PDHonline Course E283 (6 PDH)

Karnaugh Maps (Digital Logic Optimization)

Instructor: David A. Snyder, PE

2020

PDH Online | PDH Center

5272 Meadow Estates Drive
Fairfax, VA 22030-6658
Phone: 703-988-0088
www.PDHonline.com

An Approved Continuing Education Provider

Karnaugh Maps
(Digital Logic Optimization)

David A. Snyder, PE

Table of Contents:

Introduction:

This course describes the use of Karnaugh Maps (also known as Veitch diagrams) to optimize a binary (digital) function, allowing us to design the logic to use the fewest number of gates, or, in the case of relay logic, the fewest number of relay contacts. This course can be a refresher for readers who have already been exposed to Karnaugh maps, but it can also be a simple introduction to Karnaugh maps for readers who are not yet familiar with them. It is assumed that the reader is familiar with binary and hexadecimal numbering. Boolean algebra will be used as a checking tool in this course, but the quiz will not require any Boolean algebra.

A binary function can be represented as a truth table of outputs based on all of the possible inputs, such as in Truth Table 7. It is a straightforward task to design the required gates or relay logic to represent each output based on the results of the truth table (see Figure 1a), but it is usually possible to reduce or optimize the logic (see Figure 1b) prior to designing the relay contact or logic gate implementation of the function described by the truth table. Karnaugh maps are similar to truth tables and are used to graphically represent the same information in a format that makes it easier to optimize the logic by grouping together the inputs and outputs such that adjacent cells are more logically related to each other. It is a common opinion that Karnaugh maps tend to lose their effectiveness for more than 6 input variables (6 input variables would be uvwxyz with 2^6 or 64 outputs), due to the size and complexity of the resulting maps.

Note:
Only 2-input AND & OR gates are considered in this document. Implementation using NOR and NAND gates will not be discussed in this document.

Why is it important to be able to optimize a binary function? Refer again to Figure 1a and Figure 1b. These figures illustrate the same binary function, but Figure 1b is the optimized version of the function. It may not seem very important to eliminate a few gates or relay contacts from the design of a single logic function, but a typical project could involve dozens of functions, so the reduction of a few gates for each function could result in significant space savings on a circuit board design. Also, some logic designs will be replicated many thousands of times for mass-produced products, so the savings of a few dollars per item can have a large impact. Similarly, the reduction of the number of relay contacts required to implement a function might allow you to complete a project in the field by using an extra relay that has been rattling around in your toolbox, instead of having to take half a day to run to the electrical supply house and back to the job site.

This document uses a value of 1 for a True (On) statement and a value of 0 for a False (Off) statement. This document places variables together with no intervening symbol to represent AND (xy = x AND y) and the + symbol for OR (x + y = x OR y). The complement or prime of a variable or equation is represented by an apostrophe (x', also known as x-prime, is the complement of x). Numbers that appear to be binary, such as 0101, are binary, unless otherwise noted.

Truth Tables:

A truth table is one way of representing the input and output values of a binary function. For example, let's say the function F is described by Truth Table 1.

x	*F*
0	*0*
1	*1*

$$F = x$$

Truth Table 1

The simple example in Truth Table 1 shows us that the output function F is equal to input x, since the function F is true (equal to 1) whenever input variable x is true (equal to 1). Similarly, the output function F of Truth Table 2 is equal to the complement or inverse of input x, since F is true (1) when input x if false (0), and vice-versa.

x	*F*
0	*1*
1	*0*

$$F = x'$$

Truth Table 2

The previous truth tables are functions of only one input variable, called x. Since there is only one input variable and it has two valid states (1 or 0), the truth table has 2^1 output cells, which means that there are 2 possible outputs. Let's add another input variable, which we will call y. That will give us a truth table that has 2^2 output cells, which gives us 4 possible outputs, as illustrated in Truth Table 3.

x	y	*F*
0	0	*0*
0	1	*0*
1	0	*0*
1	1	*1*

F = xy

Truth Table 3

It is easy to deduce from Truth Table 3 that the output function F is equal to xy (x AND y), since the output function is only true (equal to 1) when both x and y are true (equal to 1). Another 2^2 example is presented in Truth Table 4.

x	y	*F*
0	0	*0*
0	1	*1*
1	0	*1*
1	1	*1*

F = x + y

Truth Table 4

Truth Table 4 represents an output function F that is equal to x + y (x OR y), since the output function is true (equal to 1) when either x or y are true (equal to 1).

Minterms:

Each of the output cells (shown in the red text column of the previous truth tables) has a possible minterm associated with it, as shown in Truth Table 5. It is described as a *possible* minterm because we will only use a minterm if the function F is true (1) for that particular input combination.

x	y	Minterms
0	0	x'y'
0	1	x'y
1	0	xy'
1	1	xy

Introduction to Minterms

Truth Table 5

Truth Table 5 shows that if a 1 (true output) were present at input xy = 00, the minterm for that row of the truth table would be x'y'. Likewise, if a 1 were present at the output function for input xy = 01, the minterm for that row of the truth table would be x'y. If a 1 were present at the output of input combination xy = 10, the minterm for that row of the truth table would be xy'. Finally, if a 1 were present at xy = 11, the minterm for that row of the truth table would be xy. Notice how the minterm has an x' term (such as x'y') when the x input is 0, but has an x term (such as xy') when the x input is 1. Similarly, the minterm has a y' term (such as x'y') when the y input is 0, but has a y term (such as xy) when the y input is 1.

Using the minterm definitions of Truth Table 5 and the output values of Truth Table 4, we can assemble the minterms of Truth Table 4 to be:

F = x'y + xy' + xy

We did this by including only the minterms from Truth Table 5 where the output value is 1 on Truth Table 4. This happens on the last three rows of Truth Table 4, so we ORed together the last three rows of Truth Table 5. It might look like we arrived at a different answer than shown in red text at the bottom of Truth Table 4 (F = x + y), but let's use Boolean algebra to simplify our most recent result. We start by grouping similar terms together, then factoring out a common term (just like in 'regular' algebra):

F = x'y + xy' + xy =

x'y + x(y' + y)

Since (y' + y) is always true (see Appendix – Boolean Algebra), whether y = 0 or y = 1, the term (y' + y) can be replaced with a 1. Since x AND 1 is always x, the term x(y' + y) is simply x. That simplifies our results to:

F = x'y + x

The distributive property of Boolean algebra (see Appendix – Boolean Algebra) tells us that:

A + BC = (A + B)(A + C), so we can set A = x and BC = x'y and say:

F = x + x'y =

(x + x')(x + y)

Since (x + x') is always equal to 1, the (x + x') term can be dropped, giving us:

F = (x + y)

This matches the result of Truth Table 4.

Notice that a minterm expression, such as F = x'y + xy' + xy, looks like a sum of three products.

Maxterms:

Each of the output cells (shown in the red text column of Truth Table 4) also has a possible maxterm associated with it, as shown in Truth Table 6. They are described as *possible* maxterms because we will only use a maxterm if the function is false (equal to 0) for that particular input condition.

x	y	Maxterms
0	0	*x + y*
0	1	*x + y'*
1	0	*x' + y*
1	1	*x' + y'*

Introduction to Maxterms

Truth Table 6

Maxterms will be discussed in more detail later. For a quick exercise, however, if we reconsider Truth Table 4 from the point of view of maxterms, we see that only one of the output cells has a zero in it. This corresponds to when input variables xy = 00, which defines a maxterm of (x + y) in Truth Table 6. This maxterm result of F = x + y agrees with the minterm result in the previous discussion and with the result of Truth Table 4.

Checking with Truth Tables:

Another two-input (2^2 output) example is presented in Truth Table 7.

x	y	*F*
0	0	*1*
0	1	*1*
1	0	*1*
1	1	*0*

$$F = x'y' + x'y + xy'$$
$$F = (x'y' + x'y) + xy'$$
$$F = x'(y' + y) + xy'$$
$$F = x' + xy'$$
$$F = x' + y'$$

Truth Table 7

In Truth Table 7, the output function is listed in the red text underneath the truth table as a minterm expression or a sum of products. To do this, look for each occurrence where the output function F is true (equal to 1) and write out the minterm for each, then OR all of the minterms together. As previously stated, we do this by writing the 'regular' input variable (such as x) when x = 1, and the complement (x') when x = 0, as demonstrated with Truth Table 5. Since there are three outputs that have a value of 1, we will start out with three minterms. The minterms in Truth Table 7, starting at the top at xy = 00, are x'y' (since x = 0 and y = 0); on the second line at xy = 01, the minterm is x'y (since x = 0 and y = 1); and on the third line at xy = 10, which is the last line where output function F is true, the minterm is xy' (since x = 1 and y = 0). When these minterms are ORed, they are x'y' + x'y + xy', which defines the three input combinations (xy = 00, 01, & 10) that have an output of 1. The first line of red text under Truth Table 7 lists the three minterms. The second line of red text under Truth Table 7 groups two of the minterms together within parentheses in preparation to reduce and simplify the equation using Boolean algebra. The third line of red text under Truth Table 7 shows the x' being factored out of the parentheses, since x' is part of both of those minterms. Now there is y + y' inside of the parentheses, which means the value inside the parentheses will always be 1, whether y = 0 or y = 1. The fourth line of red text under Truth Table 7 is a simplified version of output function F, but we can use the distributive property of Boolean algebra (see Appendix – Boolean Algebra) with A = x' and BC = xy' to simplify it further. The fifth line of red text beneath Truth Table 7 is a simpler expression of the output function F, and it would obviously require fewer gates (or relay contacts) and fewer interconnecting conductors than the way the function was described on the first line of red text under Truth Table 7.

Even though we are fairly certain that the results of simplifying output function F are correct, we can check our work by substituting the input values into the simplified function and comparing the results with Truth Table 7. This is performed in Truth Table 8.

x	x'	y	y'	x'+y'	F
0	1	0	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	0	1	0	0	0

$$F = x' + y'$$

Checking Simplified Output Function F Against Truth Table 7

Truth Table 8

The 'regular' inputs (x & y) and their complements (x' & y') are both shown in Truth Table 8 to make it easier to calculate the results of the minterm expression (in blue). The column of red text in Truth Table 8 is copied from Truth Table 7. This checking of simplified output function F = x' + y' has shown that it is a valid representation of Truth Table 7, since the outputs shown in the blue column and the red column of Truth Table 8 are the same.

It will be obvious to many readers that another very simple solution to Truth Table 7 is F = (xy)'. It might not be obvious that (xy)' = x' + y', but we can do another truth table to prove that it is true, as shown in Truth Table 9.

x	x'	y	y'	x'+y'	(xy)'
0	1	0	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	0	1	0	0	0

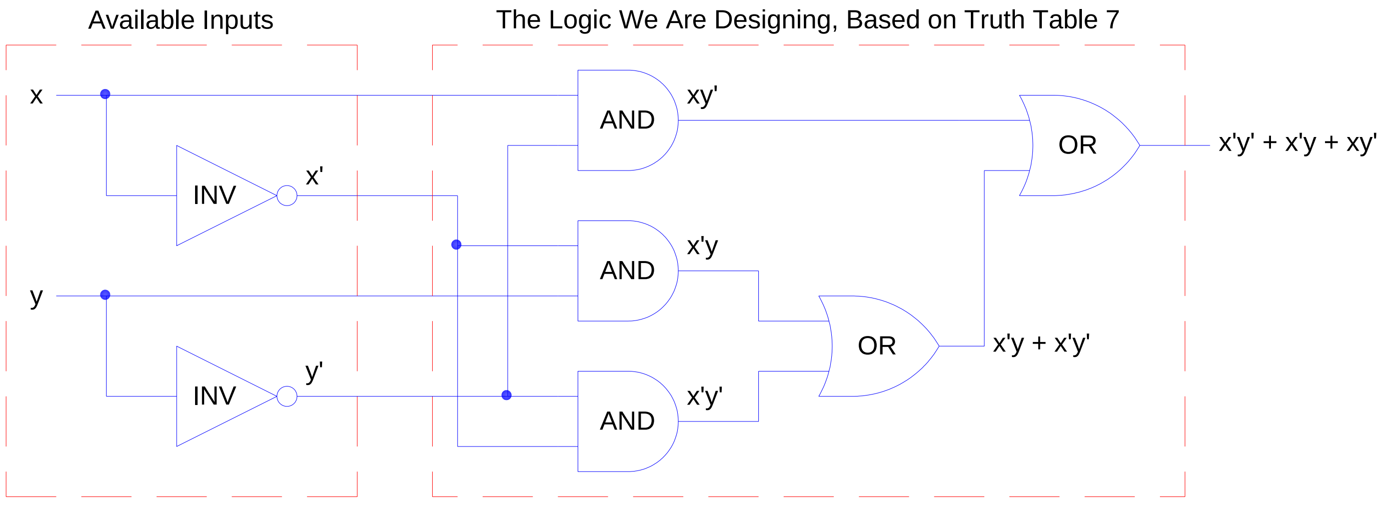
Truth Table 9

It is clear from Truth Table 9 that (xy)’ = x’ + y’. This relationship is also listed in Appendix – Boolean Algebra, under the DeMorgan heading.

Karnaugh Maps:

In addition to the use of Boolean algebra, there are other ways to arrive at simplified or reduced output functions, including the graphical method known as a Karnaugh map, which is just a rearrangement of a truth table. A Karnaugh map will show us how to minimize the number of relay contacts or logic gates required to produce the logical function by presenting the same information as the truth table, but in a different format.

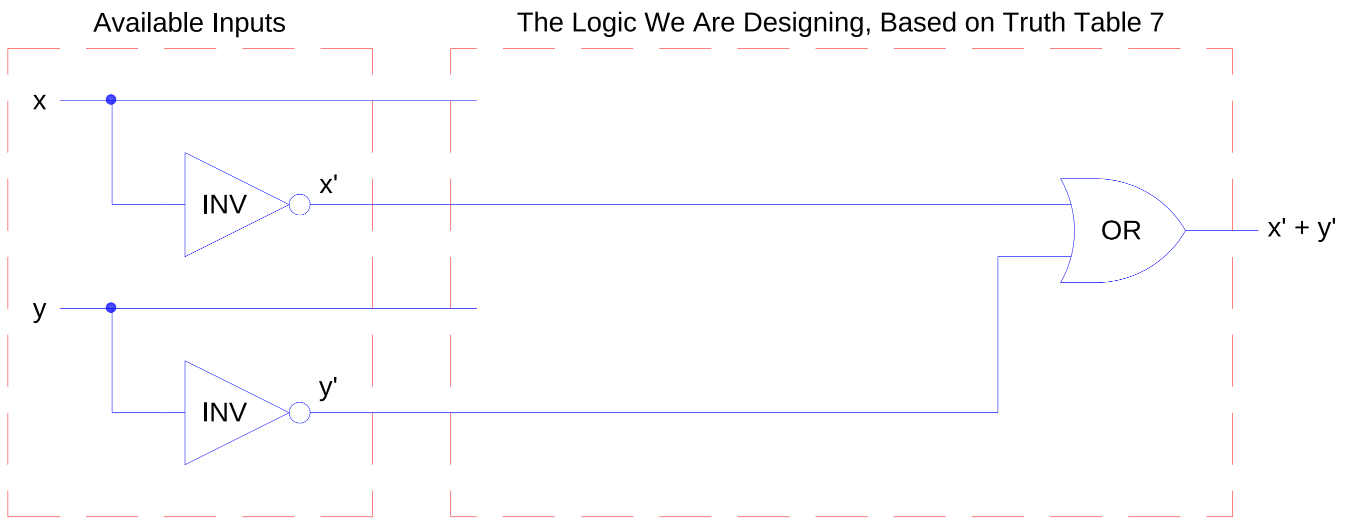
Suppose we were told to design a logic circuit to implement the function described in Truth Table 7. If we decided to scurry off and start designing the logic based on the first red line of text beneath Truth Table 7, without any kind of logic optimization, we would come up with the design in Figure 1a.



Non-Optimized Minterm Implementation with Gates
Figure 1a

Non-Optimized Minterm Implementation of Truth Table 7 with Gates
Figure 1a

If we take the time to perform some type of logic optimization, using Boolean algebra, Karnaugh maps, or some other technique, we can typically reduce and optimize the design of the logic, as shown in Figure 1b. Both of the designs in Figure 1a and Figure 1b will produce the same outputs, but the design in Figure 1b is obviously simpler and would therefore cost less to build.



Optimized Minterm Implementation of Truth Table 7 with Gates

Optimized Minterm Implementation with Gates
Figure 1b

Figure 1b

Let's rearrange the 2x2 example in Truth Table 7 into a Karnaugh map, which shows input variable x as rows and input variable y as columns, illustrated in Map 1.

F	0	1	y
0	*1*	*1*	
1	*1*	*0*	
x			

Map 1

Map 1 is equivalent to Truth Table 7. The upper-left cell of Map 1 represents x = 0 and y = 0, which is an output value of 1. The upper-right cell represents x = 0 and y = 1, which is an output value of 1. The lower-left cell represents x = 1 and y = 0, which is an output value of 1. The lower-right cell represents x = 1 and y =1, which has an output value of 0. All inputs in which x = 0 are represented on the first row of red text and all inputs in which x = 1 are represented in the second row. Likewise, all inputs in which y = 0 are represented in the first column of red text and all inputs in which y = 1 are represented in the second column. When we start using Karnaugh maps in earnest later in this document, we will usually not show output values of both 0 and 1, but rather one or the other.

A Karnaugh map, when properly executed, will give us an optimized solution, but there may be more than one equally-optimized solution, depending on how the groups are made. This is shown in Figure 9a and Figure 16.

Like truth tables, Karnaugh maps represent all the possible outputs for all the possible inputs. We will discuss minterms first (which occur where output values are 1), then maxterms (which occur where output values are 0).

Minterms:

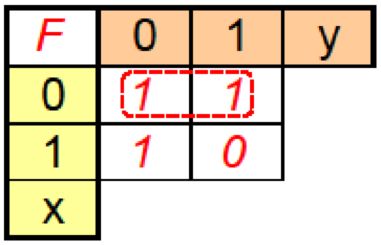
The four minterms of a 2-input function are shown on Map 2a. The outputs are defined as x'y', x'y, xy', & xy. Again, this Karnaugh map is simply a rearrangement of a truth table, which would be Truth Table 5, in this case. Each individual minterm is expressed by ANDing its input variables, resulting in x'y' when the input is xy = 00. Another example is the minterm xy' which is when the input is xy = 10. To reiterate, we only use minterms when they are true (equal to 1).

F	0	1	y
0	x'y'	x'y	
1	xy'	xy	
x			

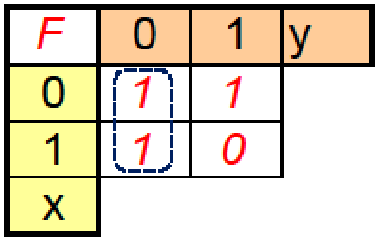
Minterms (Sum of Products)

Map 2a

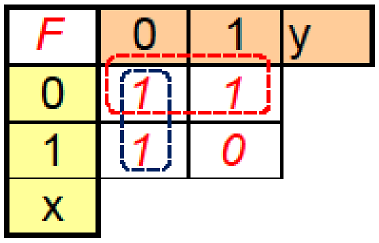
Karnaugh maps are used to graphically represent the output values in such a way that they can be grouped. Consider Maps 3a, 3b, and 3c, which are repeats of Map 1.



This minterm = x'



This minterm = y'



Combining all minterms:
F = x' + y'

Map 3a ***Map 3b*** ***Map 3c***

Adjacent cells with matching values (all 1s or all 0s) on a Karnaugh map can be grouped together in sets of 2, 4, 8, 16, or other powers of 2. Sometimes, a cell with a one or zero is not adjacent to another cell with a one or zero in it, so that cell would be all by itself in a group of 1, which is a group of 2^0 cells. Forming a group of 5 or 6 or 10 or some other quantity of cells that is not a power of 2 is not appropriate. Map 3a shows how output values of 1 appearing in two adjacent cells in the same row can be grouped together. In the group within the red shape on Map 3a, output values are 1 when xy = 00 and when xy = 01. Since the value of y changes from 0 to 1 in this group, y can be excluded or dropped out because the value of y has no effect on the truth of this particular grouping. The resulting minterm is x', which means the output is 1 in this group whenever x = 0.

Map 3b shows how output values of 1 in the same column appearing in two adjacent cells can be grouped together. In the group within the blue shape on Map 3b, output values are 1 when xy = 00 and when xy = 10. Since the value of x changes from 0 to 1 in this group, x can be excluded or dropped out because the value of x has no effect on the truth of this particular grouping. The resulting minterm is y', which means the output is 1 in this group whenever y = 0.

Map 3c shows the two groups at once and illustrates that the groups can overlap each other, as long as all of the 1s are included in at least one group. The output function F is arrived at by ORing all of the minterms together to get F = x' + y'.

If a truth table has all of its output cells filled in with ones, then the function would be F = 1, since F would always be true.

Maxterms:

Another way to optimize a function using Karnaugh maps is to find the maxterms, which are the inputs that yield a false (equal to zero) output. This requires an opposite way of thinking. Let's revisit Map 1 and look at the inputs that give false (0) outputs. There is only one input combination that yields a false output, which is xy = 11.

F	0	1	y
0	$x+y$	$x+y'$	
1	$x'+y$	$x'+y'$	
x			

Maxterms (Product of Sums)

Map 2b

Look at Map 2b, which illustrates the opposite way of thinking of primes and non-primes (such as x' and x) that is required when using maxterms. Maxterms were already discussed briefly at Truth Table 6. Let's expand on that. The four maxterms of a 2-input function are shown on Map 2b. As previously discussed, a Karnaugh map is simply a rearrangement of a truth table, so Map 2b is a rearrangement of Truth Table 6. The four possible maxterm outputs are defined as x + y, x + y', x' + y, & x' + y'. Each maxterm is expressed by ORing the complement of its inputs, such as x + y when the input is xy = 00. Another example is the maxterm x + y’ which is defined as when the input is xy = 01.

We describe maxterms as “an opposite way of thinking” because instead of cell 00 being equal to x’y’, cell 00 is now equal to x + y, which is the opposite or inverse or complement of x’y’. Likewise, cell 01 is x + y’ instead of x’y, cell 10 is x’ + y instead of xy’, and cell 11 is x’ + y’ instead of xy. When we reconsider Map 1 from the standpoint of false (0) outputs, we find that only cell 11 has a false output, which means the only maxterm is x’ + y’, as shown in Map 2b. We AND all of the maxterms together, but there is only one maxterm in this case, so F = x’ + y’. This agrees with the solution we found using minterms in the previous section.

We have thus far considered truth tables with one input variable (x) and also truth tables with 2 input variables (x & y), Now, let’s consider Truth Table 10, which has 3 input variables (x, y, & z), which combine for a total of $2^3 = 8$ possible outputs.

x	y	z	*F*
0	0	0	*1*
0	0	1	*1*
0	1	0	*1*
0	1	1	*0*
1	0	0	*1*
1	0	1	*1*
1	1	0	*1*
1	1	1	*0*

Minterms (Sum of Products):

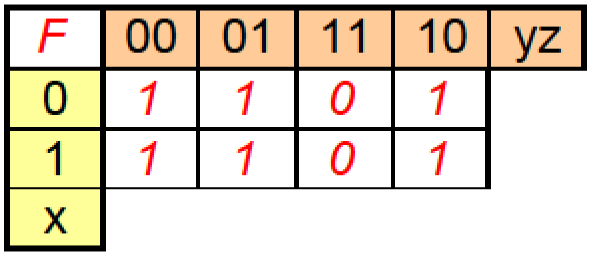
$F = x'y'z' + x'y'z + x'yz' + xy'z' + xy'z + xyz'$

Maxterms (Product of Sums):

$F = (x + y' + z')(x' + y' + z')$

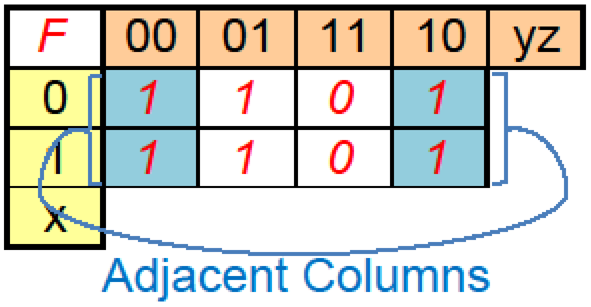
Truth Table 10

Since there are six output values that are 1 (true), we could start by listing six minterms for output function F in Truth Table 10. We could then try to simplify the output function by using Boolean algebra, but let’s try using a Karnaugh map instead. Truth Table 10 has been rearranged as Map 4a, which has exactly the same output values (1 or 0) for the same combination of inputs (x, y, & z).



F	00	01	11	10	yz
0	1	1	0	1	
1	1	1	0	1	
x					

Map 4a



F	00	01	11	10	yz
0	1	1	0	1	
1	1	1	0	1	
x					

Map 4b

The numbering along the top of Map 4 is slightly different than one might expect. Specifically, in Map 4a the columns are numbered 00, 01, 11, 10, instead of going in numerical order of 00, 01, 10, 11. The reason for this is to limit how many input variables can change when going from one column to the next. If the columns were numbered in numerical order (00, 01, 10, 11), then two input variables (y & z) would change simultaneously when going from yz = 01 to yz = 10 (and vice-versa) and also when wrapping around from 11 to 00 (and vice-versa). Using the numbering method of the Karnaugh map allows only one input variable to change at a time when moving from column to column (or from row to row, as we'll see later in 4-bit Karnaugh maps). In other words, when going from column 00 to 01, 01 to 11, 11 to 10, and wrapping around from 10 to 00, only one input variable at a time changes state. Map 4a shows the same information as Truth Table 10, just in a different arrangement. Notice that, when xyz = 000, the output is 1 in Map 4a and Truth Table 10. Likewise, when xyz = 011, the output is 0, and so on for all of the values in Map 4a and Truth Table 10.

With a pair of scissors, cut out Figure 2 on the dashed red lines and wrap it around an aluminum 12-ounce beverage can, such that the column with cells 000 and 100 wrap around the beverage can to touch the column with cells 010 and 110. Notice that the blue-shaded cells of this Karnaugh map are adjacent to each other. Going from cell 000 to cell 010 only changes one input variable, y. Going from cell 100 to cell 110 only changes one input variable, y again.

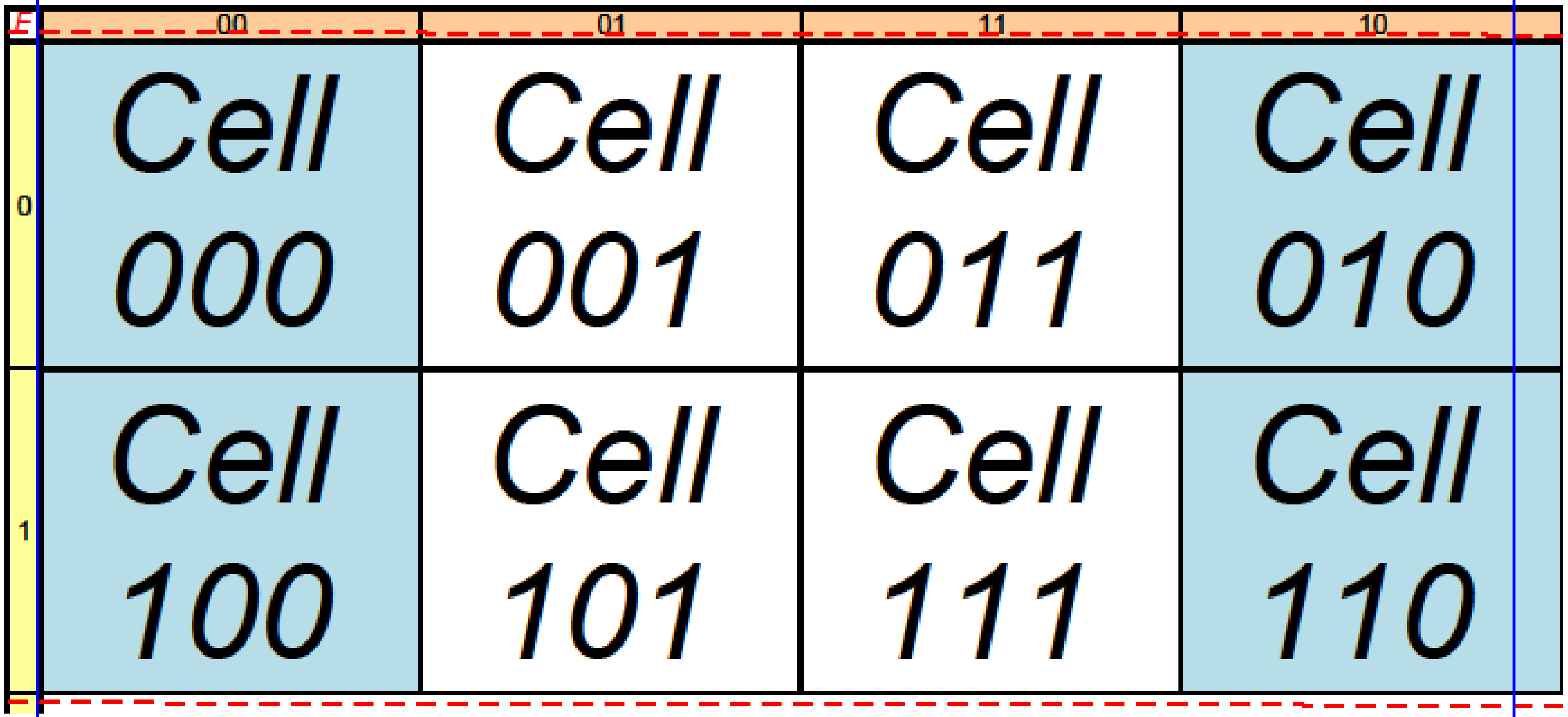


Illustration of Wrap-Around Feature for 3-Input Karnaugh Maps

Figure 2

The order or sequence in which the columns are numbered in Map 4a (and all Karnaugh maps of three or more input variables) is known as a Gray code, of which a special type is called reflected Gray code. As referred to above, a Gray code has the property of changing only one bit at a time from one position or value to an adjacent position or value. A reflected Gray code is a special type of Gray code that is symmetrical about its middle. Some examples of four-bit (wxyz) Gray codes are illustrated in Figure 3. If you take the type called Reflected Gray Code in Figure 3 and fold it in half on the line between decimal #7 and decimal #8, you will see that the facing cells are logically adjacent to each other, a sort of mirror-image symmetry. For example, cell 0100 is logically adjacent to cell 1100. Moving outward from the mirror-image or center line, cell 0101 is logically adjacent to cell 1101, and so on for cells 0111 & 1111, etc. Also, the first address (0000) wraps around to the last address (1000) while only one input variable changes (w, in this case).

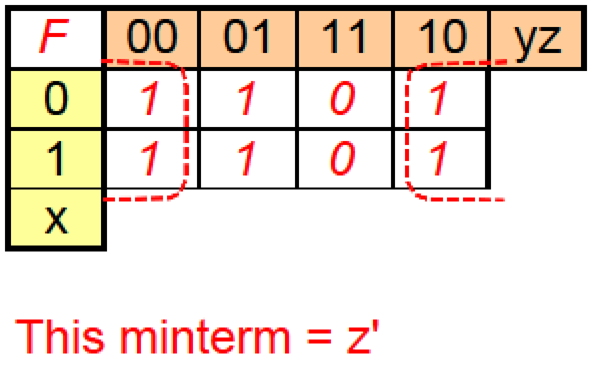
Dec.	"Regular" Binary Value				Reflected Gray Code				Another Gray Code				Yet Another Gray Code			
#	w	x	y	z	w	x	y	z	w	x	y	z	w	x	y	z
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	0	1	1	0	0	0	0	1	1	1
2	0	0	1	0	0	0	1	1	1	1	0	0	0	0	1	1
3	0	0	1	1	0	0	1	0	1	1	1	0	0	0	0	1
4	0	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
5	0	1	0	1	0	1	1	1	1	1	0	1	0	1	0	0
6	0	1	1	0	0	1	0	1	1	0	0	1	0	1	0	1
7	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	1
8	1	0	0	0	1	1	0	0	1	0	1	0	1	0	0	1
9	1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	0
10	1	0	1	0	1	1	1	1	0	1	1	0	1	1	0	0
11	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	0
13	1	1	0	1	1	0	1	1	0	0	0	1	0	0	1	0
14	1	1	1	0	1	0	0	1	0	1	0	1	1	0	1	0
15	1	1	1	1	1	0	0	0	0	1	0	0	1	0	1	1

Examples of Four-Bit Gray Code

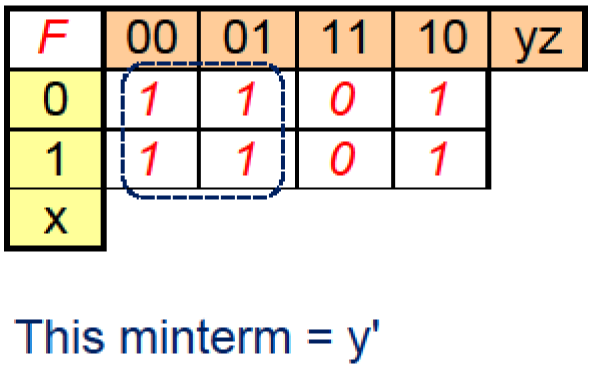
Figure 3

The reflected Gray code used in the 4-bit Karnaugh maps in this document is the one labeled “Reflected Gray Code” in Figure 3. The sequence of the reflected Gray code is shown in Figure 5 as a red path, starting at cell 0000 (assigned decimal #0 in Figure 3) and ending at cell 1000 (assigned decimal #15 in Figure 3). Notice that the red path takes a back-and-forth serpentine route – it is not like reading a book, going left to right, then dropping down to the next line.

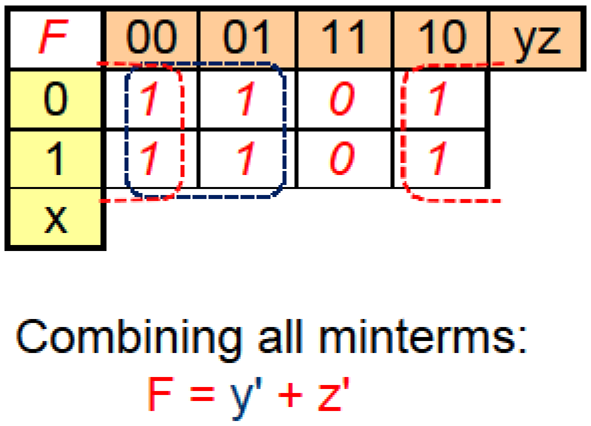
Getting back to the information in Truth Table 10, when there are three (xyz) or more input variables, it becomes important to discuss the ways in which the Karnaugh map folds back on itself. Specifically, the left-most column is logically adjacent to the right-most column, in terms of logic, since only one bit changes when travelling between these two columns, as illustrated in Map 4b, Figure 2, and Map 5a.



Map 5a



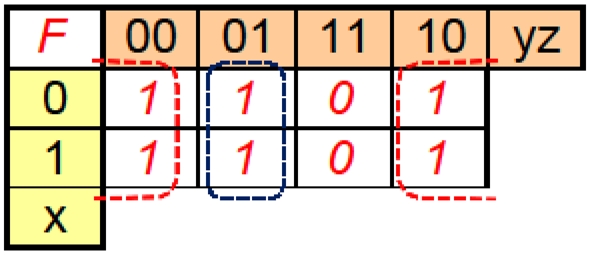
Map 5b



Map 5c

The four logically adjacent cells grouped in the two halves of the red shape in Map 5a give us the minterm z’, since z = 0 for this group and since x & y both change value within this group. The four adjacent cells shown grouped in Map 5b give us the minterm y’, since y = 0 for this group, while x & z both change value within this group. Combining both groups, which covers all of the true (1) outputs of function F gives us F = y’ + z’, as shown in Map 5c. Notice that the two groups overlap, which is perfectly acceptable. The main focus, the most important task, is to make the group or groups as large as possible, so as to cover a group of 2, 4, 8, etc.

We could have decided to group only the two cells x’y’z and xy’z in Map 5b and that would have covered all six of the true outpus (since the other four were covered in Map 5a), but that would given us a final result that was not as reduced as F = y’ + z’. This less-than-ideal decision is illustrated in Map 5d.



Combining all minterms:

F = z' + y'z

Less-Than-Ideal Optimization

Map 5d

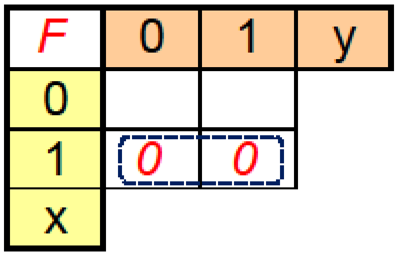
In Map 5d, by only including two cells in the blue shape, we have cheated ourselves out of a more-optimized solution. Comparing Map 5c with 5d, we now have an additional z term in Map 5d that is not really required. If we expand the blue shape to include the adjacent left-most column, we can drop the extra z term and have the more-optimized equation expressed in Map 5c.

If a truth table has all of its output cells filled in with zeroes, then the function would be F = 0, since F would always be false.

Before we go any further, let's have a quick review.

Example 1:

Consider Karnaugh Map 6a. What would be the maxterm expression of this map?



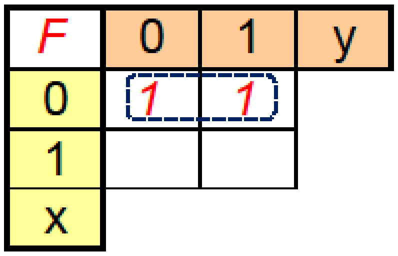
F = x'

Maxterms

Map 6a

As previously discussed, the maxterms are the input combinations that result in an output of false or zero (0). However, the conventions for primes and non-primes, as illustrated in Map 2b, are opposite of that for minterms (which represent an output of true or one). Looking at Map 6a, there are two cells that are filled in with zeroes, so we can assume that the other two cells are minterms, which have a value of 1. Looking at the two zeroes, it is clear that the output function of this Karnaugh Map is F = x', since the output values are zero when x = 1. If there were more than one maxterm, we would AND them all together.

Let's look at the same function as minterms (sum of products), as shown in Map 6b, instead of maxterms (product of sums). As we know, minterms are the input conditions that result in an output value of true or one (1).



$F = x'$

Minterms

Map 6b

Looking at Map 6b, there are two cells that are filled in with ones, so we can assume that the other two empty cells are maxterms, which have a value of 0 (see Map 6a). Looking at the two ones, it is clear that the output function of Karnaugh Map 6b is F = x', since the output values are one when x = 0. This matches the result of Map 6a. If there were more than one minterm, we would OR them all together.

It is unusual for a Karnaugh map to have both ones and zeroes filled in. It is more common to have either ones shown or zeroes shown, and to assume that the empty cells are zeroes or ones, respectively. Map 6c shows an example, however, in which both ones and zeroes are shown.

F	0	1	y
0	1	1	
1	0	0	
x			

$$F = x'$$

Map 6c

In Map 6c, it is up to the reader to decide whether to solve for maxterms (output = 0) or minterms (output = 1).

END OF EXAMPLE

Example 2:

Consider Karnaugh Map 7a. What would be the minterm expression of this map?

F	00	01	11	10	yz
0		1	1	1	
1				1	
x					

$$F = x'z + yz'$$

Minterms

Map 7a

By grouping the two pairs of ones as shown in Map 7a, it is clear that this function can be expressed by the sum of products F = x'z + yz'.

Let's look at the same function as a product of sums, using maxterms as shown in Map 7b. What would be the function described by that arrangement of false outputs?

F	00	01	11	10	yz
0	0				
1	0	0	0		
x					

$$F = (y + z)(x' + z')$$

Maxterms

Map 7b

By grouping the two pairs of zeroes as shown in Map 7b, it can be seen that the function described by the product of maxterms is F = (y + z)(x' + z'). Is this the same function that was described by Map 7a? Let's use Truth Table 11 to find out.

x	x'	y	y'	z	z'	x'z	yz'	x'z+yz'	y+z	x'+z'	(y+z)(x'+z')
0	1	0	1	0	1	0	0	0	0	1	0
0	1	0	1	1	0	1	0	1	1	1	1
0	1	1	0	0	1	0	1	1	1	1	1
0	1	1	0	1	0	1	0	1	1	1	1
1	0	0	1	0	1	0	0	0	0	1	0
1	0	0	1	1	0	0	0	0	1	0	0
1	0	1	0	0	1	0	1	1	1	1	1
1	0	1	0	1	0	0	0	0	1	0	0
								Map 7a			Map 7b

Checking Logical Equivalence of Map 7a & Map 7b

Truth Table 11

It can be seen from Truth Table 11 that the two formulas have the same results, so they are logically equivalent. Let's double-check those same results by using the comparatively more painful method of Boolean algebra. Proving the equivalence of logical functions by using Boolean algebra won't be on the quiz, but it is included here for completeness.

Let’s start with the minterm result of Map 7a: x’z + yz’

The distributive property of Boolean algebra is: A + BC = (A + B)(A + C). Let’s use that property (with A = x’z, B = y, and C = z’) to convert our starting equation to:

(x’z + y)(x’z + z’), which we will re-order as:

(y + x’z)(z’ + x’z).

Applying the distributive property to both of the parenthetical groups gives us:

[(y + x’)(y + z)][(z’ + x’)(z’ + z)]

Since (z’ + z) is always true (1), we can drop it from the equation, leaving us with:

(x’ + y)(y + z)(x’ + z’)

Let’s multiply (AND) the first two parenthetical groups together and merely copy down the third group. Expanding a Boolean algebra formula like the one above is just like multiplication in ‘regular’ algebra, which multiplies the first terms of both parenthetical groups, the outside terms, the inside terms, and the last terms (the acronym is FOIL):

[x’y + x’z + yy + yz](x’ + z’)

Multiplying (ANDing) the expression in brackets by the expression in parentheses gives us this lengthy result:

x’x’y + x’yz’ + x’x’z + x’zz’ + x’yy +yyz’ + x’yz +yzz’

There are a couple of terms (in red) that have zz’ in them, so we’ll drop those terms, since zz’ will always be zero. Also, x’x’y is simply x’y, x’yy is xy, and yyz’ is yz’. We can reduce the equation to:

x’y + x’yz’ + x’z + x’y + yz’ + x’yz

There are some duplications of terms (in blue) which are redundant, so we will drop the multiple occurrences of those terms, leaving only one copy of each term:

x’y + x’yz’ + x’z + yz’ + x’yz

Let’s rearrange the formula to group together the terms x’yz’ and x’yz (in green):

x’y + x’z + yz’ + x’y(z’ + z)

Since (z’ + z) is always true, we now have:

x'y + x'z + yz' + x'y

We have two occurrences of x'y, so we can drop one of them:

x'y + x'z + yz'

We know zz' is always false (0), so we can OR it with any function without affecting the outcome of that function. Let's add (OR) it to our equation:

x'y + x'z + yz' + zz'

Grouping similar terms together:

x'(y + z) + z'(y + z) =

(x' + z')(y + z) , which agrees with the minterm solution of Map 7b. Therefore,

x'z + yz' = (y + z)(x' + z') **This re-confirms our result.**

Let's triple-check our results by going in the opposite direction, starting with the maxterm result of Map 7b: (y + z)(x' + z')

Let's multiply (AND) the two parenthetical terms together:

(y + z)(x' + z') = x'y + yz' + x'z + zz'

The last term, zz', will always be false (0), so we can drop it from the equation, since zero ORed with a function does not affect the result of that function. We now have, with slight re-ordering of the terms:

x'y + x'z + yz'

Notice that z is common to two of the three terms. Let's try introducing z into the first term to see if that helps. We know that the statement (z + z') is always one, so we can "multiply" (AND) any function we want by (z + z') without affecting the results of the function. Let's apply this to the first term only and see what happens:

(x'y)(z + z') +x'z + yz' =

[x'yz + x'yz'] + x'z + yz'

Grouping similar terms together:

{x'yz + x'z} + {x'yz' + yz'} =

(y + 1)(x'z) + (x' + 1)(yz')

Whenever a function is ORed with 1, the answer is always 1, so the terms (y + 1) and (x' + 1) are always 1 and can disappear from the equation. We can rewrite the equation as:

x'z + yz' , which agrees with the maxterm solution of Map 7a. Therefore,

(y + z)(x' + z') = x'z + yz' **This confirms our result yet again.**

Clearly, it is often easier to use a truth table to check if the maxterm and minterm equations are equal, rather than resorting to Boolean algebra. This example was rather lengthy, but it covered several different methods. We found the minterm expression for F, found the maxterm expression for F, checked the results with a truth table, double-checked the results by converting the minterm expression to the maxterm expression using Boolean algebra, then triple-checked the results by converting the maxterm expression to the minterm expression using Boolean algebra.

END OF EXAMPLE

Example 3:

What function is described by Karnaugh Map 8?

F	00	01	11	10	yz
0		*1*	*1*		
1		*0*	*0*		
x					

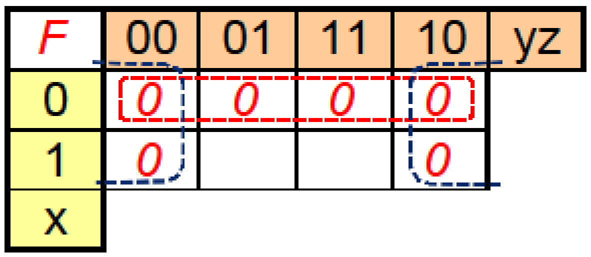
Map 8

It is not possible to determine the function described by Map 8 because it shows both ones and zeroes with blank cells. If only ones were shown with some blank cells, it would be safe to assume that the blank cells were zeroes. Conversely, if only zeroes were shown with some blank cells, it would be safe to assume that the blank cells were ones. If ones and zeroes were shown, with no blank cells, we would be able to derive this function. Since Map 8 shows ones and zeroes with blank cells, we don't know whether the blank cells are zeroes or ones, so we can't derive this function.

END OF EXAMPLE

Example 4:

What is the function described by Karnaugh Map 9a?

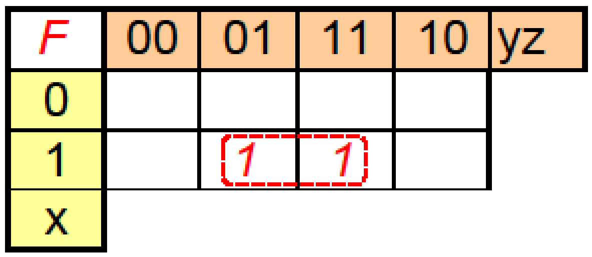


F	00	01	11	10	yz
0	0	0	0	0	
1	0			0	
x					

$$F = (x)(z) = xz$$

Map 9a

The maxterms defined by Map 9a are x (grouped in red) and z (grouped in blue). When these two maxterms are combined by ANDing them together (to get a product of sums), we get the function F = xz. Consider the same function as a minterm expression (sum of products) described by Karnaugh Map 9b.



F	00	01	11	10	yz
0					
1		1	1		
x					

$$F = xz$$

Map 9b

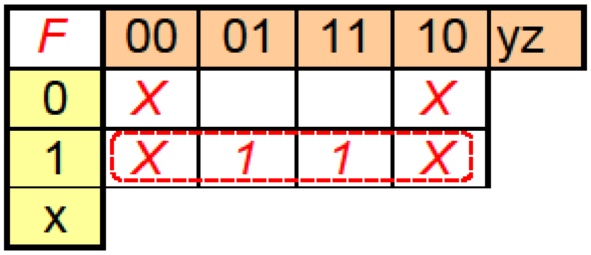
The minterm described in Map 9b gives us an output function of F = xz, which agrees with the results of Map 9a.

END OF EXAMPLE

"Don't Care" Conditions:

It is not unusual to have input and output conditions on a truth table or Karnaugh map that will never occur. A typical scenario is the BCD to seven-segment LED display logic shown later in Example 6.

A "don't care" condition could represent an input that will never occur or an output that has no effect on what we're trying to accomplish. Consider the function illustrated by Map 10. This is similar to Map 9b, but some "don't care" conditions have been added in place of some of the zeroes. Each "don't care" condition is indicated by an X in the appropriate cell.



F	00	01	11	10	yz
0	X			X	
1	X	1	1	X	
x					

F = x

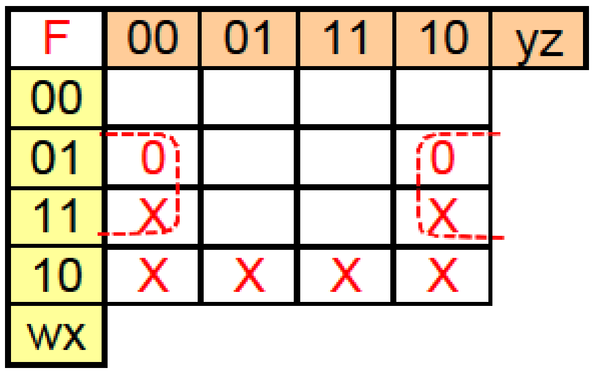
Map 10

It can be seen from the red grouping in Map 10 that this function can be completely expressed by the function F = x. As will be discussed later, we don't have to use all of the "don't care" conditions.

It is also true that Map 10 represents the function F = xz, but it is usually better to go with the simplest result, which is F = x. It is useful to remember, however, that there could be more than one correct answer to a Karnaugh map.

It would be a simple matter to picture the two zeroes above the two ones and determine that the maxterm equivalent of this function is also F = x by forming a 4-cell horizontal group using the two zeroes and the two "don't care" conditions on the same row.

Let’s try a slightly larger Karnaugh map with “don’t care” conditions, as illustrated in Map 11a.



F	00	01	11	10	yz
00					
01	0			0	
11	X			X	
10	X	X	X	X	
wx					

Maxterms:

$F = (x' + z)$

Map 11a

It can be seen from Map 11a that this new function can be expressed as the maxterm (x’ + z), based on the four-cell grouping surrounded by the red shape. Notice that there are some “don’t care” conditions that have not been included in the red shape. When we are deriving maxterms, we must cover all of the zeroes, but we only use the “don’t care” conditions that actually help to simplify the logic. We do not have to cover all of the “don’t care” conditions – we can take them or leave them.

Likewise, when we are deriving minterms, we must cover all of the ones, but we use only those “don’t care” conditions that help us, and we ignore the rest of the “don’t care” conditions.

We have already discussed how the left-most and right-most columns of a Karnaugh map are logically adjacent to each other (see Map 4b and Figure 2), so it is now time to say that the top and bottom rows of a Karnaugh map are also adjacent to each other, as illustrated in Figure 4 and Figure 5.

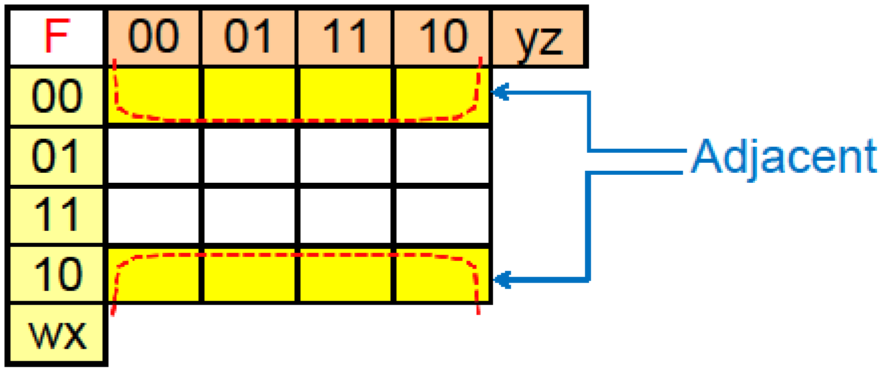


Figure 4

Figure 4 shows that the top and bottom rows (shaded in yellow) are actually adjacent to each other, and the largest possible grouping of these cells is shown in the two halves of the red shape. As is required, when wrapping around from the top row to the bottom row, only one of the input variables will change. This is more evident in Figure 5, where we can see that going from cell 0011 on the top row to cell 1011 on the bottom row will only change one input bit (the w bit), which goes from a 0 to a 1.

The red path shown on Figure 5 is the path of the reflected Gray code listed in Figure 3.

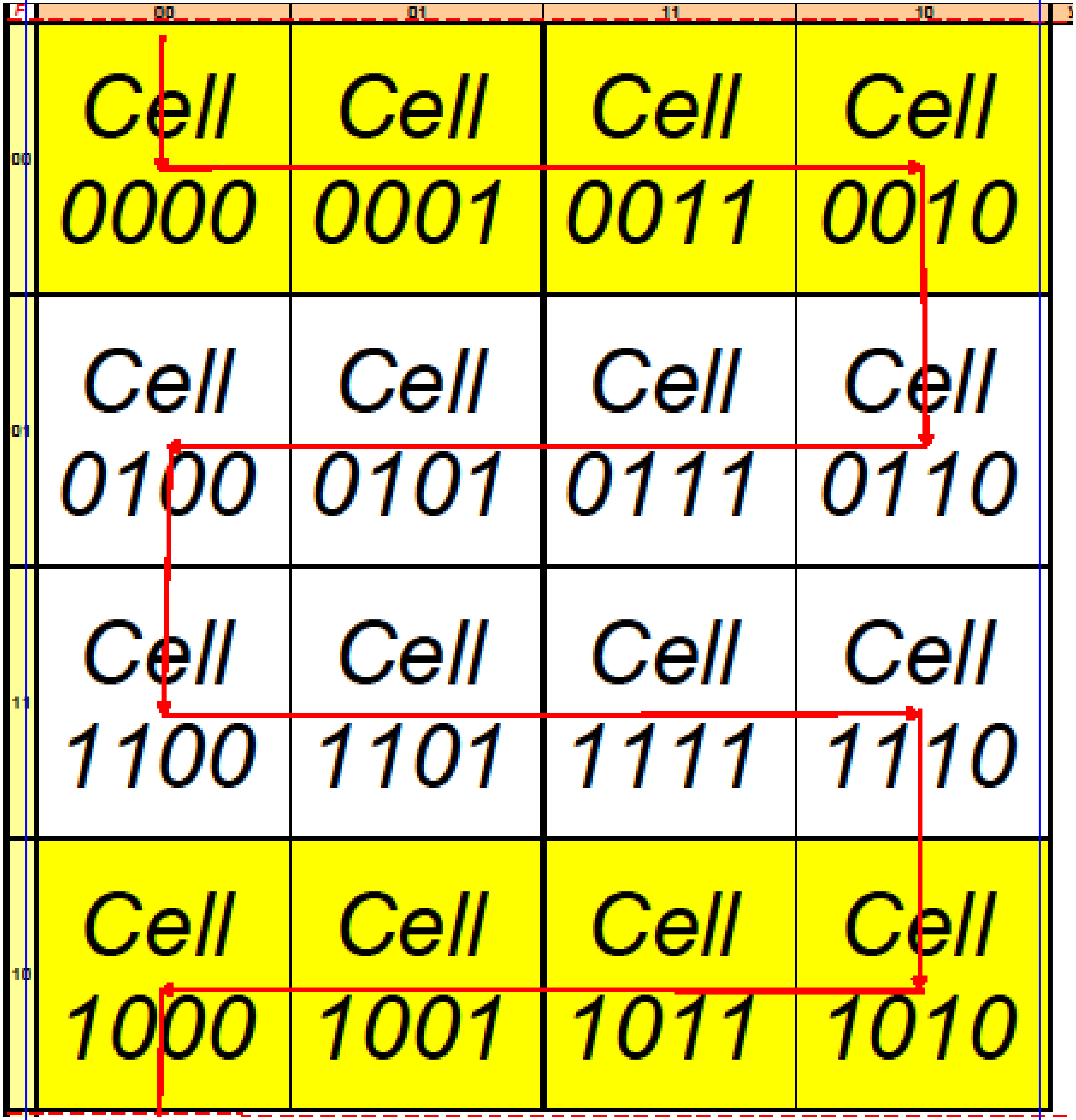
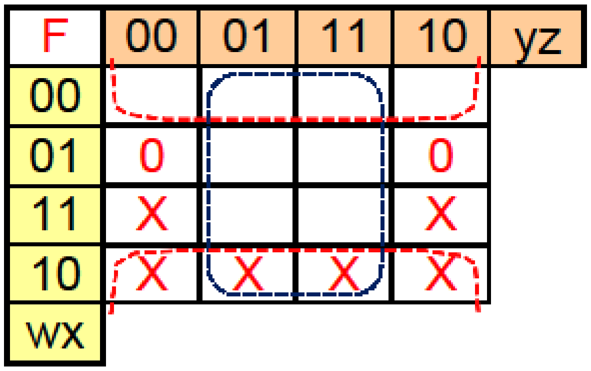

00
01
11
10
00
Cell 0000
Cell 0001
Cell 0011
Cell 0010
01
Cell 0100
Cell 0101
Cell 0111
Cell 0110
11
Cell 1100
Cell 1101
Cell 1111
Cell 1110
10
Cell 1000
Cell 1001
Cell 1011
Cell 1010

Figure 5

Using scissors, cut Figure 5 along the dashed red lines and wrap it around an aluminum 12-ounce beverage can. Line up the yellow-shaded cells so they are adjacent, top row touching bottom row. When you are satisfied that the top row is adjacent to the bottom row, turn the cut-out 90 degrees of angle to line up the left-most column adjacent with the right-most column, as we did with Figure 2. Cell 0000 is adjacent to cells 0001, 0010, 0100, and 1000, since only one digit changes when going from cell 0000 to any of those cells.

Using this information let's re-examine Map 11a. We've already solved for the maxterm expression, so let's derive the minterm expression, as illustrated in Map 11b. Imagine that there is a one (1) in each empty cell.



F	00	01	11	10	yz
00					
01	0			0	
11	X			X	
10	X	X	X	X	
wx					

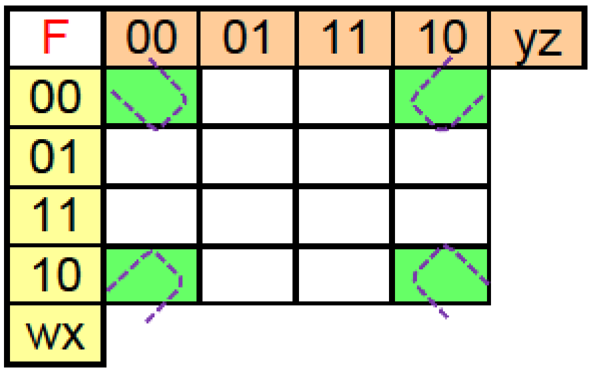
Minterms:

F = x' + z

Map 11b

Map 11b groups the cells with ones (1s) to derive the minterm expression. Since there are zeroes (0s) and "don't cares" (Xs) shown, it is safe to assume that all of the blank cells are ones (1s). The minterm expression shown in Map 11b matches the maxterm expression derived from Map 11a, as we would expect.

Given the top row to bottom row adjacency illustrated in Figure 4 and Figure 5, and the property of the left-most and right-most columns being adjacent, as illustrated in Figure 2, it is a logical extension to say that the four corners of the Karnaugh map can be used to form a 4-cell group, if the values in the cells are appropriate. This is shown in Figure 6.



Grouping of Four Corners

Figure 6

Figure 6 illustrates that the four corners of a Karnaugh map can be used to form a group of four cells, as shown by the green-shaded cells. If the grouping defined by the four pieces of the purple shape (the green cells) on Figure 6 were filled with ones, or ones and “don’t cares”, the minterm function would be x’z’. If the four indicated cells were filled with zeroes, or zeroes and “don’t cares”, the maxterm function would be (x + z), which is the opposite or complement of x’z’.

Example 5:

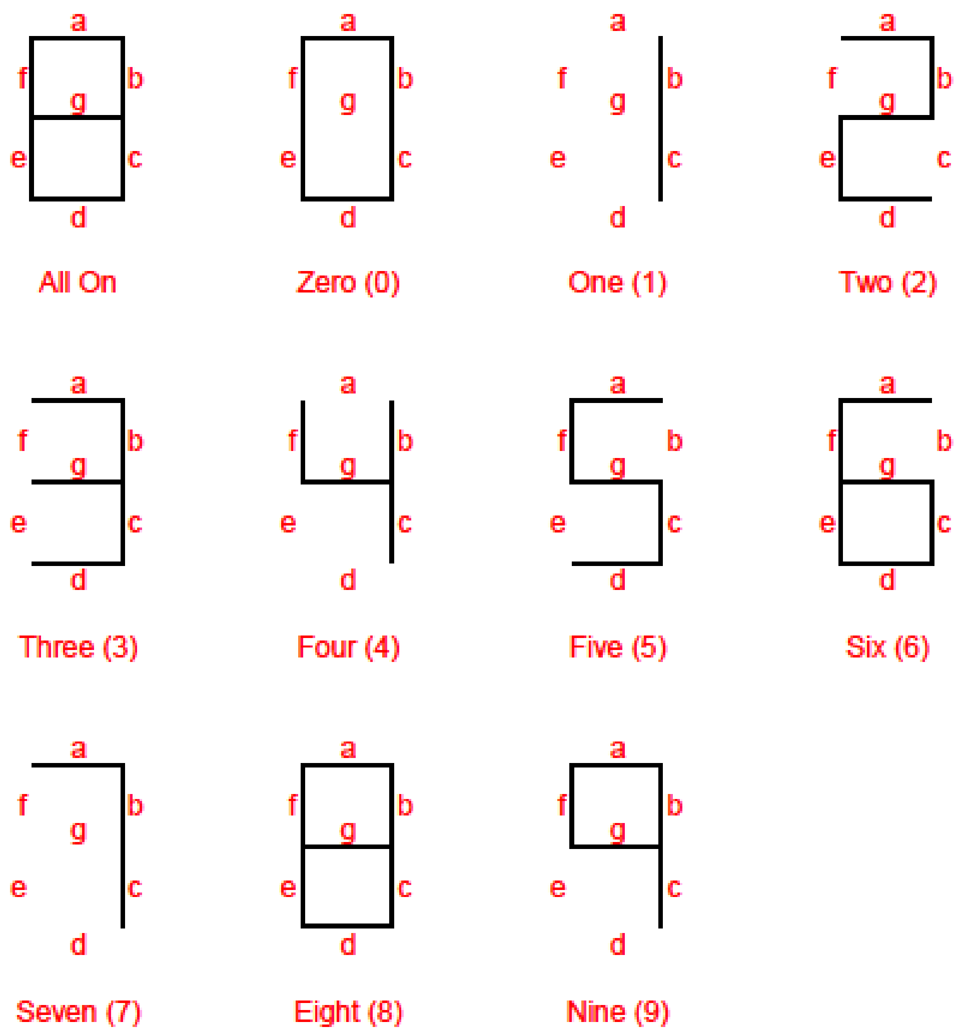
Even though many numerical displays nowadays are LCD (liquid crystal display), there are still some applications for LED (light-emitting diode) displays. This well-worn example of digital logic optimization is a useful application of the topics discussed above. This example assumes that there aren’t any “don’t care” conditions, but the example directly following this one will incorporate some given “don’t care” conditions.

Figure 7 shows the seven segments (a through g) of a typical seven-segment single-digit LED display. To display the decimal number four (4), segments b, c, f, & g must be turned on. To display the decimal number six (6), all of the segments except segment b must be turned on. There is usually a trailing decimal point to the lower right of a typical seven-segment display, near segment c, but we are not utilizing that function in this example.

Looking at it from the point of view of one of the seven segments, the question it may ask would be, when should I turn on? Segment a will turn on when the binary equivalent of the decimal numbers 0, 2, 3, 5, 6, 7, 8, and 9 are present. Likewise, segment g will turn on whenever the binary equivalent of the decimal numbers 2, 3, 4, 5, 6, 8, and 9 are present. These turn-on conditions are captured for each segment in Truth Table 12. Notice that there are 16 different input combinations, but we’ll only be displaying 10 of those input conditions (0 through 9 Decimal). We will design the logic in this example to make the display blank (no segments turned on) whenever the binary equivalents of decimal numbers 10, 11, 12, 13, 14, and 15 are present.

Truth Table 12 shows each of the seven segments as a separate column, with a 1 for turning on that segment and a 0 for turning off that segment. Each segment's column in Truth Table 12 has been converted to its own Karnaugh map in Figure 9a through Figure 9g. For example, Figure 9e shows the Karnaugh map for segment e, which only turns on for four input conditions. The minterm expression is shown under the Karnaugh maps in Figures 9a through 9g. The colors of the individual minterms correspond to the colors of the shapes that enclose the appropriate ones (1s) on the Karnaugh map (this will be explained again later).

The maxterm expression for each segment's function is also shown in Figures 9a through 9g, located below the minterm expression, but the derivation of the maxterm expression is left for the reader to enjoy.



BCD Seven-Segment LED Display (0 – 9)

Figure 7

Dec.	Inputs				Outputs (Seven Segments)						
#	w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10	1	0	1	0	0	0	0	0	0	0	0
11	1	0	1	1	0	0	0	0	0	0	0
12	1	1	0	0	0	0	0	0	0	0	0
13	1	1	0	1	0	0	0	0	0	0	0
14	1	1	1	0	0	0	0	0	0	0	0
15	1	1	1	1	0	0	0	0	0	0	0

BCD Seven-Segment LED Display
Decimal Digits 0 through 9, Blank Display for Invalid Inputs

Truth Table 12

As was already mentioned, Truth Table 12 is derived from Figure 7. Using segment f as an example, this segment is required to turn on whenever the logic we are designing receives as an input the binary equivalent of decimal values 0, 4, 5, 6, 8, and 9. As can be seen on Truth Table 12, the column for segment f has an output of 1 for each of those six inputs. This slice of the truth table appears in Figure 9f, along with the associated Karnaugh map.

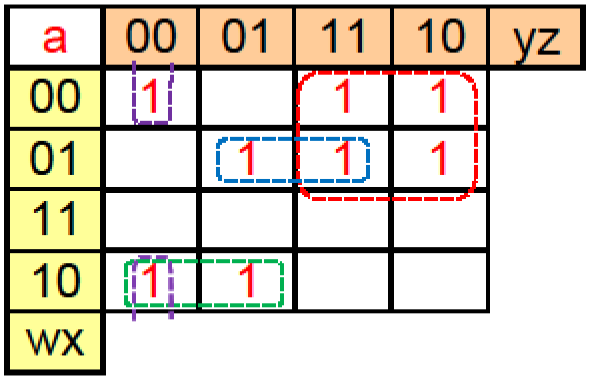
The Karnaugh maps have thus far been presented with binary numbers (such as 1110) for cell addresses, but Figure 8 shows a 4-bit Karnaugh map with decimal addresses in the cells. This should make it easier to go from the slice of Truth Table 12 to the associated Karnaugh map for each of the seven segments in Figures 9a through 9g.

F	00	01	11	10	yz
00	0	1	3	2	
01	4	5	7	6	
11	12	13	15	14	
10	8	9	11	10	
wx					

Decimal Equivalents of Binary Addresses

Figure 8

Dec.	O/P
#	a
0	1
1	0
2	1
3	1
4	0
5	1
6	1
7	1
8	1
9	1
10	0
11	0
12	0
13	0
14	0
15	0



a	00	01	11	10	yz
00	1		1	1	
01		1	1	1	
11					
10	1	1			
wx					

Minterms:

a = w'y + w'xz + wx'y' + x'y'z'

Another Equally-Optimized Minterm Expression:

a = w'y + w'xz + wx'y' + w'x'z'

Maxterms:

a = (w' + x')(w' + y')(x' + y + z)(w + x + y +z')

BCD Segment a Truth Table and Karnaugh Map

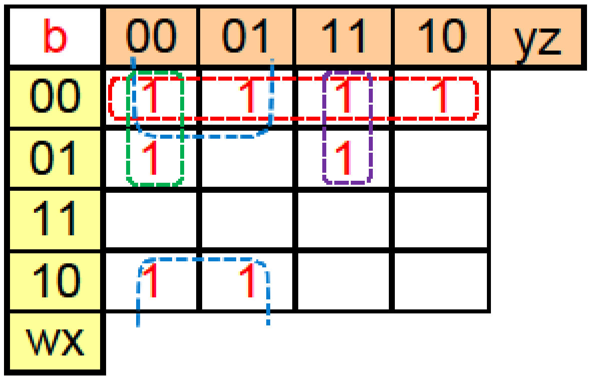
Figure 9a

Our first task after filling in the ones or zeroes on a Karnaugh map is to look for the largest possible group that covers 2, 4, 8, or 2^n cells. In Figure 9a, the largest minterm grouping that we can use is the 4 cells in the red shape. This red shape corresponds to the red minterm of w'y. The rest of the groups are composed of 2 cells each. Luckily, we aren't stuck with any single-cell minterm groups on this particular Karnaugh map. The blue shape covering the 2-cell group in Figure 9a corresponds to the blue minterm of w'xz, and so on for the green shape and purple shape.

Notice that a second minterm expression is shown in Figure 9a. This second minterm expression is just as optimized as the first one and was derived by grouping cell 0000 with cell 0010, instead of with cell 1000.

To derive the maxterm expression, imagine (or write) a zero in each of the blank cells. As can be seen in the maxterm solution in Figure 9a above, the author started with the group of four cells on the wx = 11 row for the first maxterm, then the 2x2 group of four cells in the bottom right-hand corner for the second maxterm, then the group of 2 cells for the third maxterm, then the remaining isolated single-cell group for the last maxterm. Let's move on to segment b.

Dec.	O/P
#	b
0	1
1	1
2	1
3	1
4	1
5	0
6	0
7	1
8	1
9	1
10	0
11	0
12	0
13	0
14	0
15	0



b	00	01	11	10	yz
00	1	1	1	1	
01	1		1		
11					
10	1	1			
wx					

Minterms:

$b = w'x' + x'y' + w'y'z' + w'yz$

Maxterms:

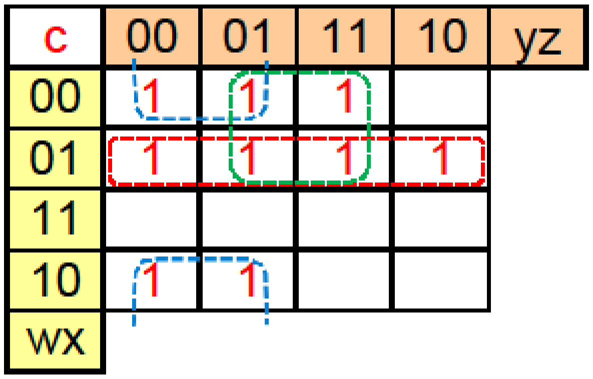
$b = (w' + x')(w' + y')(x' + y + z')(x' + y' + z)$

BCD Segment b Truth Table and Karnaugh Map

Figure 9b

Compare the complexity of Figure 9b without "don't care" conditions to the simplicity of Figure 12b with "don't care" conditions.

Dec.	O/P
#	c
0	1
1	1
2	0
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	0
11	0
12	0
13	0
14	0
15	0



c	00	01	11	10	yz
00	1	1	1		
01	1	1	1	1	
11					
10	1	1			
wx					

Minterms:

$c = w'x + x'y' + w'z$

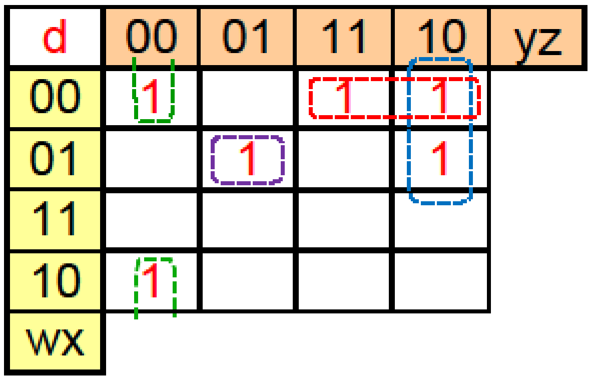
Maxterms:

$c = (w' + x')(w' + y')(x + y' + z)$

BCD Segment c Truth Table and Karnaugh Map

Figure 9c

Dec.	O/P
#	d
0	1
1	0
2	1
3	1
4	0
5	1
6	1
7	0
8	1
9	0
10	0
11	0
12	0
13	0
14	0
15	0



d	00	01	11	10	yz
00	1		1	1	
01		1		1	
11					
10	1				
wx					

Minterms:

$d = w'x'y + w'yz' + x'y'z' + w'xy'z$

Maxterms:

$d = (w'+x')(w'+y')(x'+y+z)(x+y+z')(x'+y'+z')$

BCD Segment d Truth Table and Karnaugh Map

Figure 9d

Notice the lonely 1 in the purple shape in Figure 9d. It is not adjacent to any other 1s, so it must be expressed by using all four of the input variables.

Dec.	O/P
#	e
0	1
1	0
2	1
3	0
4	0
5	0
6	1
7	0
8	1
9	0
10	0
11	0
12	0
13	0
14	0
15	0

e	00	01	11	10	yz
00	1			1	
01				1	
11					
10	1				
wx					

Minterms:
e = x'y'z' + w'yz'

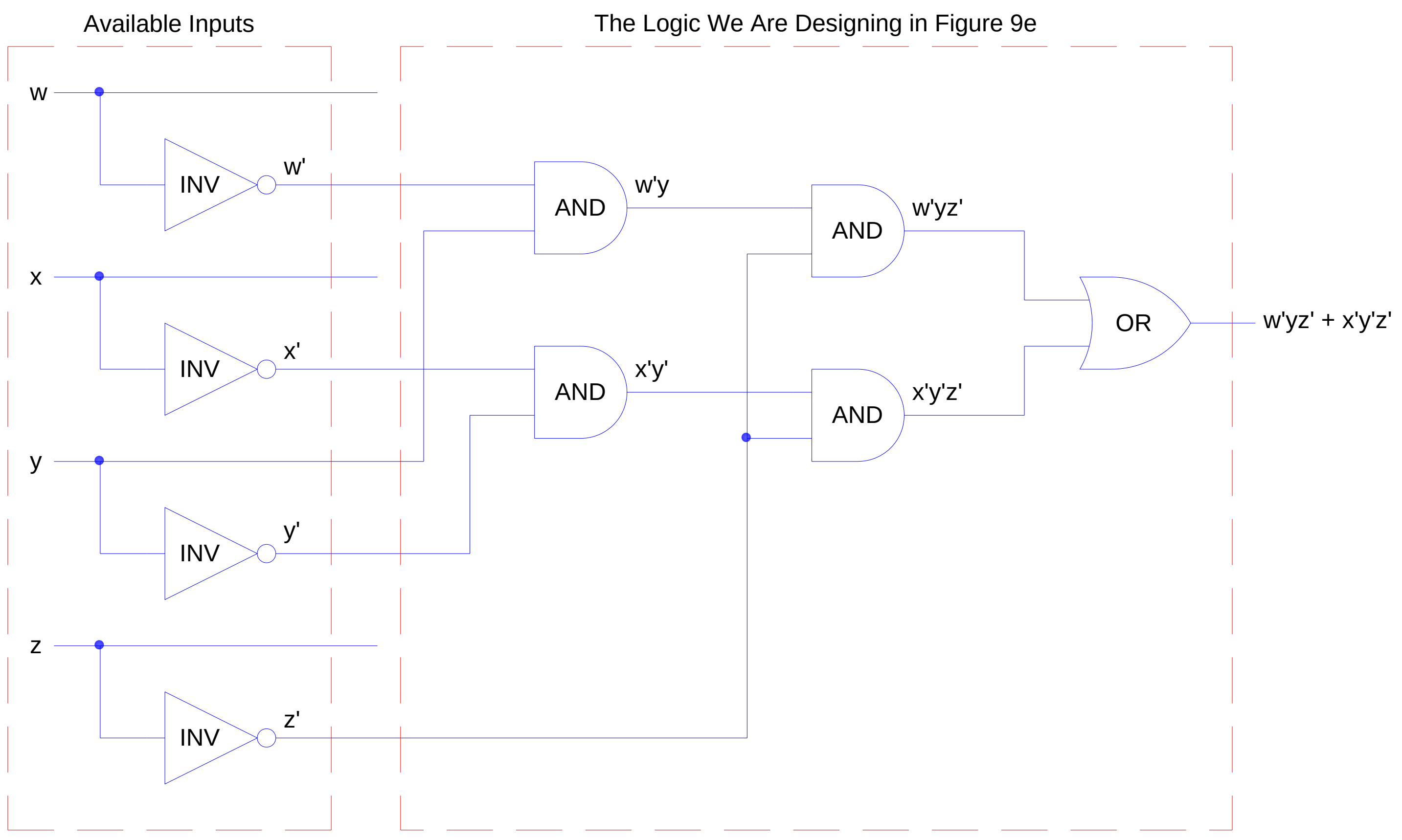
Maxterms:
e = (z')(w' + y')(x' + y)

BCD Segment e Truth Table and Karnaugh Map

Figure 9e

Although the logic for segment e shown in Figure 9e is relatively simple, it is even simpler in Figure 12e, when "don't care" conditions are included.

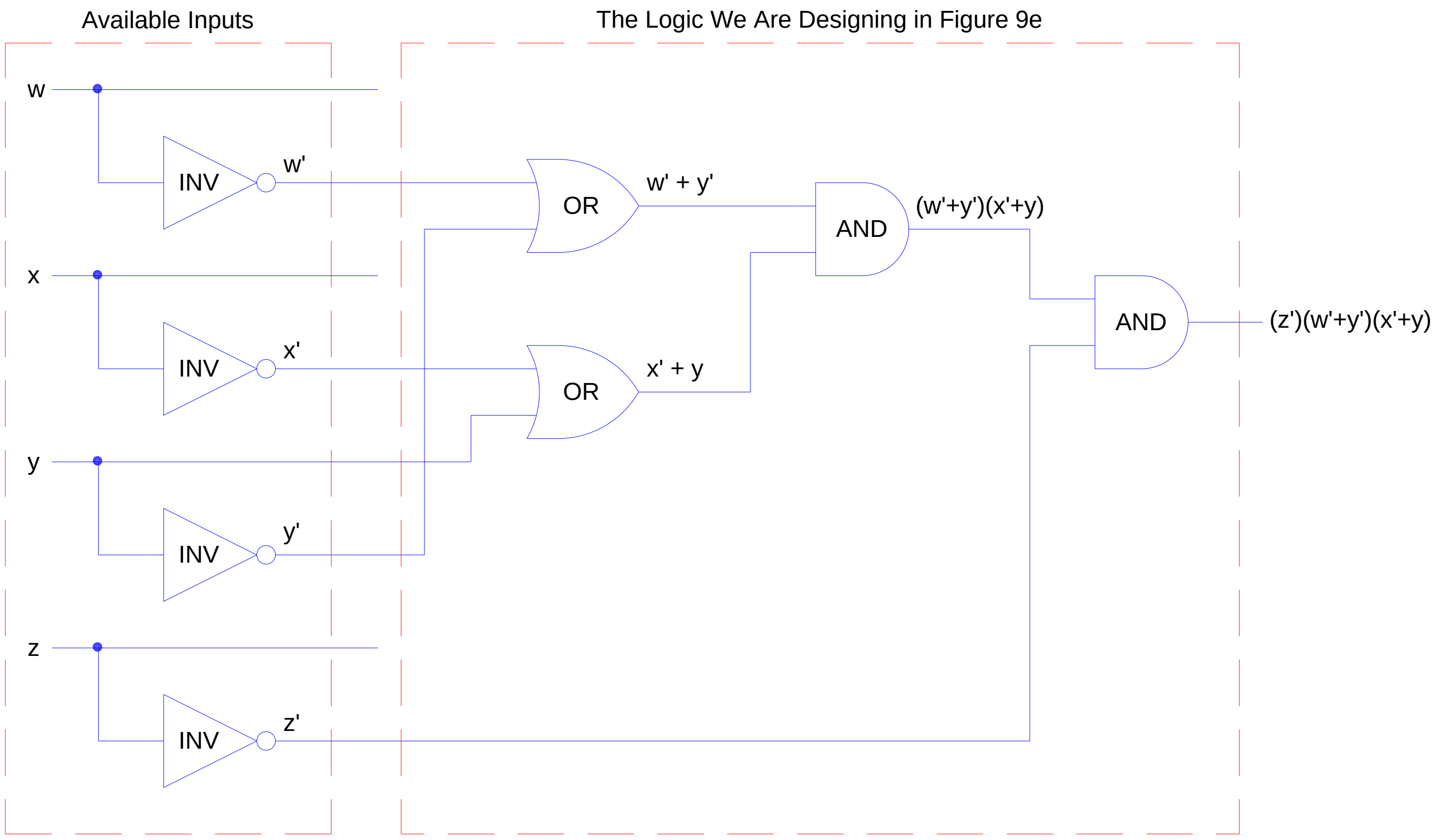
Let’s look at how to implement the logic design shown in Figure 9e, since the logic for segment e is one of the simpler of the seven segments. Figure 10a shows how the minterms would be implemented using AND & OR gates.



Segment e
Minterm Implementation with Gates
Figure 10a

Segment e
Minterm Implementation with Gates
Figure 10a

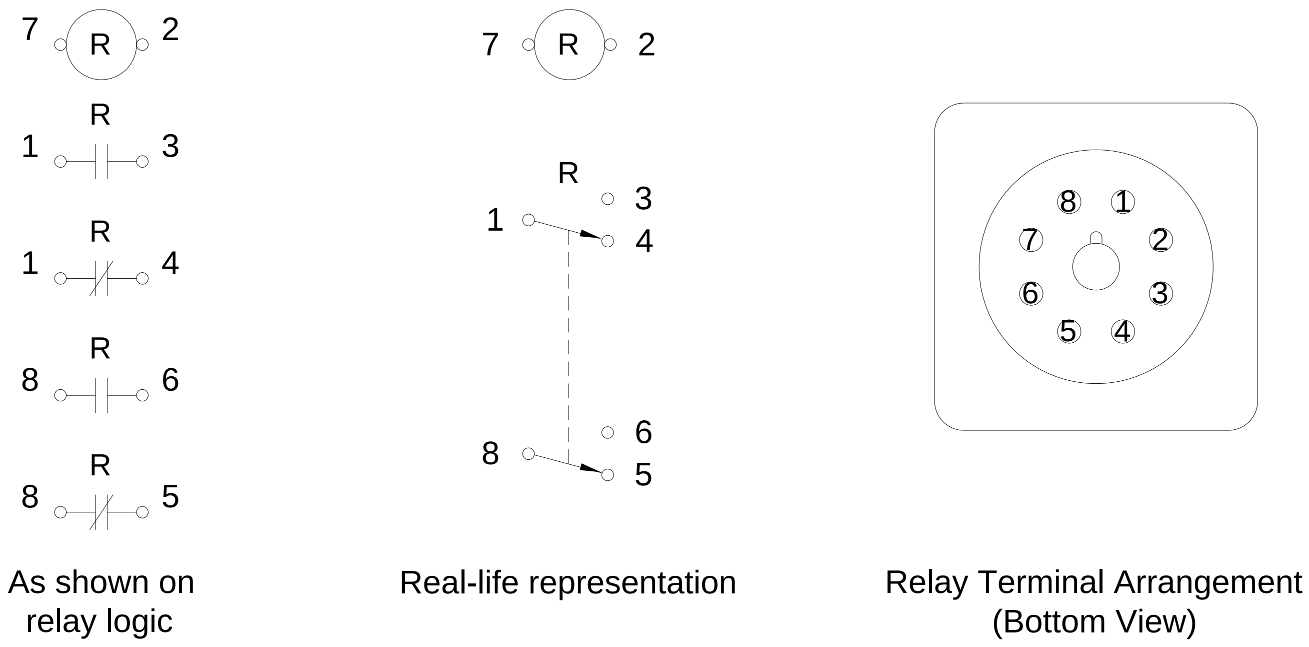
Figure 10b shows how the maxterms would be implemented using AND & OR gates.



Segment e
Maxterm Implementation with Gates
Figure 10b

Segment e
Maxterm Implementation with Gates
Figure 10b

Figure 10c shows how the minterms would be implemented using relay logic. But first, Figure 11 describes the type of relay we will be considering in this document. The relay coil is between terminals 2 and 7. The first set of contacts is 1 & 3 [Normally Open (NO)] and 1 & 4 [Normally Closed NC)], with terminal 1 obviously being a common point between the NO and NC positions. The second set of contacts is 8 & 6 [NO] and 8 & 5 [NC].



General Purpose DPDT Relay (8-Pin or Octal) Terminal Designations

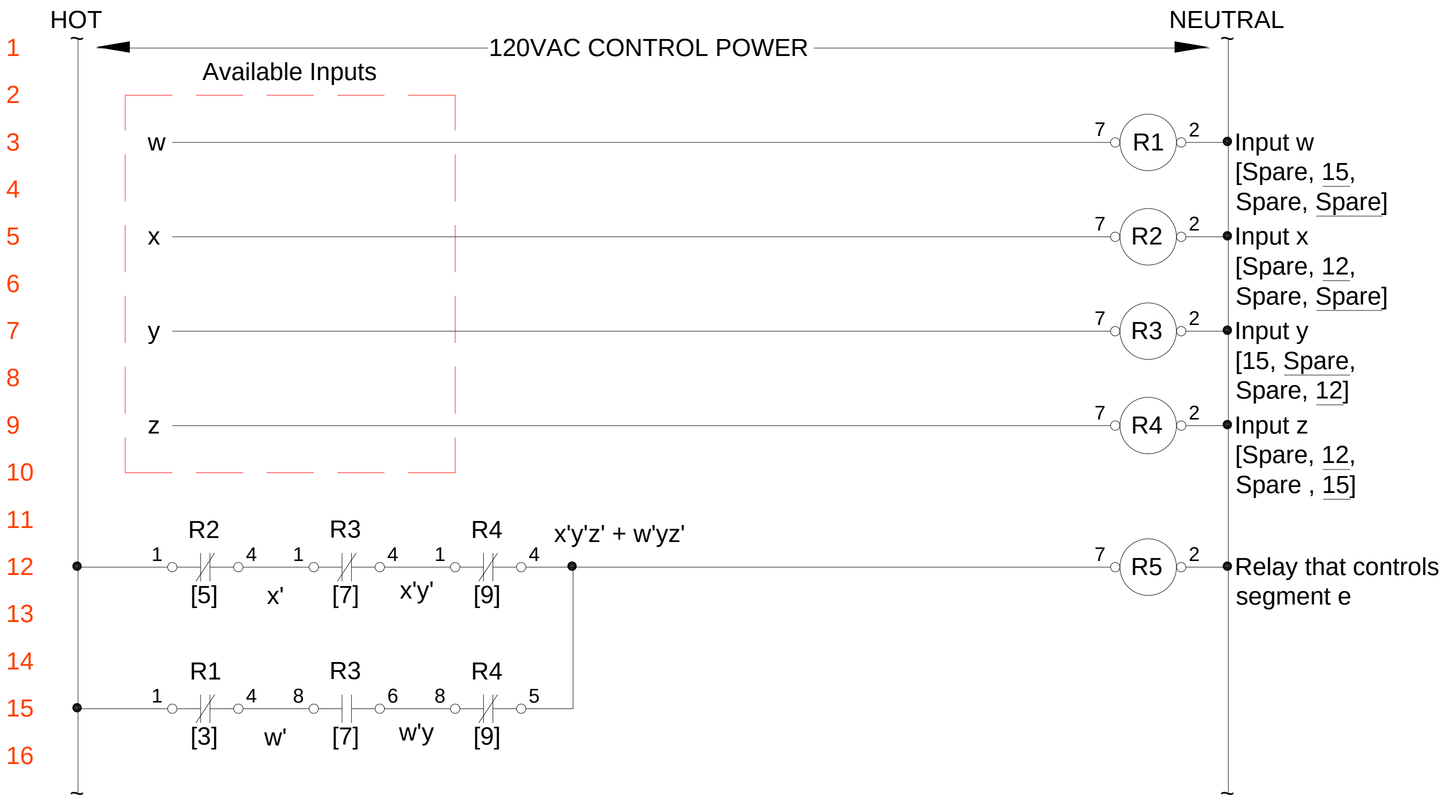
Figure 11

General Purpose DPDT Relay (8-Pin or Octal) Terminal Designation Example Figure 11

The four input variables (wxyz) are used to turn on relays R1 through R4 respectively. Input variable w turns on relay R1 when w is true, so a normally open (NO) R1 contact will represent w and a normally closed (NC) R1 contact will represent w'.

The numbers in [brackets] in Figure 10c and other relay logic drawings represent the rung number of the relay coils or contacts. For example, looking at the normally closed (NC) contact of relay R2 on rung number 12 of that figure, we see that the coil of relay R2 is at rung number 5. If we look at the coil for relay R2 at rung number 5, we see that one of its NC contacts is used at rung number 12 and that the rest of the contacts are spare (not used). The underlined <u>12</u> & <u>Spare</u> represent the NC contacts, while the non-underlined rung number or Spare represent the normally open (NO) contacts. In some documents, the rung numbers of the NC contacts are represented by a line drawn over the top of the rung numbers, rather than underneath, similar to the way some documents show the complement of Boolean variables or expressions by drawing a line on top of them. Underlining the NC contacts is used in this document because it is easier.

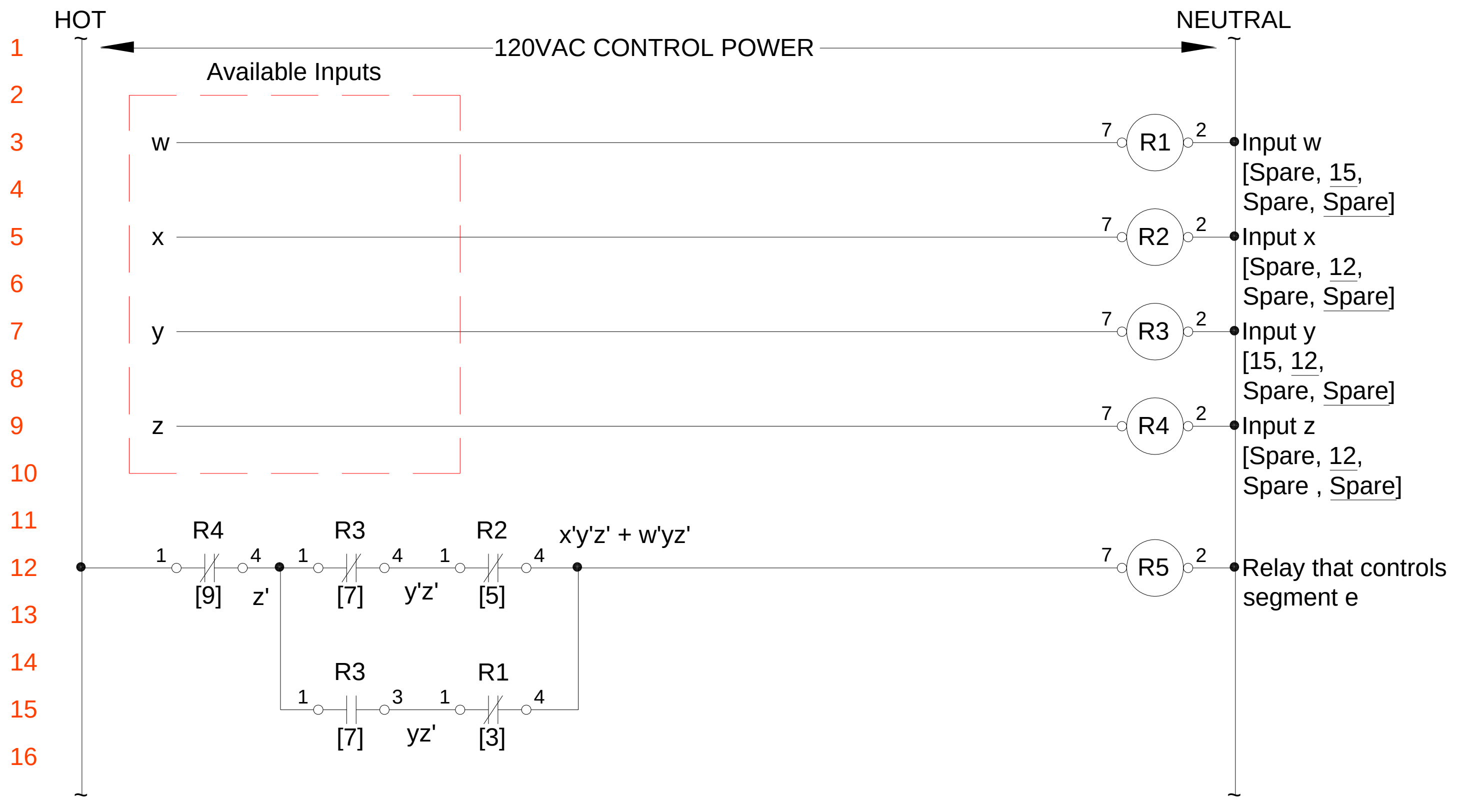
Notice in Figure 10c that the NO contact of R3 is in rung number 12, while the NC contact of this relay is in rung number 15. Since there is no direct connection between these two contacts in the wiring, we have to use both of the sets of contacts of R3, instead of using just one set of contacts, as shown in Figure 10c-1. In other words, we have to state [15, Spare, Spare, 12] at the coil for R3 in Figure 10c, which takes up both sets of contacts of R3, but we can state [15, 12, Spare, Spare] in Figure 10c-1, since one set of contacts is still available. In Figure 10c-1, the R3 relay contacts have been moved over to the right, allowing the 1-3-4 set of contacts of R3 to be wired such that the NO & NC contacts have a direct, common connection.


HOT
NEUTRAL
120VAC CONTROL POWER
Available Inputs
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
w
x
y
z
R1
R2
R3
R4
R5
Input w [Spare, 15, Spare, Spare]
Input x [Spare, 12, Spare, Spare]
Input y [15, Spare, Spare, 12]
Input z [Spare, 12, Spare , 15]
Relay that controls segment e
x'y'z' + w'yz'
x'
x'y'
w'
w'y
[5]
[7]
[9]
[3]
[7]
[9]

Segment e
Minterm Implementation with Relays
Figure 10c

Segment e
Minterm Implementation with Relays
Figure 10c

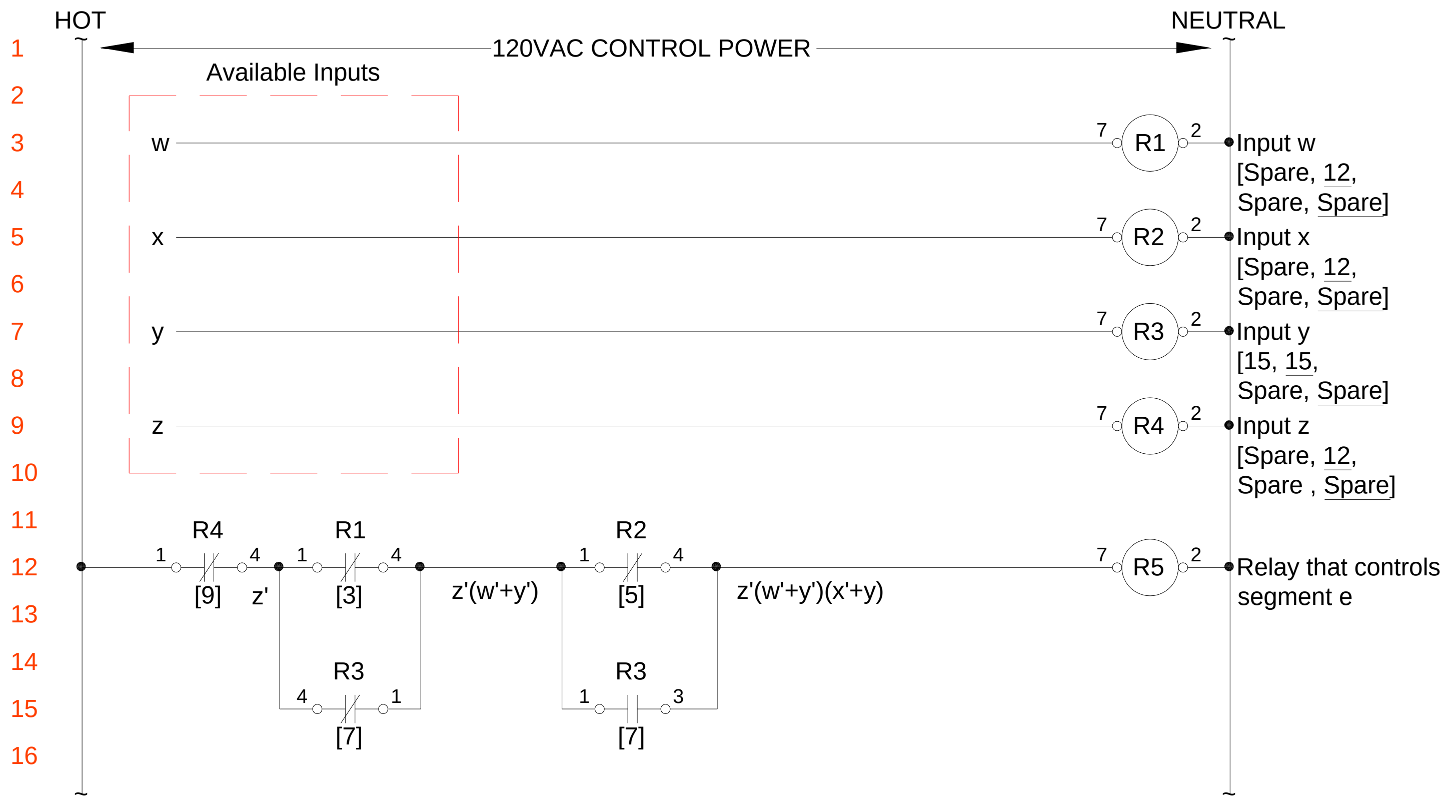
There are additional improvements that we could make to Figure 10c. We could have wired the relay logic in Figure 10c to use fewer sets of contacts and still produce the same function by using the z' (R4) NC relay contact only once, since z' is common to both minterms, then putting the y (R3) relay NO & NC contacts second, using the common point of the 1-3-4 set of contacts, as mentioned previously. This wiring optimization (as opposed to logic optimization) is shown in Figure 10c-1.



Improved Segment e
Minterm Implementation with Relays
Figure 10c-1

Improved Segment e
Minterm Implementation with Relays
Figure 10c-1

We have looked at how to implement a minterm solution using relay logic, now let's consider maxterms. Figure 10d shows how the maxterms would be implemented using relay logic.


HOT
NEUTRAL
120VAC CONTROL POWER
Available Inputs
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
w
x
y
z
R1
R2
R3
R4
R5
Input w [Spare, 12, Spare, Spare]
Input x [Spare, 12, Spare, Spare]
Input y [15, 15, Spare, Spare]
Input z [Spare, 12, Spare , Spare]
Relay that controls segment e
[9]
z'
[3]
[7]
z'(w'+y')
[5]
z'(w'+y')(x'+y)

Segment e
Maxterm Implementation with Relays
Figure 10d

Segment e
Maxterm Implementation with Relays
Figure 10d

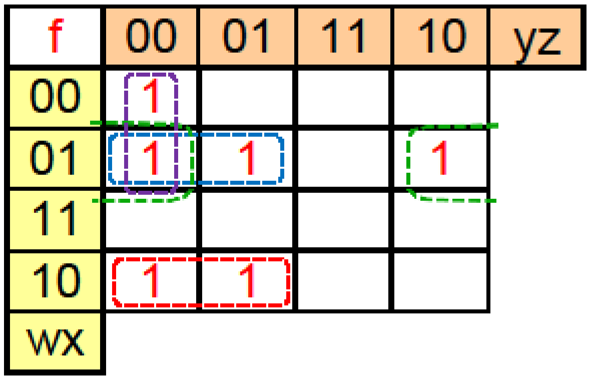
Notice in Figure 10d that we turned the NC contact of R3 around so that it can share its common connection (terminal 1) with the NO contact of R3, thus completely freeing up the other NO/NC set of contacts of R3.

Even though the first NO contact for R2 is described as Spare in Figure 10d, it is actually already partially committed, since the common terminal #1 is already connected to the function z'(w' + y'). If we used the so-called spare NO contact of R2, it would only be able to pass through a function called xz'(w' + y'). This contact is not available for any other function as presently wired in Figure 10d.

Relay Logic –vs- Gate Logic:
When using relay logic to implement a function, two separate functions can be ORed together simply by wiring the appropriate relay contacts together, as was done for the function x'y'z' + w'yz' in Figure 10c. When using gate logic to implement a function, however, it is not advisable to directly wire together the outputs of gates because those outputs might have a tendency to fight with each other (because of their electronic output circuitry), which could result in unpredictable output states. When using gate logic, it is recommended to use an OR gate to perform the OR function.

Let’s get back to deriving the minterm and maxterm expressions for the rest of the segments.

Dec.	O/P
#	f
0	1
1	0
2	0
3	0
4	1
5	1
6	1
7	0
8	1
9	1
10	0
11	0
12	0
13	0
14	0
15	0



Minterms:

$f = wx'y' + w'xy' + w'xz' + w'y'z'$

Maxterms:

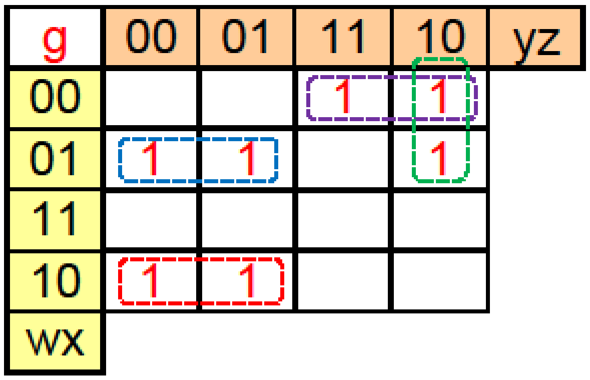
$f = (w' + x')(y' + z')(x + y')(w + x + z')$

BCD Segment f Truth Table and Karnaugh Map

Figure 9f

It is easy to see that the purple group in Figure 9f could have been re-shaped to make the last minterm x’y’z’, instead of w’y’z’. Both results would be equally optimized.

Dec.	O/P
#	g
0	0
1	0
2	1
3	1
4	1
5	1
6	1
7	0
8	1
9	1
10	0
11	0
12	0
13	0
14	0
15	0



g	00	01	11	10	yz
00			1	1	
01	1	1		1	
11					
10	1	1			
wx					

Minterms:

$$g = wx'y' + w'xy' + w'yz' + w'x'y$$

Maxterms:

$$g = (w' + x')(w' + y')(w + x + y)(x' + y' + z')$$

BCD Segment g Truth Table and Karnaugh Map

Figure 9g

This concludes our design of a BCD to seven segment decoder without "don't care" conditions.

END OF EXAMPLE

The next example assumes that we won't receive any inputs that aren't appropriate. In other words, the unseen upstream control circuitry won't send our logic any useless inputs. (See Appendix – Malfunction of "Don't Care" Conditions for what would happen if the upstream control circuitry were to malfunction and start sending useless inputs to our logic.) In the next example, our BCD display will display the decimal numbers 0 through 9, so we shouldn't receive any inputs for decimal numbers 10, 11, 12, 13, 14, and 15. Those non-valid inputs are "don't care" conditions because the upstream control circuitry won't ever send us the binary equivalent of those values. Each "don't care" condition is marked with an X on the truth tables and Karnaugh maps.

Example 6:

Truth Table 13 is basically the same as Truth Table 12, but "don't care" conditions have been added for inputs 10 through 15 decimal.

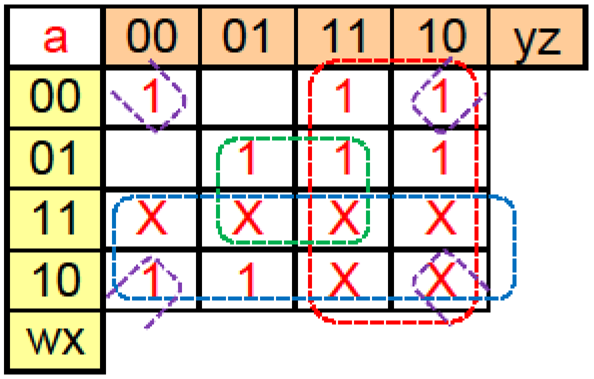
Dec.	Inputs				Outputs (Seven Segments)						
#	w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10	1	0	1	0	X	X	X	X	X	X	X
11	1	0	1	1	X	X	X	X	X	X	X
12	1	1	0	0	X	X	X	X	X	X	X
13	1	1	0	1	X	X	X	X	X	X	X
14	1	1	1	0	X	X	X	X	X	X	X
15	1	1	1	1	X	X	X	X	X	X	X

BCD Seven-Segment LED Display with "Don't Cares"

Truth Table 13

As was done in the previous example, Figures 12a through 12g represent each of the seven segments. Each of these figures has a slice of Truth Table 13 pertaining to that particular segment, as well as a Karnaugh map based on that slice of truth table. The minterms are color-coded to match the groupings shown on the Karnaugh map. The derivation of the minterm expressions is illustrated, but the derivation of the maxterm expressions is left as an exercise for the reader.

Dec.	O/P
#	a
0	1
1	0
2	1
3	1
4	0
5	1
6	1
7	1
8	1
9	1
10	X
11	X
12	X
13	X
14	X
15	X



a	00	01	11	10	yz
00	1		1	1	
01		1	1	1	
11	X	X	X	X	
10	1	1	X	X	
wx					

Minterms:

a = y + w + xz + x'z'

Maxterms:

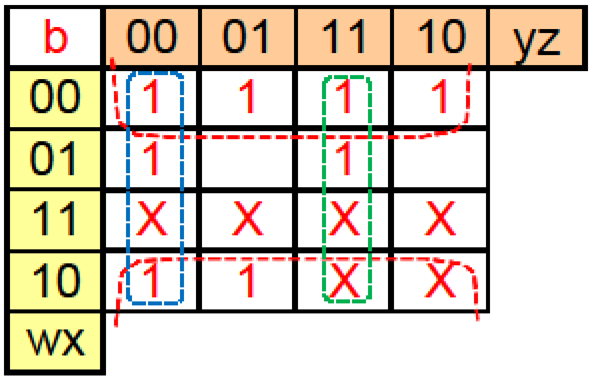
a = (x' + y + z)(w + x + y + z')

BCD Segment a Truth Table and Karnaugh Map with "Don't Cares"

Figure 12a

Compare Figure 12a to Figure 9a. The design with the "don't care" conditions in Figure 12a is simpler (fewer terms) than the design without the "don't care" conditions Figure 9a. This is true for the maxterm expression, as well as the minterm expression.

Dec. #	O/P b
0	1
1	1
2	1
3	1
4	1
5	0
6	0
7	1
8	1
9	1
10	X
11	X
12	X
13	X
14	X
15	X



b	00	01	11	10	yz
00	1	1	1	1	
01	1		1		
11	X	X	X	X	
10	1	1	X	X	
wx					

Minterms:
b = x' + y'z' + yz

Maxterms:
b = (x' + y + z')(x' + y' + z)

BCD Segment b Truth Table and Karnaugh Map with "Don't Cares"

Figure 12b

To reinforce what was previously discussed, let's apply Boolean algebra to the results of Figure 12b to see if the maxterm expression is logically equivalent to the minterm expression.

(x' + y + z')(x' + y' + z)

"Multiplying" (ANDing) the two parenthetical expressions together:

x'x' + x'y' + x'z + x'y + yy' + yz + x'z' + y'z' + zz'

Taking out yy' & zz' and grouping similar terms together:

x' + x'y' + x'y + x'z + x'z'+ yz + y'z'

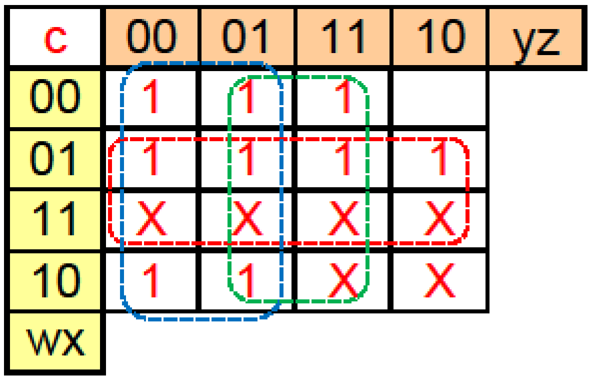
x’ + x’(y’ + y) + x’(z + z’) + yz + y’z’

x’ + x’ + x’ + yz + y’z’

x’ + yz + y’z’

This confirms that the maxterm expression is logically equivalent to the minterm expression in Figure 12b. Let’s get back on task with segment c.

Dec.	O/P
#	c
0	1
1	1
2	0
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	X
11	X
12	X
13	X
14	X
15	X



c	00	01	11	10	yz
00	1	1	1		
01	1	1	1	1	
11	X	X	X	X	
10	1	1	X	X	
wx					

Minterms:
c = x + y' + z

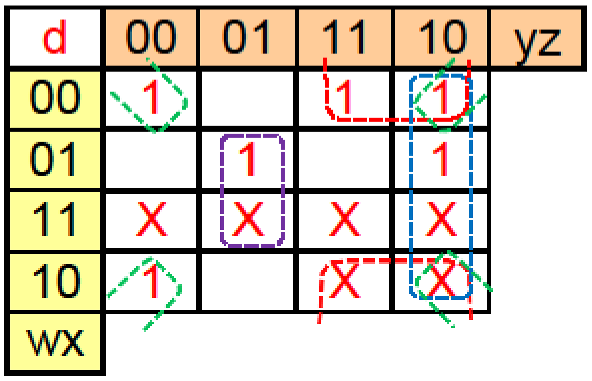
Maxterms:
c = (x + y' + z)

BCD Segment c Truth Table and Karnaugh Map with “Don’t Cares”

Figure 12c

Compare Figure 12c with "don't care" conditions to Figure 9c without "don't care" conditions. Clearly, the inclusion of "don't care" conditions can provide opportunities for significant logic reduction and optimization.

Dec.	O/P
#	d
0	1
1	0
2	1
3	1
4	0
5	1
6	1
7	0
8	1
9	0
10	X
11	X
12	X
13	X
14	X
15	X



d	00	01	11	10	yz
00	1		1	1	
01		1		1	
11	X	X	X	X	
10	1		X	X	
wx					

Minterms:

$d = x'y + yz' + x'z' + xy'z$

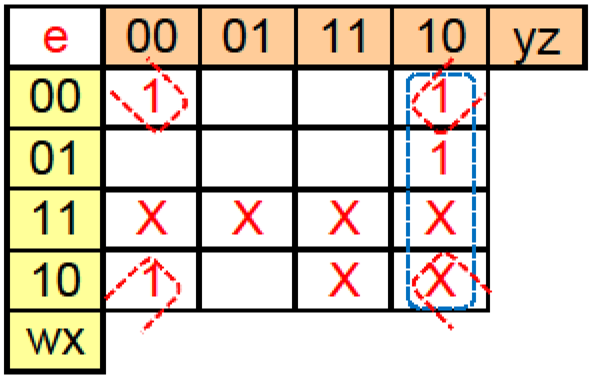
Maxterms:

$d = (x' + y + z)(x + y + z')(x' + y' + z')$

BCD Segment d Truth Table and Karnaugh Map with "Don't Cares"

Figure 12d

Dec.	O/P
#	e
0	1
1	0
2	1
3	0
4	0
5	0
6	1
7	0
8	1
9	0
10	X
11	X
12	X
13	X
14	X
15	X



e	00	01	11	10	yz
00	1			1	
01				1	
11	X	X	X	X	
10	1		X	X	
wx					

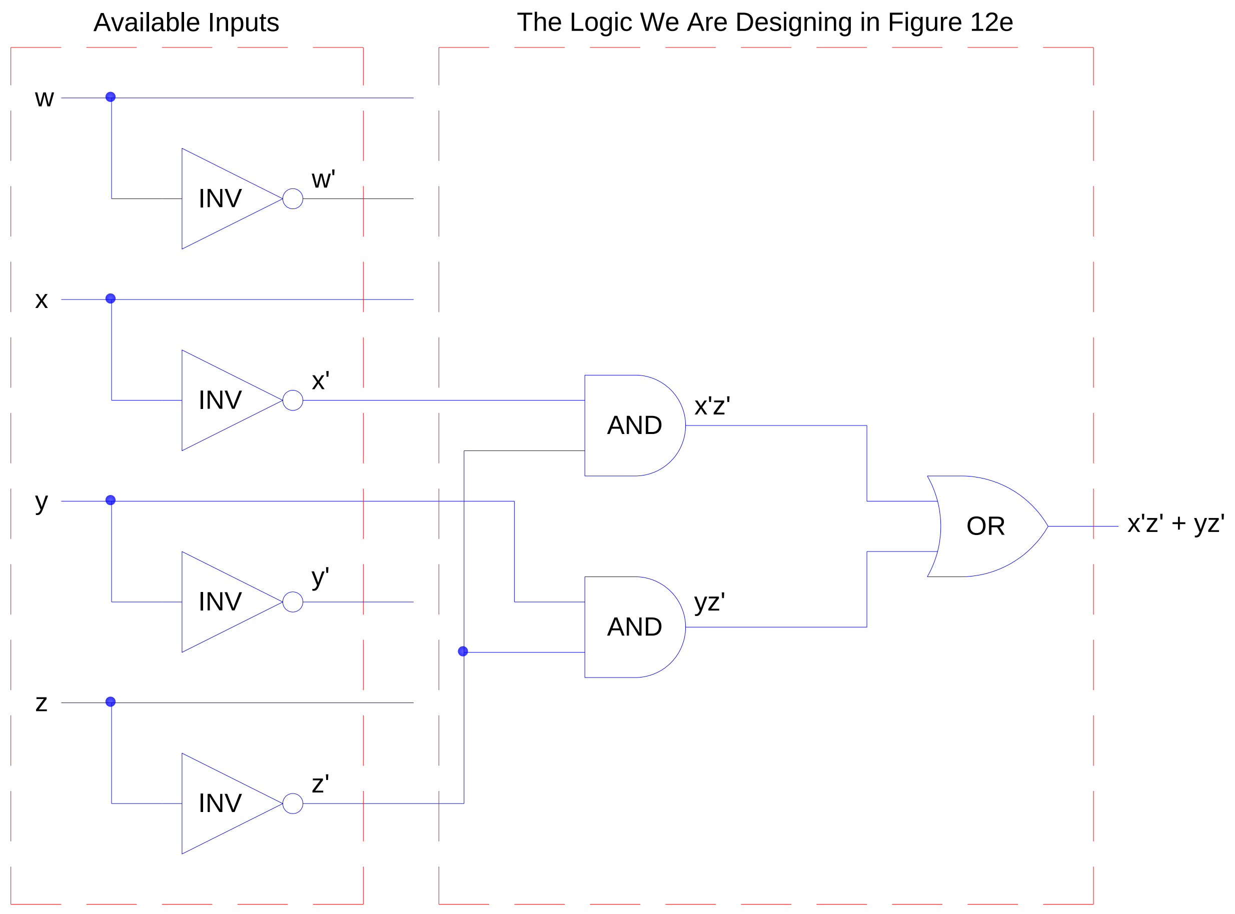
Minterms:
e = x'z' + yz'

Maxterms:
e = (z')(x' + y)

BCD Segment e Truth Table and Karnaugh Map with "Don't Cares"

Figure 12e

It is easy to see that the minterm and maxterm expressions in Figure 12e are logically equivalent. Figure 13a shows the minterm implementation of the segment e function using logic gates. Compare this to Figure 10a, which was the segment e function without "don't care" conditions.



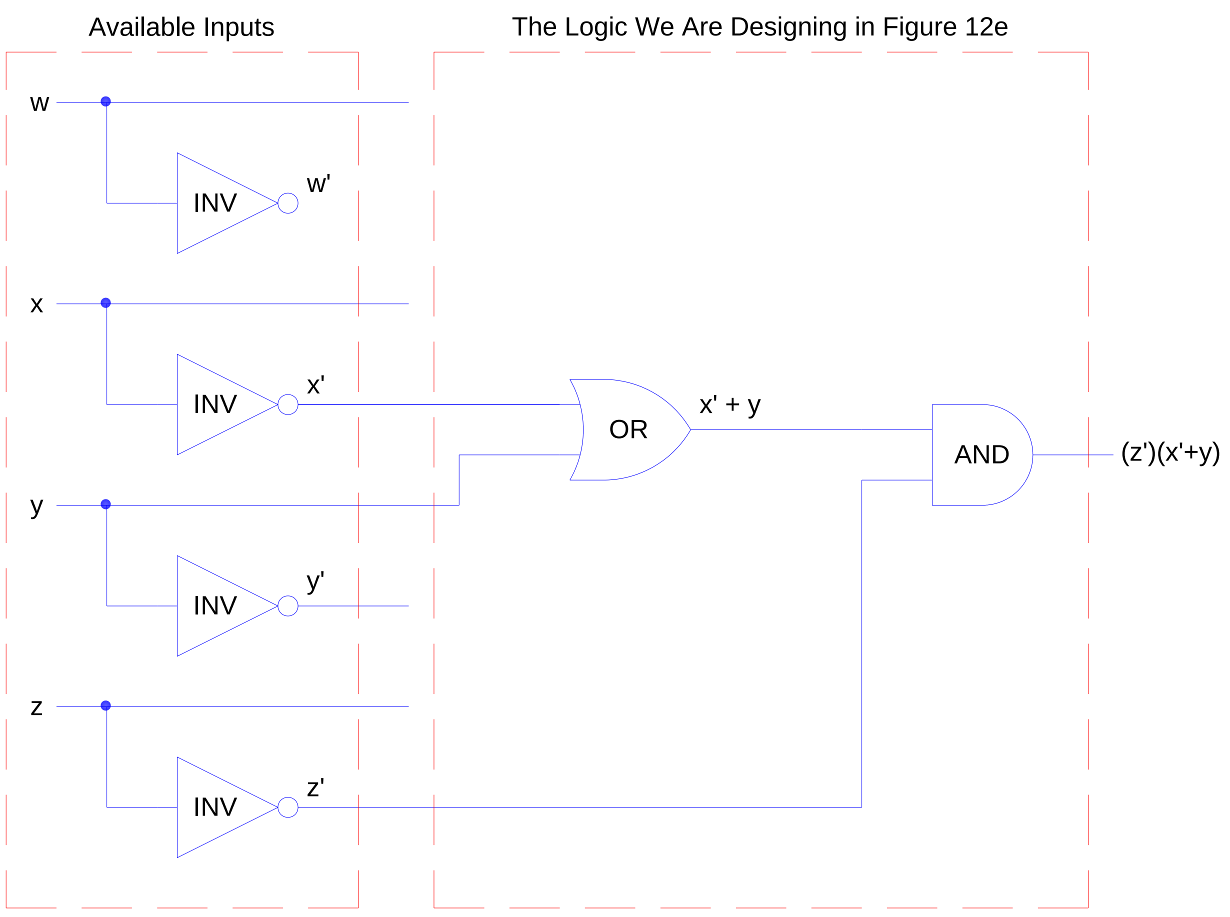
Segment e with "Don't Care" Conditions
Minterm Implementation with Gates
Figure 13a

Segment e with "Don't Cares"
Minterm Implementation with Gates
Figure 13a

Notice that there are two fewer gates in Figure 13a, when compared to Figure 10a.

Figure 13b shows the maxterm implementation of the segment e function using logic gates. Compare this to Figure 10b, which was the segment e function without "don't care" conditions.

We might have noticed in Figure 13a that we could easily factor z' out of the minterm expression to get z'(x' + y), which would only require two gates and which is also the maxterm expression.

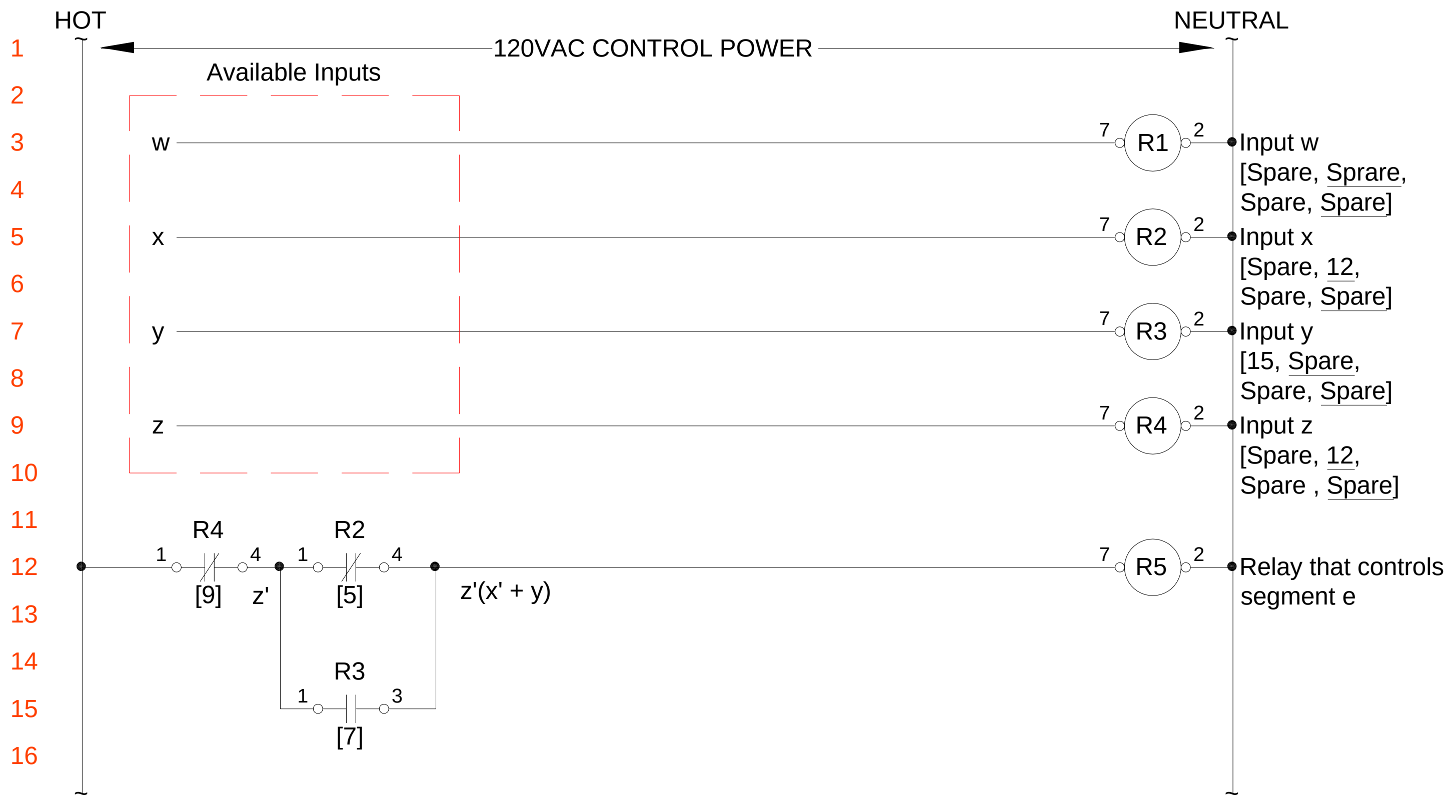


Segment e with "Don't Care" Conditions
Maxterm Implementation with Gates
Figure 13b

Segment e with "Don't Care" Conditions
Maxterm Implementation with Gates
Figure 13b

Notice that there are two fewer gates in Figure 13b, when compared to Figure 10b.

The minterm expression for segment e with "don't care" conditions is so simple and so easily converted to the maxterm expression that only one relay implementation drawing is presented to cover both the minterm and the maxterm expression, as shown in Figure 13c.



Segment e with "Don't Care" Conditions
Implementation with Relays

Figure 13c

Segment e with "Don't Cares"
Implementation with Relays

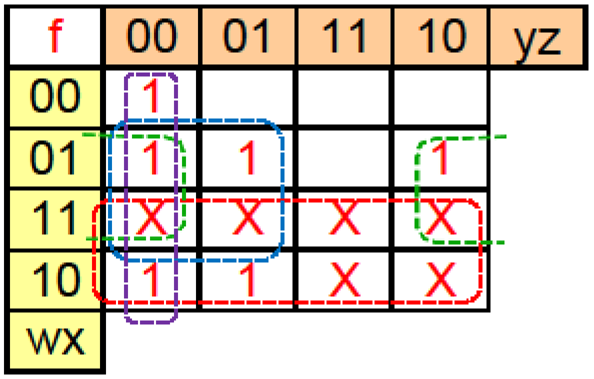
Figure 13c

Notice that Figure 13c uses a fraction of the relay contacts called for in Figures 10c and 10d. Notice also in Figure 13c that relay R1 (w input) is not even required for the implementation of segment e with "don't care" conditions.

Have You Noticed:

If a grouping of 8 cells appears on a 4-bit Karnaugh map, it results in a minterm or maxterm of only 1 input variable for that particular grouping (see Figure 12c). If a grouping of 4 cells appears on a 4-bit Karnaugh map, it results in a minterm or maxterm of two input variables for that grouping (see Figure 12e). In general, if i is the number of input variables (meaning there are 2^i output cells) and 2^c is the number of cells covered by a grouping, it will result in a minterm or maxterm that contains (i – c) input variables. For example, if a grouping of 4 cells (2^2 cells) appears on a 3-bit (xyz) Karnaugh map (with 2^3 output cells), it will result in a mintern or maxterm of 3 - 2 = 1 input variable (such as x) for that particular grouping (see Map 10).

Dec.	O/P
#	f
0	1
1	0
2	0
3	0
4	1
5	1
6	1
7	0
8	1
9	1
10	X
11	X
12	X
13	X
14	X
15	X



f	00	01	11	10	yz
00	1				
01	1	1		1	
11	X	X	X	X	
10	1	1	X	X	
wx					

Minterms:

$f = w + xy' + xz' + y'z'$

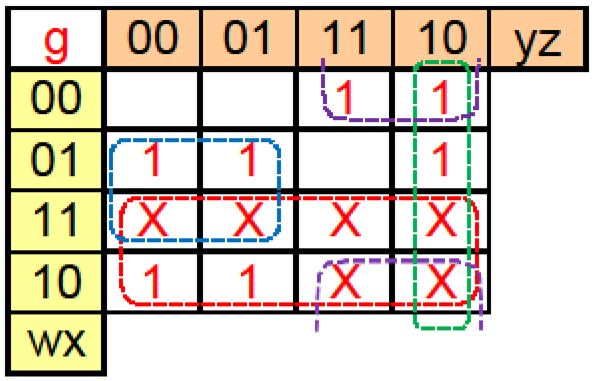
Maxterms:

$f = (x + y')(y' + z')(w + x + z')$

BCD Segment f Truth Table and Karnaugh Map with "Don't Cares"

Figure 12f

Dec.	O/P
#	g
0	0
1	0
2	1
3	1
4	1
5	1
6	1
7	0
8	1
9	1
10	X
11	X
12	X
13	X
14	X



g	00	01	11	10	yz
00			1	1	
01	1	1		1	
11	X	X	X	X	
10	1	1	X	X	
wx					

Minterms:

$g = w + xy' + yz' + x'y$

Maxterms:

$g = (w + x + y)(x' + y' + z')$

BCD Segment g Truth Table and Karnaugh Map with "Don't Cares"

Figure 12g

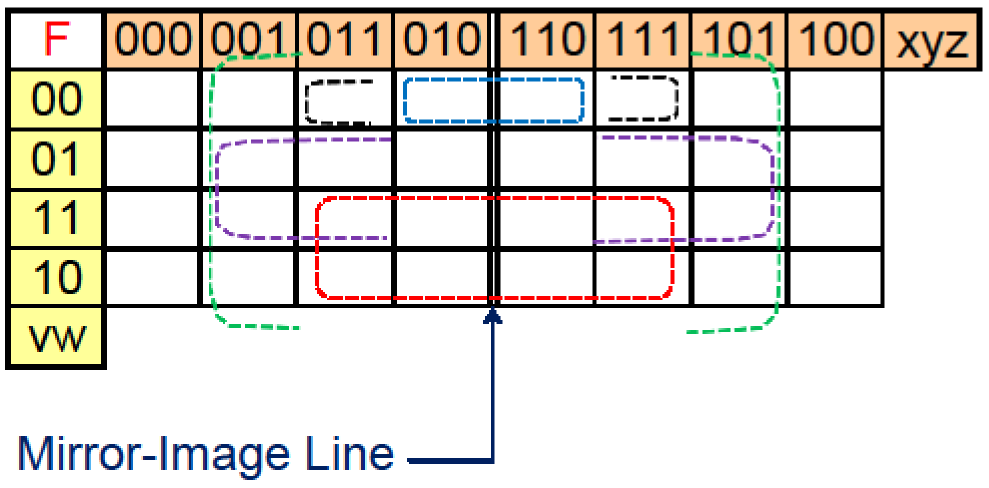
This concludes our design of a BCD to seven segment decoder with "don't care" conditions.

END OF EXAMPLE

Larger Karnaugh Maps:

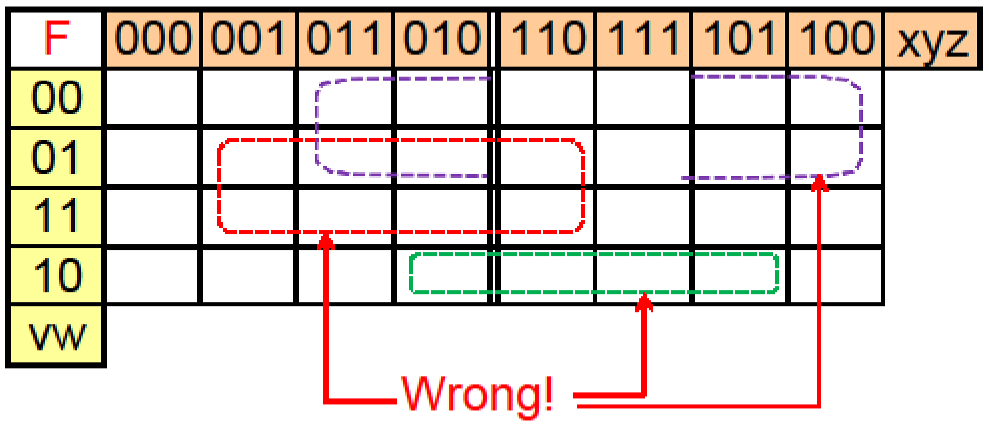
Figure 14a shows a larger Karnaugh map, this one having five input variables, but the row and column numbering system still allows only one input variable to change state when moving up/down the rows or left/right along the columns on the map.

In addition to the adjacency properties we've already discussed for 4-bit and smaller Karnaugh maps, the 5-bit Karnaugh map in Figure 14a also has the adjacency property defined by the vertical mirror-image line running through the middle of the map, as illustrated in Figure 14a. The cells facing each other when a 5-bit Karnaugh map is folded in half are logically adjacent to each other. Futhermore, if a group crosses the mirror-image line, that group must be symmetrical around the mirror-image line. Some examples of acceptable groups are shown in Figure 14a, and some examples of unacceptable (non-symmetrical around the mirror-image line) groups are shown in Figure 14b.



5-Bit Karnaugh Map with Some Appropriate Groups

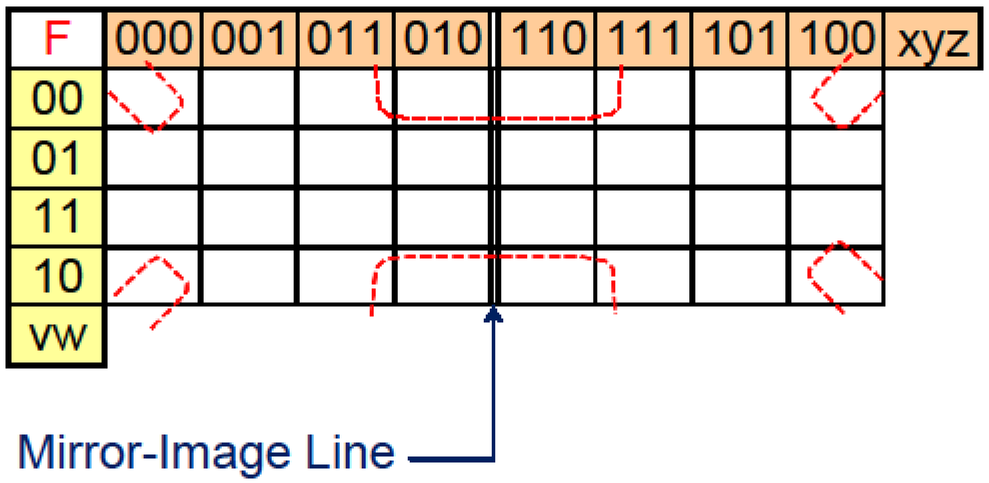
Figure 14a



5-Bit Karnaugh Map with Some Incorrect Groups

Figure 14b

Figure 14a shows some of the groupings that are peculiar to 5-bit and larger Karnaugh maps, but all of the other groupings we've discuss for 4-bit and smaller Karnaugh maps still apply. For example, the top row is still adjacent to the bottom row in Figure 14a, and the left-most column is still adjacent to the right-most column. The four corners are still adjacent to each other, and, though it's not shown on Figure 14a, cells 10000 and 10010 are adjacent, and cell 10110 is adjacent to cell 10100. You can think of a 5-bit Karnaugh map as two adjacent 4-bit Karnaugh maps. Therefore, it would be valid to have a group composed of the 8 corners (four outside corners and 4 inside corners) of the two adjacent 4-bit maps, as illustrated in Figure 14c.



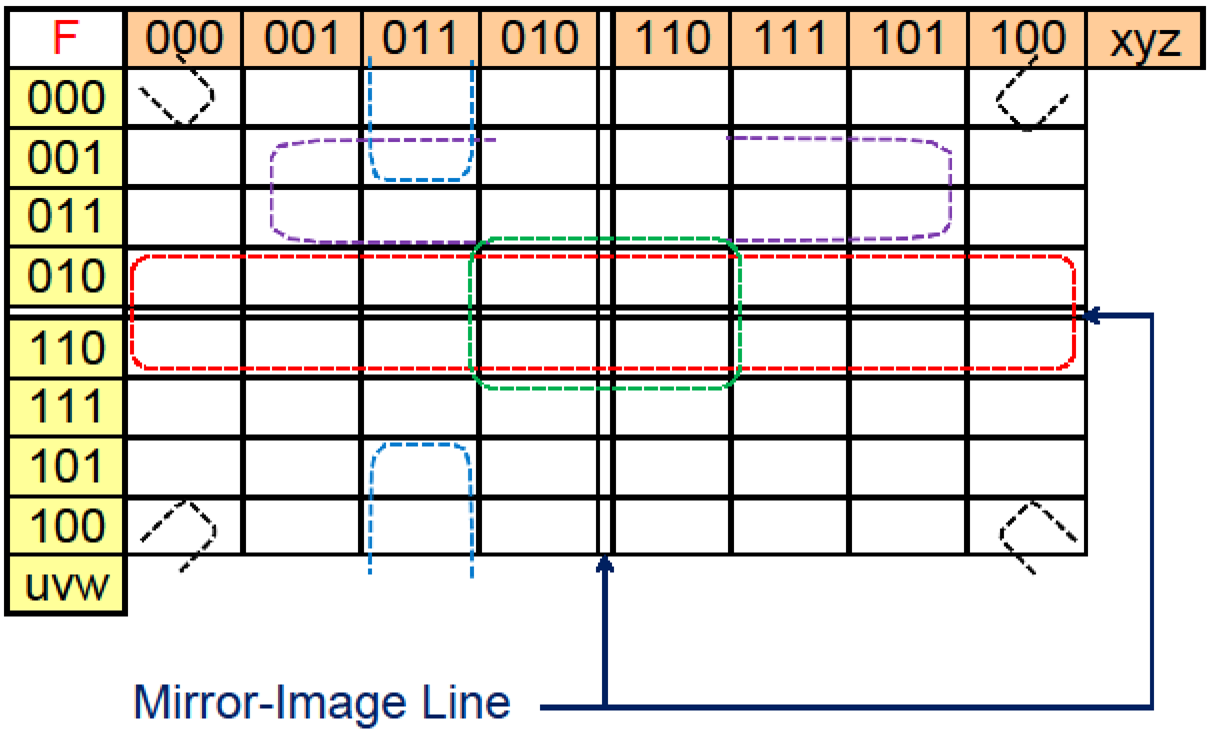
5-Bit Karnaugh Map with 8-Corner Group

Figure 14c

If the 8-corner group in Figure 14c had all 1s, or 1s and Xs, the minterm would be w'z'. If the 8-corner group in Figure 14c had all 0s, or 0s and Xs, the maxterm would be (w + z), which is the inverse or complement of w'z'.

In addition to the 5-bit Karnaugh map format shown in Figures 14a through 14c, which utilize mirror-image symmetry, there is also a 5-bit Karnaugh map format known as 'overlay', which is presented in Appendix – Overlay Type of 5-Bit and 6-Bit Karnaugh Maps.

A six input variable map (uvwxyz) would be constructed by numbering the rows in the same order as shown for the columns in Figure 14a for five input variables. This reflected Gray code must have mirror-image symmetry such that, when the resulting Karnaugh map is folded in the middle in either direction, the cells that face each other can only change by one digit. A six-bit Karnaugh map is shown in Figure 15.



6-Bit Karnaugh Map with Some Appropriate Groups

Figure 15

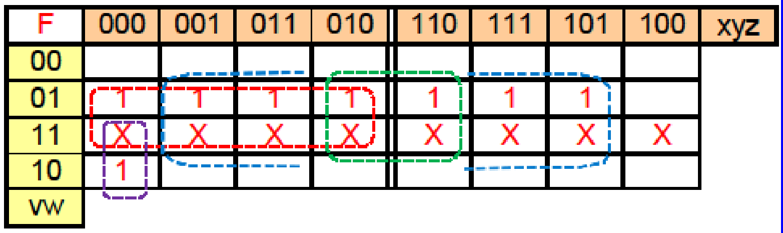
In addition to the 6-bit Karnaugh map format shown in Figure 15, which has mirror-image symmetry, there is also the 6-bit Karnaugh map known as the overlay type, as shown in Appendix – Overlay Type of 5-Bit and 6-Bit Karnaugh Maps.

Let's try an example with a 5-bit Karnaugh map and some "don't care" conditions.

Example 7:

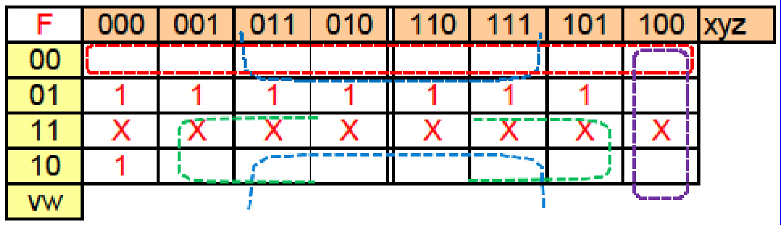
Assume that there is a large room that is full of office workers during the workday. This room has many desks, but no partitions or cubicles. There is a light mounted on the ceiling that is illuminated during working hours, but turned off during lunch and after hours. The work schedule is 8am to noon, lunch from noon to 1pm, then back to work from 1pm to 5pm. There is a master clock system that outputs a digital (binary) signal representing the time in hours, minutes, and seconds. We will only be looking at the hours portion of the time signal, not the minutes or seconds, for this design. When the workers look up at the ceiling and see that the light is on, they know they should get back to work. If they look up and see that the light is off, they know that it is 1) not time to start yet, 2) time for lunch, or 3) time to go home, depending on whether it's morning, mid-day, or afternoon. The truth table for this light is part of Figure 16 and the associated Karnaugh map is presented several times for several different solutions. Notice the "don't care" conditions for decimal numbers 24 through 31, since these numbers are not valid for a 24-hour time format.

Time	Dec.	Inputs					
Hr.	#	v	w	x	y	z	F
12mid	0	0	0	0	0	0	0
1am	1	0	0	0	0	1	0
2am	2	0	0	0	1	0	0
3am	3	0	0	0	1	1	0
4am	4	0	0	1	0	0	0
5am	5	0	0	1	0	1	0
6am	6	0	0	1	1	0	0
7am	7	0	0	1	1	1	0
8am	8	0	1	0	0	0	1
9am	9	0	1	0	0	1	1
10am	10	0	1	0	1	0	1
11am	11	0	1	0	1	1	1
12noon	12	0	1	1	0	0	0
1pm	13	0	1	1	0	1	1
2pm	14	0	1	0	1	0	1
3pm	15	0	1	1	1	1	1
4pm	16	1	0	0	0	0	1
5pm	17	1	0	0	0	1	0
6pm	18	1	0	0	1	0	0
7pm	19	1	0	0	1	1	0
8pm	20	1	0	1	0	0	0
9pm	21	1	0	1	0	1	0
10pm	22	1	0	1	1	0	0
11pm	23	1	0	1	1	1	0
---	24	1	1	0	0	0	X
---	25	1	1	0	0	1	X
---	26	1	1	0	1	0	X
---	27	1	1	0	1	1	X
---	28	1	1	1	0	0	X
---	29	1	1	1	0	1	X
---	30	1	1	0	1	0	X
---	31	1	1	1	1	1	X



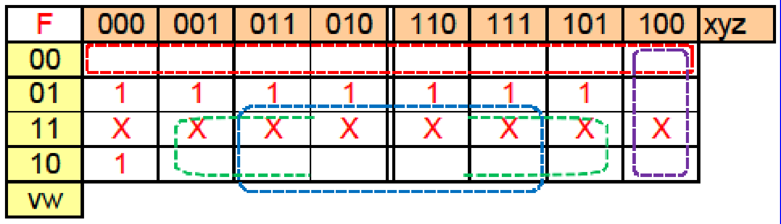
F	000	001	011	010	110	111	101	100	xyz
00									
01	1	1	1	1	1	1	1		
11	X	X	X	X	X	X	X	X	
10	1								
vw									

Minterms:
F = wx' + wz + wyz' + vx'y'z'



F	000	001	011	010	110	111	101	100	xyz
00									
01	1	1	1	1	1	1	1		
11	X	X	X	X	X	X	X	X	
10	1								
vw									

Maxterms:
F = (v + w)(w + y')(v' + z')(x' + y + z)



F	000	001	011	010	110	111	101	100	xyz
00									
01	1	1	1	1	1	1	1		
11	X	X	X	X	X	X	X	X	
10	1								
vw									

Another Equally-Optimized Maxterm Expression:
F = (v + w)(v' + y')(v' + z')(x' + y + z)

"Get Back to Work" Light Truth Table & Karnaugh Map

Figure 16

There are eight outputs that are true (1) in the truth table in Figure 16, which corresponds to an 8-hour day (not counting the hour for lunch). In the first Karnaugh map of this figure, the red shape covers a group of 8 cells, resulting in the red minterm. The blue shape covers 8 cells (4 on one side of the mirror-image line and the 4 appropriate cells on the other side of the mirror-image line), resulting in the blue midterm. The green shape is also symmetrical about the mirror-image line and the purple shape covers two cells, which is the best we could do to capture the lonely, isolated 1.

Notice in Figure 16 that two maxterm expressions are presented, both of which are equally optimized.

Let's check the validity of the minterm expression using Truth Table 14.

Time	Dec.	Inputs										Minterms				O/P
Hr.	#	v	v'	w	w'	x	x'	y	y'	z	z'	wx'	wz	wyz'	vx'y'z'	F
12mid	0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	*0*
1am	1	0	1	0	1	0	1	0	1	1	0	0	0	0	0	*0*
2am	2	0	1	0	1	0	1	1	0	0	1	0	0	0	0	*0*
3am	3	0	1	0	1	0	1	1	0	1	0	0	0	0	0	*0*
4am	4	0	1	0	1	1	0	0	1	0	1	0	0	0	0	*0*
5am	5	0	1	0	1	1	0	0	1	1	0	0	0	0	0	*0*
6am	6	0	1	0	1	1	0	1	0	0	1	0	0	0	0	*0*
7am	7	0	1	0	1	1	0	1	0	1	0	0	0	0	0	*0*
8am	8	0	1	1	0	0	1	0	1	0	1	1	0	0	0	*1*
9am	9	0	1	1	0	0	1	0	1	1	0	1	1	0	0	*1*
10am	10	0	1	1	0	0	1	1	0	0	1	1	0	1	0	*1*
11am	11	0	1	1	0	0	1	1	0	1	0	1	1	0	0	*1*
12noon	12	0	1	1	0	1	0	0	1	0	1	0	0	0	0	*0*
1pm	13	0	1	1	0	1	0	0	1	1	0	0	1	0	0	*1*
2pm	14	0	1	1	0	0	1	1	0	0	1	1	0	1	0	*1*
3pm	15	0	1	1	0	1	0	1	0	1	0	0	1	0	0	*1*
4pm	16	1	0	0	1	0	1	0	1	0	1	0	0	0	1	*1*
5pm	17	1	0	0	1	0	1	0	1	1	0	0	0	0	0	*0*
6pm	18	1	0	0	1	0	1	1	0	0	1	0	0	0	0	*0*
7pm	19	1	0	0	1	0	1	1	0	1	0	0	0	0	0	*0*
8pm	20	1	0	0	1	1	0	0	1	0	1	0	0	0	0	*0*
9pm	21	1	0	0	1	1	0	0	1	1	0	0	0	0	0	*0*
10pm	22	1	0	0	1	1	0	1	0	0	1	0	0	0	0	*0*
11pm	23	1	0	0	1	1	0	1	0	1	0	0	0	0	0	*0*
---	24	1	0	1	0	0	1	0	1	0	1	1	0	0	1	*1*
---	25	1	0	1	0	0	1	0	1	1	0	1	1	0	0	*1*
---	26	1	0	1	0	0	1	1	0	0	1	1	0	1	0	*1*
---	27	1	0	1	0	0	1	1	0	1	0	1	1	0	0	*1*
---	28	1	0	1	0	1	0	0	1	0	1	0	0	0	0	*0*
---	29	1	0	1	0	1	0	0	1	1	0	0	1	0	0	*1*
---	30	1	0	1	0	0	1	1	0	0	1	1	0	1	0	*1*
---	31	1	0	1	0	1	0	1	0	1	0	0	1	0	0	*1*

Checking Minterm Expression from Figure 16

Truth Table 14

The output column (red text) of Truth Table 14 matches the output column in Figure 16, except that Truth Table 14 shows the outputs for the "don't care" conditions, which should never be seen by the logic we have designed.

Let’s step through the functioning of this light. Starting at midnight, which is decimal 0 on the truth table in Figure 16, the “Get Back to Work” light is off until the transition from 7:59am to 8:00am occurs. At 8:00am, the minterm wx’ is true (equal to 1) and the light turns on. This minterm (wx’) keeps the light on until 12noon, at which point the minterm wx’ is no longer true, so the light turns off. When 1pm rolls around, wx’ is still not true, but the minterm wz is true, so the light turns on. The minterm wz was true earlier in the day, but the minterm wx’ was also true at those times. All of the minterms are ORed together, so any of them can cause the light to turn on. All of the minterms would have to be false in order for the light to turn off. The minterm wz is not true at 2pm, but that’s when minterm wyz’ fills in, from 2pm to 2:59pm. At 3pm, wz is true again, so it keeps the light on from 3pm to 4pm. At 4pm, none of the three previously-discussed minterms are true, but vx’y’z keeps the light on from 4pm to 4:59pm. At 5pm, none of the minterms are true, so the light stays off until 8am the next morning, at which time wx’ is true again. There is, obviously, no consideration of days of the week in this design.

END OF EXAMPLE

In Closing:

Karnaugh maps are a simple way to visually determine how to simplify or optimize a binary (digital) function. If the number of input or output conditions is restricted such that there are some “don’t care” output conditions, it is often possible to simplify the logic even further.

Abbreviations:

BCD – Binary-Coded Decimal (binary 0000 through 1001, decimal 0 through 9)
Dec. – Decimal or Base-10 (numerals 0 through 9)
DPDT – Double-Pole Double-Throw
H or Hex – Hexadecimal or base-16 (numerals 0 through F)
LCD – Liquid Crystal Display
LED – Light-Emitting Diode
NC – Normally Closed (relay contact is closed when relay coil is not energized)
NO – Normally Open (relay contact is open when relay coil is not energized)

Appendix

Appendix – Boolean Algebra:

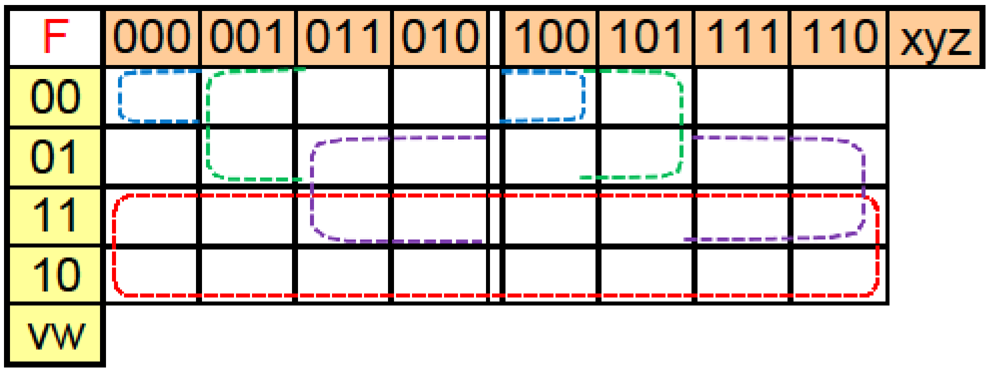
Boolean algebra will not be on the quiz, but a discussion of this topic would be useful as a trouble-shooting and cross-checking aid. Presented below are some common relationships.

Distributive:	A + BC = (A + B)(A + C)
	A(B + C) = AB + AC
Associative:	A + (B + C) = (A + B) + C
	A(BC) = (AB)C
DeMorgan:	(A + B)’ = A’B’
	(AB)’ = A’ + B’
Involution:	(A’)’ = A
Others:	A’ + A = 1
	A + A = A
	A’A = 0
	AA = A
	(1)(A) = A
	(0)(A) = 0
	1 + A = 1
	0 + A = A

Appendix – Overlay Type of 5-Bit and 6-Bit Karnaugh Maps:

The 5-bit Karnaugh map shown in Figure 17 is the overlay type, as opposed to the reflected Gray code type shown in Figure 14a. As mentioned previously, a 5-bit Karnaugh map can be thought of as two 4-bit Karnaugh maps. In the case of the overlay type of 5-bit Karnaugh map, the symmetry of the two halves of the map is realized by placing one of the 4-bit Karnaugh maps on top of the other one, like stacking two sheets of paper. In other words, looking at figure 17, cell 00000 is logically adjacent to cell 00100, but cell 00010 is not logically adjacent to cell 00100. The dividing line between the two 4-bit Karnaugh maps in Figure 17 is not a mirror-image line, it is simply a divider between the two 4-bit Karnaugh maps.

Appendix

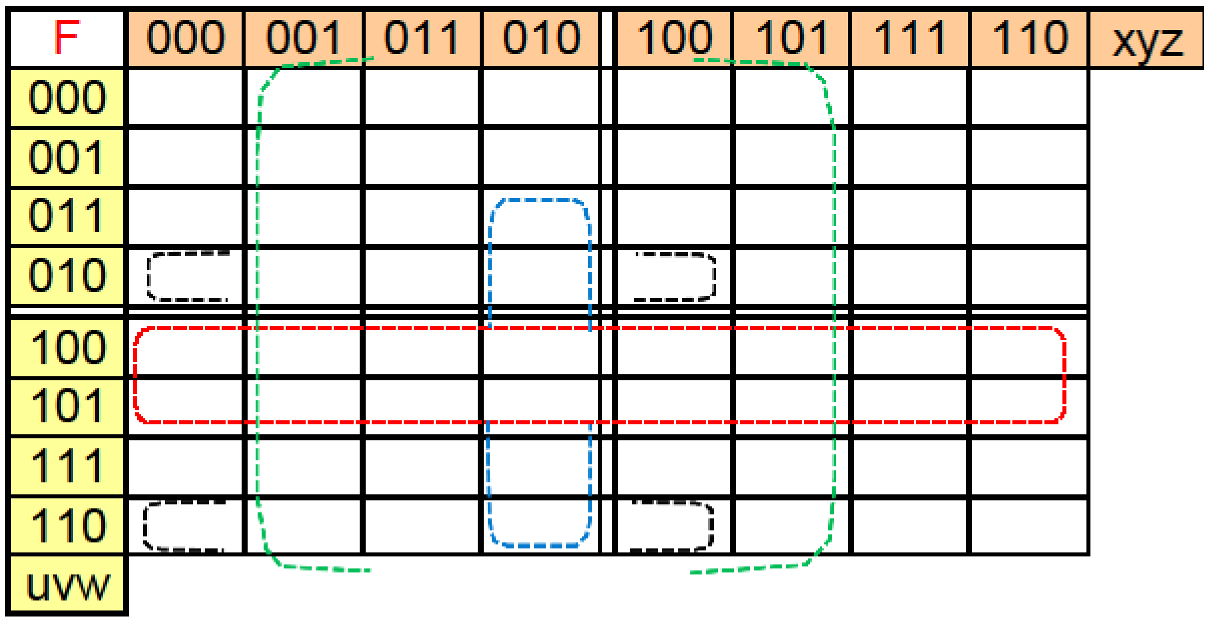


5-Bit Overlay Type of Karnaugh Map with Some Appropriate Groups

Figure 17

The 6-bit Karnaugh map shown in Figure 18 is the overlay type, as opposed to the reflected Gray code type shown in Figure 15. We can think of a 6-bit Karnaugh map as four 4-bit Karnaugh maps. The symmetry of this type of Karnaugh map is realized by placing one of the 4-bit Karnaugh maps on top of or underneath the 4-bit Karnaugh map to the left or right, or on top of or underneath the one shown above or below it. You can't place a 4-bit piece directly on top of or underneath a 4-bit piece that is located in a diagonal corner because the facing cells would not be logically adjacent to each other. For example, cell 000000 is adjacent to cell 000100, but cell 000000 is <u>not</u> adjacent to cell 100100. If you can find a matching group on all of the four-bit maps, you can stack all four of the 4-bit maps, working in a clockwise or a counter-clockwise direction, such that the diagonal 4-bit map is the third one in the stack. Consider the example of the four-cell black group illustrated in Figure 18, which would have a minterm of vw'y'z' if those cells all had 1s, or 1s and Xs. Starting with cell 010000, working clockwise, the next cell that would be adjacent is 010100, then to the diagonal corner for cell 110100, which is adjacent to the previous cell, 010100, then continuing clockwise to cell 110000, which is adjacent to 110100. This entire series wraps back around to the beginning cell, 010000, which is adjacent to cell 110000.

Appendix



6-Bit Overlay Type of Karnaugh Map with Some Appropriate Groups

Figure 18

Some people find the overlay type of Karnaugh map to be easier to use than the reflected Gray code type of 5-bit and 6-bit Karnaugh map. Whether an overlay type or reflected Gray code type of Karnaugh map is used, the resulting minterm and maxterm expressions will be the same.

Appendix – Malfunction of "Don't Care" Conditions:

As with any design, we have to work with the information we are given. If we are told that there are "don't care" inputs that will never be presented to the logic we are designing, we need to document it and proceed with our design. Still, it might be advisable to consider, for instance, what would happen if the unseen upstream logic in Example 6 had a malfunction that allowed it to transmit erroneous inputs (inputs that were defined as "don't care" conditions) to our seven-segment LED display logic. In other words, what if our logic received as an input the binary equivalent of decimal #14? The binary equivalent of decimal #14 is wxyz = 1110. If we look at the minterm expression for segments a through g (listed in Figures 12a through 12g), we find that all of the segments except segment b would turn on. This would look just like the decimal #6 was displaying on the seven-segment LED display.

Truth Table 15 shows us what the state (on or off) of each segment would be for "don't care" inputs of decimal #10 through #15. This truth table was constructed by plugging the values of w, x, y, and z into the minterm expressions in Figures 12a through 12g. Looking at the minterms is easier than looking at the maxterms for this exercise because all we have to do is find one minterm that is true (1) within the minterm expression and we'll know that the segment will turn

Appendix

on. Segment f, for example, will turn on any time w = 1, which is true for all decimal numbers 10 through 15. There are other parts to the minterm expression for segment f, but we already went far enough to determine that segment f will be on for all of the inputs presently under consideration.

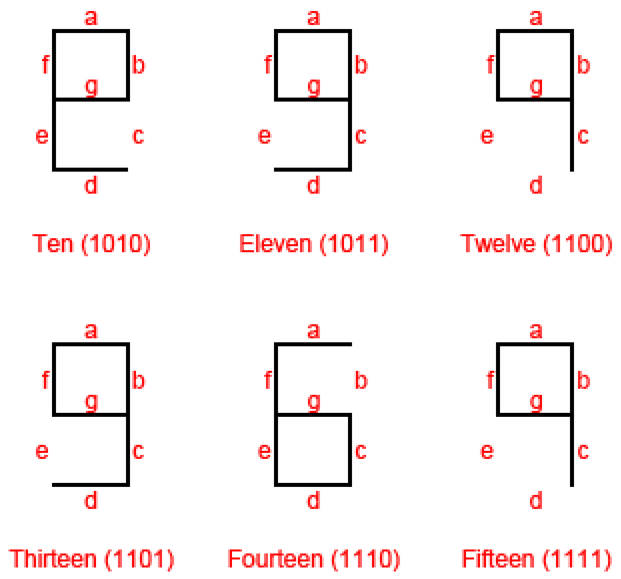
Dec.	Inputs				Outputs (Seven Segments)						
#	w	x	y	z	a	b	c	d	e	f	g
10	1	0	1	0	1	1	0	1	1	1	1
11	1	0	1	1	1	1	1	1	0	1	1
12	1	1	0	0	1	1	1	0	0	1	1
13	1	1	0	1	1	1	1	1	0	1	1
14	1	1	1	0	1	0	1	1	1	1	1
15	1	1	1	1	1	1	1	0	0	1	1

What if the Upstream Logic Malfunctioned and Was Able to Send Binary Numbers 1010 through 1111 to Example 6?

Truth Table 15

Using the logic designed in Example 6, which included "don't care" conditions, Figure 19 shows what the seven-segment LED display would look like if the erroneous inputs of decimal #10 through #15 were received from the upstream logic.

Appendix



What if the Upstream Logic Malfunctioned and Was Able to Send Binary Numbers 1010 through 1111 to Example 6?

Figure 19

The displays in Figure 19 don’t look like obvious errors, except for decimal #10 (1010). The other five displays look like a 6 or a 9, and so might pass without comment. We saved several gates by incorporating the “don’t care” conditions, but at the cost of displaying erroneous outputs for erroneous inputs. If the upstream logic starts sending out erroneous inputs to our logic, there are probably other problems with the upstream logic, such that the entire functioning of the device has already been compromised.

The logic we designed in Example 5, which required more relay contacts or gates and did not include “don’t care” conditions, would show a blank display for inputs of the binary equivalents of decimal #10 through #15.

End of Appendix

— ——— ——— —— ••— —•—• •••• ••—• ••— —•