Chapter 3

Manipulating Logic Specifications

3.1 Equations From Truth Tables

If truth tables and logic expressions are to serve as design aids, they must work together, hand in hand. As we shall see both representations lead directly to hardware implementations, each in different technologies, suggesting that we need ways to move from one representation to the other.

Consider first the question of deriving a logic equation from a truth-table specification. Take the example

Row No.	A	B	Q
(0)	0	0	0
(1)	0	1	0
(2)	1	0	1
(3)	1	1	0

In words, Row (2) tells us W is true if A is true and B is false, and the remaining rows tell us that W is false in all other cases. In other words, W is true only if A is true and B is false. Formally, $W = A \cdot \overline{B}$. Here is another example:

	A	B	R
(0)	0	0	1
(1)	0	1	0
(2)	1	0	1
(3)	1	1	1

This table says there are three cases when R may be true, according to rows (0), (2) and (3), when $A = B = 0$, or when $A = 1$ and $B = 0$, or when $A = B = 1$. In symbolic terms,

$$R = \overline{A} \cdot \overline{B} + A \cdot \overline{B} + A \cdot B \tag{3.1}$$

We might use a bit more insight in our intuitive translation, recognizing, for example, that when A is true, so is R regardless of what B is. This would lead us to formulate,

$$R = \overline{A} \cdot \overline{B} + A \tag{3.2}$$

Looking at it still another way, we could say that R is *false* only if $A = 0$ and $B = 1$, that is,

$$\overline{R} = \overline{A} \cdot B \tag{3.3}$$

So our intuition can yield two, and in fact several, equations describing the same simple truth table. Are they all correct? Does this make the table ambiguous?

The answer to the first question, is "yes," as can be verified by algebraic derivation (See Exercise **??**). The answer to the second question is "no," although this says nothing about either our intuition, which may easily fail us in more complex cases.

Just as standard truth table is desirable, we need a standard expression for a given function, and a systematic way to derive it.

Sum-of-Products Form. Equations 3.1 and 3.2 express the function R in *sum-of-products*, or *SoP*, form. This is the most common formulation of logic function. As we have already seen, it readily reflects our intuition about what the table says.

The name "sum-of-products" comes from the analogy to arithmetic operators. This family of expressions is characterized as being a sum of terms, each consisting of simple products of variables or their negations. No further nesting of operations is allowed; no duplication of variables in a product term is allowed; and no duplication of product terms is allowed. Here is the "language" of SOPs in for form of a grammar.

$$\begin{aligned}
\langle SOP \rangle &::= 0 \;\; [] \;\; \langle Trm \rangle + \cdots + \langle Trm \rangle \\
\langle Trm \rangle &::= \langle Lit \rangle \cdot \cdots \cdot \langle Lit \rangle \\
\langle Lit \rangle &::= \langle Var \rangle \;\; [] \;\; \overline{\langle Var \rangle} \\
\langle Var \rangle &::= \textit{a fixed set of variable symbols}
\end{aligned}$$

The ellipses above are intended to mean "one or more occurrences." As we have already noted this is not truely a context-free grammar: there may be no repetitions.

In logic, SOPs are called *disjunctive forms*; the product terms are called *clauses*; and the variables, whether negated or not, are called *literals*. In that context, we would ordinarily use the logical symbols, T, F, $\wedge$, $\vee$ and $\neg$, rather than than the boolean symbols.

Derivation of SOPs from Truth Tables. To derive an SOP from a standard truth table, write the sum containing product terms representing each row of the table for which the function value is 1. Consider again the truth table for function X,

	A	B	C	X		
(0)	0	0	0	0	$\mathbf{m}_0 = \overline{A}\,\overline{B}\,\overline{C}$	
(1)	0	0	1	1	$\mathbf{m}_1 = \overline{A}\,\overline{B}\,C$	$\leftarrow$
(2)	0	1	0	1	$\mathbf{m}_2 = \overline{A}\,B\,\overline{C}$	$\leftarrow$
(3)	0	1	1	0	$\mathbf{m}_3 = \overline{A}\,B\,C$	
(4)	1	0	0	1	$\mathbf{m}_4 = A\,\overline{B}\,\overline{C}$	$\leftarrow$
(5)	1	0	1	0	$\mathbf{m}_5 = A\,\overline{B}\,C$	
(6)	1	1	0	0	$\mathbf{m}_6 = A\,B\,\overline{C}$	
(7)	1	1	1	0	$\mathbf{m}_7 = A\,B\,C$	

An SOP for X will contain three terms representing rows (1), (2) and (4). These terms can be written immediately from the entries for A, B and C in the row. If the entry is 1, write the variable in its positive form; if the entry is 0, use the variable's negative form:

$$X = \underbrace{\overline{A}\,\overline{B}C}_{\mathbf{m}_1} + \underbrace{\overline{A}B\overline{C}}_{\mathbf{m}_2} + \underbrace{A\overline{B}\overline{C}}_{\mathbf{m}_4}$$

A product term that contains an occurance of every variable is called a *minterm*. The SOP we have just derived is called the *normal form sum-of-products* because it consists entirely of minterms. Equations 3.1 and 3.2 are also valid SOPs, but they are not in normal form. Equation 3.3 is also in SOP form, it is a sum consisting of just one term. The normal-form SOP for $\overline{Y}$ is $R = \mathbf{m}_1 = \overline{A}B$. (Remember that the index of a standard truth table depends on both the number and order of the input variables, in this case $[A, B]$.)

We shall also allow the vacuous SOP consisting on zero terms, and use 0 to represent it. By convention, 0 is the value of the boolean sum of zero terms, just as is the case with arithmetic sums.

In the context of logic, the canonical SOP expression is called the *disjunctive normal form*, or *DNF*.

Product-of-Sums Form. Get used to the fact that for every concept in logic or boolean algebra there will be a dual concept. Another characteristic family of expressions, dual to SOP, is *product-of-sums* (*POS*) form, in which the roles of $+$ and $\cdot$ are exchanged.

$$\begin{aligned}
\langle POS \rangle &::= \mathsf{T} \;\; [] \;\; \langle Trm \rangle \cdot \cdots \cdot \langle Trm \rangle \\
\langle Trm \rangle &::= \langle Lit \rangle + \cdots + \langle Lit \rangle \\
\langle Lit \rangle &::= \langle Var \rangle \;\; [] \;\; \overline{\langle Var \rangle} \\
\langle Var \rangle &::= \textit{a fixed set of variable symbols}
\end{aligned}$$

In logic, these are called *conjunctive forms*. A sum term, or *clause*, is called a *maxterm* if it contains an occurance of every variable. An POS expression is canonical if it contains only maxterms, and in logic is calle a *conjuctive normal form*, or *CNF*.

A minterm has an intuitive counterpart in a truth table, a single row. And a canonical SOP, in essence, lists the rows that are true for the function specified by the expression. Let us consider what maxterms and canonical POS represent in relation to truth tables, starting with an example.

	J	K	L	T
(0)	0	0	0	0
(1)	0	0	1	1
(2)	0	1	0	1
(3)	0	1	1	1
(4)	1	0	0	0
(5)	1	0	1	0
(6)	1	1	0	0
(7)	1	1	1	0

Before looking at the function T, think about an arbitary maxterm, say $\mathbf{M}_3 \equiv \overline{J} + K + L$. What row or set rows does it describe? In words it says, "Either $J = 0$ (i.e. rows 1–3) or $K = 1$ (i.e. rows 2, 3, 6 and 7) or $L = 1$ (i.e. the odd-numbered rows)." So $\mathbf{M}_3$ describes the set of all the rows *except Row 4*. Notice that 6 is the binary complement of 3, $\overline{011} = 100$.

Indeed, by identity *DeMorgan's Law* (DM),

$$\overline{\mathbf{m}_3} \equiv \overline{\overline{J} \cdot K \cdot L} \stackrel{\text{DM}}{=} J + \overline{KL} \equiv \mathbf{M}_6 = \mathbf{M}_{\overline{3}}$$

and in general,

$$\overline{\mathbf{m}_i} = \mathbf{M}_{\overline{i}}$$

for any index i. Thus, a maxterm $\mathbf{M}_i$ describes the set-complement of its complementary row, a truth-table in which every entry is 1 except for row $\overline{i}$. The *and* of two maxterms, $\mathbf{M}_i$ and $\mathbf{M}_j$ describes the intersection of their two complementary row-sets, a truth table whose only 0 entries are rows $\overline{i}$ and $\overline{j}$. Conversely, the 0s of a truth table, tell us what the maxterms of a canonical truth table are.

Derivation of POSs from Truth Tables. A boolean function is 1 everywhere it isn't 0, so the way to reduce a truth table to a *canonical POS*, or *conjunctive normal form* in logic, is to write down a maxterm for each row for which the function's value is zero. We must remember, however, to complement the index to obtain the literals of the sum term. For example the canonical POS for function T, whose truth table is shown above is

$$T = \underbrace{(J + K + L)}_{\mathbf{M}_{\overline{0}}} \cdot \underbrace{(\overline{J} + K + L)}_{\mathbf{M}_{\overline{4}}} \cdot \underbrace{(\overline{J} + K + \overline{L})}_{\mathbf{M}_{\overline{5}}} \cdot \underbrace{(\overline{J} + \overline{K} + L)}_{\mathbf{M}_{\overline{6}}} \cdot \underbrace{(\overline{J} + \overline{K} + \overline{L})}_{\mathbf{M}_{\overline{7}}}$$

We now have two ways to systematically derive canonical expressions for both positive and negative forms of a boolean function from its truth-table specification. We get the negative form by listing minterms (or maxterms) corresponding to the function's 0s (1s). For instance,

$$\begin{aligned}
\overline{T} &= \underbrace{(\overline{J}+\overline{K}+L)}_{\mathbf{M}_1}\cdot\underbrace{(\overline{J}+K+\overline{L})}_{\mathbf{M}_2}\cdot\underbrace{(\overline{J}+K+L)}_{\mathbf{M}_3} \\
T &= \underbrace{\overline{J}\,\overline{K}\,L}_{\mathbf{m}_1}+\underbrace{\overline{J}\,K\,\overline{L}}_{\mathbf{m}_2}+\underbrace{\overline{J}\,K\,L}_{\mathbf{m}_3} \\
\overline{T} &= \underbrace{\overline{J}\,\overline{K}\,\overline{L}}_{\mathbf{m}_0}+\underbrace{J\,\overline{K}\,\overline{L}}_{\mathbf{m}_4}+\underbrace{J\,\overline{K}\,L}_{\mathbf{m}_5}+\underbrace{J\,K\,\overline{L}}_{\mathbf{m}_6}+\underbrace{J\,K\,L}_{\mathbf{m}_7}
\end{aligned}$$

We will see a third, more informal, way to generate equations from truth tables in Section 3.1.2.

3.1.1 Generating Truth Tables From Equations

If you need to generate a truth table from a given boolean expression there are two ways ways to do it, depending on the form of the expression. For an arbitrary expression $calE$, it may be necessary to *evaluate* it once for every combination of values its variables can assume. To do this systematically, write down an empty truth table in canonical form with the variables' values listed in numerical order and the expression in the place of the function:

$\cdots$	A	B	C	$\mathcal{E}$
	0	0	0·	
	0	0	1·	
	0	1	0·	
	⋮	⋮	⋱	

You can then begin evaluating subexpressions of $\mathcal{E}$ according to the precedence rules. The tabular layout can serve to organize this process. If you do not have experience with expression evaluation of this kind, pay special attention to the exercises at the end of this chapter that ask you to do it.

If the expression happens to be in SOP form, then conversion to a truth table is easy, in light of the previous discussions. Each product term specifies one or more rows of the canonical truth table, assuming that the variables occur in the proper order.

For example, consider the equation

$$U = \underbrace{\overline{D}\,\overline{B}}_{\text{Term 1}} + \underbrace{\overline{D}\,\overline{C}\,B}_{\text{Term2}} + \underbrace{B\,C}_{\text{Term3}}$$

- Begin by surveying the expression to see what variables are involved. In this case there are three: B, C and D.

- Choose a variable order for the truth table. The choice is arbitrary. As usual we will use alphabetical order in the absence of any reason to do otherwise. Although it is not necessary to do so in practice, for this example, for clarity we shall rewrite the equation to use the chosen ordering.

$$U = \underbrace{\overline{B}\,\overline{D}}_{\text{Term1}} + \underbrace{B\overline{C}\,\overline{D}}_{\text{Term2}} + \underbrace{B\,C}_{\text{Term3}}$$

- Note that only one of the product terms for U is a minterm. The other two are missing a variable, either C or D, and so are not canonical. According to identity **??** in Fig. fig:BooleanIdentities (and commutativity),

$$B\,C = B\,C\,D + B\,C\,\overline{D}$$

  So the term $B \cdot C$ stands for two minterms, hence two rows of the truth table, and so does for $\overline{B}D$. Simlarly, a term that was missing two variable occurances would represent four rows of the table, and in general, a term missing k variables stands for 2^k rows.

Putting all this together, we get the truth table

	B	C	D	U
0	0	0	1	From Term 1
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	1	From Term 2
1	0	1	1	From Term 3
1	1	0	0	
1	1	1	1	From Term 3

The procedure for generating tables from POS forms is similar. Recall from the discussion on Page 4 that a maxterm is best though of as the set-complement of its "complementary" row. In other words, maxterm $\mathbf{M}_i$ specifies that there is a *0 in Row* $\overline{i}$, where $\overline{i}$ stands for the row in which all variables hold the opposite values as those specified by i. For instance, given

$$V = \underbrace{(\overline{A} + B + C)}_{\text{Term1}} \cdot \underbrace{(\overline{A} + B)}_{\text{Term2}} + \underbrace{(\overline{A} + \overline{B} + \overline{C})}_{\text{Term3}}$$

Term 1 makes V false for row 4 (100); Term 3 specifies a 0 for row 7 (111); and Term 2, with its missing variable C specifies 0s for rows 4 and 5 (10X). Note

that Term 1 is redundant.

A	B	C	V	
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	0	From Terms 1 and 2
1	0	1	0	From Term 2
1	1	0	1	
1	1	1	0	From Term 3

3.1.2 Condensing Truth Tables

We use truth tables a great deal and so have various ways to express them more compactly. As an illustration, consider the function W defined by the truth table below:

A	B	C	W
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

In both cases where $A = 1$ and $B = 0$ the value of the function W is 0; and similarly, when $A = 1$ and $B = 1$, W is 1. We employ a *don't care* symbol, 'X', to collapse each of these pairs into a single row, and can do likewise for two pairs of cases when $A = 1$:

A	B	C	W
0	0	–	0
0	1	–	1
1	–	0	1
1	–	1	0

To condense further, notice that W's value in the first two rows is the same as that of B. Below, we introduce a mathematical variable b to represent input B's value, and use b to specify W:

A	B	C	W
0	b	–	b
1	–	0	1
1	–	1	0

We could go even further, noting that a logical relationship exists between the values for C and W in the second and third rows. We could again introduce a

variable c for C's table entry and reduce the table to

A	B	C	W
0	b	$-$	b
1	$-$	c	$\overline{c}$

Of course, it is possible to carry such reductions too far, depending on what purpose the table is serving in the first place. A logical function can always be expressed as a term over its input variables. Introducing a for A table entry, we could even reduce to

A	B	C	W
a	b	c	$(\overline{a} \cdot b) + (a \cdot \overline{c})$

At this point, we might as well write

$$W = \overline{A} \cdot B + A \cdot \overline{C}$$

and dispense with the table entirely (See Exercise **??**).

3.1.3 Don't Care Outputs in Truth Tables

Truth tables enjoy a useful property that logic equations cannot express. We often know from the nature of a problem that a function's value is irrelevant for certain combinations of input values. This situation typically arises when we know that certain input combinations cannot or do not legitimately arise. It can also happen that, under certain conditions, a function's value is inaccessible to the surrounding system.

In such cases, it is good design practice to *defer* the decision of what value the function will take as long as possible. Making an unnecessary and arbitrary choice of value is called *overspecification*. It is a bad practice because it unecessarily constrains later implementation choices.

To avoid overspecification in truth tables, we place hyphen, '-' in the table, rather than a 1 or a 0. Like the 'X' that is used in input entries, this symbol is called a *don't care*. In the case of outputs, the '-' literally means that we are free, at a later time, to implement the function using either a 0 or a 1 in this case.

For example, consider the condensed truth table below:

A	B	Y
0	X	1
1	0	-
1	1	0

Should we choose to "implement" this table using a 1 for the don't care entry, we get the equation

$$Y = \overline{A} + A\overline{B} = \overline{A} + \overline{B} \text{(by Eq. ??}$$

If, on the other hand, we choose 0 for the don't-care case, our implementation becomes

$$Y = \overline{A}$$

The don't care gives us the freedom to use either implementation. The choice may depend on other factors arising in the problem design.

3.2 Minimization and Karnaugh Maps

We are of course interested in manipulating boolean expressions, and in particular, making them simpler and smaller—which are often but not always compatable goals!. In the early years of digital hardware, each logic device in a circuit was large, consumed a lot of power, and generated a lot of heat. Consequently, there was great emphasis on reducing the number of elements to a bare minimum in order to conserve implementation costs. Furthermore, while circuits were small compared to Today's standards, they were still complex insofar as our intellectual capacity to design them; hence, a great deal of design efforts was spent on circuit simplification, and a proportionate fraction of a designer's education was devoted to the practice of elaborate minimization techniques.

Today the situation is different for two reasons. First, the cost of an individual logic device, an *and* or *or*, is quite small, however one chooses to measure it. Second, there have been great advances in the automation of circuit minimization; tools are readily available that do a more reliable job than a human can in correctly manipulating boolean expressions. The designer's concerns are focused at much higher levels of problem solving in most circumstances.

Still, there are valid reasons to spend some time developing basic skills at logic minimization. From a practical standpoint, the most important reason is one of convenience. Technological advances have enabled designers to focus on higher level aspects of design, placing far greater emphasis on systematic methodology and *design decomposition*—breaking designs down into managable conceptual pieces. In response, a curcuit industry has evolved to provide designers with generalized components that are suited to this methodology. Such components can be configured, combined, or programmed in application to a range of design problems. As design becomes involved with composing these components, a key task is to implement the "*glue logic*" that integrates them into a single system. These design subtasks often involve the implementation of a number of small logic functions, consisting of just a few input variables, and a degree of proficiency in dealing with small functions can make this process go more smoothly.

A few other reasons to learn a bit about the "arcane" skill of logic minimization are listed below.

- It helps develop insight into what design automation tools are doing. It sometimes happens that the design tools cannot produce implementations that meet the designer's performance goals. When this happens, the designer must become involved in guiding the tool to a better outcome.

This can be done in several ways, including adjustment of optimization parameters, addition of advisory pragmas, reformulation of the source description, or (all too often) manual alterations to the tool output. Any of these tactics benefit from knowledge of how the tool does its job: its basic algorithms, heuristics, and knowledge base.

- It is a form of focused relaxation. If all routine tasks are delegated to someone or something else, the designer would spend all the time working on the most challenging tasks. Studies have suggested that in the absence of intellectual "breaks" productivity declines—either progress slows or the number of mistakes increases—owing to the stress of constant decision making. Pausing occasionally to solve a straightforward subproblem, or even to straighten up the workbench, can be beneficial in the long run.

- It's good for you. Some people believe practicing basic skills, such as arithmetic, algebra, and so on, cultivates a higher understanding of the underlying concepts, just as physical exercise strengthens the body. Of course, opinions vary widely about how much and what kind of practice is healthy and useful.

3.2.1 Karnaugh Maps

The *Karnaugh Map*, or *K-map* for short, is a visual aid for simplifying sum-of-products expressions[1]. A K-map is a canonical two-dimensional rendering of a truth table. The first example, below, shows the K-map of a two-input function Y over variables A and B.

	A	B	Y
(0)	0	0	Y_0
(1)	0	1	Y_1
(2)	1	0	Y_2
(3)	1	1	Y_3

B \ A	0	1
0	Y_0	Y_2
1	Y_1	Y_3

A three-variable K-map contains eight squares, corresponding to the eight rows of a canonical truth table. Two notations for the three-variable map are shown in Fig. 3.1. This book uses the form on the left. Although it takes slightly more effort to draw, the form on the right conveys the important fact that each half region of the map represents all those cases for which one of the variables has a *true* value. It is the content of the table that matters, not the external annotations, so you may use either form. Note carefully the the order of the labels in the left-hand form. In moving from square to square in the map only one bit changes at a time. This holds horizontally and vertically, and also around the corners. As you are about to see, this *unit distance* property is what makes the K-map useful for simplifying logic expressions.

Extending K-maps to four variables adds an additional variable to the vertical side, with the same indexing scheme as just discussed. The scheme is shown

[1] Karnaugh maps are also known as *Veitch diagrams*

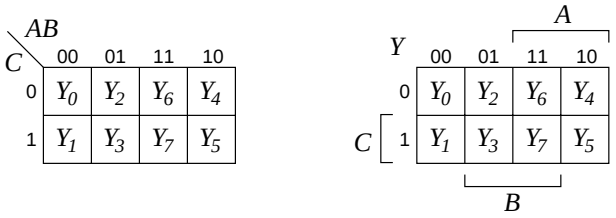


Figure 3.1: Two forms of the three-variable K-map.

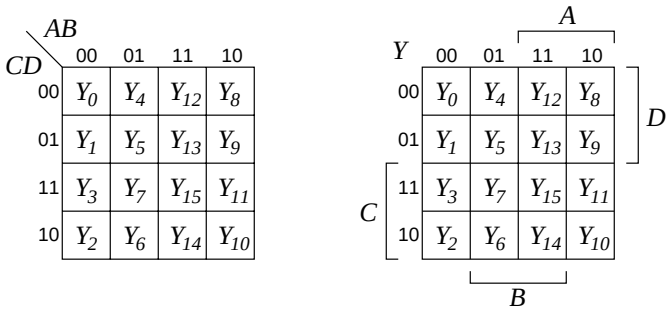


Figure 3.2: Two forms of the four-variable K-map.

in Fig. 3.2. At this point we have run out of dimensions to extend the tables for additional variables, and Karnaugh maps are nearing the limit of their useful range as a visual aid. As we shall see later in this section, there are techniques for extending to five-variable (and sometimes even six) problems.

3.2.2 Building K-Maps

There is a direct correspondence between K-maps and canonical truth tables, and hence also between K-maps and canonical SOP expressions. K-maps and truth tables are just different arrangements of the same information. In a condensed truth table, each don't-care (X) input yields values for a contiguous block of squares in a K-map, as we shall see in a moment.

We can construct the K-map of a logic equation in just the same way that we derive a truth table. If the equation is in canonical SOP form, each product term corresponds to square in the map. Figure 3.3 illustrates the correspondences among three representations of the same function, X.

Derivation of a K-map from an SOP that is not in full canonical form, is still straightforward. For example, consider the equation

$$V = A \cdot \overline{B} + B + A \cdot \overline{B} \cdot C$$

Since there are three variables involved, we will have a three-variable K-map. Each of the product terms contributes a block of 1s to the map, which has 0s

$$X = \overline{A} \cdot \overline{B} + A \cdot \overline{B}$$

$$\Updownarrow$$

A	B	X
0	0	1
0	1	0
1	0	1
1	1	0

$\Leftrightarrow$

B \ A	0	1
0	0	1
1	1	1

Figure 3.3: Three representations of a boolean function.

everywhere else. The first term, $A\overline{B}$ yields 1s in the K-map in every square for which A, $B = 10$ and C equals anything. To have a true value, the second term requires B to be 1, but A and C can be anything—a region of four squares. Finally, the term $A\overline{B}C$ specifies just one square in the K-map, where A, B, $C = 101$. The process of deriving the K-map is illustrated in Figure 3.4

3.2.3 Simplifying K-maps

As we have already seen, certain regions in K-maps correspond to reduced, or non-canonical, product terms. These terms arise when a function's value is independent of one or more of its inputs, the same condition that allows us to condense truth tables. This property is a consequence of Eq. **??** in Fig. **??**. Whenever we have two product terms that differ only in their value for a particular variable, we can replace them with a shorter term in which that variable is omitted. For instance, the function X in Fig. 3.3 can be simplified to:

$$X = A \cdot \overline{B} + \overline{A} \cdot \overline{B} \stackrel{??}{=} \overline{B}$$

The K-map is a visualization aid that helps us find and apply instances of this identity. Let's circle the region specified by this SOP:

B \ A	0	1
0	1	1
1	0	0

The circle is a reminder that this block of 1s share the property that $B = 0$ and span all of the possible values for A. That is, the circle represents the term "B." Each individual 1 represents a canonical product term; and each

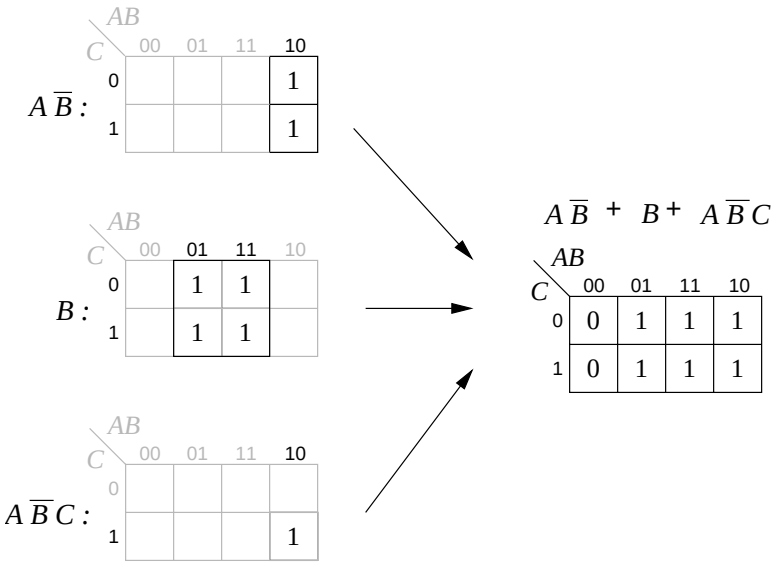
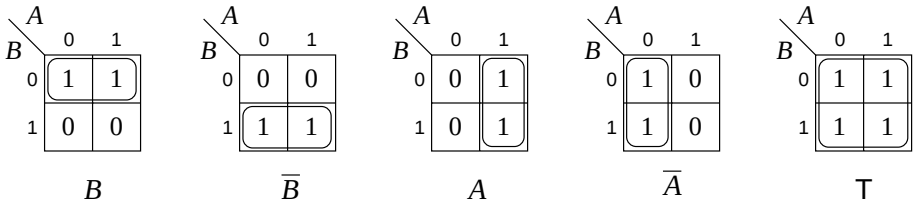


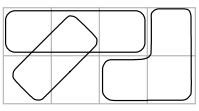
Figure 3.4: Derivation of a 3-variable K-map

rectangular region in the 2×2 K-map represents a reduced term:



In using the K-map to perform simplification, we look for blocks to circle. The larger the block, the smaller the corresponding product term, so the goal is to "cover" all the 1s with the fewest and largest circles. That is, each 1 must appear in at least one circled region, even if it is by itself.

For K-maps of three or more variables, some additional issues arise. For one thing, we discover that not all regions are proper blocks. Regions must be rectangular, so circlings such as those shown below are *improper*.



Of these examples the diagonal and the 'L'-shaped circlings are perhaps obiously wrong, but what about the 1-by-3 region? It is improper because, no matter where it is put on the map, it fails to represent a single value for any input. The

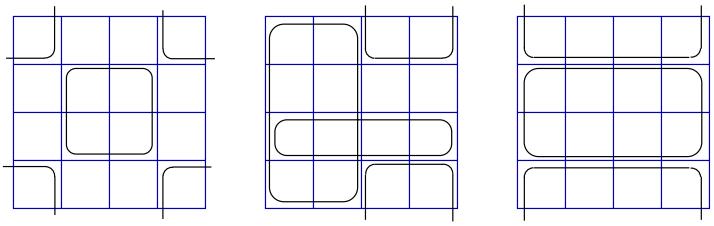
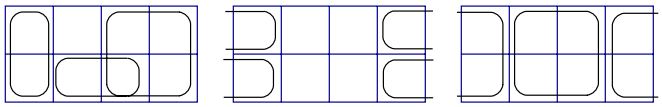


Figure 3.5: Typical circlings involving four 1s and eight 1s on K-maps of four variables.

valid regions are all rectangles of size $2^i \times 2^j$— including the cases where $i = 0$, $j = 0$ or $i = j = 0$, of course.

There are also proper regions that cannot be encircled entirely within the K-map. In the 2-by-1 K-map, there are three such regions:



K-maps of four variables have four squares on a side. Figure 3.5 shows some forms involving correct circlings of four and eight 1s. The proper circlings shown earlier of 1 and 2 squares remain valid.

Given the K-map of a function, the circling procedure is as follows:

- (a) Draw circles, ovals, or around-the-corner patterns enclosing properly sized blocks of 1s.

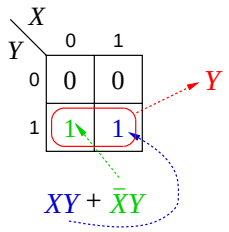
- (b) Enclose each 1 in the largest circle containing only 1s and don't-cares, but *no 0s*. Overlapping is allowed, however, never draw a circle that is entirely contained in another circle—this would result in a redundant term.

  - i. Take care of the "lonely" variables first. These are the variables admitting only one useful circling.
  - ii. Then work from the largest blocks toward the smaller ones.

- (c) Circling is finished when all the 1s are circled. It is not necessary to encircle all the don't-cares. The point of using K-maps is to use the drawing process to derive the result in a systematic and mechanical fashion.

- (d) Write down the product term corresponding to each circle in the map.

Example 1. A proof of the boolean identity Eq. **??** in the form of a K-map looks like:



Example 2. The four-variable SOP below is already a minimal SOP, as its K-map admits no consolidation of regions. As we shall see in Sec. 3.3, further simplification is possible, but does not result in an SOP form.

$$\begin{aligned} X = {} & \bar{N}\bar{P}\bar{Q} \\ & + \bar{M}NP \\ & + NP\bar{Q} \\ & + M\bar{N}PQ \\ & + MN\bar{P}Q \end{aligned}$$

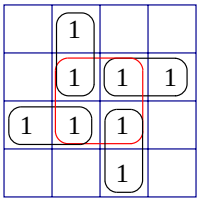
PQ \ MN	00	01	11	10
00	1	0	0	1
01	0	0	1	0
11	0	1	0	1
10	0	1	1	0

Example 3. The function Y, below, another four-variable SOP, reduces from five product terms, including a total of 16 *and* and *or* operations; to three product terms with a total of 7 *and*s and *or*s. For reasons that will become clear in Chapter 2, we often do not count all the *not* operations when estimating the *cost* of an expression (this time we have chosen not to count any of them).

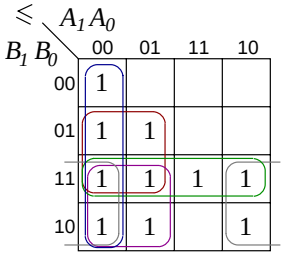
$$\begin{aligned} X = {} & \bar{N}\bar{P}\bar{Q} \\ & + \bar{M}NP \\ & + NP\bar{Q} \\ & + M\bar{N}PQ \\ & + MN\bar{P}Q \end{aligned}$$

PQ \ MN	00	01	11	10
00	1	0	0	1
01	0	0	1	0
11	0	1	0	1
10	0	1	1	0

Example 4. The K-map below illustrates a pathology explaining the strategy of starting with *lonely variables*.[2] If we start first by circling the largest regions and later pick up the "outlying" 1s, there is a risk of introducing redundant circlings, as below. Spend a few moments thinking about how this pathology arises. You will see that the combination of necessary conditions are comparatively rare—all the more reason to exercise care and adopt a systematic circling strategy.

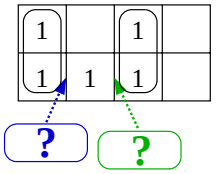


Example 5. We want to implement a two-bit *comparator* function that returns T if the binary value $A_1\,A_0 \leq B_1\,B_0$. If we arrange the indices carefully, there is no need to construct a truth table or even write down an expression defining the function. We can instead simply fill in the K-map (taking care to remember the unit-distance ordering of the indices!), and then derive the minimized SOP.



$$\Longrightarrow \overline{A_1}\,\overline{A_0} + A_1\,\overline{A_0}\,B_1 + B_1\,B_0 + \overline{A_1}\,B_0 + \overline{A_0}\,B_0$$

You may already have noticed that the best circling is not always unique. As illustrated below, situations can arise when there is a choice as to how to cover all the 1s.



From the standpoint of the expression being simplified, either choice is acceptable, but using both is not, nor is using a smaller circle "for the sake of symmetry."

[2] Thanks to Tony Zamora for pointing out this example.

At the same time, the surrounding context may make one choice perferable over the other. The designer may know, for example, that one of the variables is available in its positive form but not its negative form, and that global design knowledge may guide the choice. In the absence of a definite choice, make a decision and move on.[3]

3.2.4 K-map Simplification Blunders

The most common mistake made in simplifying expressions with K-maps is failing to circle the largest possible groupings of 1s. Often, this is due to a failure to notice a region that wraps around a corner. Figure **??** illustrates typical mistakes—try to find them without looking at the corrections on the right. A far less common mistake is to enclose one circling within another. Also bear in mind the more subtle redundancy illustrated in Example 4 Either of these mistakes, correctly translated, still produce valid SOP expressions, but they are not as simple (i.e. small) as they could be.

Beginners sometimes use inappropriate circlings, such as a 1×3 or a 2×3 region. This is a true blunder, because it can lead to an invalid result; but it is easily seen and corrected as soon as one has had some practice writing down the minterms that correspond to valid regions.

- Take care of the "lonely" variables first. These are the variables with only one useful circling.
- Then work from the largest blocks toward the smaller ones.

See Exercise **??**.

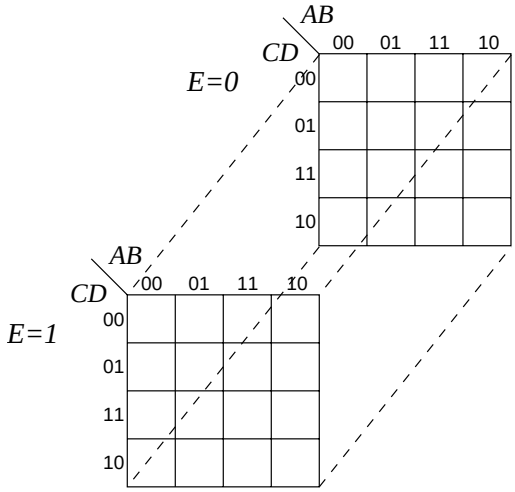
Other Ways to Read K-maps. As always, and just as with truth tables, we have four ways of interpreting K-maps, two of which are consequences of duality. We list them below without further discussion, and encourage the Reader to demonstrate them as a self-test of concepts presented thoughout this chapter.

> *Method 1:* (The standard method). Circle 1s and generate a simplified expression in SOP form.
>
> *Method 2:* Circle 0s and generate the inverse of the function in simplified SOP form.
>
> *Method 3:* Circle 0s and generate a simplified expression in POS form.
>
> *Method 4:* Circle 1s and generate a the inverse of the function in simplified POS form.

[3] Too many of these kinds of arbitrary choices can be stressful because, as one accumulates experience, one tends to remember the choices that prove later to be wrong. Making the right choice in the absence of any obvious reason for doing so is what some call "the wisdom of experience."

3.2.5 Dealing with five or more variables

With 5 or more variables, the K-map is no longer effective as a visual aid. We cannot extend the map any further in two dimensions and would have to try something like a three-dimensional extension, with a one copy of the four-variable map for each value of the fifth variable.

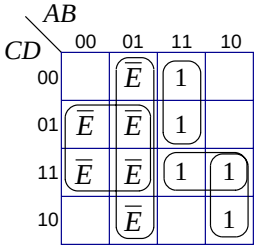


Our "circlings" would likewise become three-dimensional. All in all this is not a very effective way to visualize simplification.

Recall that in Section **??** we introduced mathematical variables as a means to condense truth tables by incorporating partial function expressions in the output column. We can use a similar technique to extend a K-map's utility. Consider, for example, the function

$$Y = A\,\overline{B}\,C + \overline{A}\,\overline{B}\,D\,\overline{E} + A\,B\,\overline{D}\,\overline{D} + A\,B\,D + \overline{A}\,B\,\overline{E}$$

We use *map entered* variables, E and $\overline{E}$ in the K-map below, preserving positive and negative occurances of E in those product terms that contain them. For the term $A\,\overline{B}\,C$, 1s are entered in the corresponding region, while for the term $\overline{A}\,\overline{B}\,D\,\overline{E}$ we enter $\overline{E}$s rather than 1s in the region designated by $\overline{A}\,\overline{B}\,D$, and so forth.



$$Y = A\,B\,\overline{C} + A\,B\,C + A\overline{B}\,C + \overline{A}\,B\,\overline{E} + \overline{A}\,B\,\overline{E}$$

The procedure for circling map-entered K-maps generalizes that on page 14 for basic K-maps in two ways:

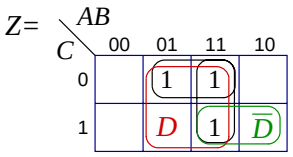
- (b') Enclose each 1 in the largest proper block containing only 1s and don't cares. In addition, enclose each mapped-variable instance V in the largest proper block containing only Vs, 1s and don't-cares. Note positive and negative instances are regarded as *different* for circling purposes.

- (d') Write down a product term for each circle. For circles enclosing a mapped variable instance V, include V as a literal in the term.

The fact that we can use 1s in circlings for mapped variables is a consequence of identity **??** in Fig. **??**:

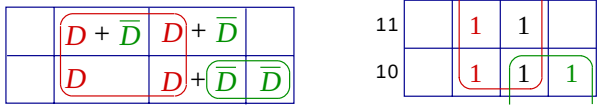
$$1 = V + \overline{V}$$

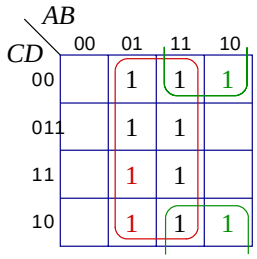
For insight into what is going on, let us consider the example



$$Z = D\,B + B\,\overline{C} + A\,B + A\,C\,\overline{D}$$

If we replace the 1s with $D + \overline{D}$, or "expand" the C-rows of the K-map (as we might expand rows of a truth table), we can see more clearly how the 1s contribute to the mapped-variable circlings.





Mapped variables are a useful extension of K-maps, but they should be used with caution because the no longer allow us to systematically and visually simplify an expression to its minimal SOP form. Used correctly, simplification will always produce a valid expression, but it may not be the simplest one, because, in effect, we are solving two or more simplification problems with the same map.

As a general rule, when simplifying with mapped variables, one should select the variable occuring in the fewest product terms of a canonical SOP expansion. This tactic assumes that we are starting with a canonical expansion in the first place, and even if we are, there may be two or more candidates to choose from. In any event, keep in mind that mapped variables are a means to extend the usefulness of K-maps at the possible price of sacrificing the ability to always find a minimal SOP form.

3.2.6 Generalization and Automation

We have introduced a visual, "back-of-the-envelope" method for minimizing SOP expressions based on Karnaugh maps. The examples show that it is a technique that is effective for small examples, but also entails enough subtlety that practice is needed before one can gain confidence in using it. The method is in the number of variables it can handle and therefore also in the size of the expressions one can maniuplate with it.

One can, in fact, overcome the graphical limitation to around four variables by using a tabular technique called *Quine-McCluskey minimization*. Like Karnaugh maps, the procedure is systematic and, if correctly applied, assured that a (one of possibly several) minimal SOP form is obtained. Based on the foundation of boolean identities as K-maps, the Quine-McCluskey method is also well suited for automation, as is the basis for most SOP optimization tools.

If you have need to simplify an expression larger than can be handled with a K-map, it is advisable to employ an automated minimization tool. Minimization is a routine but tedious task which grows rapidly as expression size grows. Essentially, each additional variable doubles the size of the problem, and considering human fallability, more than doubles the liklihood of that an error will occur when solving the problem manually.

At this point it is timely to mention that, just as we should learn how far to trust our own capabilities and always to cross-check our calculations, we should develop an appropriate level of mistrust in design automation tools and verify their results. Should the situation call for the use of minimization tool, it calls also for validation of its output. Though unlikely, it is possible for the tool to make a mistake. Much more often, we tend to trust not just the results produced by the tool, but the input we provided it as well. Automated tasks such as minimization simplify work but also magnify errors by making it more difficult to diagnose their source.

3.3 Multi-level optimization

Example 6. Recall from Example 5 that

$$(A_1 A_0 \leq B_1 B_0) = -A_1 \; - A_0 + A_1 \; - A_0 \, B_1 + B_1 \, B_0 + -A_1 \, B_0 + -A_0 \, B_0$$

This is a minimal SOP form involving six binary *and* operations and four binary *or*s (As mentioned on Page 15 and explained in Chapters 2 and 3, we generally don't count individual *not*s). Although we cannot find a smaller SOP expression, we can reduce this expression further using the distributive laws (Eq. **??**, Fig. **??**) and other boolean identities:

$$\begin{array}{lcll} (A_1 A_0 \leq B_1 B_0) & = & \overline{A_1}\,\overline{A_0} + A_1\,\overline{A_0}\,B_1 + B_1\,B_0 + \overline{A_1}\,B_0 + \overline{A_0}\,B_0 & \\ & = & \overline{A_1}\,\overline{A_0} + A_1\,\overline{A_0}\,B_1 + B_0\,(B_1 + \overline{A_1} + \overline{A_0}) & \text{(Eq. ??)} \\ & = & \overline{A_0}\,(\overline{A_1} + A_1\,B_1) + B_0\,(B_1 + \overline{A_1} + \overline{A_0}) & \text{(Eq. ??)} \\ & = & \overline{A_0}\,(\overline{A_1} + B_1) + B_0\,(B_1 + \overline{A_1} + \overline{A_0}) & \text{(Eq. ??)} \end{array}$$

The result of this algebraic derivation is an expression with two *and*s and four *or*s, and by that measure it is simpler than the SOP derived just above. It is not an SOP, however.

SOPs and POSs are sometimes called *two-level* logic expressions because they nest operations of different kinds just two levels deep. Here we are thinking of a uniform n-ary operation

$$x_1 + x_2 + \cdots + x_n \text{ or } y_1 * y_2 * \cdots * y_n$$

as a single "level."

More deeply nested expressions like the one just derived are called *multi-level* logic expressions. We sometimes want to derive equivalent boolean expressions that are minimal in the sense we have just seen: containing the fewest possible number of (say) binary logic operations. When the expressions are small, it may be sufficient to perform boolean algebra with the goal of minimization.

There are no visualization aids like K-maps for doing simplification purpose. The situation is similar[4] to that for ordinary algebra, manual derivations can go awry, and even become circular; and it is hard to tell when you are done. As usual, practice improves proficiency, but so to does the judicious use of design automation tools. Our vast experience with ordinary algebra helps, but may sometimes fail us, owing to duality.

Example 7. Here is an important function that we shall see again in Chapter 3, given as a truth table and again as a K-map.

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Z:

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

$$Z = -A - BC + -AB - C + ABC + A - B - C$$

Z's SOP form does not admit any simplification; there are no adjacent 1s in it's K-map, so no block's larger than 1×1 can be circled. This is an extreme example a quality that can be seen in K-maps to indicate an expensive SOP implementations. It takes 11 boolean operations (excluding negation) to implement Z as an expression.

[4] Unlike the case for algebra over the real numbers, a set of algebraic rewriting rules exists that can be used deterministically to reduce an expression to its simplest terms. I am referring here to boolean algebra as exercised by human beings.

The derivation below simplifies Z to a multi-level expression involving

$$
\begin{array}{lll}
Z = & \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,B\,C + A\,\overline{B}\,\overline{C} & \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(B\,C + \overline{B}\,\overline{C}) & \text{(distr.)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{\overline{B\,C} + \overline{B}\,\overline{C}}) & \text{(Eq. ??)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{B\,C}\,\overline{\overline{B}\,\overline{C}}) & \text{(DeMorgan)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{(\overline{B} + \overline{C})(\overline{B} + \overline{C})}) & \text{(DeMorgan)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{(\overline{B} + \overline{C})(B + C)}) & \text{(dbl. negation)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{(\overline{B} + \overline{C})B + (\overline{B} + \overline{C})C}) & \text{(distr.)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{(\overline{B}\,B + \overline{C}\,B) + (\overline{B}\,C + \overline{C}\,C)}) & \text{(distr.)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{(0 + \overline{C}\,B) + (\overline{B}\,C + 0)}) & \text{(inverse)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{\overline{C}\,B + \overline{B}\,C}) & \text{(identity)} \\
& \overline{A}\,(\overline{B}\,C + B\,\overline{C}) + A\,(\overline{\overline{B}\,C + B\overline{C}}) & \text{(?)}
\end{array}
$$

Why stop here? Is this the minimal expression as measured in terms of boolean operations? See Exercise **??**. Another important property of the derived expression above is its use a repeated pattern, which we might define as follows:

$$f(x, y) =_{\text{def}} \overline{x}\,y + x\,\overline{y}$$

You may recognize this as the *exclusive-or* function, a binary function that is *true* when exactly one of its arguments is *true*. This is such a common and useful funtion that we shall give it an infix symbol, writing $x \oplus y$ rather than $f(x, y)$. With this function defined, our derived expression above becomes

$$Z = \cdots = A \oplus (B \oplus C)$$

The lesson here is that a derivation may have goals other than, or in addition to, strict minimization of the number of operators in an expression. In fact, our limited practice of boolean algebra is just as likely to involve a balance of objectives in manipulating an expression.

As we have just seen, the introduction of auxiliary function definitions, like *exclusive or* in this example, can greatly increase the economy of expressions by introducing a special term to represent a parameterized boolean expression. This does not give us all the generatity we need, however. Local function definitions give us the means to write down hierarchical expressions, but not the means to describe identify common subexpression.

For instance, consider the expression

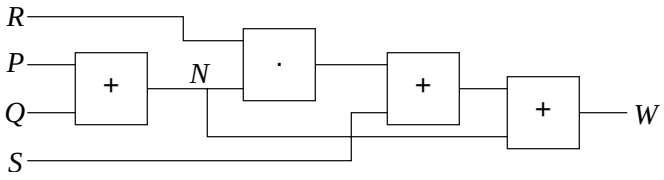
$$W = (P + Q)\,R + S + (P + Q)$$

How "big" is W as measured in the number of operations it uses? Reading the expression, the answer would be "W uses four *or*s and one *and*. But the same

subexpression $P+Q$ appears twice and were we to implement W—as a program expression for example—we would have the option of computing $P+Q$ just once and using the result twice. Of course, hardware gives us the same option, and one way to express it is to give the term $P+Q$ a name and then use that name twice:

$$\begin{aligned} N &= P+Q \\ W &= N\,R+N+S \end{aligned}$$

This *system*, or set, of equations may be interpreted as describing a circuit containing two *and* and two *or* devices connected as the prescribed by the equations:



3.4 Conclusions and Directions

Logic is for reasoning about truth; algebra is for reasoning about equality. In the case of propositional logic and boolean algebra, these two notions merge into a tool set for dealing manipulating expressions in a two-valued system reflecting the binary digital devices we call gates.

Exercises 3.4

1. Figure defines five binary (two-input) logic functions.

(a) How many two-input logic functions are there?

(b) In this chapter you have learned that any logic function can be expressed in terms of '$\wedge$', '$\vee$' and '$\neg$'. Show that each of '$\wedge$', '$\vee$' and '$\neg$' can be expressed in terms of the *nand* operation, defined to the right.

nand, $\downarrow$		
A	B	$A \downarrow B$
F	F	T
F	T	F
T	F	F
T	T	F

(c) Are there any other logic functions, distinct from *nand*, that can implement '$\wedge$', '$\vee$' and '$\neg$', and therefore all other functions?

2. On Page **??** a *nand* gate is described by an "equation," $Z = A\ nand\ B$. Discuss what the '=' sign might represent in this context? Is it mathematical equality? A defining expression? An assignment statement?

3. Verify that the truth table for Y on page **??** specifies the function

$$Y = S \wedge A \vee \neg S \wedge B$$

4. By inserting full parentheses, show the order of evaluation of these function.

(a) $B \cdot \overline{A \cdot C} + D + \overline{E}$
(b) $\overline{A + B} \cdot C + D$
(c) $\overline{A + B} \cdot (\overline{C} + D)$

5. By inspecting their truth tables, deduce the value of each function, X, Y, Z and W. Do not formally derive any logic expression; simply write down the minimal formulation of each function.

A	B	C	W	X	Y	Z
0	0	0	0	1	0	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	1	0
1	1	0	0	1	1	1
1	1	1	0	1	1	0

6. Write logic equations corresponding to the following:

(a) $X(A, B, C) = \mathbf{m}_0 + \mathbf{m}_2 + \mathbf{m}_5$
(b) $\overline{Y}(P, Q) = \mathbf{M}_1 \cdot \mathbf{M}_3$
(c) $U(V, W, X) = \mathbf{M}_2 \cdot \mathbf{M}_3 \cdot \mathbf{M}_5 \cdot \mathbf{M}_6$
(d) $V(C, B, G) = \mathbf{m}_1 + \mathbf{m}_2 + \mathbf{m}_7$

7. Write canonical truth tables for each of the functions defined in Exercise 6

8. Consider the following truth table:

A	X	YZ	G
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	0

Derive canonical equations for G or $\overline{G}$ in the following forms:

(a) Sum of products on *true* outputs.

(b) Sum of products on *false* outputs.

(c) Product of sums on *true* outputs.

(d) Product of sums on *false* outputs.

9. Derive the canonical truth tables the correspond to each of the following K-maps.

0	0	0	0
0	0	1	0

0	1	0	0
0	1	-	0
-	1	1	0
0	1	1	0

10. Verify that the truth table for W on page 7 specifies the function

$$W = (\neg A \wedge B) \vee (A \wedge \neg C)$$

11. Do all systems of boolean equations describe meaningful boolean circuits like the one on Page 23. Can you think of a property a system might have that would be impossible to draw as a schematic? Can you think of a property that would result in schematics that are ill-formed in some way?

12. Carefully compare the binary tree (BT) and the binary decision diagram (BDD) in Fig. **??** on page **??**. Explain the correspondence between the root-to-leaf paths in both structures. Discuss the relationship to K-maps.

13. As a project, research and implement the *Quine-McCluskey* procedure for boolean simplification.

