

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 2.1

Optimizing Logic Expressions

The second part of the course covers digital design more deeply than does the textbook. The lecture notes will explain the additional material, and we will provide further examples in lectures and in discussion sections. Please let us know if you need further material for study.

In the last notes, we introduced Boolean logic operations and showed that with AND, OR, and NOT, we can express any Boolean function on any number of variables. Before you begin these notes, please read the first two sections in Chapter 3 of the textbook, which discuss the operation of **complementary metal-oxide semiconductor** (**CMOS**) transistors, illustrate how gates implementing the AND, OR, and NOT operations can be built using transistors, and introduce DeMorgan's laws.

This set of notes exposes you to a mix of techniques, terminology, tools, and philosophy. Some of the material is not critical to our class (and will not be tested), but is useful for your broader education, and may help you in later classes. The value of this material has changed substantially in the last couple of decades, and particularly in the last few years, as algorithms for tools that help with hardware design have undergone rapid advances. We talk about these issues as we introduce the ideas.

The notes begin with a discussion of the "best" way to express a Boolean function and some techniques used historically to evaluate such decisions. We next introduce the terminology necessary to understand manipulation of expressions, and use these terms to explain the Karnaugh map, or K-map, a tool that we will use for many purposes this semester. We illustrate the use of K-maps with a couple of examples, then touch on a few important questions and useful ways of thinking about Boolean logic. We conclude with a discussion of the general problem of multi-metric optimization, introducing some ideas and approaches of general use to engineers.

Defining Optimality

In the notes on logic operations, you learned how to express an arbitrary function on bits as an OR of minterms (ANDs with one input per variable on which the function operates). Although this approach demonstrates logical completeness, the results often seem inefficient, as you can see by comparing the following expressions for the carry out C from the addition of two 2-bit unsigned numbers, $A = A_1A_0$ and $B = B_1B_0$.

$$\begin{aligned} C &= A_1B_1 + (A_1 + B_1)A_0B_0 && (1)\\ &= A_1B_1 + A_1A_0B_0 + A_0B_1B_0 && (2)\\ &= \overline{A_1}\ A_0\ B_1\ B_0 + A_1\ \overline{A_0}\ B_1\ \overline{B_0} + A_1\ \overline{A_0}\ B_1\ B_0 + \\ &\quad A_1\ A_0\ \overline{B_1}\ B_0 + A_1\ A_0\ B_1\ \overline{B_0} + A_1\ A_0\ B_1\ B_0 && (3)\end{aligned}$$

These three expressions are identical in the sense that they have the same truth tables—they are the same mathematical function. Equation (1) is the form that we gave when we introduced the idea of using logic to calculate overflow. In this form, we were able to explain the terms intuitively. Equation (2) results from distributing the parenthesized OR in Equation (1). Equation (3) is the result of our logical completeness construction.

Since the functions are identical, does the form actually matter at all? Certainly either of the first two forms is easier for us to write than is the third. If we think of the form of an expression as a mapping from the function that we are trying to calculate into the AND, OR, and NOT functions that we use as logical building blocks, we might also say that the first two versions use fewer building blocks. That observation does have some truth, but let's try to be more precise by framing a question. For any given function, there are an infinite number of ways that we can express the function (for example, given one variable A on which the function depends, you can OR together any number of copies of $A\overline{A}$ without changing the function). *What exactly makes one expression better than another?*

In 1952, Edward Veitch wrote an article on simplifying truth functions. In the introduction, he said, "This general problem can be very complicated and difficult. Not only does the complexity increase greatly with the number of inputs and outputs, but the criteria of the best circuit will vary with the equipment involved." Sixty years later, the answer is largely the same: the criteria depend strongly on the underlying technology (the gates and the devices used to construct the gates), and no single **metric**, or way of measuring, is sufficient to capture the important differences between expressions in all cases.

Three high-level metrics commonly used to evaluate chip designs are cost, power, and performance. Cost usually represents the manufacturing cost, which is closely related to the physical silicon area required for the design: the larger the chip, the more expensive the chip is to produce. Power measures energy consumption over time. A chip that consumes more power means that a user's energy bill is higher and, in a portable device, either that the device is heavier or has a shorter battery life. Performance measures the speed at which the design operates. A faster design can offer more functionality, such as supporting the latest games, or can just finish the same work in less time than a slower design. These metrics are sometimes related: if a chip finishes its work, the chip can turn itself off, saving energy.

How do such high-level metrics relate to the problem at hand? Only indirectly in practice. There are too many factors involved to make direct calculations of cost, power, or performance at the level of logic expressions. Finding an **optimal** solution—the best formulation of a specific logic function for a given metric—is often impossible using the computational resources and algorithms available to us. Instead, tools typically use heuristic approaches to find solutions that strike a balance between these metrics. A **heuristic** approach is one that is believed to yield fairly good solutions to a problem, but does not necessarily find an optimal solution. A human engineer can typically impose **constraints**, such as limits on the chip area or limits on the minimum performance, in order to guide the process. Human engineers may also restructure the implementation of a larger design, such as a design to perform floating-point arithmetic, so as to change the logic functions used in the design.

Today, manipulation of logic expressions for the purposes of optimization is performed almost entirely by computers. Humans must supply the logic functions of interest, and must program the acceptable transformations between equivalent forms, but computers do the grunt work of comparing alternative formulations and deciding which one is best to use in context.

Although we believe that hand optimization of Boolean expressions is no longer an important skill for our graduates, we do think that you should be exposed to the ideas and metrics historically used for such optimization. The rationale for retaining this exposure is threefold. First, we believe that you still need to be able to perform basic logic reformulations (slowly is acceptable) and logical equivalence checking (answering the question, "Do two expressions represent the same function?"). Second, the complexity of the problem is a good way to introduce you to real engineering. Finally, the contextual information will help you to develop a better understanding of finite state machines and higher-level abstractions that form the core of digital systems and are still defined directly by humans today.

Towards that end, we conclude this introduction by discussing two metrics that engineers traditionally used to optimize logic expressions. These metrics are now embedded in **computer-aided design** (**CAD**) tools and tuned to specific underlying technologies, but the reasons for their use are still interesting.

The first metric of interest is a heuristic for the area needed for a design. The measurement is simple: count the number of variable occurrences in an expression. Simply go through and add up how many variables you see. Using our example function C, Equation (1) gives a count of 6, Equation (2) gives a count of 8, and Equation (3) gives a count of 24. Smaller numbers represent better expressions, so Equation (1) is the best choice by this metric. Why is this metric interesting? Recall how gates are built from transistors. An N-input gate requires roughly $2N$ transistors, so if you count up the number of variables in the expression, you get an estimate of the number of transistors needed, which is in turn an estimate for the area required for the design.

A variation on variable counting is to add the number of operations, since each gate also takes space for wiring (within as well as between gates). Note that we ignore the number of inputs to the operations, so a 2-input AND counts as 1, but a 10-input AND also counts as 1. We do not usually count complementing

variables as an operation for this metric because the complements of variables are sometimes available at no extra cost in gates or wires. If we add the number of operations in our example, we get a count of 10 for Equation (1)—two ANDs, two ORs, and 6 variables, a count of 12 for Equation (2)—three ANDS, one OR, and 8 variables, and a count of 31 for Equation (3)—six ANDs, one OR, and 24 variables. The relative differences between these equations are reduced when one counts operations.

A second metric of interest is a heuristic for the performance of a design. Performance is inversely related to the delay necessary for a design to produce an output once its inputs are available. For example, if you know how many seconds it takes to produce a result, you can easily calculate the number of results that can be produced per second, which measures performance. The measurement needed is the longest chain of operations performed on any instance of a variable. The complement of a variable is included if the variable's complement is not available without using an inverter. The rationale for this metric is that gate outputs do not change instantaneously when their inputs change. Once an input to a gate has reached an appropriate voltage to represent a 0 or a 1, the transistors in the gate switch (on or off) and electrons start to move. Only when the output of the gate reaches the appropriate new voltage can the gates driven by the output start to change. If we count each function/gate as one delay (we call this time a **gate delay**), we get an estimate of the time needed to compute the function. Referring again to our example equations, we find that Equation (1) requires 3 gate delays, Equation (2) requires 2 gate delays, Equation (3) requires 2 or 3 gate delays, depending on whether we have variable complements available. Now Equation (2) looks more attractive: better performance than Equation (1) in return for a small extra cost in area.

Heuristics for estimating energy use are too complex to introduce at this point, but you should be aware that every time electrons move, they generate heat, so we might favor an expression that minimizes the number of bit transitions inside the computation. Such a measurement is not easy to calculate by hand, since you need to know the likelihood of input combinations.

Terminology

We use many technical terms when we talk about simplification of logic expressions, so we now introduce those terms so as to make the description of the tools and processes easier to understand.

Let's assume that we have a logic function $F(A, B, C, D)$ that we want to express concisely. A **literal** in an expression of F refers to either one of the variables or its complement. In other words, for our function F, the following is a complete set of literals: A, $\overline{A}$, B, $\overline{B}$, C, $\overline{C}$, D, and $\overline{D}$.

When we introduced the AND and OR functions, we also introduced notation borrowed from arithmetic, using multiplication to represent AND and addition to represent OR. We also borrow the related terminology, so a **sum** in Boolean algebra refers to a number of terms OR'd together (for example, $A + B$, or $AB + CD$), and a **product** in Boolean algebra refers to a number of terms AND'd together (for example, $A\overline{D}$, or $AB(C + D)$. Note that the terms in a sum or product may themselves be sums, products, or other types of expressions (for example, $A \oplus \overline{B}$).

The construction method that we used to demonstrate logical completeness made use of minterms for each input combination for which the function F produces a 1. We can now use the idea of a literal to give a simpler definition of minterm: a **minterm** for a function on N variables is a product (AND function) of N literals in which each variable or its complement appears exactly once. For our function F, examples of minterms include $ABC\overline{D}$, $A\overline{B}CD$, and $\overline{A}BC\overline{D}$. As you know, a minterm produces a 1 for exactly one combination of inputs.

When we sum minterms for each output value of 1 in a truth table to express a function, as we did to obtain Equation (3), we produce an example of the sum-of-products form. In particular, a **sum-of-products** (**SOP**) is a sum composed of products of literals. Terms in a sum-of-products need not be minterms, however. Equation (2) is also in sum-of-products form. Equation (1), however, is not, since the last term in the sum is not a product of literals.

Analogously to the idea of a minterm, we define a **maxterm** for a function on N variables as a sum (OR function) of N literals in which each variable or its complement appears exactly once. Examples for F

include $(A + B + \overline{C} + D)$, $(A + \overline{B} + \overline{C} + D)$, and $(\overline{A} + \overline{B} + C + \overline{D})$. A maxterm produces a 0 for exactly one combination of inputs. Just as we did with minterms, we can multiply a maxterm corresponding to each input combination for which a function produces 0 (each row in a truth table that produces a 0 output) to create an expression for the function. The resulting expression is in a **product-of-sums** (**POS**) form: a product of sums of literals. The carry out function that we used to produce Equation (3) has 10 input combinations that produce 0, so the expression formed in this way is unpleasantly long:

$$\begin{aligned} C \quad = \quad & (\overline{A_1} + \overline{A_0} + B_1 + B_0)(\overline{A_1} + A_0 + B_1 + \overline{B_0})(\overline{A_1} + A_0 + B_1 + B_0)(A_1 + \overline{A_0} + \overline{B_1} + B_0) \\ & (A_1 + \overline{A_0} + B_1 + \overline{B_0})(A_1 + \overline{A_0} + B_1 + B_0)(A_1 + A_0 + \overline{B_1} + \overline{B_0})(A_1 + A_0 + \overline{B_1} + B_0) \\ & (A_1 + A_0 + B_1 + \overline{B_0})(A_1 + A_0 + B_1 + B_0) \end{aligned}$$

However, the approach can be helpful with functions that produce mostly 1s. The literals in maxterms are complemented with respect to the literals used in minterms. For example, the maxterm $(\overline{A_1} + \overline{A_0} + B_1 + B_0)$ in the equation above produces a zero for input combination $A_1 = 1$, $A_0 = 1$, $B_1 = 0$, $B_0 = 0$.

An **implicant** G of a function F is defined to be a second function operating on the same variables for which the implication $G \rightarrow F$ is true. In terms of logic functions that produce 0s and 1s, *if G is an implicant of F, the input combinations for which G produces 1s are a subset of the input combinations for which F produces 1s.* Any minterm for which F produces a 1, for example, is an implicant of F.

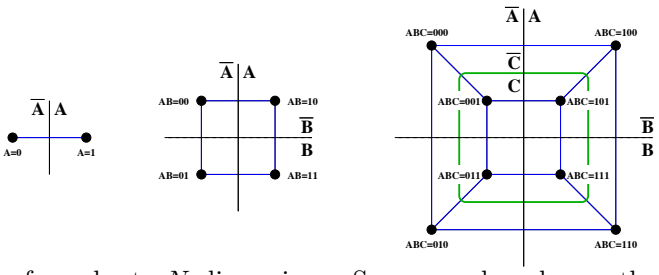
In the context of logic design, *the term implicant is used to refer to a single product of literals.* In other words, if we have a function $F(A, B, C, D)$, examples of possible implicants of F include AB, $B\overline{C}$, $ABC\overline{D}$, and $\overline{A}$. In contrast, although they may technically imply F, we typically do not call expressions such as $(A + B)$, $C(\overline{A} + D)$, nor $A\overline{B} + C$ implicants.

Let's say that we have expressed function F in sum-of-products form. All of the individual product terms in the expression are implicants of F. As a first step in simplification, we can ask: for each implicant, is it possible to remove any of the literals that make up the product? If we have an implicant G for which the answer is no, we call G a **prime implicant** of F. In other words, if one removes any of the literals from a prime implicant G of F, the resulting product is not an implicant of F.

Prime implicants are the main idea that we use to simplify logic expressions, both algebraically and with graphical tools (computer tools use algebra internally—by graphical here we mean drawings on paper).

Veitch Charts and Karnaugh Maps

Veitch's 1952 paper was the first to introduce the idea of using a graphical representation to simplify logic expressions. Earlier approaches were algebraic. A year later, Maurice Karnaugh published a paper showing a similar idea with a twist. The twist makes the use of **Karnaugh maps** to simplify expressions much easier than the use of Veitch charts. As a result, few engineers have heard of Veitch, but everyone who has ever taken a class on digital logic knows how to make use of a **K-map**.



Before we introduce the Karnaugh map, let's think about the structure of the domain of a logic function. Recall that a function's **domain** is the space on which the function is defined, that is, for which the function produces values. For a Boolean logic function on N variables, you can think of the domain as sequences of N bits, but you can also visualize the domain as an N-dimensional hypercube. An **N-dimensional hypercube** is the generalization of a cube to N dimensions. Some people only use the term hypercube when $N \geq 4$, since we have other names for the smaller values: a point for $N = 0$, a line segment for $N = 1$, a square for $N = 2$, and a cube for $N = 3$. The diagrams above and to the right illustrate the cases that are easily drawn on paper. The black dots represent specific input combinations, and the blue edges connect input combinations that differ in exactly one input value (one bit).

By viewing a function's domain in this way, we can make a connection between a product of literals and the structure of the domain. Let's use the 3-dimensional version as an example. We call the variables A, B, and C, and note that the cube has $2^3 = 8$ corners corresponding to the 2^3 possible combinations of A, B, and C. The simplest product of literals in this case is 1, which is the product of 0 literals. Obviously, the product 1 evaluates to 1 for any variable values. We can thus think of it as covering the entire domain of the function. In the case of our example, the product 1 covers the whole cube. In order for the product 1 to be an implicant of a function, the function itself must be the function 1.

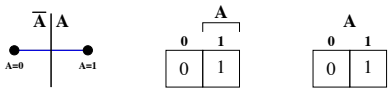
What about a product consisting of a single literal, such as A or $\overline{C}$? The dividing lines in the diagram illustrate the answer: any such product term evaluates to 1 on a face of the cube, which includes $2^2 = 4$ of the corners. If a function evaluates to 1 on any of the six faces of the cube, the corresponding product term (consisting of a single literal) is an implicant of the function.

Continuing with products of two literals, we see that any product of two literals, such as $A\overline{B}$ or $\overline{B}C$, corresponds to an edge of our 3-dimensional cube. The edge includes $2^1 = 2$ corners. And, if a function evaluates to 1 on any of the 12 edges of the cube, the corresponding product term (consisting of two literals) is an implicant of the function.

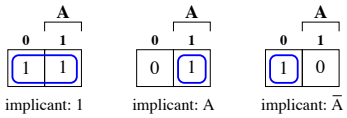
Finally, any product of three literals, such as $\overline{A}B\overline{C}$, corresponds to a corner of the cube. But for a function on three variables, these are just the minterms. As you know, if a function evaluates to 1 on any of the 8 corners of the cube, that minterm is an implicant of the function (we used this idea to construct the function to prove logical completeness).

How do these connections help us to simplify functions? If we're careful, we can map cubes onto paper in such a way that product terms (the possible implicants of the function) usually form contiguous groups of 1s, allowing us to spot them easily. Let's work upwards starting from one variable to see how this idea works. The end result is called a Karnaugh map.

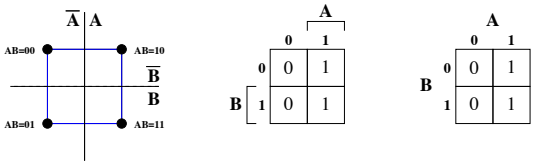
The first drawing shown to the right replicates our view of the 1-dimensional hypercube, corresponding to the domain of a function on one variable, in this case the variable A. To the right of the hypercube (line segment) are two variants of a Karnaugh map on one variable. The middle variant clearly indicates the column corresponding to the product A (the other column corresponds to $\overline{A}$). The right variant simply labels the column with values for A.



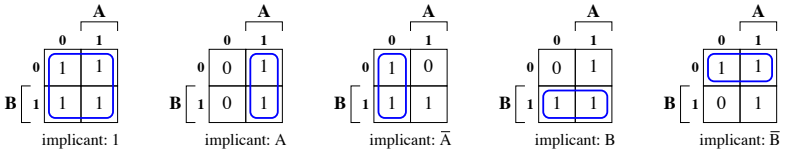
The three drawings shown to the right illustrate the three possible product terms on one variable. *The functions shown in these Karnaugh maps are arbitrary, except that we have chosen them such that each implicant shown is a prime implicant for the illustrated function.*



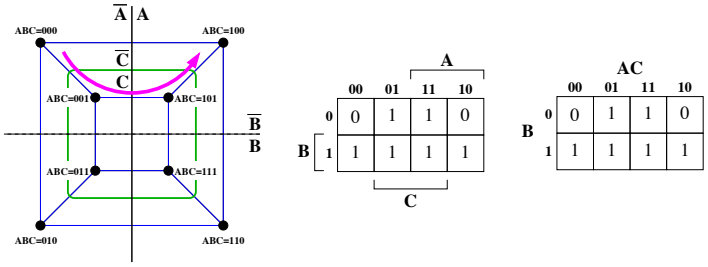
Let's now look at two-variable functions. We have replicated our drawing of the 2-dimensional hypercube (square) to the right along with two variants of Karnaugh maps on two variables. With only two variables (A and B), the extension is fairly straightforward, since we can use the second dimension of the paper (vertical) to express the second variable (B).



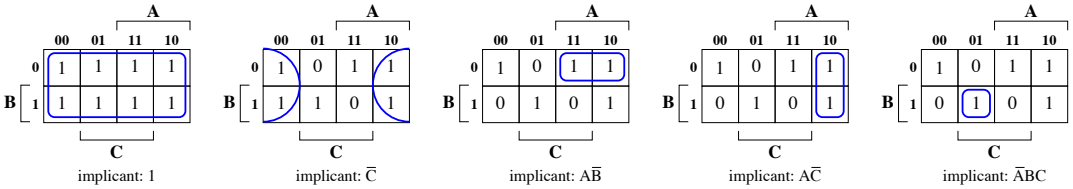
The number of possible products of literals grows rapidly with the number of variables. For two variables, nine are possible, as shown to the right. Notice that all implicants have two properties. First, they occupy contiguous regions of the grid. And, second, their height and width are always powers of two. These properties seem somewhat trivial at this stage, but they are the key to the utility of K-maps on more variables.



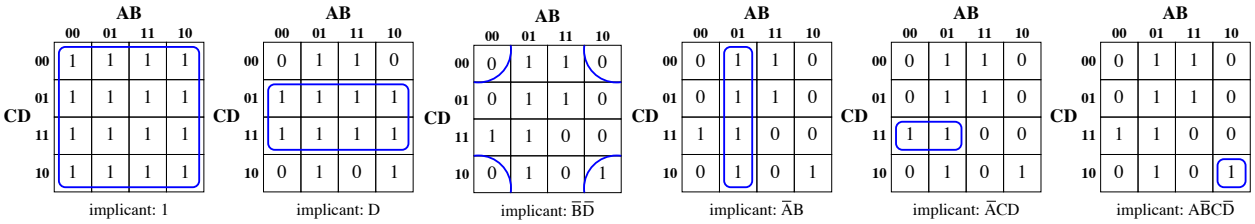
Three-variable functions are next. The cube diagram is again replicated to the right. But now we have a problem: how can we map four points (say, from the top half of the cube) into a line in such a way that any points connected by a blue line are adjacent in the K-map? The answer is that we cannot, but we can preserve most of the connections by choosing an order such as the one illustrated by the arrow. The result is called a Gray code. Two K-map variants again appear to the right of the cube. Look closely at the order of the two-variable combinations along the top, which allows us to have as many contiguous products of literals as possible. Any product of literals that contains $\overline{C}$ but not A nor $\overline{A}$ wraps around the edges of the K-map, so you should think of it as rolling up into a cylinder rather than a grid. Or you can think that we're unfolding the cube to fit the corners onto a sheet of paper, but the place that we split the cube should still be considered to be adjacent when looking for implicants. The use of a Gray code is the one difference between a K-map and a Veitch chart; Veitch used the base 2 order, which makes some implicants hard to spot.



With three variables, we have 27 possible products of literals. You may have noticed that the count scales as 3^N for N variables; can you explain why? We illustrate several product terms below. Note that we sometimes need to wrap around the end of the K-map, but that if we account for wrapping, the squares covered by all product terms are contiguous. Also notice that both the width and the height of all product terms are powers of two. *Any square or rectangle that meets these two constraints corresponds to a product term!* And any such square or rectangle that is filled with 1s is an implicant of the function in the K-map.



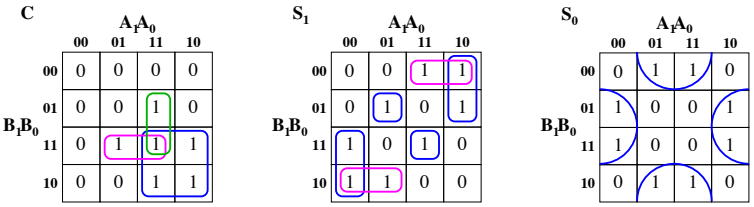
Let's keep going. With a function on four variables—A, B, C, and D—we can use a Gray code order on two of the variables in each dimension. Which variables go with which dimension in the grid really doesn't matter, so we'll assign AB to the horizontal dimension and CD to the vertical dimension. A few of the 81 possible product terms are illustrated at the top of the next page. Notice that while wrapping can now occur in both dimensions, we have exactly the same rule for finding implicants of the function: any square or rectangle (allowing for wrapping) that is filled with 1s and has both height and width equal to (possibly different) powers of two is an implicant of the function. *Furthermore, unless such a square or rectangle is part of a larger square or rectangle that meets these criteria, the corresponding implicant is a prime implicant of the function.*



Finding a simple expression for a function using a K-map then consists of solving the following problem: pick a minimal set of prime implicants such that every 1 produced by the function is covered by at least one prime implicant. The metric that you choose to minimize the set may vary in practice, but for simplicity, let's say that we minimize the number of prime implicants chosen.

Let's try a few! The table on the left below reproduces (from Notes Set 1.4) the truth table for addition of two 2-bit unsigned numbers, A_1A_0 and B_1B_0, to produce a sum S_1S_0 and a carry out C. K-maps for each output bit appear to the right. The colors are used only to make the different prime implicants easier to distinguish. The equations produced by summing these prime implicants appear below the K-maps.

inputs				outputs		
A_1	A_0	B_1	B_0	C	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0



$$C = A_1\ B_1 + A_1\ A_0\ B_0 + A_0\ B_1\ B_0$$

$$S_1 = A_1\ \overline{B_1}\ \overline{B_0} + A_1\ \overline{A_0}\ \overline{B_1} + \overline{A_1}\ \overline{A_0}\ B_1 + \overline{A_1}\ B_1\ \overline{B_0} + \overline{A_1}\ A_0\ \overline{B_1}\ B_0 + A_1\ A_0\ B_1\ B_0$$

$$S_0 = A_0\ \overline{B_0} + \overline{A_0}\ B_0$$

In theory, K-maps extend to an arbitrary number of variables. Certainly Gray codes can be extended. An **N-bit Gray code** is a sequence of N-bit patterns that includes all possible patterns such that any two adjacent patterns differ in only one bit. The code is actually a cycle: the first and last patterns also differ in only one bit. You can construct a Gray code recursively as follows: for an $(N+1)$-bit Gray code, write the sequence for an N-bit Gray code, then add a 0 in front of all patterns. After this sequence, append a second copy of the N-bit Gray code in reverse order, then put a 1 in front of all patterns in the second copy. The result is an $(N+1)$-bit Gray code. For example, the following are Gray codes:

1-bit 0, 1
2-bit 00, 01, 11, 10
3-bit 000, 001, 011, 010, 110, 111, 101, 100
4-bit 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

Unfortunately, some of the beneficial properties of K-maps do not extend beyond two variables in a dimension. *Once you have three variables in one dimension*, as is necessary if a function operates on five or more variables, *not all product terms are contiguous in the grid*. The terms still require a total number of rows and columns equal to a power of two, but they don't all need to be a contiguous group. Furthermore, *some contiguous groups of appropriate size do not correspond to product terms*. So you can still make use of K-maps if you have more variables, but their use is a little trickier.

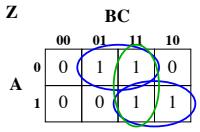
Canonical Forms

What if we want to compare two expressions to determine whether they represent the same logic function? Such a comparison is a test of **logical equivalence**, and is an important part of hardware design. Tools today provide help with this problem, but you should understand the problem.

You know that any given function can be expressed in many ways, and that two expressions that look quite different may in fact represent the same function (look back at Equations (1) to (3) for an example). But what if we rewrite the function using only prime implicants? Is the result unique? Unfortunately, no.

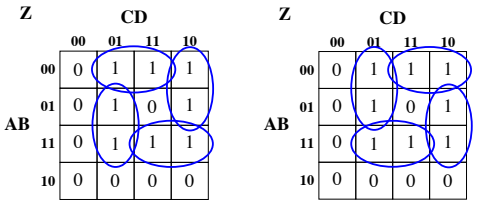
In general, *a sum of products is not unique (nor is a product of sums), even if the sum contains only prime implicants.*

For example, consensus terms may or may not be included in our expressions. (They are necessary for reliable design of certain types of systems, as you will learn in a later ECE class.) The green ellipse in the K-map to the right represents the consensus term BC.



$$\begin{aligned} Z &= A\,C + \overline{A}\,B + B\,C \\ Z &= A\,C + \overline{A}\,B \end{aligned}$$

Some functions allow several equivalent formulations as sums of prime implicants, even without consensus terms. The K-maps shown to the right, for example, illustrate how one function might be written in either of the following ways:

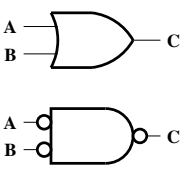


$$\begin{aligned} Z &= \overline{A}\,\overline{B}\,D + \overline{A}\,C\,\overline{D} + A\,B\,C + B\,\overline{C}\,D \\ Z &= \overline{A}\,\overline{B}\,C + B\,C\,\overline{D} + A\,B\,D + \overline{A}\,\overline{C}\,D \end{aligned}$$

When we need to compare two things (such as functions), we need to transform them into what in mathematics is known as a **canonical form**, which simply means a form that is defined so as to be unique for each thing of the given type. What can we use for logic functions? You already know two answers! The **canonical sum** of a function (sometimes called the **canonical SOP form**) is the sum of minterms. The **canonical product** of a function (sometimes called the **canonical POS form**) is the product of maxterms. These forms technically only meet the mathematical definition of canonical if we agree on an order for the min/maxterms, but that problem is solvable. However, as you already know, the forms are not particularly convenient to use. In practice, people and tools in the industry use more compact approaches when comparing functions, but those solutions are a subject for a later class (such as ECE 462).

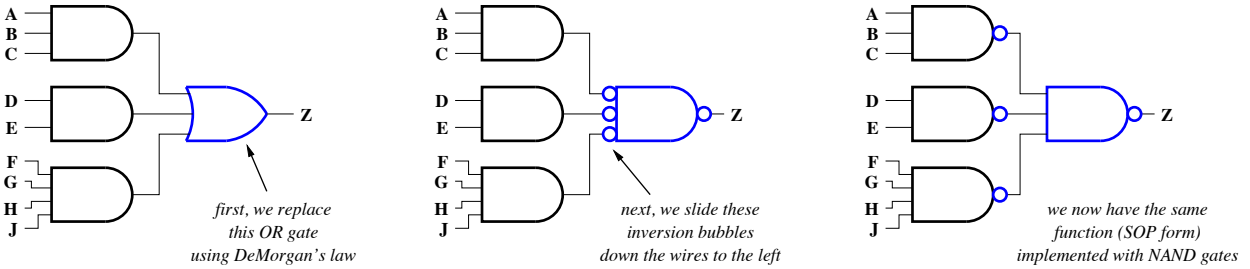
Two-Level Logic

Two-level logic is a popular way of expressing logic functions. The two levels refer simply to the number of functions through which an input passes to reach an output, and both the SOP and POS forms are examples of two-level logic. In this section, we illustrate one of the reasons for this popularity and show you how to graphically manipulate expressions, which can sometimes help when trying to understand gate diagrams.

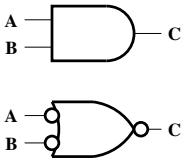


We begin with one of DeMorgan's laws, which we can illustrate both algebraically and graphically: $C \;=\; B + A \;=\; \overline{\overline{B}\;\overline{A}}$

Let's say that we have a function expressed in SOP form, such as $Z = ABC + DE + FGHJ$. The diagram on the left below shows the function constructed from three AND gates and an OR gate. Using DeMorgan's law, we can replace the OR gate with a NAND with inverted inputs. But the bubbles that correspond to inversion do not need to sit at the input to the gate. We can invert at any point along the wire, so we slide each bubble down the wire to the output of the first column of AND gates. *Be careful: if the wire splits, which does not happen in our example, you have to replicate the inverter onto the other output paths as you slide past the split point!* The end result is shown on the right: we have not changed the function, but now we use only NAND gates. Since CMOS technology only supports NAND and NOR directly, using two-level logic makes it simple to map our expression into CMOS gates.



You may want to make use of DeMorgan's other law, illustrated graphically to the right, to perform the same transformation on a POS expression. What do you get?



Multi-Metric Optimization

As engineers, almost every real problem that you encounter will admit multiple metrics for evaluating possible designs. Becoming a good engineer thus requires not only that you be able to solve problems creatively so as to improve the quality of your solutions, but also that you are aware of how people might evaluate those solutions and are able both to identify the most important metrics and to balance your design effectively according to them. In this section, we introduce some general ideas and methods that may be of use to you in this regard. *We will not test you on the concepts in this section.*

When you start thinking about a new problem, your first step should be to think carefully about metrics of possible interest. Some important metrics may not be easy to quantify. For example, compatibility of a design with other products already owned by a customer has frequently defined the success or failure of computer hardware and software solutions. But how can you compute the compability of your approach as a number?

Humans—including engineers—are not good at comparing multiple metrics simultaneously. Thus, once you have a set of metrics that you feel is complete, your next step is to get rid of as many as you can. Towards this end, you may identify metrics that have no practical impact in current technology, set threshold values for other metrics to simplify reasoning about them, eliminate redundant metrics, calculate linear sums to reduce the count of metrics, and, finally, make use of the notion of Pareto optimality. All of these ideas are described in the rest of this section.

Let's start by considering metrics that we can quantify as real numbers. For a given metric, we can divide possible measurement values into three ranges. In the first range, all measurement values are equivalently useful. In the second range, possible values are ordered and interesting with respect to one another. Values in the third range are all impossible to use in practice. Using power consumption as our example, the first range corresponds to systems in which when a processor's power consumption in a digital system is extremely low relative to the power consumption of the system. For example, the processor in a computer might use less than 1% of the total used by the system including the disk drive, the monitor, the power supply, and so forth. One power consumption value in this range is just as good as any another, and no one cares about the power consumption of the processor in such cases. In the second range, power consumption of the processor

makes a difference. Cell phones use most of their energy in radio operation, for example, but if you own a phone with a powerful processor, you may have noticed that you can turn off the phone and drain the battery fairly quickly by playing a game. Designing a processor that uses half as much power lengthens the battery life in such cases. Finally, the third region of power consumption measurements is impossible: if you use so much power, your chip will overheat or even burst into flames. Consumers get unhappy when such things happen.

As a first step, you can remove any metrics for which all solutions are effectively equivalent. Until a little less than a decade ago, for example, the power consumption of a desktop processor actually was in the first range that we discussed. Power was simply not a concern to engineers: all designs of interest consumed so little power that no one cared. Unfortunately, at that point, power consumption jumped into the third range rather quickly. Processors hit a wall, and products had to be cancelled. Given that the time spent designing a processor has historically been about five years, a lot of engineering effort was wasted because people had not thought carefully enough about power (since it had never mattered in the past). Today, power is an important metric that engineers must take into account in their designs.

However, in some areas, such as desktop and high-end server processors, other metrics (such as performance) may be so important that we always want to operate at the edge of the interesting range. In such cases, we might choose to treat a metric such as power consumption as a **threshold**: stay below 150 Watts for a desktop processor, for example. One still has to make a coordinated effort to ensure that the system as a whole does not exceed the threshold, but reasoning about threshold values, a form of constraint, is easier than trying to think about multiple metrics at once.

Some metrics may only allow discrete quantification. For example, one could choose to define compatibility with previous processor generations as binary: either an existing piece of software (or operating system) runs out of the box on your new processor, or it does not. If you want people who own that software to make use of your new processor, you must ensure that the value of this binary metric is 1, which can also be viewed as a threshold.

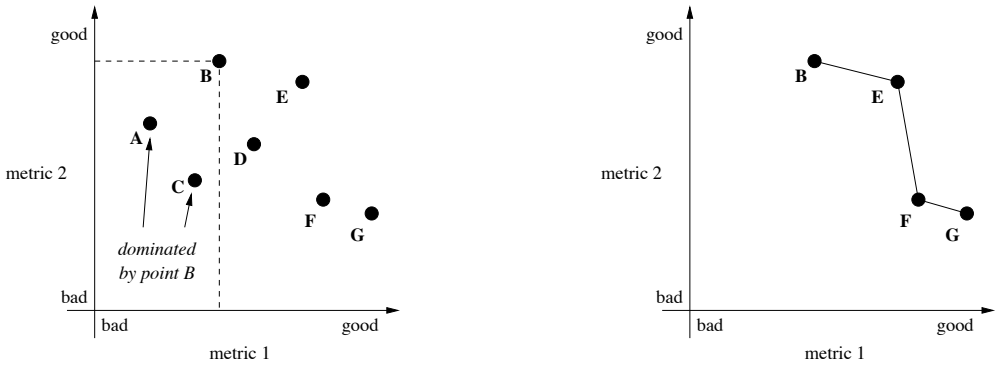
In some cases, two metrics may be strongly **correlated**, meaning that a design that is good for one of the metrics is frequently good for the other metric as well. Chip area and cost, for example, are technically distinct ways to measure a digital design, but we rarely consider them separately. A design that requires a larger chip is probably more complex, and thus takes more engineering time to get right (engineering time costs money). Each silicon wafer costs money to fabricate, and fewer copies of a large design fit on one wafer, so large chips mean more fabrication cost. Physical defects in silicon can cause some chips not to work. A large chip uses more silicon than a small one, and is thus more likely to suffer from defects (and not work). Cost thus goes up again for large chips relative to small ones. Finally, large chips usually require more careful testing to ensure that they work properly (even ignoring the cost of getting the design right, we have to test for the presence of defects), which adds still more cost for a larger chip. All of these factors tend to correlate chip area and chip cost, to the point that most engineers do not consider both metrics.

After you have tried to reduce your set of metrics as much as possible, or simplified them by turning them into thresholds, you should consider turning the last few metrics into a weighted linear sum. All remaining metrics must be quantifiable in this case. For example, if you are left with three metrics for which a given design has values A, B, and C, you might reduce these to one metric by calculating $D = w_A A + w_B B + w_C C$. What are the w values? They are weights for the three metrics. Their values represent the relative importance of the three metrics to the overall evaluation. Here we've assumed that larger values of A, B, and C are either all good or all bad. If you have metrics with different senses, use the reciprocal values. For example, if a large value of A is good, a small value of $1/A$ is also good.

The difficulty with linearizing metrics is that not everyone agrees on the weights. Is using less power more important than having a cheaper chip? The answer may depend on many factors.

When you are left with several metrics of interest, you can use the idea of Pareto optimality to identify interesting designs. Let's say that you have two metrics. If a design D_1 is better than a second design D_2 for both metrics, we say that D_1 **dominates** D_2. A design D is then said to be **Pareto optimal** if no other design dominates D. Consider the figure on the left below, which illustrates seven possible designs measured

with two metrics. The design corresponding to point B dominates the designs corresponding to points A and C, so neither of the latter designs is Pareto optimal. No other point in the figure dominates B, however, so that design is Pareto optimal. If we remove all points that do not represent Pareto optimal designs, and instead include only those designs that are Pareto optimal, we obtain the version shown on the right. These are points in a two-dimensional space, not a line, but we can imagine a line going through the points, as illustrated in the figure: the points that make up the line are called a **Pareto curve**, or, if you have more than two metrics, a **Pareto surface**.



As an example of the use of Pareto optimality, consider the figure to the right, which is copied with permission from Neal Crago's Ph.D. dissertation (UIUC ECE, 2012). The figure compares hundreds of thousands of possible designs based on a handful of different core approaches for implementing a processor. The axes in the graph are two metrics of interest. The horizontal axis measures the average performance of a design when executing a set of benchmark applications, normalized to a baseline processor design. The vertical axis measures the energy consumed by a design when executing the same benchmarks, normalized again to the energy consumed by a baseline design. The six sets of points in the graph represent alternative design techniques for the processor, most of which are in commercial use today. The points shown for each set are the subset of many thousands of possible variants that are Pareto optimal. In this case, more performance and less energy consumption are the good directions, so any point in a set for which another point is both further to the right and further down is not shown in the graph. The black line represents an absolute power consumption of 150 Watts, which is a nominal threshold for a desktop environment. Designs above and to the right of that line are not as interesting for desktop use. The **design-space exploration** that Neal reported in this figure was of course done by many computers using many hours of computation, but he had to design the process by which the computers calculated each of the points.