# Minimization of Boolean Functions

We now continue our study of Boolean circuits to consider the possibility that there might be more than one implementation of a specific Boolean function. We are particularly focused on the idea of simplifying a Boolean function in the sense of reducing the number of basic logic gates (NOT, AND, and OR gates) required to implement the function.

There are a number of methods for simplifying Boolean expressions: algebraic, Karnaugh maps, and Quine-McCluskey being the more popular. We have already discussed algebraic simplification in an unstructured way. We now study the tabular methods Karnaugh maps (K-Maps) and Quine-McCluskey (**QM**). Most students prefer K-Maps as a simplification method. We shall study QM both within the context of K-Maps and separately as a method for simplifying expressions that are a bit too complex for K-maps.

Logical Adjacency

**Logical adjacency** is the basis for all Boolean simplification methods. The facility of the K-Map approach is that it transforms logical adjacency into physical adjacency so that simplifications can be done by inspection. To understand the idea of logical adjacency, we review two simplifications based on the fundamental properties of Boolean algebra.
For any Boolean variables X and Y:

$$X \bullet Y + X \bullet \overline{Y} = X \bullet (Y + \overline{Y}) = X \bullet 1 = X$$

$$\begin{aligned}(X + Y) \bullet (X + \overline{Y}) &= X \bullet X + X \bullet \overline{Y} + Y \bullet X + Y \bullet \overline{Y} \\ &= X \bullet X + X \bullet \overline{Y} + X \bullet Y + 0 \\ &= X + X \bullet (\overline{Y} + Y) = X + X = X\end{aligned}$$

Two Boolean terms are said to be **logically adjacent** when they contain the same variables and differ in the form of exactly one variable; i.e., one variable will appear negated in one term and in true form in the other term and all other variables have the same appearance in both terms. Consider the following lists of terms, the first in 1 variable and the others in 2.

| | | | |
|---|---|---|---|
| $X$ | $\overline{X}$ | | |
| $X \bullet Y$ | $X \bullet \overline{Y}$ | $\overline{X} \bullet \overline{Y}$ | $\overline{X} \bullet Y$ |
| $(X + Y)$ | $(X + \overline{Y})$ | $(\overline{X} + \overline{Y})$ | $(\overline{X} + Y)$ |

The terms in the first list are easily seen to be logically adjacent. The first term has a single variable in the true form and the next has the same variable in the negated form.


Copyright © 2005 by Edward Bosworth

We now examine the second list, which is a list of product terms each with two variables. Note that each of the terms differs from the term following it in exactly one variable and thus is logically adjacent to it: X•Y is logically adjacent to $X \bullet \overline{Y}$, $X \bullet \overline{Y}$ is logically adjacent to $\overline{X} \bullet \overline{Y}$, $\overline{X} \bullet \overline{Y}$ is logically adjacent to $\overline{X} \bullet Y$, and $\overline{X} \bullet Y$ is logically adjacent to X•Y. Note that logical adjacency is a commutative relation thus $X \bullet \overline{Y}$ is logically adjacent to both X•Y and $\overline{X} \bullet \overline{Y}$. Using the SOP notation, we represent this list as 11, 10, 00, 01.

The third list also displays logical adjacencies in its sequence: (X + Y) is logically adjacent to $(X + \overline{Y})$, which is logically adjacent to $(\overline{X} + \overline{Y})$, which is logically adjacent to $(\overline{X} + Y)$. Using POS notation, we represent this list as 00, 01, 11, 10.

Consider the list of product terms when written in the more usual sequence

$\overline{X} \bullet \overline{Y}$ $\overline{X} \bullet Y$ $X \bullet \overline{Y}$ X•Y, or 00, 01, 10, 11 in the SOP notation.

In viewing this list, we see that the first term is logically adjacent to the second term, but that the second term is not logically adjacent to the third term: X'•Y and X•Y' differ in two variables. This is seen also in viewing the numeric list 00, 01, 10, and 11. Note that each of the digits in 01 and 10 is different, so that 01 and 10 can't represent logically adjacent terms.

Karnaugh Maps for 2, 3, and 4 variables

All books seem to define K-Maps for 2, 3, 4, 5, and 6 variables. It is this author's opinion that K-Maps for 5 and 6 variables are a waste of time, so he discuss them only to persuade the student to avoid them. The reason for this opinion is that K-Maps are designed to be a simple tool for simplifying Boolean expressions; K-Maps with 5 or more variables are hopelessly complex. One can also envision a K-Map for a single variable, although it is hard to envision any productive use for such a K-Map.

This figure shows the basic K-Maps for 2, 3, and 4 variables. Note that there are two equivalent forms of the 3-variable K-Map; the student should pick one style and use it. This instructor prefers to use the second form (two rows and four columns) in these notes because the format fits the page better, but uses either form when presenting on the board.

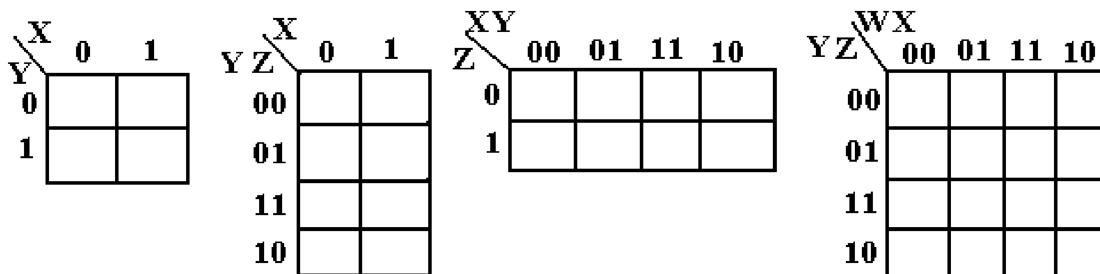



**Figure: The Basic K-Map Forms**


Copyright © 2005 by Edward Bosworth

We now examine three equivalent forms of the K-Map of an unspecified function. We show these K-Maps only to comment on the form of K-Maps and not to discuss simplification.

| Z \ XY | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |

| Z \ XY | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | | 1 | 1 | 1 |

| Z \ XY | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | | 0 |
| 1 | 0 | | | |

Each of these K-Maps represents the same function, shown at right in the truth-table form. One way to view a K-Map is as a truth-table with the main exception of the ordering 00, 01, 11, 10 seen on the top. For those interested, this ordering is called a **Gray code**.

| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

The full K-Map is shown at left, with each square filled in either with a 0 or a 1. K-Maps are never written in this fashion – either one omits the 0's or one omits the 1's. The form omitting the 0's is used when simplifying SOP expressions; to simplify POS one omits the 1's.

**A Different Truth-Table**

Just for fun, we now present the truth table for the above function in a form in which the row order of the table is dictated by the Karnaugh map.

| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 |

This truth-table contains the same information as the traditional truth-table, but is organized in a form in which the variable entries in each row are logically adjacent to those of both the pervious and following row. Note that the row prior to X = 0, Y = 0, and Z = 0 is the last row X = 1, Y = 0, and Z = 0, and that the row following the last row is the first row.

The attentive student would note that there are two other equivalent presentations of this Gray code truth table, one with the second row being 0 1 0 and the other with 1 0 0.

One final note – K-Maps are used to simplify Boolean expressions written in canonical form.


Copyright © 2005 by Edward Bosworth

## K-Maps for Sum of Products (SOP)

Consider the Canonical SOP expression $F(X,Y,Z) = \overline{X}\bullet Y\bullet Z + X\bullet \overline{Y}\bullet Z + X\bullet Y\bullet \overline{Z} + X\bullet Y\bullet Z$. The first step in using K-Maps to simplify this expression is to use the SOP numbering to represent these as 0's and 1's. The SOP copy rule is that the negated variable is written as a 0, the plain as a 1. Thus, this function is represented as 011, 101, 110, and 111.

| Z \ XY | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | 1 | |
| 1 | | 1 | 1 | 1 |

Place a 1 in each of the squares with the "coordinates" given in the list above. In the K-Map at left, the entry in the top row corresponds to 110 and the entries in the bottom row correspond to 011, 111, and 101 respectively. Remember that we do not write the 0's when we are simplifying expressions in SOP form.

The next step is to notice the physical adjacencies. We group adjacent 1's into "rectangular" groupings of 2, 4, or 8 boxes. Here there are no groupings of 4 boxes in the form or a rectangle, so we group by two's. There are three such groupings, labeled A, B, and C.

The grouping labeled A represents the product term XY. The B group represents the product term YZ and the C group represents the product term XZ. Examine the B grouping: it has 011 and 111. In this we have Y and Z staying the same and X having both values; thus the product term YZ. This function is $X\bullet Y + X\bullet Z + Y\bullet Z$.

We now apply Quine-McCluskey to this problem as a way to introduce the method.
The first step in the QM procedure is to list the terms in the binary form of the Σ–list; for this problem the list is (011, 101, 110, 111).

We then group the terms by the number of 1's; in this case the listing is immediate.

| | |
|---|---|
| 0 | None |
| 1 | None |
| 2 | 011, 101, 110 |
| 3 | 111 |

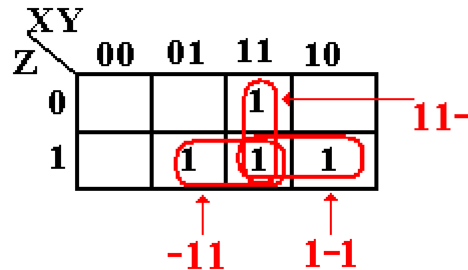
The QM rule for combination is that a term in row N may be combined with either a term in row (N – 1) or row (N + 1) or both if the terms to be combined differ only in one place. The result of combining a 0 and a 1 is denoted by the symbol "–".

| | |
|---|---|
| 011 and 111 | combine to form –11, and |
| 101 and 111 | combine to form 1–1, and |
| 110 and 111 | combine to form 11–. |

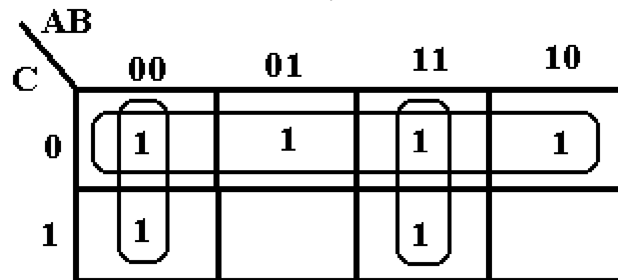

Copyright © 2005 by Edward Bosworth

The reduced terms –11, 1–1, and 11– translate to Y•Z, X•Z, and X•Y, so the result of the QM simplification is F(X, Y, Z) = Y•Z + X•Z + X•Y, which can be rewritten in a more standard form as F(X, Y, Z) = X•Y + X•Z + Y•Z, the result from the K-Maps.

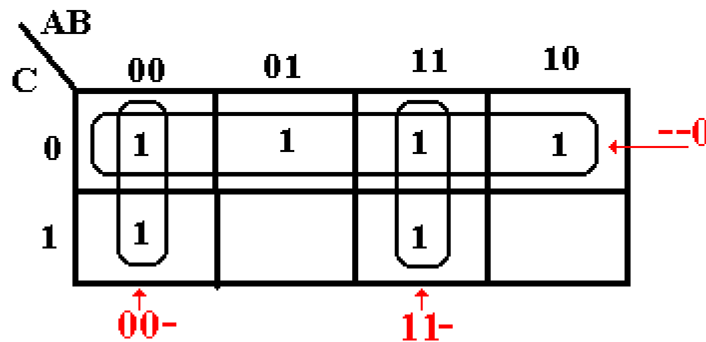
The link between K-Maps and QM is best illustrated by the use of QM labeling on a K-map. Consider the K-map for the problem above with each of the combined terms bearing the QM labels, so that the product terms 110 and 111 combine to form 11– , etc.




The next example is to simplify F(A, B, C) = Π(3, 5). We shall consider use of K-Maps to simplify POS expressions, but for now the solution is to convert the expression to the SOP form F(A, B, C) = Σ(0, 1, 2, 4, 6, 7). We could write each of the six product terms, but the easiest solution is to write the numbers as binary: 000, 001, 010, 100, 110, and 111.




The top row of the K-Map corresponds to the entries 000, 010, 100, and 110, arranged in the order 000, 010, 110, and 100 to preserver logical adjacency. The bottom row corresponds to the entries 001 and 111. The top row simplifies to $\overline{C}$. The first column simplifies to $\overline{A} \bullet \overline{B}$ and the third column to A•B. Thus we have F(A, B, C) = $\overline{A} \bullet \overline{B}$ + A•B + $\overline{C}$.




Here is the K-Map with the QM labeling. We now apply the QM procedure directly.


Copyright © 2005 by Edward Bosworth

Before applying the Quine-McCluskey method, we restate the problem. We are asked to simplify the function F(A, B, C) = $\Sigma$(0, 1, 2, 4, 6, 7). We begin by writing these numbers in binary as 000, 001, 010, 100, 110, and 111. We should next order the list by the number of 1's in each term, but note that it is already so ordered. We move to the next step.

| | |
|---|---|
| 0 No 1's | 000 |
| 1 One 1 | 001, 010, 100 |
| 2 Two 1's | 110 |
| 3 Three 1's | 111 |

The first step in applying the method is to combine terms in adjacent rows if the terms differ in only one place. Here is the first phase.

Rows 0 and 1
000 and 001 combine to form 00–
000 and 010 combine to form 0–0
000 and 100 combine to form –00

Rows 1 and 2
100 and 110 combine to form 1–0

Rows 2 and 3
110 and 111 combine to form 11–

Note that rows 3 and 0 are not adjacent. Unlike the K-Map procedure, the rows in QM are ordered so that there is no "wrap around".

Here is the situation after the first round of simplifications.

| | |
|---|---|
| 000 | |
| | 00–, 0–0, –00 |
| 001, 010, 100 | |
| | 1–0 |
| 110 | |
| | 11– |
| 111 | |

We now apply the second phase of the simplification. This and future rounds follow 2 rules.

1) Any two terms that differ in a single position, with one term having a 0 where the other term has a 1, may be combined as above.
2) If there are two terms (call them T1 and T2) that match according to the following criteria, then the term T2 may be removed.
   a) When T1 has a digit (either 0 or 1), then T2 has the same digit in that position.
   b) When T1 has a –, T2 has either a digit or a –.


Copyright © 2005 by Edward Bosworth

Applying these rules, here are the results after the second round of simplifications, having combined the terms `0-0` and `1-0` to form `--0`.

```
000
                00-, 0-0, -00
001, 010, 100                     --0
                1-0
110
                11-
111
```

Note that the terms `--0` and `-00`can now be combined to drop the latter term. The results of the complete simplification procedure are the three terms `00-`, `11-`, and `--0`. These three terms correspond to $\overline{A} \bullet \overline{B}$ , A•B, and $\overline{C}$ , respectively. Thus we have the result F(A, B, C) = $\overline{A} \bullet \overline{B}$ + A•B + $\overline{C}$ , the same as that obtained by K-Maps.

Earlier we noted that K-Maps (and by extension, the QM procedures) are used only on canonical forms, either SOP or POS. We now show how one might apply K-Maps to expressions in normal form. The trick is first to make the canonical form.

We next consider this example.

F(W, X, Y, Z) = W'•X'•Y'•Z' + W'•X'•Y'•Z + W•X'•Y'.

The trouble with K-Maps is that the technique is designed to be used only with expressions in canonical form. In order to use the K-Map method we need to convert the term W•X'•Y' to its equivalent W•X'•Y'•Z' + W•X'•Y'•Z, thus obtaining a four-term canonical SOP.

Now that we see where we need to go with the tool, we draw the four-variable K-Map. F(W, X, Y, Z) = W'•X'•Y'•Z' + W'•X'•Y'•Z + W•X'•Y'•Z' + W•X'•Y'•Z. Using the SOP encoding method, these are terms 0000, 0001, 1000, and 1001. The K-Map is

| YZ \ WX | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | | | 1 |
| 01 | 1 | | | 1 |
| 11 | | | | |
| 10 | | | | |

The first row in the K-Map represents the entries 0000 and 1000. The second row in the K-Map represents the entries 0001 and 1001. The trick here is to see that the last column is adjacent to the first column The four cells in the K-Map are thus adjacent and can be grouped into a square. We simplify by noting the values that are constant in the square: X = 0 and Y = 0. Thus, the expression simplifies to X'•Y', as required.


Copyright © 2005 by Edward Bosworth

To verify the K-Map, we apply simple algebraic simplification to F.

$$
\begin{aligned}
F(W, X, Y, Z) &= W'\bullet X'\bullet Y'\bullet Z' + W'\bullet X'\bullet Y'\bullet Z + W\bullet X'\bullet Y' \\
&= W'\bullet X'\bullet Y'\bullet (Z' + Z) + W\bullet X'\bullet Y' \\
&= W'\bullet X'\bullet Y' + W\bullet X'\bullet Y' \\
&= (W' + W)\bullet X'\bullet Y' = X'\bullet Y'
\end{aligned}
$$

We close the discussion of SOP K-Maps with the example at right, which shows that the four corners of the square are adjacent and can be grouped into a 2 by 2 square. This K-Map represents the terms 0000, 0010, 1000, 1010 or $W'\bullet X'\bullet Y'\bullet Z' + W'\bullet X'\bullet Y\bullet Z' + W\bullet X'\bullet Y'\bullet Z' + W\bullet X'\bullet Y\bullet Z'$. The values in the square that are constant are X = 0 and Z = 0, thus the expression simplifies to $X'\bullet Z'$.

| YZ \ WX | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | | | 1 |
| 01 | | | | |
| 11 | | | | |
| 10 | 1 | | | 1 |

To convince the student that these four corners are indeed adjacent, we redraw the K-Map using another acceptable Gray code for the rows and columns: 01, 11, 10, 00.

| YZ \ WX | 01 | 11 | 10 | 00 |
|---|---|---|---|---|
| 01 | | | | |
| 11 | | | | |
| 10 | | | 1 | 1 |
| 00 | | | 1 | 1 |

**Figure: The Redrawn K-Map**

The logical adjacencies in this expression are clarified by this equally valid K-Map. In combining four cells (arranged in a 2-by-2 square), we eliminate two variables. We note in examining the square, that we have the following:

1) Both W = 0 and W = 1 — W is eliminated
2) Both Y = 0 and Y = 1 — Y is eliminated
3) X = 0 only and Z = 0 only, — so the term is $X'\bullet Z'$.


Copyright © 2005 by Edward Bosworth

## K-Maps for POS

K-Maps for Product of Sums simplification are constructed similarly to those for Sum of Products simplification, except that the POS copy rule must be enforced: 1 for a negated variable and 0 for a non-negated (plain) variable.

As our first example we consider F(A, B, C) = $\Pi$(3, 5) = (A + B' + C')•(A' + B + C'). Recall that the term (A + B' + C') corresponds to 011 and that (A' + B + C') to 101.

| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | 0 | | 0 |

This is really somewhat of a trick question used only to illustrate placing of the terms for POS. Place a 0 at each location, rather than the 1 placed for SOP. Note that the two 0's placed are not adjacent, so we cannot simplify the expression.

For the next example consider F2 = (A + B + C)•(A + B + C')•(A + B' + C)•(A' + B + C). Using the POS copy rule, we translate this to 000, 001, 010, and 100.

Before we attempt to simplify F2, we note that it is a very good candidate for simplification. Compare the first term 000 to each of the following three terms. The term 000 differs from the term 001 in exactly one position. The same applies for comparison to the other two terms. Any two terms that differ in exactly one position can be combined in a simplification.

| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | | 0 |
| 1 | 0 | | | |

We begin the K-Map for POS simplification by placing a 0 in each of the four positions 000, 001, 010, 100. Noting that 000 is adjacent to 001, just below it, we combine to get 00– or (A + B). The term 000 is adjacent to 010 to its right to get 0–0 or (A + C). The term 000 is adjacent to 100 to its "left" to get –00 or (B + C). As a result, we get the simplified form. F2 = (A + B)•(A + C)•(B + C)

Just for fun, we simplify this expression algebraically, using the derived Boolean identity X•X•X = X for any Boolean expression X.

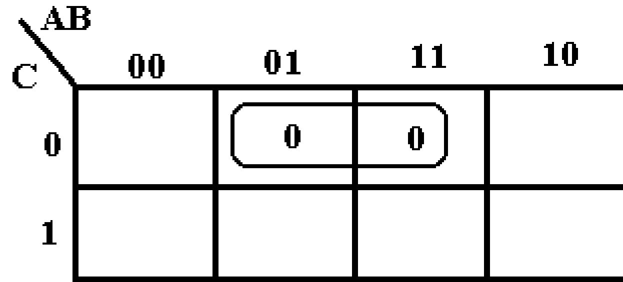
F2 = (A + B + C)•(A + B + C')•(A + B' + C)•(A' + B + C)
= (A + B + C)•(A + B + C')•(A + B + C)•(A + B' + C)•(A + B + C)•(A' + B + C)
= (A + B)•(A + C)•(B + C)

It is encouraging that we get the same answer.


Copyright © 2005 by Edward Bosworth

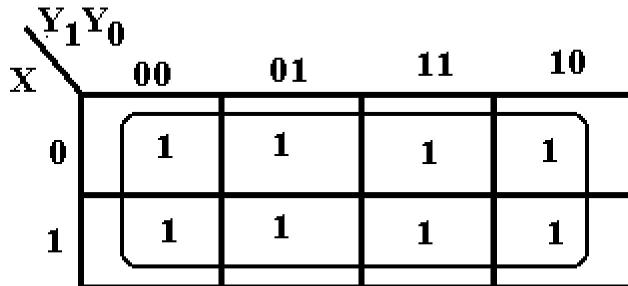
We now consider simplification of a POS function specified by a truth table.

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |



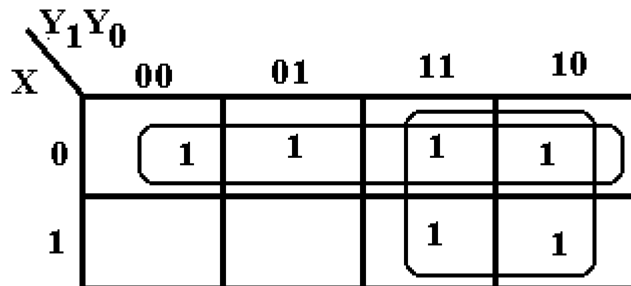

We plot two 0's for the POS representation of the function – one at 010 and one at 110. The two are combined to get –10, which translates to (B' + C).

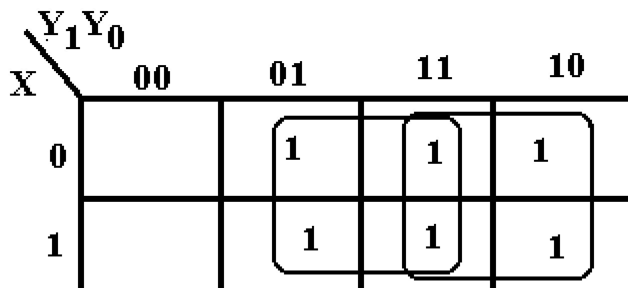
<u>More Examples of K-Maps</u>




The sample at left, based on an earlier design shows a particularly simple problem. We find that all the entries in the K-Map are covered with a single grouping, thus removing all three variables. Since the entire K-Map is covered, the simplification is F = 1.

The K-Map at right shows an example with overlap of two groupings of 1's. All 1's in the map must be covered and some should be covered twice. The top row corresponds to X'. We then form the 2-by-2 grouping at the right to obtain the term $Y_1$. Thus F = X' + $Y_1$.




There is another simplification that should be considered. This corresponds to two 2-by-2 groupings. The 2-by-2 grouping at the right still corresponds to $Y_1$. The new 2-by-2 grouping in the middle gives rise to $Y_0$, so we get another simplification F = $Y_0$ + $Y_1$.


Copyright © 2005 by Edward Bosworth

Just One More K-Map: Overlapping Circles

We close the discussion of K-Maps with a technique that applies to both SOP and POS simplifications. We shall apply it to SOP simplification.

Consider the following K-Map.

| YZ \ WX | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | 1 | 1 | 1 | |
| 11 | 1 | 1 | 1 | |
| 10 | | | | |

The six ones can be grouped in a number of ways. Consider the following.

| YZ \ WX | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | 1 | 1 | 1 | |
| 11 | 1 | 1 | 1 | |
| 10 | | | | |

This grouping of four and two covers the six one's in the K-Map.

The four ones in the square form the term W'•Z. The two ones in the rectangle form the term W•X•Z.

The K-Map simplifies to W'•Z + W•X•Z.


Copyright © 2005 by Edward Bosworth

Another way to consider the simplification of the K-Map is to group the rectangle and the square as in the figure at right.

The rectangle corresponds to the term W'•X'•Z.

The square corresponds to the term X•Z.

This simplification yields W'•X'•Z + X•Z.

| YZ \ WX | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | 1 | 1 | 1 | |
| 11 | 1 | 1 | 1 | |
| 10 | | | | |

It is important to note that the groupings can overlap if this yields a simpler reduction.

| YZ \ WX | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | 1 | 1 | 1 | |
| 11 | 1 | 1 | 1 | |
| 10 | | | | |

Here we show two overlapping squares.

The square at left corresponds to the term W'•Z.

The square at right corresponds to the term X•Z.

This simplification yields W'•Z + X•Z, which is simpler than either of the other two forms validly produced by the K-Map method.
Try 1: W'•Z + W•X•Z
Try 2: W'•X'•Z + X•Z

Try 3: W'•Z + X•Z. This seems better.


Copyright © 2005 by Edward Bosworth

Simplification with Don't-Care Conditions

We now consider the use of K-Maps to simplify expressions that include the "d" or Don't-Care condition often generated when considering digital designs using flip-flops. We give a number of examples related to our previous designs of sequential circuits.

| | X = 0 | X = 1 |
|---|---|---|
| $Y_1Y_0$ | $J_1$ | $J_1$ |
| 0 0 | 0 | 1 |
| 0 1 | 1 | 0 |
| 1 0 | d | d |
| 1 1 | d | d |

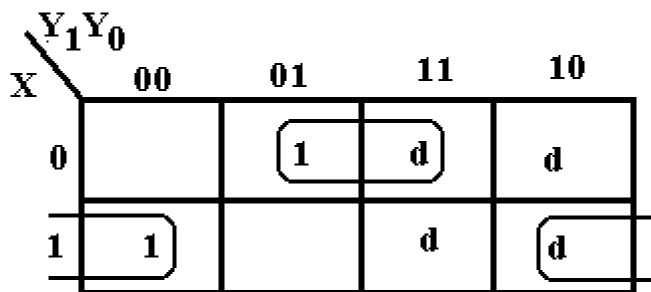
The general rule in considering a simplification with the Don't-Care conditions is to count the number of 0's and number of 1's in the table and to use SOP simplification when the number of 1's is greater and POS simplification when the number of 0's is greater. Again we admit that most students prefer the SOP simplification. With a two-two split, we try SOP simplification.

First we should explain the above table in some detail. The first thing to say about it is that we shall see similar tables again when we study flip-flops. For the moment, we call it a "folded over" truth table, equivalent to the full truth table at right. The function to be represented is $J_1$. Lines 0, 1, 4, and 5 of the truth table seem to be standard, but what of the other rows in which $J_1$ has a value of "d". This indicates that in these rows it is equally acceptable to have $J_1 = 0$ or $J_1 = 1$. We have four "Don't-Cares" or "d" in this table; each can be a 0 or 1 independently of the others – in other words we are not setting the value of *d* as a variable.

| $Y_1$ | $Y_0$ | X | $J_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | d |
| 0 | 1 | 1 | d |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | d |
| 1 | 1 | 1 | d |

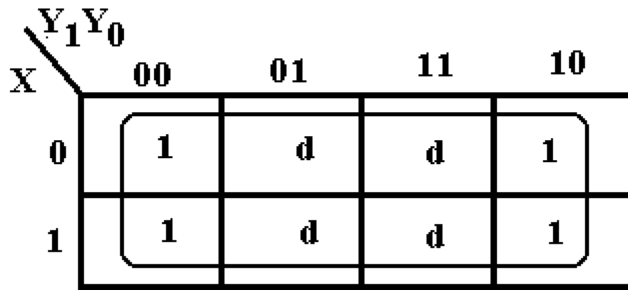
Design with flip-flops is the subject of another course.

When attempting a K-Map for SOP simplification, we drop the 0's and plot the 1's and d's. We then attempt to group the 1's into 2-by-1, 2-by-2 groupings, etc. We use the d's as are convenient and have no requirement to cover any or all of them. Note that 3-by-1 groupings are not valid and that the 2-by-2 grouping of d's does not add anything to the simplification, but only adds an extra useless term.



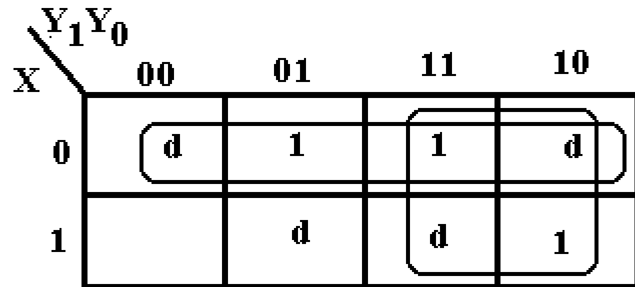

The terms in the top row, labeled 001 and 011 for $X'Y_1'Y_0$ and $X'Y_1Y_0$, simplify to 0–1 for $X'Y_0$, and the terms in the bottom row, labeled 100 and 110 for $XY_1'Y_0'$ and $XY_1Y_0'$, simplify to 1–0 for $XY_0'$, so the simplified expression is $X' \bullet Y_0 + X \bullet Y_0' = X \oplus Y_0$.

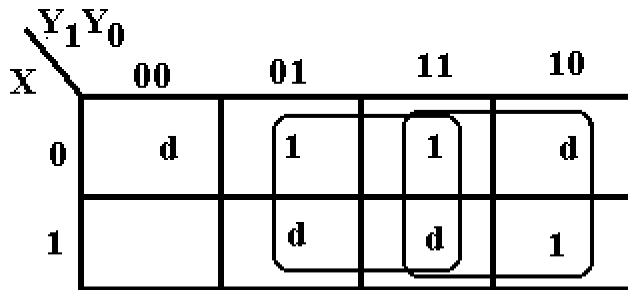

Copyright © 2005 by Edward Bosworth




The sample at left, based on an earlier design shows a particularly simple problem. We find that using the d's to combine with the 1's to produce a 4-by-2 grouping of 1's. Since the entire K-Map is covered, the simplification is F = 1.

The K-Map at right corresponds to an input table with one 0 and three 1's. This immediately suggests a SOP approach to the K-Map; we plot the 1's and d's and drop the 0. The top row corresponds to X'. We then form the 2-by-2 grouping at the right to obtain the term $Y_1$. Thus F = X' + $Y_1$.



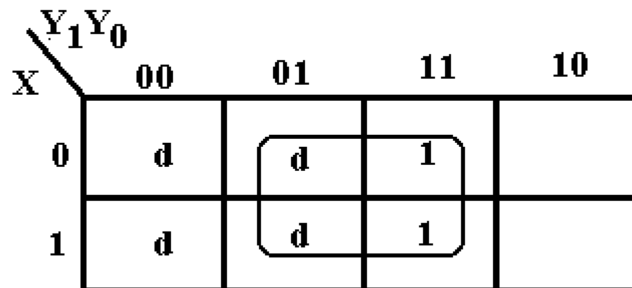




There is another simplification that should be considered. This corresponds to two 2-by-2 groupings. The 2-by-2 grouping at the right still corresponds to $Y_1$. The new 2-by-2 grouping in the middle gives rise to $Y_0$, so we get another simplification F = $Y_0$ + $Y_1$.

| | X = 0 | X = 1 |
|---|---|---|
| $Y_1Y_0$ | $K_0$ | $K_0$ |
| 0 0 | d | d |
| 0 1 | d | d |
| 1 0 | 0 | 0 |
| 1 1 | 1 | 1 |

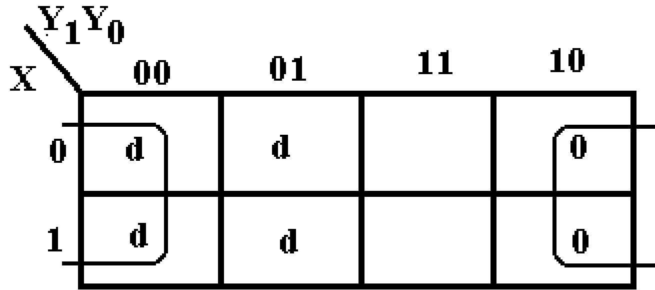
As a final example, we consider the input table, which contains two 0's and two 1's. According to the theory, this could be simplified equally well either as a SOP or POS expression. To gain confidence, we do both simplifications, with the SOP first.

Considered as a SOP problem, we plot the 1's and d's, then form the largest possible group that covers all of the 1's. Note that 3-by-2 is not a valid grouping, so we go with the 2-by-2 grouping. The square corresponds to $Y_0$. The top row simplifies to 0–1 and the bottom to 1–1, thus we have – – 1 or $Y_0$.





Copyright © 2005 by Edward Bosworth




The POS simplification is shown at left. The top row simplifies to 0–0 and the bottom row simplifies to 1–0 so the K-Map simplifies to – – 0. Using the POS copy rule, this translates to $Y_0$, as before. It's a good thing that the two methods agree.

A Diversion: Application of Simplification Techniques to Programming

This next section attempts to apply some of the Boolean simplification techniques to issues sometimes seen in software development, especially C++ programming.

Consider the Boolean expressions in C++ that relate to equality. For variable x, we can have expressions such as (x == 0), (x != 0), and !(x == 0). The last two are logically identical, and all are distinct from the assignment statement (x = 0), which evaluates to False.

In our diversion, we consider three variables: x, y, and z. The only assumption made here is that each of the three is of a type that can validly be compared to 0; assuming that all are integer variables is one valid way to read these examples. Each of the expressions (x == 0), (y == 0), and (z == 0) evaluates to either T (True) or F (False).

Consider a function that is to be called conditionally based on the values of three variables: x, y, and z. We write the Boolean expression as follows

```
if (      (  (x != 0)   && (y != 0)   && (z != 0) )
       ||  (  (x != 0)   && (y != 0)   && (z == 0) )
       ||  (  (x != 0)   && (y == 0)   && (z == 0) )
       ||  (  (x == 0)   && (y != 0)   && (z != 0) )
       ||  (  (x == 0)   && (y != 0)   && (z == 0) )
       ||  (  (x == 0)   && (y == 0)   && (z == 0) ) ) y = fzero( )
```

We can apply the truth-table approach to analysis of the conditions under which the function fzero is invoked. The following table illustrates when the function is to be called.

| (x == 0) | (y == 0) | (z == 0) | Call fzero |
|---|---|---|---|
| F | F | F | Yes |
| F | F | T | Yes |
| F | T | F | No |
| F | T | T | Yes |
| T | F | F | Yes |
| T | F | T | Yes |
| T | T | F | No |
| T | T | T | Yes |

If this looks a bit like a truth table, it is because it is equivalent to a truth table and can be converted to one. Consider the following definitions of Boolean variables A, B, and C.

A = (x == 0)
B = (y == 0)
C = (z == 0)


Copyright © 2005 by Edward Bosworth

Consider the expression `A = (x == 0)` in the C++ programming language. It may seem a bit strange, but is perfectly legitimate. The expression `(x == 0)` is a Boolean expression – it evaluates to True or False. The variable `A` is a Boolean variable, it also takes on one of the Boolean values. In order to translate the table above into a truth table that we recognize, we replace the expressions `(x == 0)`, `(y == 0)`, and `(z == 0)` by their equivalents – the Boolean variables A, B, and C. We are beginning to construct a Truth Table.

In order to apply the truth table approach to this problem, we must define a Boolean function. For our purpose, we define F(A, B, C) as follows

F(A, B, C) = 1 if fzero is called
= 0 if fzero is not called

Returning to our convention of 0 for False and 1 for True, we have the truth table.

| A | B | C | F(A, B, C) |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

This is a truth table that we have considered and simplified in an earlier section of the work. Using terminology we have already discussed, we see that this function F(A, B, C) can be expressed as either a SOP with six product terms or a POS with two sum terms. Reading this as a POS, we get $F(A, B, C) = (A + \overline{B} + C) \bullet (\overline{A} + \overline{B} + C)$, which simplifies to $F(A, B, C) = (\overline{B} + C)$.

We now convert back to the original notation. Recalling that B = (y == 0) and C = (z == 0), we note that B' = (y != 0) and the condition for calling the function fzero becomes ( (y != 0) || (z == 0) ). So the equivalent (and much simpler) expression is

```
if ( (y != 0) || (z == 0) ) y = fzero( )
```

Consider now the Boolean expression ( (x == 0) || ( (x != 0) && (y == 0) ) ). In an attempt to simplify this expression we define two Boolean variables

A = (x == 0)
B = (y == 0)

Recall that (x != 0) = !(x == 0) = $\overline{A}$. In our terminology, the expression is

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

There are a number of ways to simplify this expression. The first, and least obvious, is to invoke the theorem of absorption, which states that the formula equals A + B.


Copyright © 2005 by Edward Bosworth

To illustrate other options, we expand the above to canonical SOP and then examine it by means of both a truth table and a K-map. To expand the expression into canonical SOP, we need to have the first term contain a literal for the variable B.

$$A + \overline{A} \bullet B = A \bullet \left(\overline{B} + B\right) + \overline{A} \bullet B = \overline{A} \bullet B + A \bullet \overline{B} + A \bullet B$$

The truth table for this expression is

| A | B | F(A, B) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

Representing this as a POS formula, we immediately get F(A, B) = (A + B), which translates to the C++ expression ( (x == 0) || (y == 0) ).

As a final example, consider the following Boolean expression in C++

```
( (x == 0) || (y == 0) || (z == 0) ) && ( (x == 0) || (y == 0) || (z !=0)) &&
( (x == 0) || (y != 0) || (z == 0) ) && ( ( x != 0) || (y == 0) || (z == 0) )
```

Define A = (x == 0)
B = (y == 0)
C = (z == 0)

With these definitions our expression becomes

$$F(A, B, C) = (A + B + C) \bullet \left(A + B + \overline{C}\right) \bullet \left(A + \overline{B} + C\right) \bullet \left(\overline{A} + B + C\right)$$

This is known to simplify to

F(A, B, C) = (A + B)•(A + C)•(B + C)

So our Boolean expression in C++ simplifies to

```
( (x == 0) || (y == 0) ) && ( (x == 0) || (z == 0) ) && (y == 0) || (z == 0) )
```

Inspection of the above shows that we want at least two of (x == 0), (y == 0), and (z == 0) to be true. Compare this with the original derivation of the function

F(A, B, C) = (A + B)•(A + C)•(B + C)

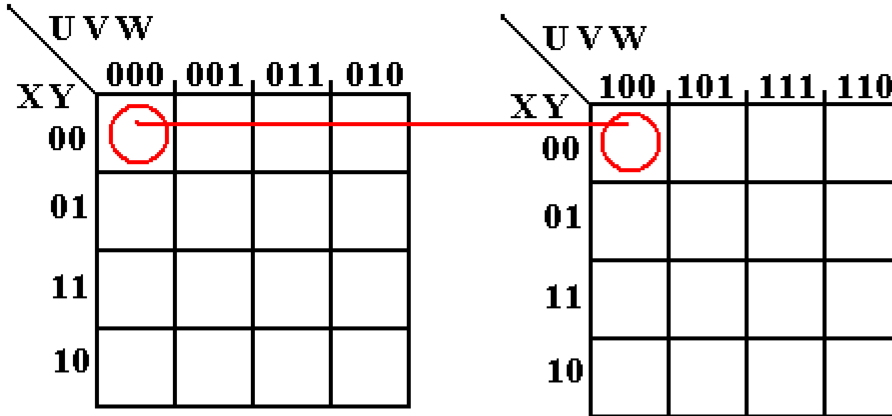
used for the carry-out of a Full-Adder, which is 1 if two or three of the inputs are 1.


Copyright © 2005 by Edward Bosworth

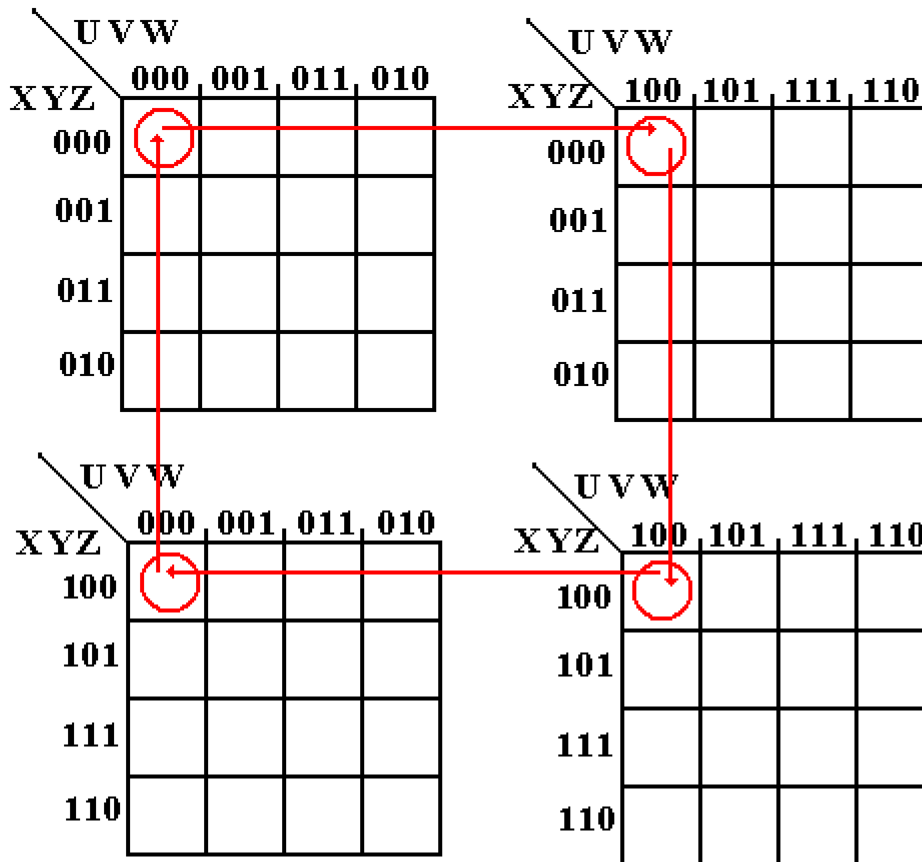
**Big K-Maps**

Before focusing on the QM procedure, let's look at K-maps for 5 and 6 variables.

**Five-Variable K-Maps.** **F(U, V, W, X, Y)**




**Six Variable K-Maps** **F(U, V, W, X, Y, Z)**




Note here one of the adjacencies: 000000 ↔ 100000 ↔ 100100 ↔ 000100, as well as the others. For this author, five and six variable K-Maps are much too busy.


Copyright © 2005 by Edward Bosworth

**A Formal Presentation of Quine-McCluskey**

Having decided that we do not want to use K-Maps for more than four variables, let's present the Quine-McCluskey procedure along with a five-variable simplification as an example. We shall build on our earlier informal discussions and build our presentation formally.

We present the steps of the QM procedure and illustrate with the following function.

F(V, W, X, Y, Z) = $\Sigma$(2, 3, 7, 10, 11, 15, 18, 19, 23, 24, 25, 26, 27, 28, 29, 30, 31)

**Step 1 – Determine the Number of Variables and the Range of Binary Numbers to Use**

This expression contains five Boolean variables, so we need $2^5 = 32$ binary numbers for the QM procedure. These are five-bit unsigned binary numbers in the range 0 to 31 inclusive. If not given the number of variables, derive it from the largest number in the term list. To do this, solve $2^{B-1} \leq \text{MaxTerm} < 2^B$, here $2^{B-1} \leq 31 < 2^B$, to get B = 5. For those who think that the inequality signs are misplaced, this is equivalent to $2^{B-1} < \text{MaxNumberOfTerms} \leq 2^B$.

**Step 2 – Write the Terms as Binary Numbers**

The list above can be written as five-bit unsigned binary numbers as follows.

00010, 00011, 00111, 01010, 01011, 01111, 10010, 10011, 10111,
11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111

**Step 3 – Arrange the List in Increasing Order of Number of 1's**

**Let N denote the number of binary bits, equivalently the number of variables.**

| | | |
|---|---|---|
| Group 0: | Zero 1's | None |
| Group 1: | One 1 | 00010 |
| Group 2: | Two 1's | 00011, 01010, 10010, 11000 |
| Group 3: | Three 1's | 00111, 01011, 10011, 11001, 11010, 11100 |
| Group 4: | Four 1's | 01111, 10111, 11011, 11101, 11110 |
| Group 5: | Five 1's | 11111 |

In this example, we have N = 5 as there are five variables.

**Step 4 – For I = 0 to (N – 1), Compare Each Term in Group I to Those in Group (I + 1)**
**Form a combined term for those that are adjacent.**

**I = 0**

There are no terms with zero 1's.

**I = 1**

Group 1: 00010
Group 2: 00011, 01010, 10010, 11000

00010 and 00011 form 0001–
00010 and 01010 form 0–010
00010 and 10010 form –0010


Copyright © 2005 by Edward Bosworth

**I = 2**

Group 2: 00011, 01010, 10010, 11000
Group 3: 00111, 01011, 10011, 11001, 11010, 11100

00011 and 00111 form 00–11
00011 and 01011 form 0–011
00011 and 10011 form –0011

01010 and 01011 form 0101–
01010 and 11010 form –1010

10010 and 10011 form 1001–
10010 and 11010 form 1–010

11000 and 11001 form 1100–
11000 and 11010 form 110–0
11000 and 11100 form 11–00

**I = 3**

Group 3: 00111, 01011, 10011, 11001, 11010, 11100
Group 4: 01111, 10111, 11011, 11101, 11110

00111 and 01111 form 0–111
00111 and 10111 form –0111

01011 and 01111 form 01–11
01011 and 11011 form –1011

10011 and 10111 form 10–11

11001 and 11011 form 110–1
11001 and 11101 form 11–01

11010 and 11011 form 1101–
11010 and 11110 form 11–10

11100 and 11101 form 1110–
11100 and 11110 form 111–0


Copyright © 2005 by Edward Bosworth

**I = 4**
Group 4: 01111, 10111, 11011, 11101, 11110
Group 5: 11111

01111 and 11111 form –1111
10111 and 11111 form 1–111
11011 and 11111 form 11–11
11101 and 11111 form 111–1
11110 and 11111 form 1111–

At this point, we have run into a problem due to the complexity of the problem. The method calls for us to keep combining terms until we cannot combine any more and then list those that could not be combined, calling them "**prime implicants**" or **PI's.** We need a better way to track the terms as they are combined. The answer for these notes is a number of tables.

The first table will just list what terms have been combined and what have not. Rather than just marking the terms, we list each combination and the result of that combination.

| Term | | | | |
|---|---|---|---|---|
| 00010 | 00011 => 0001– | 01010 => 0–010 | 10010 => –0010 | |
| | | | | |
| 00011 | 00010 => 0001– | 00111 => 00–11 | 01011 => 0–011 | 10011 => –0011 |
| 01010 | 00010 => 0–010 | 01011 => 0101– | 11010 => –1010 | |
| 10010 | 00010 => –0010 | 10011 => 1001– | 11010 => 1–010 | |
| 11000 | 11001 => 1100– | 11010 => 110–0 | 11100 => 11–00 | |
| | | | | |
| 00111 | 00011 => 00–11 | 01111 => 0–111 | 10111 => –0111 | |
| 01011 | 00011 => 0–011 | 01010 => 0101– | 01111 => 01–11 | 11011 => –1011 |
| 10011 | 00011 => –0011 | 10010 => 1001– | 10111 => 10–11 | |
| 11001 | 11000 => 1100– | 11011 => 110–1 | 11101 => 11–01 | |
| 11010 | 01010 => –1010 | 10010 => 1–010 | 11000 => 110–0 | |
| | 11011 => 1101– | 11110 => 11–10 | | |
| 11100 | 11000 => 11–00 | 11101 => 1110– | 11110 => 111–0 | |
| | | | | |
| 01111 | 00111 => 0–111 | 01011 => 01–11 | 11111 => –1111 | |
| 10111 | 00111 => –0111 | 10011 =>10–11 | 11111 => 1–111 | |
| 11011 | 01011 => –1011 | 11001 => 110–1 | 11010 => 1101– | 11111 => 11–11 |
| 11101 | 11001 => 11–01 | 11100 => 1110– | 11111 => 111–1 | |
| 11110 | 11010 => 11–10 | 11100 => 111–0 | 11111 => 1111– | |
| | | | | |
| 11111 | 01111 => –1111 | 10111 => 1–111 | 11011 => 11–11 | |
| | 11101 => 111–1 | 11110 => 1111– | | |


Copyright © 2005 by Edward Bosworth

We note two things as a result of the first round of combination.

1) Each of the original terms combines with at least two other terms.
   As a result of this, none of the original terms is a prime implicants.
2) We have twenty-nine intermediate terms to consider next. True, the table has fifty-eight entries, but each entry was written twice to insure correct tracking of all prime implicants.

We now sort the entries according to the following sort order: "0" < "1" < "–". From this point on, we shall list the entries in a different font to make them easier to read.

```
0001-
00-11
0101-
01-11
0-010
0-011
0-111
1001-
10-11
1-010
1100-
1101-
110-0
110-1
1110-
1111-
111-0
111-1
11-00
11-01
11-10
11-11
1-111
-0010
-0011
-0111
-1010
-1011
-1111
```

At this point, the author of these notes must make two comments.

1) That is a big list.
2) Although somewhat readable, it is sorted in the wrong order. Rather than erase his mistake and pretend it did not happen, this author will now show how to correct it.


Copyright © 2005 by Edward Bosworth

An observant reader will note that, even as written, the list can easily be simplified. The step necessary to make the list truly useful is to sort the list primarily on the position of the "–". Here is the new list. We repeat the above approach to simplification, noting that the "–" must be in the same location if two entries are to be merged. To facilitate handling, we group the clauses, with the positions being numbered left to right as 1 to 5.

**"–" in position 5**

```
        0001-
                0101- => 0-01-
                1001- => -001-
        0101-
                0001- => 0-01-
                1101- => -101-
        1001-
                0001- => -001-
                1101- => 1-01-
        1100-
                1101- => 110--
                1110- => 11-0-
        1101-
                0101- => -101-
                1001- => 1-01-
                1100- => 110--
                1111- => 11-1-
        1110-
                1100- => 11-0-
                1111- => 111--
        1111-
                1101- => 11-1-
                1110- => 111--
```

**"–" in position 4**

```
        110-0
                110-1 => 110--
                111-0 => 11--0
        110-1
                110-0 => 110--
                111-1 => 11--1
        111-0
                110-0 => 11-–0
                111-1 => 111--
        111-1
                110-1 => 11–-1
                111-0 => 111--
```


Copyright © 2005 by Edward Bosworth

**"–" in position 3**

```
00-11
        01-11 => 0—-11
        10-11 => -0-11
01-11
        00-11 => 0—-11
        11-11 => -1-11
10-11
        00-11 => -0-11
        11-11 => 1--11
11-00
        11-01 => 11-0-
        11-10 => 11--0
11-01
        11-00 => 11-0-
        11-11 => 11--1
11-10
        11-00 => 11—-0
        11-11 => 11-1-
11-11
        01-11 => -1-11
        10-11 => 1--11
        11-01 => 11—-1
        11-10 => 11-1-
```

**"–" in position 2**

```
0-010
        0-011 => 0-01-
        1-010 => --010
0-011
        0-010 => 0-01-
        0-111 => 0--11
0-111
        0-011 => 0—-11
        1-111 => --111
1-010
        0-010 => --010
1-111
        0-111 => --111
```


Copyright © 2005 by Edward Bosworth

**“–” in position 1**

```
    -0010
            -0011 => -001-
            -1010 => --010
    -0011
            -0010 => -001-
            -0111 => -0-11
            -1011 => --011
    -0111
            -0011 => -0-11
            -1111 => --111
    -1010
            -0010 => --010
            -1011 => -101-
    -1011
            -0011 => --011
            -1010 => -101-
            -1111 => -1-11
    -1111
            -0111 => --111
            -1011 => -1-11
```

At this point, we have 31 new clauses, each with two “–“. We hope that a few of these are duplicates. Again, we note that every clause that is a result of the first round has again been combined, so we do not have any prime implicants with only one “–“. The following table shows the reduced terms grouped in columns labeled by their source. Note the duplicates.

| **Pos 5** | **Pos 4** | **Pos 3** | **Pos2** | **Pos 1** |
|---|---|---|---|---|
| 110-- | 110-- | | | |
| 111-- | 111-- | | | |
| 11-0- | | 11-0- | | |
| 11-1- | | 11-1- | | |
| 0-01- | | | 0-01- | |
| 1-01- | | | | |
| -001- | | | | -001- |
| -101- | | | | -101- |
| | 11--0 | 11--0 | | |
| | 11--1 | 11--1 | | |
| | | 0--11 | 0--11 | |
| | | 1--11 | | |
| | | -0-11 | | -0-11 |
| | | -1-11 | | -1-11 |
| | | | --010 | --010 |
| | | | | --011 |
| | | | --111 | --111 |


Copyright © 2005 by Edward Bosworth

Here is the list of 17 terms with two variables removed, sorted as needed for further work.

```
110--
111--
11-0-
11-1-
0-01-
1-01-
-001-
-101-
11--0
11--1
0—11
1--11
-0-11
-1-11
--010
--011
--111
```

Here we can speed things up.

`110--` and `111--` combine to form `11---`.
`11-0-` and `11-1-` combine to form `11---`, a duplicate.
`0-01-` and `1-01-` combine to form `--01-`.
`-001-` and `-101-` combine to form `--01-`, a duplicate.
`11--0` and `11--1` combine to form `11---`, a duplicate.
`0—11` and `1--11` combine to form `---11`.
`-0-11` and `-1-11` combine to form `---11`, a duplicate.
`--010` and `--011` combine to form `--01-`, a duplicate.
`--011` and `--111` combine to form `---11`, a duplicate.

At this point, we note with some relief that the number of terms has been cut quite a bit. We also notice that all of the terms with two "–" have been involved in some combination, so that none of them is a prime implicant.

Here are the three surviving non-duplicated terms.

```
11---
--01-
---11
```

Using SOP notation, we read this simplified function as

$$F(V, W, X, Y, Z) = V \bullet W + \overline{X} \bullet Y + Y \bullet Z$$


Copyright © 2005 by Edward Bosworth