*4.4 Programmed Minimization Methods

Obviously, logic minimization can be a very involved process. In real logic-design applications, you are likely to encounter only two kinds of minimization problems: functions of a few variables that you can “eyeball” using the methods of the previous section, and more complex, multiple-output functions that are hopeless without the use of a minimization program.

We know that minimization can be performed visually for functions of a few variables using the Karnaugh-map method. We’ll show in this section that the same operations can be performed for functions of an arbitrarily large number of variables (at least in principle) using a tabular method called the *Quine-McCluskey algorithm*. Like all algorithms, the Quine-McCluskey algorithm can be translated into a computer program. And like the map method, the algorithm has two steps: (a) finding all prime implicants of the function, and (b) selecting a minimal set of prime implicants that covers the function.

Quine-McCluskey algorithm

The Quine-McCluskey algorithm is often described in terms of handwritten tables and manual check-off procedures. However, since no one ever uses these procedures manually, it’s more appropriate for us to discuss the algorithm in terms of data structures and functions in a high-level programming language. The goal of this section is to give you an appreciation for computational complexity involved in a large minimization problem. We consider only fully specified, single-output functions; don’t-cares and multiple-output functions can be handled by fairly straightforward modifications to the single-output algorithms, as discussed in the References.

*4.4.1 Representation of Product Terms

The starting point for the Quine-McCluskey minimization algorithm is the truth table or, equivalently, the minterm list of a function. If the function is specified differently, it must first be converted into this form. For example, an arbitrary n-variable logic expression can be multiplied out (perhaps using DeMorgan’s theorem along the way) to obtain a sum-of-products expression. Once we have a sum-of-products expression, each p-variable product term produces 2^{n-p} minterms in the minterm list.

We showed in Section 4.1.6 that a minterm of an n-variable logic function can be represented by an n-bit integer (the minterm number), where each bit indicates whether the corresponding variable is complemented or uncomplemented. However, a minimization algorithm must also deal with product terms that are not minterms, where some variables do not appear at all. Thus, we must represent three possibilities for each variable in a general product term:

1	Uncomplemented.
0	Complemented.
x	Doesn’t appear.

cube representation

These possibilities are represented by a string of *n* of the above digits in the *cube representation* of a product term. For example, if we are working with product terms of up to eight variables, X7, X6, …, X1, X0, we can write the following product terms and their cube representations:

$$\begin{aligned} X7' \cdot X6 \cdot X5 \cdot X4' \cdot X3 \cdot X2 \cdot X1 \cdot X0' &\equiv 01101110 \\ X3 \cdot X2 \cdot X1 \cdot X0' &\equiv \text{xxxx1110} \\ X7 \cdot X5' \cdot X4 \cdot X3 \cdot X2' \cdot X1 &\equiv \text{1x01101x} \\ X6 &\equiv \text{x1xxxxxx} \end{aligned}$$

Notice that for convenience, we named the variables just like the bit positions in *n*-bit binary integers.

In terms of the *n*-cube and *m*-subcube nomenclature of Section 2.14, the string 1x01101x represents a 2-subcube of an 8-cube, and the string 01101110 represents a 0-subcube of an 8-cube. However, in the minimization literature, the maximum dimension *n* of a cube or subcube is usually implicit, and an *m*-subcube is simply called an "*m*-cube" or a "cube" for short; we'll follow this practice in this section.

To represent a product term in a computer program, we can use a data structure with *n* elements, each of which has three possible values. In C, we might make the following declarations:

```
typedef enum {complemented, uncomplemented, doesntappear} TRIT;
typedef TRIT[16] CUBE;  /* Represents a single product
                           term with up to 16 variables */
```

However, these declarations do not lead to a particularly efficient internal representation of cubes. As we'll see, cubes are easier to manipulate using conventional computer instructions if an *n*-variable product term is represented by two n-bit computer words, as suggested by the following declarations:

```
#define MAX_VARS 16     /* Max # of variables in a product term */
typedef unsigned short WORD;   /* Use 16-bit words */
struct cube {
  WORD t;  /* Bits 1 for uncomplemented variables. */
  WORD f;  /* Bits 1 for complemented variables.   */
};
typedef struct cube CUBE;
CUBE P1, P2, P3;    /* Allocate three cubes for use by program. */
```

Here, a `WORD` is a 16-bit integer, and a 16-variable product term is represented by a record with two `WORD`s, as shown in Figure 4-41(a). The first word in a `CUBE` has a 1 for each variable in the product term that appears uncomplemented (or "true," `t`), and the second has a 1 for each variable that appears complemented (or "false," `f`). If a particular bit position has 0s in both `WORD`s, then the corresponding variable does not appear, while the case of a particular bit position

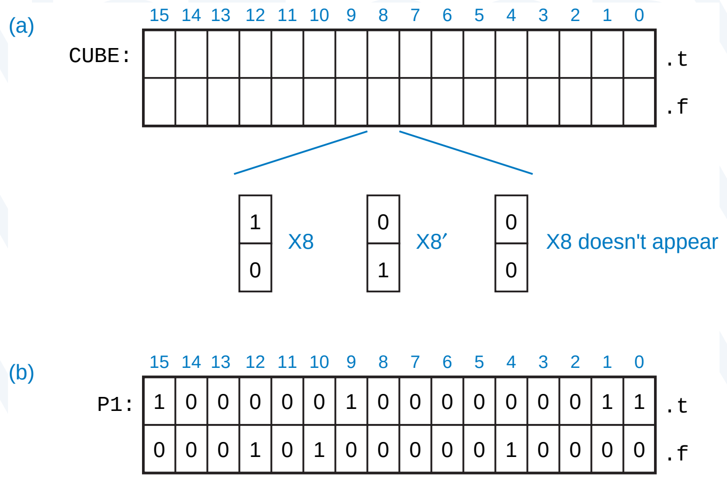

Figure 4-41 Internal representation of 16-variable product terms in a Pascal program: (a) general format; (b) P1 = X15·X12′·X10′·X9·X4′·X1·X0

having 1s in both WORDs is not used. Thus, the program variable P1 in (b) represents the product term P1 = X15 · X12′ · X10′ · X9 · X4′ · X1 · X0. If we wished to represent a logic function F of up to 16 variables, containing up to 100 product terms, we could declare an array of 100 CUBEs:

```
CUBE F[100];      /* Storage for a logic function
                     with up to 100 product terms. */
```

Using the foregoing cube representation, it is possible to write short, efficient C functions that manipulate product terms in useful ways. Table 4-8 shows several such functions. Corresponding to two of the functions, Figure 4-42 depicts how two cubes can be compared and combined if possible

Figure 4-42 Cube manipulations: (a) determining whether two cubes are combinable using theorem T10, term · X + term · X′ = term; (b) combining cubes using theorem T10.

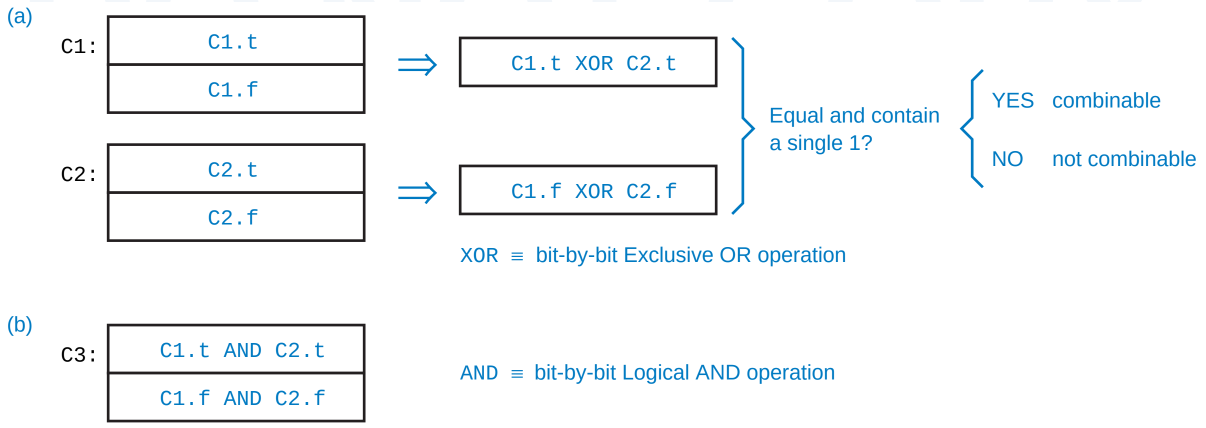

Table 4-8 Cube comparing and combining functions used in minimization program.

```
int EqualCubes(CUBE C1, CUBE C2)      /* Returns true if C1 and C2 are identical.  */
{
  return ( (C1.t == C2.t) && (C1.f == C2.f) );
}

int Oneone(WORD w)              /* Returns true if w has exactly one 1 bit.        */
{                               /* Optimizing the speed of this routine is critical    */
  int ones, b;                  /*   and is left as an exercise for the hacker.        */
  ones = 0;
  for (b=0; b<MAX_VARS; b++) {
    if (w & 1) ones++;
    w = w>>1;
  }
  return((ones==1));
}

int Combinable(CUBE C1, CUBE C2)
{                            /* Returns true if C1 and C2 differ in only one variable, */
  WORD twordt, twordf;       /* which appears true in one and false in the other.      */

  twordt = C1.t ^ C2.t;
  twordf = C1.f ^ C2.f;
  return( (twordt==twordf) && Oneone(twordt) );
}

void Combine(CUBE C1, CUBE C2, CUBE *C3)
                         /* Combines C1 and C2 using theorem T10, and stores the    */
{                        /*   result in C3.  Assumes Combinable(C1,C2) is true.     */
  C3->t = C1.t & C2.t;
  C3->f = C1.f & C2.f;
}
```

using theorem T10, $\text{term} \cdot X + \text{term} \cdot X' = \text{term}$. This theorem says that two product terms can be combined if they differ in only one variable that appears complemented in one term and uncomplemented in the other. Combining two m-cubes yields an $(m+1)$-cube. Using cube representation, we can apply the combining theorem to a few examples:

$$\begin{aligned}
010 + 000 &= 0x0 \\
00111001 + 00111000 &= 0011100x \\
101xx0x0 + 101xx1x0 &= 101xxxx0 \\
x111xx00110x000x + x111xx00010x000x &= x111xx00x10x000x
\end{aligned}$$

*4.4.2 Finding Prime Implicants by Combining Product Terms

The first step in the Quine-McCluskey algorithm is to determine all of the prime implicants of the logic function. With a Karnaugh map, we do this visually by identifying "largest possible rectangular sets of 1s." In the algorithm, this is done by systematic, repeated application of theorem T10 to combine minterms, then 1-cubes, 2-cubes, and so on, creating the largest possible cubes (smallest possible product terms) that cover only 1s of the function.

The C program in Table 4-9 applies the algorithm to functions with up to 16 variables. It uses 2-dimensional arrays, `cubes[m][j]` and `covered[m][j]`, to keep track of `MAX_VARS` *m*-cubes. The 0-cubes (minterms) are supplied by the user. Starting with the 0-cubes, the program examines every pair of cubes at each level and combines them when possible into cubes at the next level. Cubes that are combined into a next-level cube are marked as "covered"; cubes that are not covered are prime implicants.

Even though the program in Table 4-9 is short, an experienced programmer could become very pessimistic just looking at its structure. The inner `for` loop is nested four levels deep, and the number of times it might be executed is on the order of $\texttt{MAX\_VARS} \cdot \texttt{MAX\_CUBES}^3$. That's right, that's an exponent, not a footnote! We picked the value `maxCubes` = 1000 somewhat arbitrarily (in fact, too optimistically for many functions), but if you believe this number, then the inner loop can be executed *billions and billions* of times.

The maximum number of minterms of an n-variable function is 2^n, of course, and so by all rights the program in Table 4-9 should declare `maxCubes` to be 2^{16}, at least to handle the maximum possible number of 0-cubes. Such a declaration would not be overly pessimistic. If an n-variable function has a product term equal to a single variable, then 2^{n-1} minterms are in fact needed to cover that product term.

For larger cubes, the situation is actually worse. The number of possible m-subcubes of an n-cube is $\binom{n}{m} \times 2^{n-m}$, where the binomial coefficient $\binom{n}{m}$ is the number of ways to choose m variables to be x's, and 2^{n-m} is the number of ways to assign 0s and 1s to the remaining variables. For 16-variable functions, the worst case occurs with $m = 5$; there are 8,945,664 possible 5-subcubes of a 16-cube. The total number of distinct m-subcubes of an n-cube, over all values of m, is 3^n. So a general minimization program might require a *lot* more memory than we've allocated in Table 4-9.

There are a few things that we can do to optimize the storage space and execution time required in Table 4-9 (see Exercises 4.72–4.75), but they are piddling compared to the overwhelming combinatorial complexity of the problem. Thus, even with today's fast computers and huge memories, direct application of the Quine-McCluskey algorithm for generating prime implicants is generally limited to functions with only a few variables (fewer than 15–20).

Table 4-9 A C program that finds prime implicants using the Quine-McCluskey algorithm.

```
#define TRUE    1
#define FALSE   0
#define MAX_CUBES 50

void main()
{
  CUBE cubes[MAX_VARS+1][MAX_CUBES];
  int covered[MAX_VARS+1][MAX_CUBES];
  int numCubes[MAX_VARS+1];
  int m;          /* Value of m in an m-cube, i.e., ‘‘level m.’’ */
  int j, k, p;    /* Indices into the cubes or covered array.    */
  CUBE tempCube;
  int found;

  /* Initialize number of m-cubes at each level m. */
  for (m=0; m<MAX_VARS+1; m++) numCubes[m] = 0;

  /* Read a list of minterms (0-cubes) supplied by the user, storing them    */
  /* in the cubes[0,j] subarray, setting covered[0,j] to false for each      */
  /* minterm, and setting numCubes[0] to the total number of minterms read.  */
ReadMinterms;

  for (m=0; m<MAX_VARS; m++)            /* Do for all levels except the last */
    for (j=0; j<numCubes[m]; j++)       /* Do for all cubes at this level    */
      for (k=j+1; k<numCubes[m]; k++)   /* Do for other cubes at this level  */
        if (Combinable(cubes[m][j], cubes[m][k])) {
          /* Mark the cubes as covered. */
          covered[m][j] = TRUE;  covered[m][k] = TRUE;
          /* Combine into an (m+1)-cube, store in tempCube. */
          Combine(cubes[m][j], cubes[m][k], &tempCube);
          found = FALSE;  /* See if we've generated this one before. */
          for (p=0; p<numCubes[m+1]; p++)
            if (EqualCubes(cubes[m+1][p],tempCube)) found = TRUE;
          if (!found) {  /* Add the new cube to the next level. */
            numCubes[m+1] = numCubes[m+1] + 1;
            cubes[m+1][numCubes[m+1]-1] = tempCube;
            covered[m+1][numCubes[m+1]-1] = FALSE;
          }
        }
  for (m=0; m<MAX_VARS; m++)       /* Do for all levels                 */
    for (j=0; j<numCubes[m]; j++)  /* Do for all cubes at this level */
      /* Print uncovered cubes -- these are the prime implicants. */
      if (!covered[m][j]) PrintCube(cubes[m][j]);
}
```