1 & - & 1 & - & - & - & 1 & - & - \\ - & - & 1 & - & - & 1 & - & - & - & - & - \\ - & 1 & - & - & - & 1 & - & 1 & - & - & 1 \\ - & 1 & - & - & - & - & 1 & 1 & - & 1 & - \\ - & - & 1 & - & - & - & - & - & 1 & - & - \\ 0 & - & - & - & - & - & - & 1 & - & - & - \\ 0 & - & - & - & - & - & - & - & - & 1 & - \\ 0 & - & 1 & - & - & - & - & - & 1 & - & - \\ 1 & 0 & - & - & 1 & - & - & - & - & - & - \\ - & 0 & 1 & - & - & - & - & - & - & - & 1 \\ - & - & 0 & - & - & - & - & - & - & 1 & - \\ - & - & 0 & - & - & - & 1 & - & - & - & - \\ - & - & - & - & 0 & - & - & - & - & 1 & - \\ - & - & 1 & - & 0 & - & - & - & 1 & - & - \\ 1 & - & - & - & 1 & 0 & - & - & - & - & - \\ 1 & - & - & - & 1 & - & 0 & - & - & - & - \\ - & 1 & - & - & - & - & 0 & 1 & - & - & - \\ - & - & 1 & - & - & - & - & 0 & - & - & 1 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \\ 17 \\ 18 \\ 19 \\ 20 \\ 21 \end{matrix}$$

	00	01	11	10
1	–	2,0	–	5,1
2	–	1,0	3,–	–
3	4,1	3,–	1,1	5,–
4	–	–	2,–	1,–
5	6,–	3,–	2,1	–,0
6	5,–	6,1	1,–	2,–

Fig. 5.48 Flow table for Problem 5.3.

5.2 Binate Covering

Implement the BCP algorithm in your favorite programming language. It should read in a constraint matrix and output a list of columns used in the best solution found.

5.3 State Minimization

For the flow table shown in Figure 5.48:

5.3.1. Find compatible pairs using a pair chart.
5.3.2. Compute the maximal compatibles.
5.3.3. Set up and solve BCP using only the maximal compatibles.
5.3.4. Compute the prime compatibles.
5.3.5. Set up and solve BCP using the prime compatibles.
5.3.6. Form the reduced table.
5.3.7. Compare the results from 5.3.3 and 5.3.5.

5.4 State Minimization

For the flow table shown in Figure 5.49:

5.4.1. Find compatible pairs using a pair chart.
5.4.2. Compute the maximal compatibles.
5.4.3. Set up and solve BCP using only the maximal compatibles.
5.4.4. Compute the prime compatibles.
5.4.5. Set up and solve BCP using the prime compatibles.
5.4.6. Form the reduced table.
5.4.7. Compare the results from 5.4.3 and 5.4.5.

5.5 State Minimization

For the flow table shown in Figure 5.50:

5.5.1. Find compatible pairs using a pair chart.
5.5.2. Compute the maximal compatibles.
5.5.3. Compute the prime compatibles.
5.5.4. Set up and solve BCP using the prime compatibles.
5.5.5. Form the reduced table.

	00	01	11	10
1	3,–	2,–	1,1	1,–
2	6,–	–	4,1	1,–
3	–	3,1	–	–,0
4	2,1	–	–	–,0
5	–	3,0	1,–	–
6	4,0	–	5,–	–

Fig. 5.49 Flow table for Problem 5.4.

	00	01	11	10
1	3,0	1,–	–	–
2	6,–	2,0	1,–	–
3	–,1	–	4,0	–
4	1,0	–	–	5,1
5	–	5,–	2,1	1,1
6	–	2,1	6,–	4,1

Fig. 5.50 Flow table for Problem 5.5.

	00	01	11	10
1	1,0	2,0	3,0	1,0
2	1,0	2,0	3,0	2,1
3	3,1	5,0	3,0	4,0
4	1,0	–,–	4,0	4,0
5	3,1	5,0	5,1	4,0

Fig. 5.51 Flow table for Problem 5.6.

5.6 State Assignment

For the flow table shown in Figure 5.51:

5.6.1. Find a state assignment without using the outputs.

5.6.2. Find a state assignment using the outputs as state variables.

5.7 State Assignment

For the flow table shown in Figure 5.52:

5.7.1. Find a state assignment without using the outputs.

5.7.2. Find a state assignment using the outputs as state variables.

5.8 State Assignment

For the flow table shown in Figure 5.53:

5.8.1. Find a state assignment without using the outputs.

5.8.2. Find a state assignment using the outputs as state variables.

	00	01	11	10
1	1,0	2,0	3,0	1,0
2	4,0	2,0	2,0	2,1
3	4,1	2,0	3,0	3,0
4	4,0	5,0	3,0	4,0
5	1,1	5,0	5,1	5,0

Fig. 5.52 Flow table for Problem 5.7.

	00	01	11	10
1	1,0	3,0	7,0	1,0
2	5,0	2,0	6,0	2,1
3	1,1	3,0	7,0	3,0
4	4,0	7,0	6,0	4,0
5	5,1	7,0	6,1	1,0
6	1,0	2,0	6,0	3,0
7	5,1	7,1	7,1	2,1

Fig. 5.53 Flow table for Problem 5.8.

5.9 Two-Level Logic Minimization

Do the following for the Karnaugh map shown in Figure 5.54:

5.9.1. Find all prime implicants using the recursive procedure.

5.9.2. Set up and solve a covering problem to pick the minimal number of prime implicants, ignoring hazards.

5.9.3. Set up and solve a covering problem to pick the minimal number of prime implicants for a hazard-free cover assuming SIC.

5.9.4. Assume that the only transitions possible are

$$\begin{aligned} t_1 &= [\overline{a}\,b\,\overline{c}\,d, \overline{a}\,\overline{b}\,\overline{c}\,\overline{d}] \\ t_2 &= [a\,b\,\overline{c}\,\overline{d}, a\,\overline{b}\,\overline{c}\,d] \\ t_3 &= [\overline{a}\,\overline{b}\,c\,\overline{d}, \overline{a}\,b\,c\,d] \\ t_4 &= [a\,b\,c\,\overline{d}, a\,\overline{b}\,c\,d] \end{aligned}$$

Identify the type of each transition and its transition cube.

5.9.5. Determine all required cubes for the transitions above.

5.9.6. Determine all privileged cubes for the transitions above.

5.9.7. Find the dhf-prime implicants.

5.9.8. Set up and solve a covering problem to pick the minimal number of prime implicants for a hazard-free cover assuming that the only transitions are those given above.

		ab			
		00	01	11	10
cd	00	1	1	1	1
	01	1	1	1	0
	11	0	1	1	0
	10	0	0	1	1

Fig. 5.54 Karnaugh map for Problem 5.9.

		ab			
		00	01	11	10
cd	00	0	0	0	1
	01	0	0	0	1
	11	1	0	0	1
	10	1	1	0	1

Fig. 5.55 Karnaugh map for Problem 5.10.

5.10 Two-Level Logic Minimization

Do the following for the Karnaugh map shown in Figure 5.55:

5.10.1. Find all prime implicants using the recursive procedure.

5.10.2. Set up and solve a covering problem to pick the minimal number of prime implicants, ignoring hazards.

5.10.3. Set up and solve a covering problem to pick the minimal number of prime implicants for a hazard-free cover assuming SIC.

5.10.4. Assume that the only transitions possible are:

$$\begin{aligned} t_1 &= [\overline{a}\,\overline{b}\,\overline{c}\text{-}, \overline{a}\,\overline{b}\,c\text{-}] \\ t_2 &= [\overline{a}\,b\,c\,\overline{d}, \overline{a}\,b\,\overline{c}\,\overline{d}] \\ t_3 &= [a\,b\text{--}, a\,\overline{b}\text{--}] \end{aligned}$$

Identify the type of each transition and its transition cube. Assume that d is a level signal and that c is a falling directed don't care in t_3.

5.10.5. Determine all required cubes for the transitions above.

5.10.6. Determine all privileged cubes for the transitions above.

5.10.7. Find the dhf-prime implicants.

5.10.8. Set up and solve a covering problem to pick the minimal number of prime implicants for a hazard-free cover assuming that the only transitions are those given above.

5.11 Burst-Mode Synthesis

Do the following for the BM machine shown in Figure 5.56:

5.11.1. Translate the BM machine into a flow table.

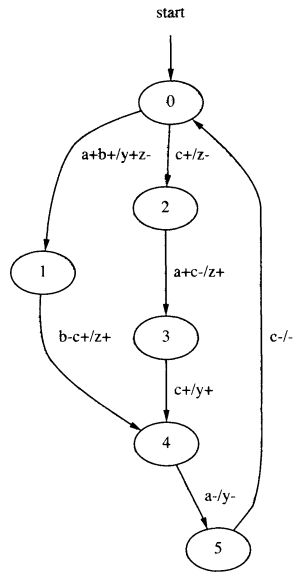


Fig. 5.56 BM machine for Problem 5.11. Note that $abc = 000$ and $yz = 01$ initially.

5.11.2. Perform state minimization on the flow table. Be sure to consider BM operation.

5.11.3. Perform state assignment on the reduced flow table.

5.11.4. Perform two-level logic minimization to find hazard-free logic to implement the output signals.

5.12 Extended Burst-Mode Synthesis

Do the following for the XBM machine shown in Figure 5.57:

5.12.1. Translate the XBM machine into a flow table.

5.12.2. Perform state minimization on the flow table. Be sure to consider XBM operation.

5.12.3. Perform state assignment on the reduced flow table.

5.12.4. Perform two-level logic minimization to find hazard-free logic to implement the output signals.

5.13 Multilevel Logic Synthesis

Apply hazard-preserving transformations to find a minimum literal factored form for your solution of Problem 5.9.

5.14 Multilevel Logic Synthesis

Apply hazard-preserving transformations to find a minimum literal factored form for your solution of Problem 5.10.

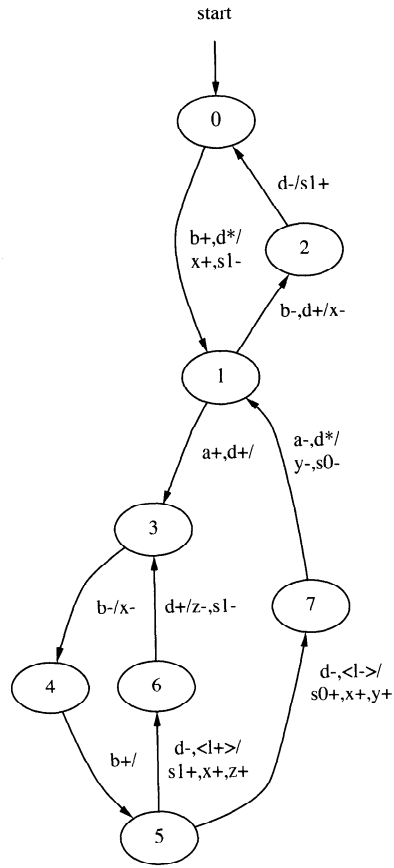


Fig. 5.57 XBM machine for Problems 5.12 and 5.21. Note that $abdl = 000-$ and $s0s1xyz = 01000$ initially.

	00	01	11	10
1	3,1	1,0	2,0	1,1
2	2,0	2,0	2,0	4,0
3	3,1	1,0	3,0	4,0
4	4,0	2,0	3,0	4,0

Fig. 5.58 Flow table for Problem 5.22.

5.15 Multilevel Logic Synthesis

Apply hazard-preserving transformations to find a minimum literal factored form for your solution of Problem 5.11.

5.16 Multilevel Logic Synthesis

Apply hazard-preserving transformations to find a minimum literal factored form for your solution of Problem 5.12.

5.17 Technology Mapping

Map your multilevel logic from Problem 5.13 using the library from Figure 5.41.

5.18 Technology Mapping

Map your multilevel logic from Problem 5.14 using the library from Figure 5.41.

5.19 Technology Mapping

Map your multilevel logic from Problem 5.15 using the library from Figure 5.41.

5.20 Technology Mapping

Map your multilevel logic from Problem 5.16 using the library from Figure 5.41.

5.21 Generalized C-Element Synthesis

Do the following for the XBM machine shown in Figure 5.57, targeting a gC implementation.

5.21.1. Translate the XBM machine into a flow table.

5.21.2. Perform state minimization on the flow table. Be sure to consider XBM operation.

5.21.3. Perform state assignment on the reduced flow table.

5.21.4. Perform two-level logic minimization to find hazard-free logic to implement the output signals using generalized C-elements.

5.22 Sequential Hazards

Find all essential hazards in the flow table shown in Figure 5.58.