Module 4: Logic Minimization

Revision: August 30, 2007

Produced in cooperation with DIGILENT®
www.digilentinc.com

Overview

The requirements for new logic circuit designs are often expressed in some loose, informal manner. For an informal behavioral description to result in an efficient, well designed circuit that meets the stated requirements, appropriate engineering design methods must be developed. As an example, the following statement might serve as the starting point for a new design: "a warning light, running from a circuit powered by a back-up battery, should be illuminated if the main power is disconnected, or if main power is OK but the reserve power source falls below 48V, or if the current exceeds 2 amps, or if the current exceeds 1 amp and the reserve falls below 48V". An initial engineering task is to state this requirement more concisely: WL <= (not P) or (P and not R) or C2 or (C1 and R). This equation removes all ambiguity from the worded description, and it can also be directly implemented as a logic circuit using two 2-input AND gates and one 4-input OR gate. But a simpler 4-input OR circuit that behaves identically under all input conditions could also be constructed (WL <= not R or not R or C1 or C2). Clearly, it would be faster, easier, less costly and less error prone to build the simpler circuit. Another engineering task involves analyzing the requirements of a logic design, with the goal of finding a minimal expression of any logic relationship.

Before beginning this module, you should…

- Be familiar with reading and constructing basic logic circuits
- Understand logic equations, and how to implement a logic circuit from a logic equation
- Know how to operate Windows computers and Windows programs

After completing this module, you should…

- Be able to minimize any given logic system
- Understand CAD tool use in basic circuit design
- Be able to implement any given combinational circuit using the Xilinx ISE schematic editor
- Be able to simulate any logic circuit
- Be able to examine the output of a logic simulator to verify whether a given circuit has been designed correctly

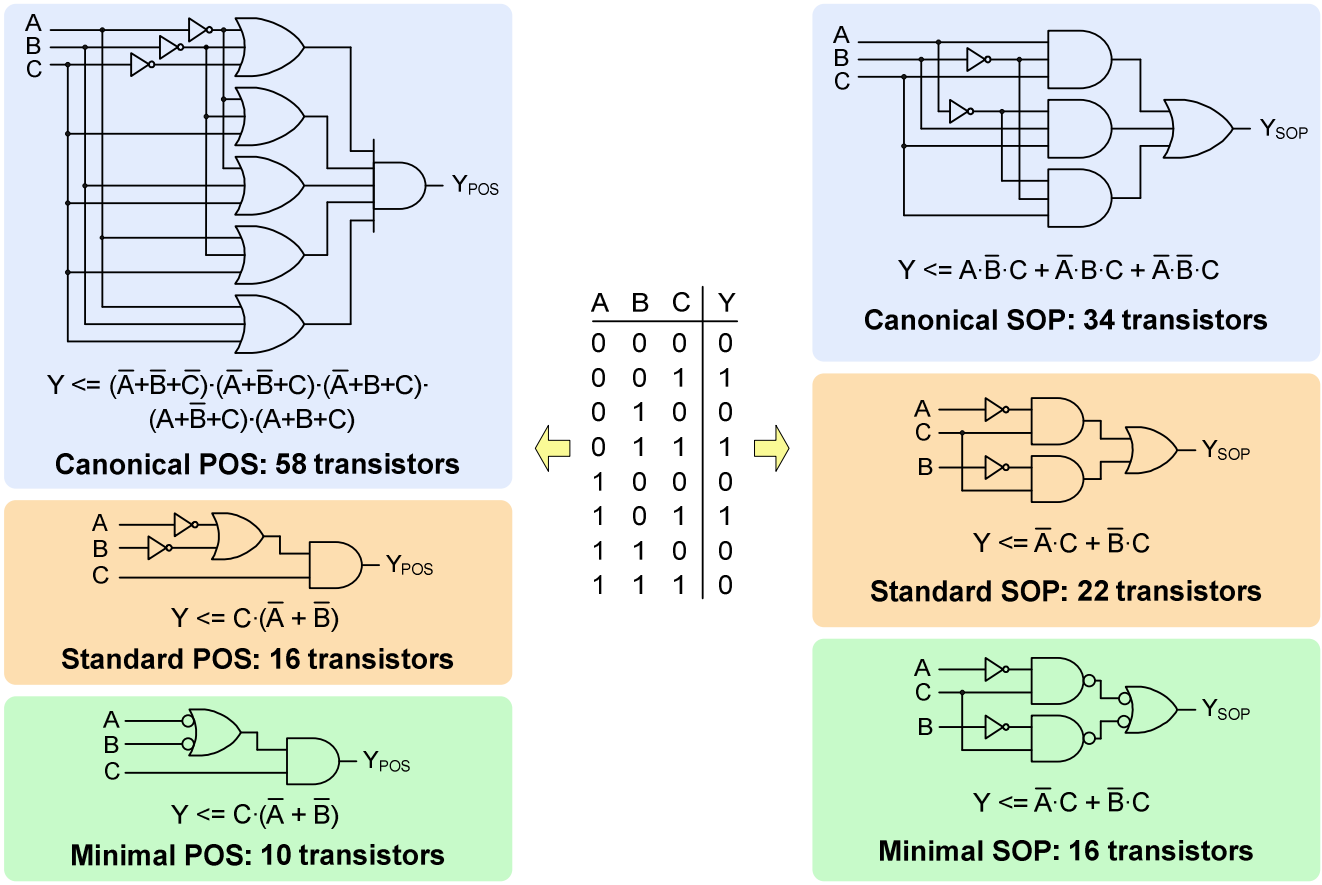
This module requires:

- A Windows PC
- The Xilinx ISE/WebPack software
- A Digilent circuit board

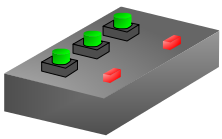
Background

A digital logic circuit consists of a collection of logic gates, the input signals that drive them, and the output signals they produce. The behavioral requirements of a logic circuit are best expressed through truth tables or logic equations, and any design problem that can be addressed with a logic circuit can be expressed in one of these forms. Both of these formalisms define the behavior of a logic circuit – how inputs are combined to drive outputs – but they do not specify how to build a circuit that meets these requirements. One goal of this module is to define engineering design methods that can produce optimum circuits based on behavioral descriptions.

Only one truth table exists for any particular logic relationship, but many different logic equations and logic circuits can be found to describe and implement the same relationship. Different (but equivalent) logic equations and circuits exist for a given truth table because it is always possible to add unneeded, redundant logic gates to a circuit without changing its logical output. Take for example the logic system introduced in the previous module (reproduced in the figure below). The system's behavior is defined by the truth table in the center of the figure, and it can be implemented by any of the logic equations and related logic circuits shown.

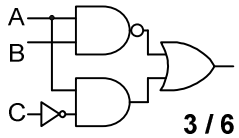


All six circuits shown are equivalent, meaning they share the same truth table, but they have different physical structures. Image a black box with three input buttons, two LEDs, and two independent circuits driving the LEDs. Any of the six circuits shown above could drive either LED, and an observer pressing buttons in any combination could not identify which circuit drove which LED. For every possible combination of button presses, the LEDs would be illuminated in exactly the same manner regardless of which circuit was used. If we have a choice of logic circuits for any given logic relationship, it follows we should first define which circuit is the best, and develop a method to ensure we find it.

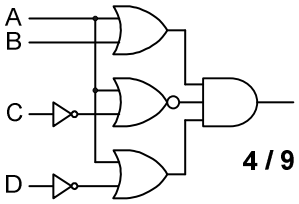


The circuits in the blue boxes above are said to be "canonical" because they contain all required minterms and maxterms. Canonical circuits typically use resources inefficiently, but they are conceptually simple. Below the canonical circuits are standard POS and SOP circuits – these two circuits behave identically to the canonical circuits, but they use fewer resources. Clearly, it would be less wasteful of resources to build the standard POS or SOP circuits. And further, replacing logic gates in the standard circuits with transistor-minimum gate equivalents (by taking advantage of NAND/NOR logic) results in the minimized POS and SOP circuits shown in the green boxes.

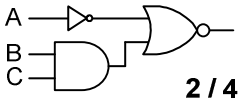
As engineers, one of our primary goals is to implement circuits efficiently. The most efficient circuit can use the fewest number of transistors, or it can operate at the highest speeds, or it can use the least amount of power. Often, these three measures of efficiency cannot all be optimized at the same time, and designers must trade-off circuit size for speed of operation, or speed for power, or power for size, etc. Here, we will define the most efficient circuit as the one that uses the minimum number of transistors, and leave speed and power considerations for later consideration. Because we have chosen the minimum-transistor measure of efficiency, we will look for "minimum" circuits. The best method of determining which of several circuits is the minimum is to count the needed transistors. For now, we will use a simpler method – the minimal circuit will be defined as the one that uses the fewest number of logic gates (or, if two forms use the same number of gates, then the one that uses the fewest number of total inputs to all gates will be considered the simplest). The following examples show circuits with the gate/input number shown below. Inverters are not included in the gate or input count, because often, they are absorbed into the logic gates themselves.



3 / 6



4 / 9



2 / 4

A minimal logic equation for a given logic system can be obtained by eliminating all non-essential or redundant inputs. Any input that can be removed from the equation without changing the input/output relationship is redundant. To find minimal equations, all redundant inputs must be identified and removed. In the truth table above, note the SOP terms generated by rows 1 and 3. The A input is '0' in both rows, and the C input is '1' in both rows, but the B input is '0' in one row and '1' in the other. Thus, for these two rows, the output is a '1' whether B is a '0' or '1' and B is therefore redundant.

The goal in "minimizing" logic systems is to find the simplest form by identifying and removing all redundant inputs. For a logic function of N inputs, there are 2^{2^N} logic functions, and for each of these functions, there exists a minimum SOP form and a minimum POS form. The SOP form may be more minimal than the POS form, or the POS form may be more minimal, or they may be equivalent (i.e., they may both require the same number of logic gates and inputs). In general, it is difficult to identify the minimum form by simply staring at a truth table. Several methods have evolved to assist with the minimization process, including the application of Boolean algebra, the use of logic graphs, and the use of searching algorithms. Although any of these methods can be employed using pen and paper, it is far easier (and more productive) to implement searching algorithms on a computer.

Boolean Algebra

Boolean algebra is perhaps the oldest method used to minimize logic equations. It provides a formal algebraic system that can be used to manipulate logic equations in an attempt to find more minimal equations. It is a proper algebraic system, with three set elements {'0', '1', and 'A'} (where 'A' is any variable that can assume the values '0' or '1'), two binary operations (***and*** or *intersection*, ***or*** or *union*), and one unary operation (***inversion*** or *complementation*). Operations between sets are closed under the three operations. The basic laws governing and, or, and inversion operations are easily derived from the logic truth tables for those operations. The associative, commutative, and distributive laws can be directly demonstrated using truth tables. Only the distributive law truth table is shown in the truth table below, with colors used to highlight the columns that show the equivalency of both sides of the distributive law equations. Truth tables to demonstrate the simpler associative and commutative laws are not shown, but they can be easily derived.

AND operations		**OR operations**		**INV operations**	
Truth table	Laws	Truth table	Laws	Truth table	Laws
0 · 0 = 0	A · 0 = 0	0 + 0 = 0	A + 0 = A	0' = 1	A'' = A
1 · 0 = 0	A · 1 = A	1 + 0 = 1	A + 1 = 1	1' = 0	
0 · 1 = 0	A · A = A	0 + 1 = 1	A + A = A		
1 · 1 = 1	A · A' = 0	1 + 1 = 1	A + A' = 1		

Associative Laws	**Commutative Laws**	**Distributive Laws**
(A·B)·C = A·(B·C) = A·B·C	A·B·C = B·A·C = ...	A·(B+C) = (A·B) + (A·C)
(A+B)+C = A+(B+C) = A+B+C	A+B+C = B+C+A = ...	A+(B·C) = (A+B) · (A+C)

Truth tables to verify distributive laws

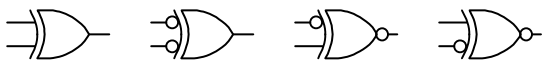
A	B	C	A+B	B+C	A+C	A·B	B·C	A·C	A·(B+C)	(A·B)+(A·C)	A+(B·C)	(A+B)·(A+C)
0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**
0	0	1	0	1	1	0	0	0	**0**	**0**	**0**	**0**
0	1	0	1	1	0	0	0	0	**0**	**0**	**0**	**0**
0	1	1	1	1	1	0	1	0	**0**	**0**	**1**	**1**
1	0	0	1	0	1	0	0	0	**0**	**0**	**1**	**1**
1	0	1	1	1	1	0	0	1	**1**	**1**	**1**	**1**
1	1	0	1	1	1	1	0	0	**1**	**1**	**1**	**1**
1	1	1	1	1	1	1	1	1	**1**	**1**	**1**	**1**

AND'ing operations take precedence over OR'ing operations. Parenthesis can be used to eliminate any possible confusion. Thus, the following two sets of equations show equivalent logic equations.

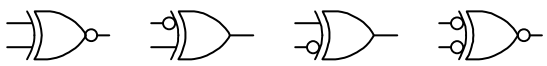
$$A \cdot B + C = (A \cdot B) + C \qquad A + B \cdot C = A + (B \cdot C)$$

DeMorgan's Law provides a formal algebraic statement for the property observed in defining the conjugate gate symbols: the same logic circuit can be interpreted as implementing either an AND or an OR function, depending how the input and output voltage levels are interpreted. DeMorgan's law, which is applicable to logic systems with any number of inputs, states

(A·B)' = A' + B' (nand form) and
(A+B)' = A'·B' (nor form).



XOR Conjugates



XNOR Conjugates

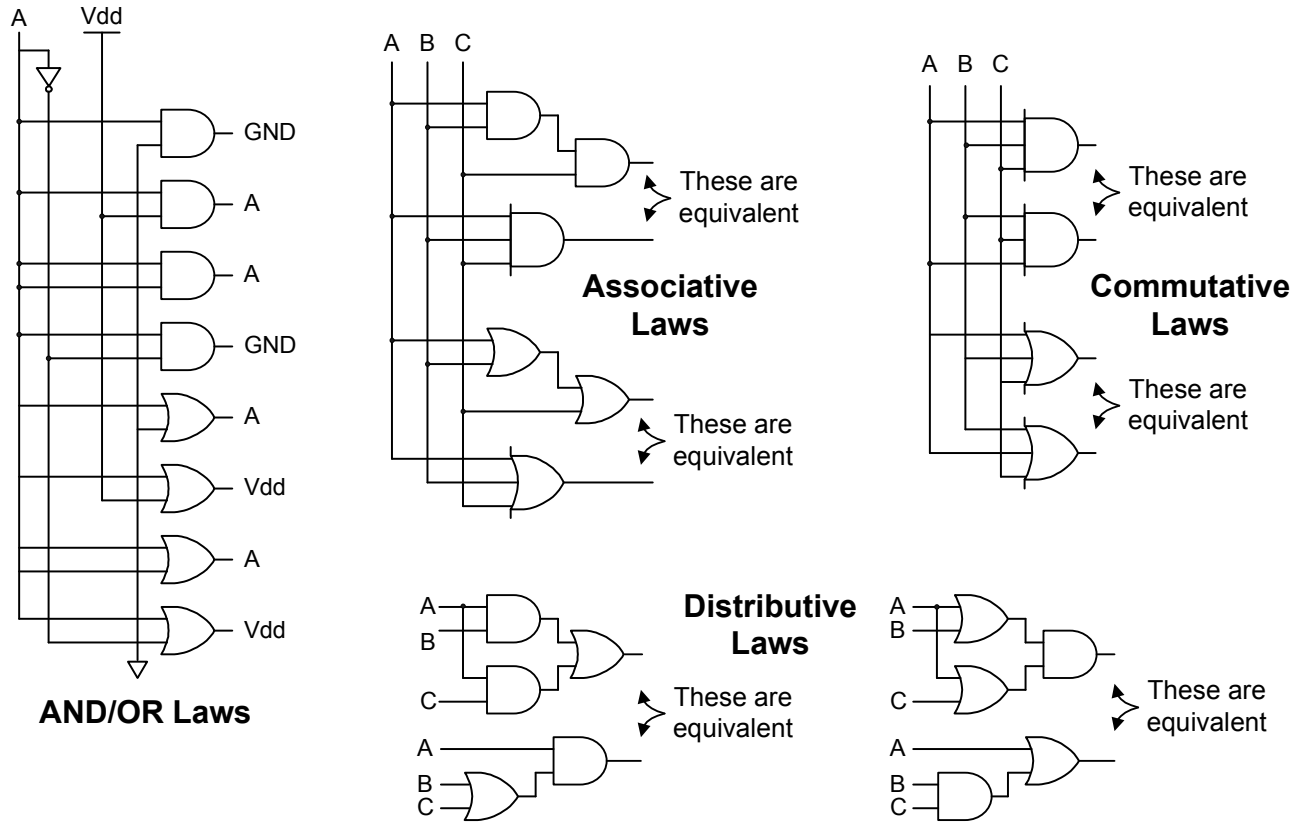
The laws of Boolean algebra generally hold for XOR functions as well, except that DeMorgan's law takes a different form. Recall from the pervious module that the XOR function output is asserted whenever an odd

number of inputs are asserted, and that the XNOR function output is asserted whenever an even number of inputs are asserted. Thus, inverting a single input to an XOR function, or inverting its output, yields the XNOR function. Likewise, inverting a single input to an XNOR function, or inverting its output, yields the XOR function. Inverting an input together with the output, or inverting two inputs, changes an XOR function to XNOR, and vice-versa. These observations lead to a version of DeMorgan's Laws that hold for XOR functions of any number of inputs:

F = A xnor B xnor C ⇔ F <= (A ⊕ B ⊕ C)' ⇔ F <= A' ⊕ B ⊕ C ⇔ F <= (A' ⊕ B' ⊕ C)' etc.
F = A xor B xor C ⇔ F <= A ⊕ B ⊕ C ⇔ F <= A' ⊕ B' ⊕ C ⇔ F <= (A ⊕ B' ⊕ C)' etc.

Note that a single input inversion can be moved to any other signal in a multi-input XOR circuit without changing the logical result. Note also that any signal inversion can be replaced with a non-inverted signal and an XNOR function. These properties will be useful in later work.

The circuits below also serve illustrate the laws of Boolean Algebra.



The following examples illustrate the use of Boolean Algebra to find simpler logic equations.

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

$F = A \cdot B \cdot (C + \overline{C}) + \overline{A} \cdot B \cdot (C+1)$	Factoring
$F = A \cdot B \cdot (1) + \overline{A} \cdot B \cdot (1)$	OR law
$F = A \cdot B + \overline{A} \cdot B$	AND law
$F = B \cdot (A + \overline{A})$	Factoring
$F = B \cdot (1)$	OR law
$F = B$	AND law

$F = (A+B+C) \cdot (A+B+\overline{C}) \cdot (A+\overline{C})$

$F = (A+B+C) \cdot (A+\overline{C}) \cdot (B+1)$	Factoring
$F = (A+B+C) \cdot (A+\overline{C}) \cdot (1)$	OR law
$F = (A+B+C) \cdot (A+\overline{C})$	AND law
$F = A + ((B+C) \cdot (\overline{C}))$	Factoring
$F = A + (B \cdot \overline{C} + C \cdot \overline{C})$	Distributive
$F = A + (B \cdot \overline{C} + 0)$	AND law
$F = A + (B \cdot \overline{C})$	OR law

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

$F = (\overline{A} + \overline{B} + \overline{C}) + \overline{A} \cdot B \cdot C + (\overline{A} + \overline{C})$	DeMorgan's
$F = \overline{A} + \overline{A} + (\overline{A} \cdot B \cdot C) + \overline{B} + \overline{C} + \overline{C}$	Commutative
$F = \overline{A} \cdot (1+1+B \cdot C) + \overline{B} + \overline{C}$	Factoring
$F = \overline{A} \cdot (1) + \overline{B} + \overline{C}$	OR law
$F = \overline{A} + \overline{B} + \overline{C}$	AND law

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

$F = \overline{A} \cdot B + A \cdot \overline{B} + \overline{A} \cdot \overline{B} + A \cdot B$	XOR expansion
$F = \overline{A} \cdot B + \overline{A} \cdot \overline{B} + A \cdot B + A \cdot \overline{B}$	Commutative
$F = \overline{A} \cdot (B + \overline{B}) + A \cdot (B + \overline{B})$	Factoring
$F = \overline{A} \cdot (1) + A \cdot (1)$	OR law
$F = \overline{A} + A$	AND law
$F = 1$	

$F = \overline{(A \oplus B)} + A \cdot B \cdot C + \overline{A \cdot B}$

$F = \overline{A} \cdot \overline{B} + A \cdot B + A \cdot B \cdot C + (\overline{A} + \overline{B})$	DeMorgan's
$F = \overline{A} \cdot \overline{B} + \overline{A} + \overline{B} + A \cdot B + A \cdot B \cdot C$	Commutative
$F = \overline{A} \cdot (B+1) + \overline{B} + A \cdot B \cdot (1+C)$	Factoring
$F = \overline{A} + \overline{B} + A \cdot B$	OR law
$F = \overline{A} + (\overline{B} + A) \cdot (\overline{B} + B)$	Factoring
$F = \overline{A} + (\overline{B} + A) \cdot (1)$	OR law
$F = \overline{A} + \overline{B} + A$	AND law
$F = 1$	OR law

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

$F = (\overline{\overline{A}} \cdot \overline{\overline{B}}) + (\overline{A} \cdot \overline{B}) + (\overline{A} \cdot B)$	DeMorgan's
$F = A \cdot B + \overline{A} \cdot \overline{B} + \overline{A} \cdot B$	NOT law
$F = A \cdot B + \overline{A} \cdot (\overline{B} + B)$	Factoring
$F = A \cdot B + \overline{A} \cdot (1)$	OR law
$F = A \cdot B + \overline{A}$	AND law
$F = (A + \overline{A}) \cdot (B + \overline{A})$	Factoring
$F = (1) \cdot (B + \overline{A})$	OR law
$F = \overline{A} + B$	AND/Commutative

$F = A + \overline{A} \cdot B$	$= A + B$
$F = (A + \overline{A}) \cdot (A + B)$	Factoring
$F = (1) \cdot (A + B)$	OR law
$F = A + B$	AND law

$F = A \cdot (\overline{A} + B)$	$= A \cdot B$
$F = (A \cdot \overline{A}) + (A \cdot B)$	Distributive
$F = (0) + (A \cdot B)$	AND law
$F = A \cdot B$	OR law

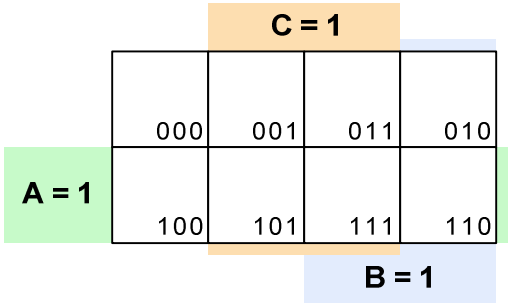
$F = A \cdot \overline{B} + \overline{B} \cdot C + \overline{A} \cdot C$	$= A \cdot \overline{B} + A \cdot \overline{C}$
$F = A \cdot \overline{B} + \overline{B} \cdot C \cdot 1 + \overline{A} \cdot C$	AND law
$F = A \cdot \overline{B} + \overline{B} \cdot C \cdot (A + \overline{A}) + \overline{A} \cdot C$	OR law
$F = A \cdot \overline{B} + A \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot C$	Distributive
$F = A \cdot \overline{B} \cdot (1+C) + \overline{A} \cdot C \cdot (\overline{B} + 1)$	Factoring
$F = A \cdot \overline{B} \cdot (1) + A \cdot \overline{C} \cdot (1)$	OR law
$F = A \cdot \overline{B} + A \cdot \overline{C}$	AND law

The last two examples on the left (with the blue boxes) shows relationships that are sometimes called the "absorptive" laws, and the example on the right (with the green box) is often called the "consensus" law. The so-called absorptive laws are easily demonstrated with other laws, so it is not necessary or even convenient to use these relationships as laws – particularly because different forms of equations can make it difficult to identify when the law might apply. The consensus law is also easily derived, if the "trick" of AND'ing a '1' into the equation, and then expanding that AND into an OR relationship is used (this trick is perfectly acceptable, if not entirely obvious).

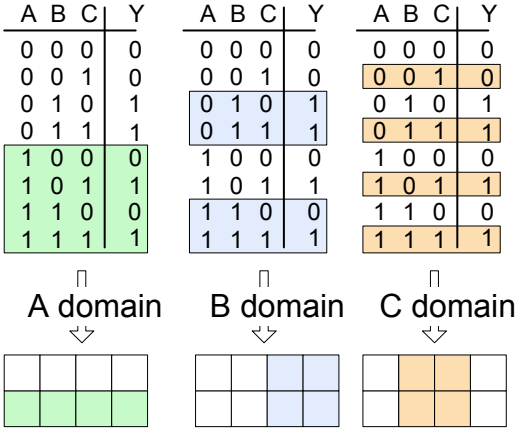
Logic Graphs

Truth tables are not very useful for minimizing logic systems, and Boolean algebra has limited utility. Logic graphs offer the easiest and most useful pen-and-paper method of minimizing a logic system. A logic graph and truth table contain identical information, but patterns that indicate redundant inputs can be readily identified in a logic graph. A logic graph is a two (or even three) dimensional construct that contains exactly the same information that a truth table does, but arranged in an array structure so that all logic domains are contiguous, and logic relationships are therefore easy to identify. Information in a truth table can easily be recast into a logic graph. The figure below shows how a three-input truth table is mapped to an 8-cell logic graph; the numbers in the logic graph cells are the numbers in the truth table rows.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



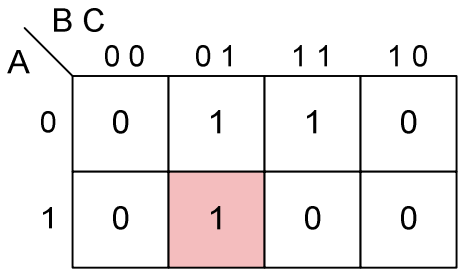
A 1-to-1 correspondence exists between the cells in the logic graph and the rows in a truth table, and that the cell numbers have been arranged so that each logic variable domain is represented by a group of four connected cells (the A domain is a row of four cells, and the B and C domains are squares of four cells). This particular arrangement of cells in the logic graph isn't the only one possible, but it has the useful property of having each domain overlap the others in exactly two cells. As can be seen in the figure, the logic domains are contiguous in the logic graph, but they are not contiguous in the truth table. It is the contiguous logic domains in the logic graphs that make them so useful.



Logic graphs are typically shown with variable names near the graph borders, and 1's and 0's near cell rows and columns to indicate the value of the variables for the rows and columns. The logic graph below shows a typical appearance. Note that the variable values on the logic graph edges can be read from left to right to find the truth table row that corresponds to a given cell. For example, the A = 1, B = 0, C= 1 row in the truth table below is shaded, and that row corresponds to the shaded cell in the logic graph.

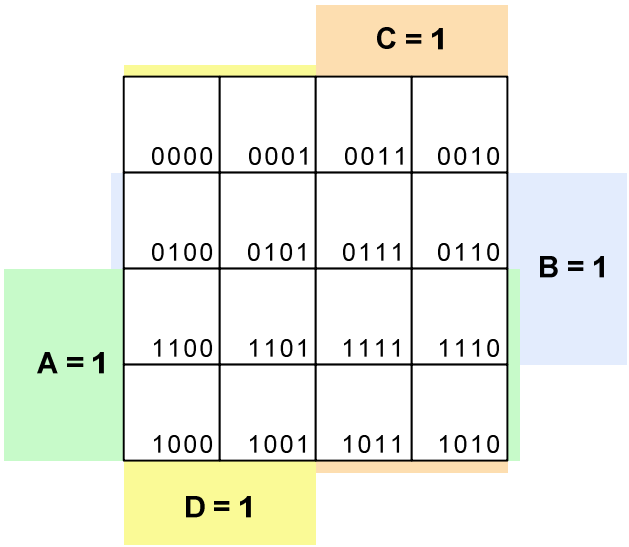
The information in the output column of the truth table below has been transferred row-for-cell into the cells of the logic graph, and so the truth table and logic graph contain identical information. In the logic graph, 1's that appear adjacent to one another (either vertically or horizontally) are said to be "logically adjacent", and these adjacencies represent opportunities to find and eliminate redundant inputs. Logic graphs used in this manner are called Karnaugh Maps (or just K-Maps) after their inventor.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



The figure below shows a four-input truth table mapped to a 16-cell K-map.

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

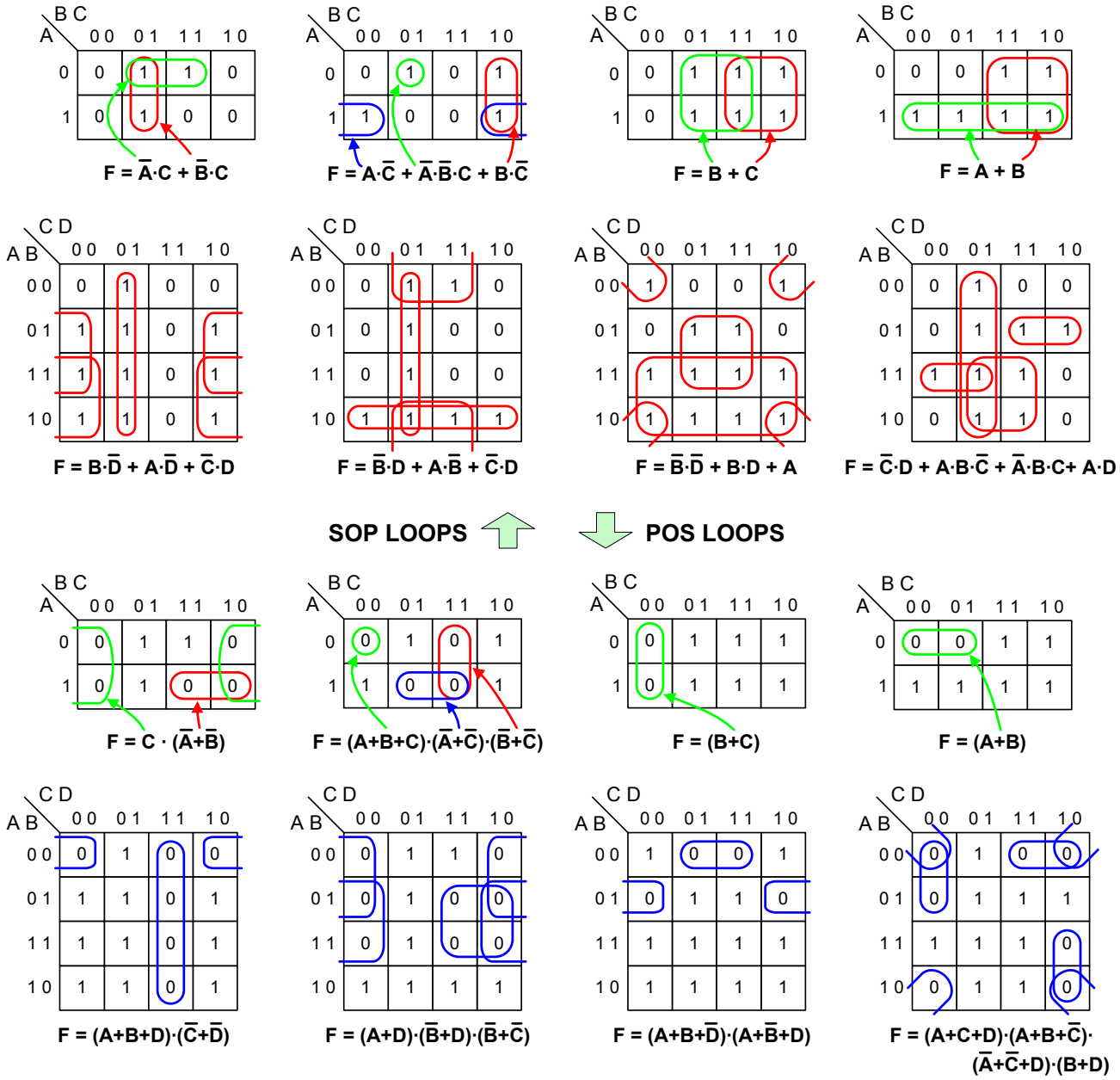


A B \ C D	00	01	11	10
00	0	1	1	0
01	0	1	0	0
11	0	1	0	0
10	1	1	1	1

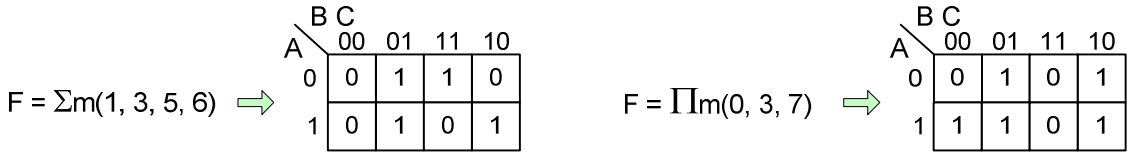
The key to using K-maps to find and eliminate redundant inputs from a logic system is to identify "groups" of 1's for SOP equations or groups of 0's for POS equations. A valid group must be a "power of 2" size (meaning that only groups of 1, 2, 4, 8, or 16 are allowed), and it must be a square or rectangle, but not a diagonal, dogleg, or other irregular shape. Each '1' in a SOP K-map must participate in at least one group, and each '1' must be in the largest possible group (and likewise for 0's in POS maps). The requirement that all 1's (or 0's) are grouped in the largest possible group may mean that some 1's (or 0's) are part of several groups. In practice, loops are drawn on a K-map to encircle the 1's (or 0's) in a given group. Once all 1's (or 0's) in a map have been grouped in the largest possible loops, the grouping process is complete and a logic equation can be read directly from the K-map. If the procedure is performed correctly, a minimal logic equation is guaranteed.

SOP logic equations are read from a K-map by writing product terms defined by each loop, and then OR'ing the product terms together. Likewise, POS logic equations are read from a K-map by writing sum terms defined by each loop, and then AND'ing all the sum terms together. A loop term is defined by the logic variables on the periphery of the K-map. SOP loop terms use minterm codes (i.e., the '0' domain of a variable results in that variable being complemented in the product term for the loop), and POS loop terms use maxterm codes (i.e., the '1' domain of an input variable results in that variable being complemented in the sum term for the loop). If a loop spans across both the '1' and '0' domain of a given logic variable, then that variable is redundant and it does not appear in the loop term. Restated, a logic variable is included in a loop term only if the loop is contained entirely in the '1' or '0' domain of that variable. The edges of maps are continuous with the opposite edges, so loops can span from one edge to the other without grouping 1's or 0's in the middle (the examples below illustrate this process).

K-maps can be used for finding minimal logic expressions for systems of 2, 3, 4, 5, or 6 input variables (beyond 6 variables and the technique becomes unwieldy). For systems of 2, 3, or 4 variables, the technique is straightforward, and it is illustrated in several examples below. In general, the looping process should be started with 1's (or 0's) that can only be grouped in one possible loop. As loops are drawn, ensure that all 1's (or 0's) are in at least one loop, and that no redundant loops exist (a redundant loop contains 1's or 0's that are *all* already grouped in other loops).

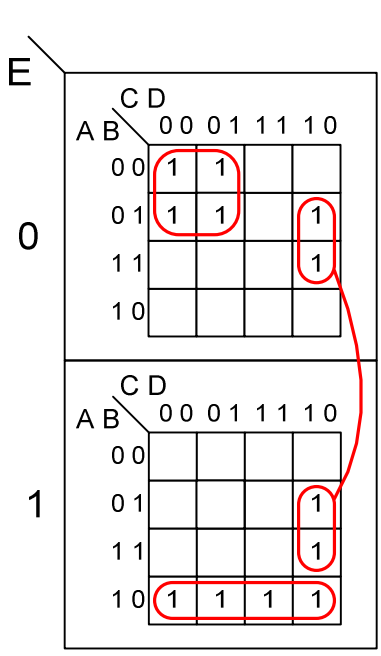


Minterm SOP equations and maxterm POS equations can be readily transferred into K-maps by simply placing 1's (for SOP equations) and 0's (POS) in the cells listed in the equation. For SOP equations, any cell not listed as receiving a '1' gets a '0', and vice-versa for POS equations. The figures below illustrate the process.

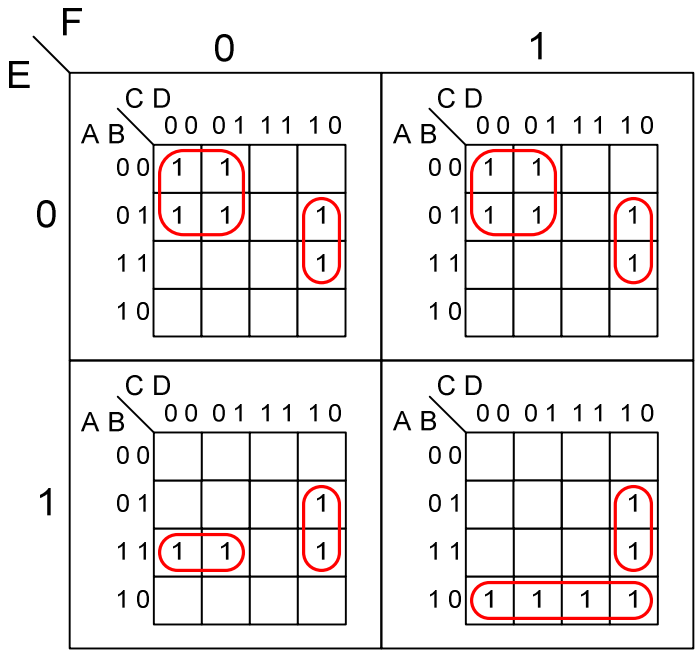


For systems of 5 or 6 variables, two different methods can be used. One method uses 4-variable K-maps nested in 1 or 2 variable "super maps", and the other method uses "map entered variables". The super-map technique for finding minimum equations for 5 or 6 variables closely follows the technique

used for 2, 3, or 4 variables, but 4-variable maps must be nested into 1 or 2-variable super-maps as shown below. Logic adjacencies between the sub-maps can be discovered by identifying 1's (or 0's) in like-numbered cells in adjacent super-map cells. The patterns in the maps show examples of adjacent cells in the K-maps. SOP equations for the maps are shown – note that the "super map" variables do not appear in product terms when 1's are located in like-numbered cells in the sub maps.



$F = \bar{A}\cdot\bar{C}\cdot\bar{E} + B\cdot C\cdot\bar{D} + A\cdot\bar{B}\cdot E$



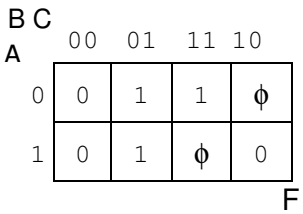
$F = \bar{A}\cdot\bar{C}\cdot\bar{E} + B\cdot C\cdot\bar{D} + A\cdot\bar{B}\cdot E\cdot F + A\cdot B\cdot\bar{C}\cdot E\cdot\bar{F}$

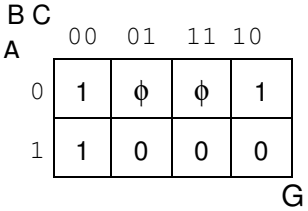
Incompletely specified logic functions (don't cares)

Situations can arise where a circuit has N input signals, but not all 2^N combinations of inputs are possible. Or, if all 2^N combinations of inputs are possible, some combinations might be irrelevant. For example, consider a television remote control unit that can switch between control of a television, VCR, or DVD. Some remotes might have operational modes where buttons like "fast forward" are physically switched out of the circuit; other remotes may use modes where such buttons are left in the circuit, but their functions are irrelevant. In either case, some combinations of input signals are completely inconsequential to the proper operation of the circuit. It is possible to take advantage of these situations to further minimize logic circuits.

Input combinations that cannot possibly effect the proper operation of a logic system can be allowed to drive circuit outputs high or low – literally, the designer doesn't care what the circuit response is to these impossible or irrelevant inputs. This information is encoded by using a special "don't care" symbol in truth tables and K-maps to

A	B	C	F	G
0	0	0	0	1
0	0	1	1	ϕ
0	1	0	ϕ	1
0	1	1	0	ϕ
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	ϕ	0

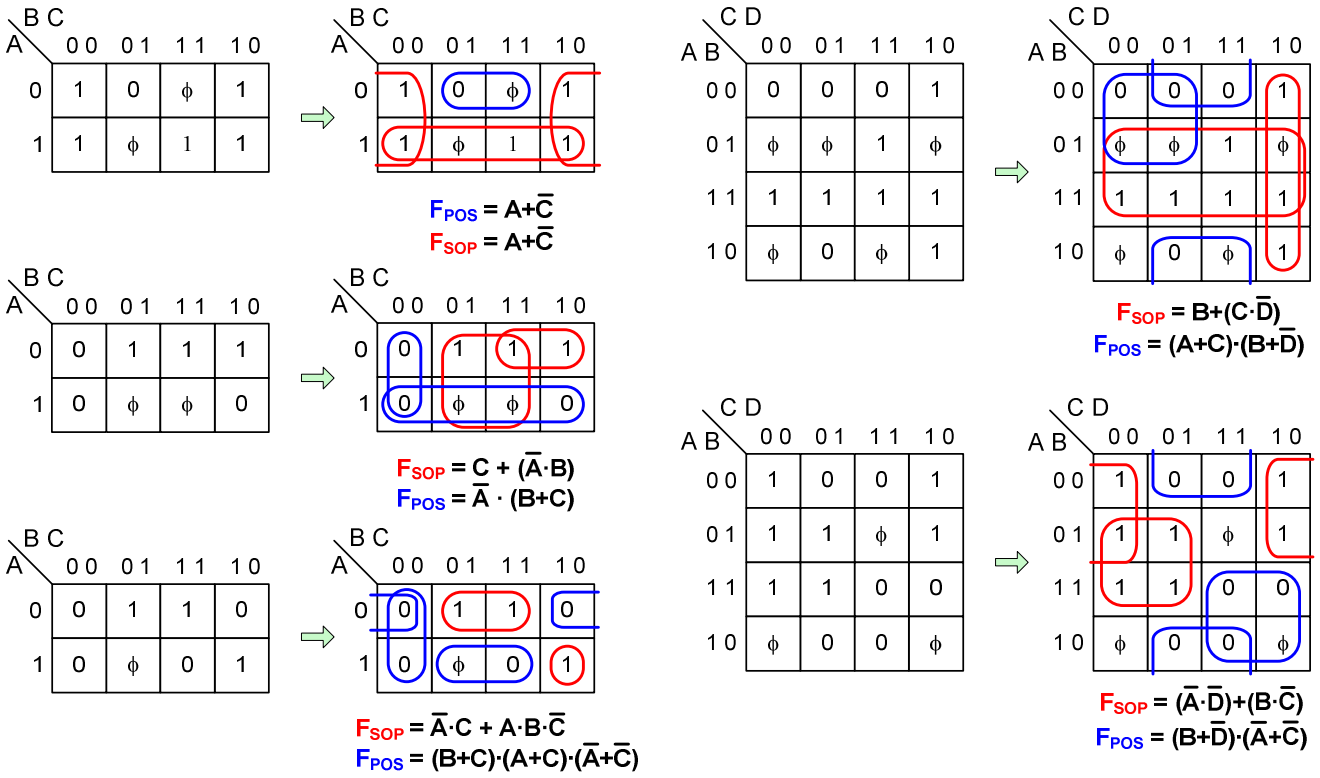




indicate that the signal can be a '1' or a '0' without effecting circuit operation. Some sources use an "X" to indicate a don't care, but this can be confused with a signal named "X". It is perhaps a better practice to use a symbol that is not normally associated with signal names – here, we choose the "ϕ" symbol.

The truth table on the right shows two output functions (F and G) for the same three inputs. Both outputs have two rows where the output is a don't care. This same information is also shown in the associated K-maps. In the "F" K-map, the designers "don't care" if the output is a '1' or a '0' for minterms 2 and 7, and so cells 2 and 7 in the K-map can be looped as either a '1' or a '0'. Clearly, looping cell 7 as a '1' and cell 2 as a '0' results in a more minimal logic circuit. In this case, both an SOP and POS looping would result in identical circuits.

In the "G" K-map, the don't cares in cells 1 and 3 can be looped as either a '1' or a '0'. In an SOP looping, both don't cares would be looped as 1's, giving a logic function of "G = A' + B'·C'". In a POS looping, however, cells 1 and 3 would be looped as 0's, giving the logic function "G = C'·(A' + B'). A little Boolean algebra reveals these two equations are not algebraically equal. Often, the SOP and POS forms of equations looped from K-maps that contain don't cares are not algebraically equal (although they would perform identically in the circuit). The following examples illustrate the use of don't cares in K-maps.

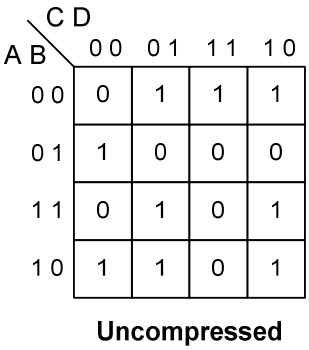


Entered Variables

Truth tables provide the best mechanism for completely specifying the behavior of a given combinational logic circuit, and K-maps provide the best mechanism for visualizing and minimizing the input-output relationships of digital logic circuits. So far, we have shown input variables across the top left of a truth table and around the periphery of K-maps. This allows every state of an output signal to

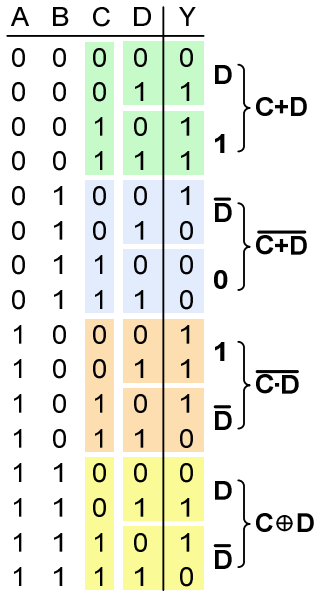
be defined as a function of the input patterns of 0's and 1's on a given row in a truth table, or as the binary coding for a given K-map cell. Without any loss of information, truth tables and K-maps can be translated into a more compact form by moving input variables from the top-left of a truth table to the output column, or from outside the K-map to inside the cells of a K-map. Although it will not be clear until later modules, the use of entered variables and compressed truth tables and K-maps often makes a multi-variable system much easier to visualize and minimize.

The translation mechanics are illustrated in the figures below, where a 16-row truth table is compressed into both 8-row and 4-row truth tables. In the 8-row truth table, the variable D is no longer used to identify an input column. Instead, it appears in the output column, where it encodes the relationship between two rows of output logic values and the D input. In the 4-row truth table, variables C and D are no longer used to identify an input column, but rather in the output column where they encode the relationship between four rows of the output logic values and the C and D inputs.



A B \ C D	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	0	1	0	1
10	1	1	0	1

Uncompressed



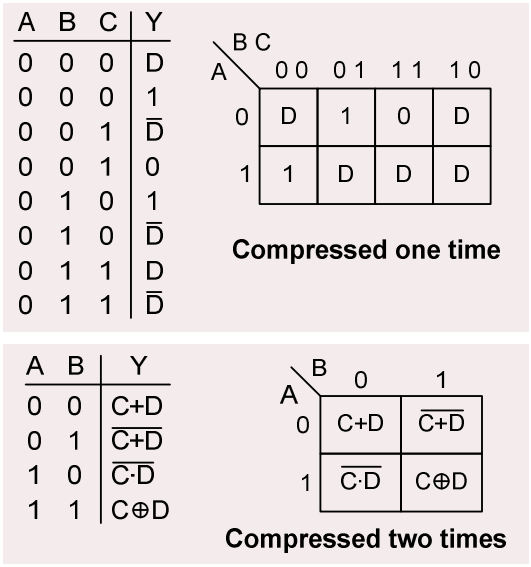
A	B	C	D	Y	
0	0	0	0	0	D
0	0	0	1	1	
0	0	1	0	1	1
0	0	1	1	1	
0	1	0	0	1	$\overline{D}$
0	1	0	1	0	
0	1	1	0	0	0
0	1	1	1	0	
1	0	0	0	1	1
1	0	0	1	1	
1	0	1	0	1	$\overline{D}$
1	0	1	1	0	
1	1	0	0	0	D
1	1	0	1	1	
1	1	1	0	1	$\overline{D}$
1	1	1	1	0	

C+D

$\overline{C+D}$

$\overline{C \cdot D}$

C⊕D



A	B	C	Y
0	0	0	D
0	0	0	1
0	0	1	$\overline{D}$
0	0	1	0
0	1	0	1
0	1	0	$\overline{D}$
0	1	1	D
0	1	1	$\overline{D}$

A \ B C	00	01	11	10
0	D	1	0	D
1	1	D	D	D

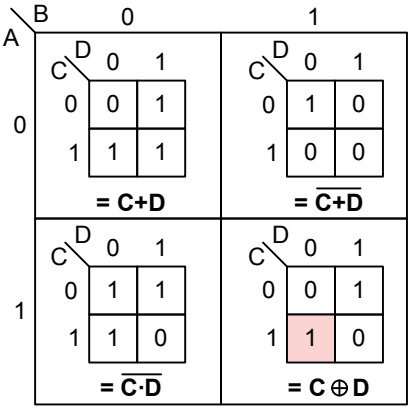
Compressed one time

A	B	Y
0	0	C+D
0	1	$\overline{C+D}$
1	0	$\overline{C \cdot D}$
1	1	C⊕D

A \ B	0	1
0	C+D	$\overline{C+D}$
1	$\overline{C \cdot D}$	C⊕D

Compressed two times

The 4-cell K-map is reproduced to the right, this time showing the implied sub-maps that illustrate the relationship between C and D for each of the four unique values of the A and B variables. For any entered variable K-map, thinking of (or actually sketching) the sub-maps can help identify the correct encoding for the entered variables. Note that truth table row numbers can be mapped to cells in the sub-maps by reading the K-map index codes, starting with the super-map code, and appending the sub-map code. For example, the shaded box in the sub-map is in box number 1110.



A \ B	0	1
0	C \ D: 0 0 1 / 1 1 1	C \ D: 0 1 0 / 1 0 0
1	C \ D: 0 1 1 / 1 1 0	C \ D: 0 0 1 / 1 1 0

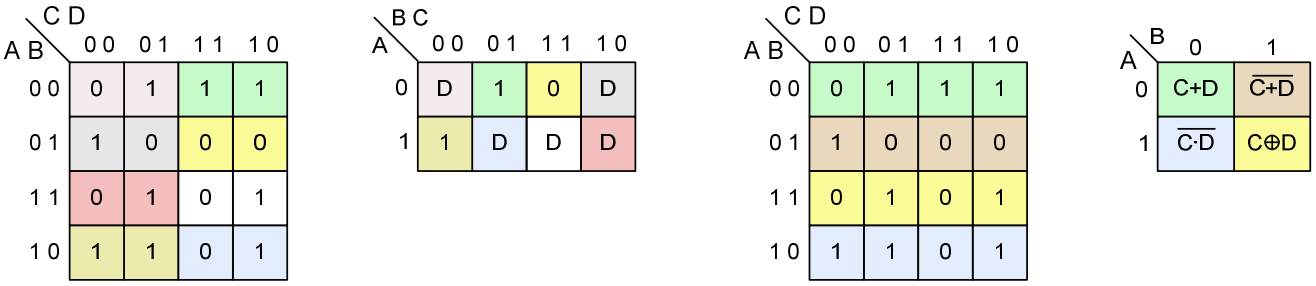
= C+D

= $\overline{C+D}$

= $\overline{C \cdot D}$

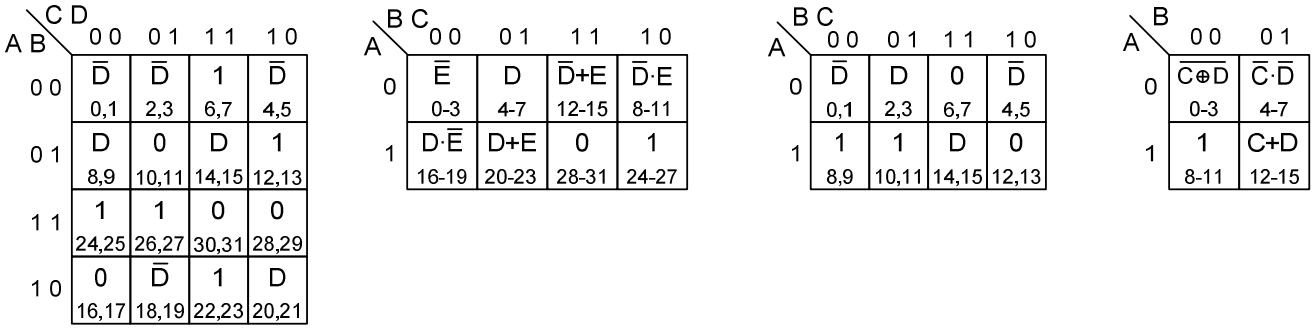
= C⊕D

This same map compression is illustrated below, showing the mapping from non-compressed K-maps directly to compressed K-maps. The colors show how cells in the uncompressed map are translated to the cells in the compressed map. Note that two cells in the 16-cell map are compressed into a single cell in the 8-cell map, and that four cells in the 16-cell map are compressed into a single cell in the 4-cell map.



Minterm SOP equations and maxterm POS equations can also be translated directly into entered variable K-maps as shown in the illustration below (the smaller numbers at the bottom of K-map cells show the minterm or maxterm numbers assigned to that cell). The minimum number of input variables is assumed when encoding minterms or maxterms into the K-maps. For example, if the largest minterm present is 14, four input variables are assumed.

$F = \Sigma m(0, 2, 4, 6, 7, 9, 12, 13, 15, 18, 21, 22, 23, 24, 25, 26, 27)$ $F = \Pi M(1, 2, 5, 6, 7, 12, 13, 14)$

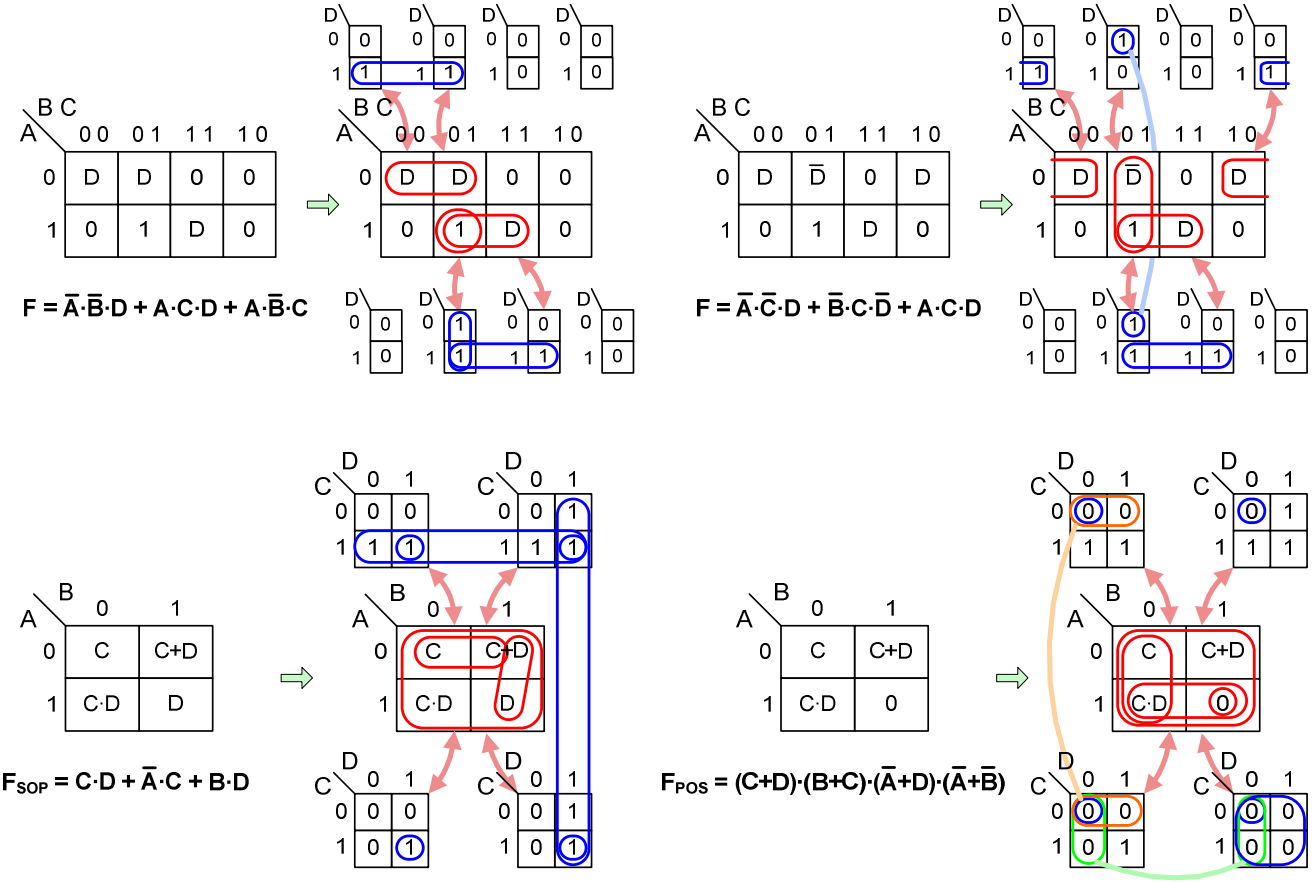


Looping entered variable K-maps follows the same general principles as looping “1-0” maps – optimal groupings of 1’s and entered variables (EVs) are sought for SOP circuits, and optimal groupings of 0’s and EVs are sought for POS circuits. The rules are similar: all EVs and all 1’s (or 0’s) must be grouped in the largest possible “power of 2” sized rectangular or square grouping, and the process is complete when all EV’s and all 1’s (or 0’s) are included in an optimal loop. The differences are that similar EVs can be included in loops by themselves or with 1’s (or 0’s), and care must be taken when looping cells with 1’s (or 0’s), because a ‘1’ (or ‘0’) indicates that all possible combinations of EV’s are present in that map cell, and loops that include 1’s (or 0’s) together with EV’s often include only a subset of the possible combinations of the EV’s (this is illustrated in the figures below). Looping an EV K-map is complete when all minterms or maxterms are contained in an adequate group. Perhaps the most challenging aspect is to ensure that all possible combinations of EV’s have been accounted for in cells that contain 1’s (or 0’s).

To help understand looping in EV K-maps, it may be helpful to think of the sub-maps implied by every K-map cell. As shown in the figures below, the variables in K-map cells can arise from looping the “1-0” information entered into cells in the implied sub-maps. A looping of information in adjacent cells in the EV K-map can include 1’s (or 0’s) in the sub-maps that appear in the same positions in the sub-maps.

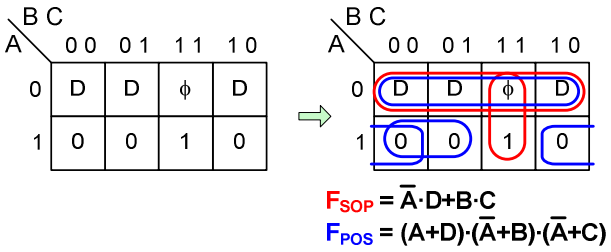
When reading the loop equations, the SOP product terms (or POS sum terms) for each loop must include the variables that define the loop domain and the EVs contained within the loop. For example,

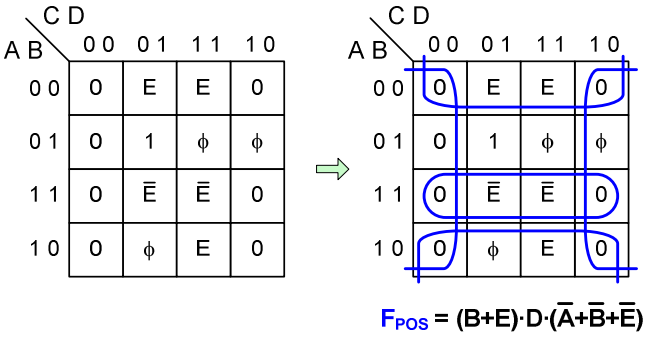
in the first example below, the first SOP term A'·B'·D includes the loop domain A'·B' and the entered variable D.

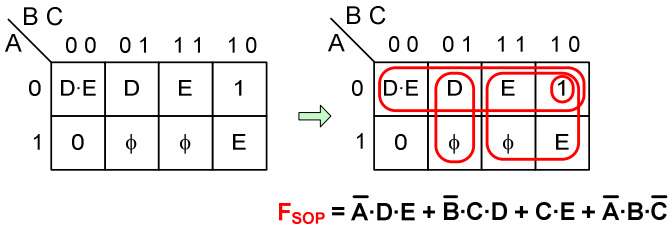


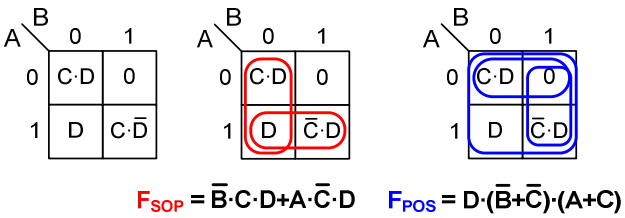
Cells in entered variable maps might contain a single entered variable or a logic expression of two or more variables. When looping cells that contain logic expressions, it helps to recognize the differences in SOP and POS looping mechanics. As compared to a single EV in a K-map cell, a product term in a cell represents a smaller SOP domain, because the more AND'ed variables in a product term, the smaller the defined logic domain. A sum term in a cell represents a larger SOP domain, because the more OR'ed variables in a sum term, the larger the defined logic domain. When looping SOP equations from an EV map, cells containing product terms have fewer 1's in their sub-maps than cells that contain single EV's, and cells with sum terms contain more 1's. Similarly, when looping POS equations from an EV map, cells containing sum terms have fewer 0's in their sub-maps than cells that contain single EV's, and cells with product terms contain more 0's.

Don't cares in entered variable K-maps serve the same purpose as they did in "1-0" maps; they indicate input conditions that cannot occur or that are irrelevant, and they can be included in groupings of 1's, 0's, or entered variables as needed to minimize logic. As shown in the examples below, a given don't care can be taken as a '1', '0', or entered variable as needed for any particular loop.









Computer-Based Logic Minimization Algorithms

Several logic minimization algorithms have been developed over the years, and many of them have been incorporated into computer-based logic minimization programs. Some of these programs, such as those based on the Quine-McCluskey algorithm, find a true minimum by exhaustively checking all possibilities. Programs based on these exhaustive search algorithms can require long execution times, especially when dealing with large numbers of inputs and outputs. Other programs, such as the popular Espresso program developed at UC Berkeley, use heuristic (or rule-based) methods instead of exhaustive searches. Although these programs run much faster (especially on moderate to large systems), they terminate upon finding a "very good" solution that may not always be minimal. In many real-world engineering situations, finding a greatly minimized solution quickly is often the best approach.

Espresso is by far the most widely used minimization algorithm, followed by Quine-McCluskey. These two algorithms will be briefly introduced, but not explained. Many good references exist in various texts and on the web that explain exactly how the algorithms function – you are encouraged to seek out and read these references to further your understanding of logic minimization techniques.

The Quine-McCluskey logic minimization algorithm was developed in the mid-1950's, and it was the first computer-based algorithm that could find truly minimal logic expressions. The algorithm finds all possible groupings of 1's through an exhaustive search, and then from that complete collection finds a minimal set that covers all minterms in the on-set (the on-set is the set of all minterms for which the function output is asserted). Because this method searches for all possible solutions, and then selects the best, it can take a fair amount of computing time, In fact, even on modern computers, this algorithm can execute for minutes to hours on moderately sized logic systems. Many free-ware programs exist that use the Q-M algorithm to minimize a single equation or multiple equations simultaneously.

Espresso was first developed in the 1960's, and it has become the most commonly used logic minimization program used in industry. Espresso is strictly "rule-based", meaning that it does not search for a guaranteed minimum solution (although in many cases, the true minimum is found). An espresso input file must be created before espresso can be run. The input file is essentially a truth table that lists all the minterms in the non-minimized function. Espresso returns an output file that

shows all the terms required in the output expression. Espresso can minimize a single logic function of several variables, or many logic functions of several variables. Espresso makes several simplifying assumptions about a logic system, and it therefore runs very quickly, even for large systems.

Digimin is a windows wrapper that allows both Boozer and Espresso to be run in a Windows environment. Digimin also provides an easy to use truth-table entry mechanism and provides output in the form of SOP and POS equations. Digimin, available from the class website, is easy and intuitive to use – simply run it, add functions (by selecting Action -> add function), and then add variables to the functions (by selecting Action -> add variables). When all functions and variables have been added, simply choose the MIN function and the Espresso or Boozer algorithm.

Since the1990's, Hardware Definition Languages (HDLs) and their associated design tools and methods have been replacing all other forms of digital circuit design. Today, the use of HDLs in virtually all aspects of digital circuit design is considered standard practice. We will introduce the use HDLs in a later module, and as we will see, any circuit defined in an HDL environment is automatically minimized before it is implemented. This feature allows a designer to focus strictly on a circuit's behavior, without getting slowed down in the details of finding efficient circuits. Although it is important to understand the structure and function of digital circuits, experience has shown that engineers can be far more productive by specifying only a circuit's behavior, and relying on computer-based tools to find efficient circuit structures that can implement those behaviors.