3 Combinational Circuits

Digital circuits are classified as either one of two types: combinational or sequential. Combinational circuits are the class of digital circuits where the outputs of the circuit are dependent only on the current inputs. They do not remember the history of past inputs and, therefore, do not require any memory elements. Sequential circuits on the other hand are circuits in which their outputs are dependent on not only the current inputs but also on past inputs. Because of their dependency on past inputs, sequential circuits must contain memory elements in order to remember the past input values. A “large” digital circuit, however, may contain both combinational components and sequential components.

Whether it is a combinational circuit or a sequential circuit, it is nevertheless a digital circuit, and so we use the same building blocks, namely the AND, OR and NOT gates. What makes them different is in the way some of the gates are connected. In order for a digital circuit to “remember” its current value or state, we have to connect the output of a logic gate directly or indirectly back to the input of that same gate. We call this a loop back circuit and it forms the basis of a sequential circuit. Combinational circuits do not have any loop backs.

In this chapter, we will focus on combinational circuits. Sequential circuits will be presented in a later chapter.

3.1 Analysis of Combinational Circuits

The analysis of combinational circuits is the process in which we are given a combinational circuit and we want to derive a precise description of the operation of the circuit. In general, a combinational circuit can be described precisely either with a truth table or with a Boolean function.

3.1.1 With a Truth Table

For example, given the combinational circuit of Figure 1, we want to derive the truth table that describes the circuit. We create the truth table by first listing all the inputs found in the circuit, one input per column, followed by all the outputs found in the circuit. Hence, we start with a table with four columns; three columns (x, y, z) for the inputs and one column (f) for the output as shown below.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>f</th>
</tr>
</thead>
</table>

The next step is to enumerate all possible combinations of 0’s and 1’s for all the input variables, usually in sequential binary counting order as shown below.

![Sample combinational circuit](image-url)
Now, for each row in the table, that is, for each combination of input values, determine what the output value is by substituting the values for the input variables and tracing the values through to the output. For example, using \(xyz = 000\), the outputs for all AND gates are 0, and ORing all the zeros gives a zero, therefore, \(f = 0\). For \(xyz = 001\), the output for the top AND gate gives a 1, and 1 OR with anything gives a 1, therefore, \(f = 1\). Continuing in this fashion, we can complete the final truth table for the circuit as shown below:

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### 3.1.2 With a Boolean Function

To derive a Boolean function that describes a combinational circuit, we simply write down the Boolean logical expressions at the output of each gate instead of substituting actual values of 0’s and 1’s for the inputs as we trace through the circuit from the primary input to the primary output. Using the sample combinational circuit of Figure 1, we note that the logical expression for the output of the top AND gate is \(x'y'z\). The logical expressions for the following AND gates are respectively \(x'yz\), \(xy'z\), and \(xyz\). Finally, the outputs from these AND gates are all ORed together. Hence, we get the final expression:

\[
f = x'y'z + x'yz + xy'z + xyz
\]

To help keep track of the expressions at the output of each logic gate, we can annotate the outputs of each logic gate with the resulting logical expression. If we substitute all possible combinations of values for the variables, we should obtain the same truth table as above.

**Example**

As another example, consider the combinational circuit below,
Starting from the primary inputs \(x, y,\) and \(z,\) we annotate the outputs of each logic gate with the resulting logical expression. Hence, we obtain the annotated circuit below.

The output of the circuit is the final function \(f = x'(xy' + (y \oplus z)).\)

### 3.2 Synthesis of Combinational Circuits

Synthesis of combinational circuits is just the reverse procedure of the analysis of combinational circuits. In synthesis, we start with a description of the operation of the circuit. From this description, we derive either the truth table or the Boolean logical function that precisely describes the operation of the circuit. Once we have either the truth table or the logical function, we can easily translate that into a circuit diagram.

For example, let us construct a 3-bit comparator circuit that outputs a 1 if the number is greater than or equal to 5, and 0 otherwise. In other words, a circuit that outputs a 0 if the input is a number between 0 and 4, and outputs a 1 if the input is a number between 5 and 7. Since we are working with decimal numbers in the range 0 to 7, we can use three input bits \((x_2, x_1,\) and \(x_0)\) to represent the number. From the description, we obtain the following truth table:

<table>
<thead>
<tr>
<th>(x_2)</th>
<th>(x_1)</th>
<th>(x_0)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In constructing the circuit, we are only interested in when the output is a 1, i.e. when the function is a 1. Thus, we need only consider the rows where the output function \(f = 1.\) From the above truth table, we see that there are three rows where \(f = 1\) which give the three AND terms \(x_2x_1x_0', x_2x_1x_0', \) and \(x_2x_1x_0.\) Notice that the variables in the AND terms are such that it is inverted if its value is a 0, and not inverted if its value is a 1. In the case of the first AND term, we want \(f = 1\) when \(x_2 = 1\) and \(x_1 = 0\) and \(x_0 = 1,\) and this is satisfied in the expression \(x_2x_1x_0.\) Finally, we want \(f = 1\) when either one of these three AND terms is equal to 1. So we ORed the three AND terms together giving us our final expression:

\[
f = x_2x_1'x_0 + x_2x_1x_0' + x_2x_1x_0
\]

In drawing the schematic diagram, we simply convert the AND operators to AND gates and OR operators to OR gates. The equation is in the sum-of-product format, meaning that it is summing (ORing) product (AND) terms. A sum-of-product equation translates to a two level circuit with the first level being made up of AND gates and the second level made up of OR gates. Each of the three AND terms contain three variables, so we use a 3-input AND gate for each of the three AND terms, The three AND terms are ORed together, so we use a 3-input OR gate to connect the output of the three AND gates. For each inverted variable, we need an inverter. The schematic diagram derived from the above equation is shown below.
3.2.1 Technology Mapping

To reduce implementation cost and turnaround time, designers often make use of off-the-shelf semi-custom gate arrays. Many gate arrays are ICs that have only NAND gates or NOR gates built in them, but their input and output connections are not yet connected. To use these gate arrays, a designer simply has to specify where to make these connections between the gates. The problem in using these gate arrays to implement our circuit is that we need to convert all AND gates, OR gates, and inverters in our circuit to use only NAND gates or NOR gates depending on what is available in the gate array. More over, these NAND and NOR gates usually have the same number of fixed inputs; usually 3-input.

The conversion of any given circuit to use only NAND or NOR gates is possible by observing the following equalities as obtained from the Boolean algebra theorems:

Rule 1: \( x' = x \)

Rule 2: \( xy = ((xy))' \)

Rule 3: \( x + y = ((x + y))' = (x' y')' \)

Rule 4: \( xy = ((xy))' = (x' + y')' \)

Rule 5: \( x + y = ((x + y))' \)

Rule 1 simply says that a double inverter can be eliminated altogether. Rule 2 applies Rule 1 to the AND operator. The resulting expression, however, gives us a NAND gate followed by an inverter. Rule 3 changes an OR gate to use two inverters and a NAND gate by first applying the double inverter rule and then De Morgan’s Theorem. Similarly, Rule 4 converts an AND gate to use two inverters and a NOR gate, and Rule 5 converts an OR gate to a NOR gate followed by an inverter.

Rules 2 and 3 are used if we want to convert an AND OR circuit to use only NAND gates, whereas, rules 4 and 5 are used if we want to use only NOR gates.

In a circuit diagram, these rules translate to the following equivalent circuits:

Rule 1: \( \)

Rule 2: \( \)

Rule 3: \( \)

Rule 4: \( \)

Rule 5: \( \)
Finally, to replace inverters with either the NAND gate or the NOR gate, we note that by simply connecting all the inputs of either the NAND or the NOR gate together, the resulting operation of the gate is like the inverter. Take the 2-input NAND gate for example, if we connect the two inputs together so that there is only one input and one output as follows:

\[ \overline{xy} \]

Since \( x \) and \( y \) are now always the same, we can simplify the NAND gate truth table by first eliminating the two rows where \( x \neq y \), and second by combining the original two columns for \( x \) and \( y \) into just one column for the one input. As a result, we get a truth table that is exactly the same as that for the inverter:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ x \quad f \]

0 1
1 0

Similarly, we can get the same functional result by connecting together the two inputs for a NOR gate:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ x \quad f \]

0 1
1 0

Another thing that we might want is to get the functionality of a 2-input NAND or NOR gate from a 3-input NAND or NOR gate respectively. The following circuit shows how you can get a 2-input NAND/NOR gate from a 3-input NAND/NOR gate. You may want to check with a truth table that they are indeed equivalent.

2-input NAND gate 2-input NOR gate

The reverse is to get the functionality of a 3-input NAND or NOR gate from 2-input NAND or NOR gates respectively. These two transformations make use of the following two equalities:

\[
(abc)' = ((ab)'c)'
\]

\[
(a+b+c)' = ((a+b)' + c)'
\]

Hence, the circuits for the 3-input NAND and NOR gates using 2-input NAND and NOR gates respectively are shown in Figure 2.

\[ \text{Figure 2.} \quad 3\text{-input NAND and NOR gates using 2\text{-input NAND and NOR gates respectively.} } \]
Example 3.1.
As an example, let us convert the following circuit to use only 3-input NAND gates.

First, we need to change the 4-input OR gate to a 3- and 2-input OR gates.

Then we will use Rule 2 to change all the AND gates to 3-input NAND gates with inverters, and Rule 3 to change all the OR gates to 3-input NAND gates with inverters. The 2-input NAND gates are replaced with 3-input NAND gates with two of its inputs connected together.

Finally, we eliminate all the double inverters, and replace the remaining inverters with NAND gates with their inputs connected together.
3.3 Minimization of Combinational Circuits

When constructing digital circuits, in addition to obtaining a functionally correct circuit, we like to optimize it in terms of circuit size, speed and power consumption. In this section, we will focus on the reduction of circuit size. Usually, by reducing the circuit size, we will also have improved on the speed and power consumption. We have seen in the previous sections that any combinational circuit can be represented using a Boolean function. The size of the circuit is directly proportional to the size or complexity of the functional expression. In fact, it is a one to one correspondence between the functional expression and the circuit size. By using the Boolean algebra theorems, we can transform an expression to another equivalent expression. If the resulting expression is simpler than the original, then we want to implement the circuit based on the simpler expression since that will give us a smaller circuit size.

Using Boolean algebra to transform an expression to one that is simpler is not an easy task, especially for the computer. There is no formula that says which is the next theorem to use. Luckily, there are easier methods for reducing Boolean expressions. The **Karnaugh map (K-map)** method is an easy way for reducing an equation manually and is discussed in section 3.3.1. The **Quine-McCluskey or tabulation** method for reducing an equation is ideal for programming the computer and is discussed in section 3.3.3.

3.3.1 Karnaugh (K) Maps

To minimize a Boolean equation in the sum-of-products form, we need to reduce the number of product terms by applying the combining Boolean Theorem (Theorem 14) from section 2.4. In so doing we will also have reduced the number of variables used in the product terms. For example, given the following 3-variable function

\[ F = xy'z' + xyz' \]

we can reduce it to

\[ F = xz' (y' + y) \]
\[ = xz' 1 \]
\[ = xz' \]

In other words, two product terms that differ in the value of only one variable can be combined together, and the variable whose value differs is dropped from the resulting term. Thus, we have reduced the number of product terms and the resulting product terms have one less variable. By reducing the number of product terms, we reduce the number of OR operators required, and by reducing the number of variables in a product term, we reduce the number of AND operators required.

Looking at a logic function’s truth table, it is sometimes difficult to see how the product terms can be combined and minimized. A **Karnaugh map or K-map** for short provides a simple and straight forward procedure for combining these product terms. A K-map is just a graphical representation of a logic function’s truth table where the minterms are grouped in such a way that it allows one to easily see which of the minterms can be combined. It is a 2-dimensional array of squares, each of which represents one minterm in the Boolean function. Thus, the map for an \( n \)-variable function is an array with \( 2^n \) squares.

Figure 3 shows the K-maps for functions with 2, 3, 4, and 5 variables. Notice the labeling of the columns and rows are such that any two adjacent columns or rows differ in only one bit change. This condition is required because we want minterms in adjacent squares to differ in the value of only one variable or one bit, and so these minterms can be combined together. This is why the labeling for the third and fourth columns and the third and fourth rows are always interchanged. When we read K-maps, we need to visualize it as such that the two end columns or rows wrap around so that the first and last columns and the first and last rows are really adjacent to each other because they differ in only one bit also.

In Figure 3 the K-map squares are annotated with its minterm and its minterm number for easy reference only. When we are actually using K-maps to minimize an equation, we will not write these in the squares. Instead, we will be putting 0’s and 1’s in the squares.

Given a Boolean function, we set the value for each K-map square to either a 0 or a 1 corresponding to whether that minterm for the function is a 0-minterm or a 1-minterm. However, since we are only interested in the 1-minterms, the 0’s are sometimes not written in the 0-minterm squares.
For example, the K-map for the 2-variable function

\[ F = x'y' + x'y + xy \]

is

\[
\begin{array}{c|cc|cc}
  & 0 & 1 & 0 & 1 \\
\hline
0 & x'y' & x'y & x'y' & x'y \\
1 & xy' & xy & xy' & xy
\end{array}
\]

The 1-minterms \(m_0\) \((x'y')\) and \(m_1\) \((x'y)\) are adjacent to each other which means that they differ in the value of only one variable. In this case, \(x\) is 0 for both minterms, but \(y\) is 0 for one and 1 for the other. Thus, variable \(y\) is dropped and the two terms are combined together giving just \(x'\). This reasoning corresponds to the expression

\[ x'y' + x'y = x'(y' + y) = x' \]

Similarly, the 1-minterms \(m_1\) \((x'y)\) and \(m_3\) \((xy)\) are also adjacent and \(y\) is the variable having the same value for both minterms, and so they can be combined to give

\[ x'y + xy = y \]

We use the term *subcube* to refer to a rectangle of adjacent 1-minterms. These subcubes must be rectangular in shape and can only have sizes that are the powers of two. Formally, for an \(n\)-variable K-map, an \(m\)-subcube is defined as that set of \(2^m\) minterms in which \(n - m\) of the variables will have the same value in every minterm while the remaining variables will take on the \(2^m\) possible combinations of 0’s and 1’s. Thus, a 1-minterm all by itself is called a 0-subcube, and two adjacent 1-minterms is a 1-subcube. In the above 2-variable K-map, there are two 1-
subcubes: one label with \( x' \) and one with \( y \). A 2-subcube will have four adjacent 1-minterms and can be in the shape of any one of those in Figure 4 (a) to (e).

Notice that Figure 4 (d) and (e) also form 2-subcubes even though the four 1-minterms are not physically adjacent to each other. They are adjacent, however, because the first and last rows, and first and last columns wrap around in a K-map. In Figure 4 (f), the four 1-minterms cannot form a 2-subcube because they do not form a rectangle. However, they can form three 1-subcubes.

We say that a subcube is characterized by the variables having the same values for all the minterms in that subcube. In general, an \( m \)-subcube for an \( n \)-variable K-map will be characterized by \( n - m \) variables. If the value that is similar for all the variables is a 1, that variable is unprimed, whereas, if the value that is similar for all the variables is a 0, that variable is primed. In an expression, this is equivalent to the resulting smaller product term when the minterms are combined together. For example, the 2-subcube in Figure 4 (d) is characterized by \( z' \) since the value of \( z \) is 0 for all the minterms, whereas the values for \( x \) and \( y \) are not all the same for all the minterms. Similarly, the 2-subcube in Figure 4 (e) is characterized by \( x'z' \).

For a 5-variable K-map as in Figure 3 (d), we need to visualize the right half of the array where \( v = 1 \) to be on top of the left half where \( v = 0 \). Thus, for example, minterm 20 is adjacent to minterm 4, and minterm 31 is adjacent to minterm 15.

The K-map method reduces a Boolean function from its canonical form to its standard form. The goal for the K-map method is to find as few subcubes as possible to cover all the 1-minterms in the given function. This naturally implies that the subcube size should be as big as possible. The reasoning is that each subcube gives a product term and all the subcubes (or product terms) must be ORed together to give the function. Larger subcubes require fewer AND gates because of fewer variables in the product term, and fewer subcubes will require fewer OR gates.

The procedure for using the K-map method is as follows:

1. Draw the appropriate K-map for the given function and place a 1 in the squares that correspond to the function’s 1-minterms.

2. For each 1-minterm, find the largest subcube that covers this 1-minterm. This largest subcube is known as a prime implicant (PI). By definition, a prime implicant is a subcube that is not contained within any other subcube. If there are more than one subcube that is the same size as the largest subcube, then they are all prime.
implicants.

3. Look for 1-minterms that are covered by only one prime implicant. Since this prime implicant is the only subcube that covers this particular 1-minterm, this prime implicant is a must have in the final solution. This prime implicant is referred to as an essential prime implicant (EPI). By definition, an essential prime implicant is a subcube that includes a 1-minterm that is not included in any other subcube.

4. Create a minimal cover list by selecting the smallest possible number of prime implicants such that every 1-minterm is contained in at least one prime implicant. This cover list must include all the essential prime implicants plus zero or more of the remaining prime implicants. It is alright that a particular 1-minterm is covered in more than one prime implicant, but all 1-minterms must be covered.

5. The final minimized function is obtained by ORing all the prime implicants from the minimal cover list.

Note that the final minimized function obtained by the K-map method may not be in its most reduced form. It is only in its most reduced standard form. Sometimes, it is possible to reduce the standard form further into a non-standard form.

Example 3.2

Use the K-map method to minimize a 4-variable \((w, x, y, z)\) function \(F\) with the 1-minterms: \(m_0, m_2, m_5, m_7, m_{10}, m_{13}, m_{14},\) and \(m_{15}\). We start with the following 4-variable K-map

```
<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The prime implicants for each of the 1-minterms are shown in the following K-map and table:

<table>
<thead>
<tr>
<th>1-minterm</th>
<th>Prime Implicant</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_0)</td>
<td>(w'x'z')</td>
</tr>
<tr>
<td>(m_2)</td>
<td>(w'x'z', x'yz')</td>
</tr>
<tr>
<td>(m_5)</td>
<td>(xz)</td>
</tr>
<tr>
<td>(m_7)</td>
<td>(xz)</td>
</tr>
<tr>
<td>(m_{10})</td>
<td>(x'y'z', wyz')</td>
</tr>
<tr>
<td>(m_{13})</td>
<td>(xz)</td>
</tr>
<tr>
<td>(m_{14})</td>
<td>(wyz', wxy)</td>
</tr>
<tr>
<td>(m_{15})</td>
<td>(xz)</td>
</tr>
</tbody>
</table>

Thus, there are five prime implicants: \(w'x'z', x'y'z', xz, wyz',\) and \(wxy\). Of these five prime implicants, \(w'x'z'\) and \(xz\) are essential prime implicants since \(m_0\) is covered only by \(w'x'z'\), and \(m_5, m_7,\) and \(m_{13}\) are covered only by \(xz\).

We start the cover list by including the two essential prime implicants \(w'x'z'\) and \(xz\). These two subcubes will have covered minterms \(m_0, m_5, m_7, m_{13}\) and \(m_{15}\). To cover the remaining two uncovered minterms \(m_{10}\) and \(m_{14}\), we want to use as few prime implicants as possible. Hence, we select the prime implicant \(wyz'\) which covers both of them.

Finally, our reduced standard form equation is obtained by ORing these three prime implicants

\[
F = w'x'z' + xz + wyz'.
\]
3.3.2 Don’t-cares

There are times when a function is not fully specified. In other words, there are some minterms for the function where we do not care whether their values are a 0 or a 1. When drawing the K-map for these “don’t-care” minterms, we assign an “×” in that square instead of a 0 or a 1. Usually, a function can be reduced even further if we remember that these ×’s can be either a 0 or a 1. As you recall when drawing K-maps, enlarging a subcube reduces the number of variables for that term. Thus, in drawing subcubes, some of them may be enlarged if we treat some of these ×’s as 1’s. On the other hand, if some of these ×’s will not enlarge a current subcube, then we want to treat them as 0’s so that we do not need to cover them. It is not necessary to treat all ×’s to be all 1’s or 0’s. We can assign some ×’s to be 0’s and some to be 1’s.

For example, given a function having the following 1-minterms and don’t-care minterms:

1-minterms: m0, m1, m2, m3, m4, m7, m8 and m9.

×-minterms: m10, m11, m12, m13, m14 and m15.

we obtain the following K-map with the prime implicants x’, yz and y’z’.

Notice that in order to get the 4-subcube characterized by x’ the two don’t-care minterms m10 and m11 are taken to have the value 1. Similarly with the minterms m12 and m15. On the other hand, the don’t-care minterms m13 and m14 are taken to have the value 0 so that they do not need to be covered in the solution. The reduced standard form function as obtained from the K-map is, therefore

\[ F = x' + yz + y'z'. \]

Again, this equation can be reduced further by recognizing that yz + y’z' = y ⊕ z. Thus,

\[ F = x' + (y ⊕ z). \]

3.3.3 Quine-McCluskey (Tabulation) Method

K-maps are useful for manually obtaining the minimized standard form Boolean function for may be up to at most six variables. However, for functions with more than six variables, it becomes very difficult to visualize how the minterms should be combined into subcubes. Moreover, the K-map algorithm is not as straight forward to program the computer with. There exist tabulation methods, one of which is the Quine-McCluskey method that are better suited for programming the computer, and thus can solve any function having any number of variables.

Example 3.3

We now illustrate the Quine-McCluskey algorithm using the same four-variable function as in example 3.2 and repeated here

\[ f(w, x, y, z) = \Sigma(0, 2, 5, 7, 10, 13, 14, 15) \]
To construct the initial table, the minterms are grouped according to the number of 1’s in that minterm number’s binary representation. For example, $m_0$ (0000) has no 1’s; $m_2$ (0010) has one 1; $m_3$ (0101) has two 1’s; etc. Thus, the initial table of 0-subcubes (i.e. subcubes having only one minterm) as obtain from the above function is

<table>
<thead>
<tr>
<th>Group</th>
<th>Subcube Minterms</th>
<th>Subcube Value</th>
<th>Subcube Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_0$</td>
<td>$m_0$</td>
<td>0 0 0 0</td>
<td>✓</td>
</tr>
<tr>
<td>$G_1$</td>
<td>$m_2$</td>
<td>0 0 1 0</td>
<td>✓</td>
</tr>
<tr>
<td>$G_2$</td>
<td>$m_5$</td>
<td>0 1 0 1</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>$m_{10}$</td>
<td>1 0 1 0</td>
<td>✓</td>
</tr>
<tr>
<td>$G_3$</td>
<td>$m_7$</td>
<td>0 1 1 1</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>$m_{13}$</td>
<td>1 1 0 1</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>$m_{14}$</td>
<td>1 1 0 0</td>
<td>✓</td>
</tr>
<tr>
<td>$G_4$</td>
<td>$m_{15}$</td>
<td>1 1 1 1</td>
<td>✓</td>
</tr>
</tbody>
</table>

The “Subcube Covered” column is filled in from the next step.

In the second step, we construct a second table by combining those minterms in adjacent groups from the first table that differ in only one bit position. For example, $m_0$ and $m_2$ differ in only the $y$ bit. Thus, this table lists all the 1-subcubes. A hyphen (–) is used in the bit position that is different in the two minterms. Since this 1-subcube covers the two individual minterms, we make a note of it by checking the two minterms in the “Subcube Covered” column in the previous table. The 1-subcube table is shown next

<table>
<thead>
<tr>
<th>Group</th>
<th>Subcube Minterms</th>
<th>Subcube Value</th>
<th>Subcube Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_0$</td>
<td>$m_0, m_2$</td>
<td>0 0 – 0</td>
<td></td>
</tr>
<tr>
<td>$G_1$</td>
<td>$m_2, m_{10}$</td>
<td>– 0 1 0</td>
<td></td>
</tr>
<tr>
<td>$G_2$</td>
<td>$m_5, m_7$</td>
<td>0 1 – 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$m_5, m_{13}$</td>
<td>– 1 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$m_{10}, m_{14}$</td>
<td>1 – 1 0</td>
<td></td>
</tr>
<tr>
<td>$G_3$</td>
<td>$m_7, m_{15}$</td>
<td>1 1 – 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$m_{11}, m_{15}$</td>
<td>1 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

We repeat the second step as long as there are adjacent subcubes that differ in only one bit position including the hyphen. These subcubes are combined to give the next subcube table. From the above 1-subcube table, we get the following 2-subcube table

<table>
<thead>
<tr>
<th>Group</th>
<th>Subcube Minterms</th>
<th>Subcube Value</th>
<th>Subcube Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_2$</td>
<td>$m_5, m_6, m_1, m_{12}$</td>
<td>1 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

We stop when there are no more subcubes that can be combined. The prime implicants are those subcubes that are not covered, i.e. those without a check mark in the Subcube Covered column. For example, from the last table (2-subcube table) the only subcube in this table has the value $x = 1$ and $z = 1$, thus we get the prime implicant $xz$. From the 1-subcube table, we have the four prime implicants $w'y'z'$, $x'y'z'$, $wy'z'$, and $wxy$. Note that these prime implicants may not necessary be all in the last table. These five prime implicants are exactly the same as those obtained in example 3.2.

### 3.4 7-Segment Decoder Example

We will now synthesize the circuit for a 7-segment decoder for driving a 7-segment LED display. The 7-segment decoder converts a 4-bit input to seven output lines for turning on the seven lights in a 7-segment LED display. The 4-bit input encodes the binary representation of a decimal digit. Given the decimal digit input, the seven
output lines are turned on in such a way so that the LED displays the corresponding digit. The 7-segment LED display schematic with the names of each segment labeled is shown below

The operation of the 7-segment decoder is specified in the truth table below.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Decimal digit</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_3$</td>
<td>$I_2$</td>
<td>$I_1$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rest of the combinations</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

From the above truth table, we are able to specify seven equations that are dependent on the four inputs for the seven segments. For example, the equation for segment $a$ is

$$a = I_3I_2I_1I_0' + I_1I_2' + I_1I_0 + I_3 + I_1I_0$$

Before implementing this equation directly in a circuit, we want to simplify it first using the K-map method. The K-map for the equation is

From evaluating the K-map, we derive the equation

$$a = I_2' + I_1 + I_0 = I_2' + (I_1 \oplus I_0)$$

Proceeding in a similar manner, we get the following remaining six equations

$$b = I_2 + I_1' + I_0$$
$$c = I_2' + I_0' + I_1' (I_2 \oplus I_0)$$
\[
\begin{align*}
    d &= I_3I_0' + I_2I_1I_0' \\
    e &= I_3 + I_2I_0' + I_1'(I_2 \oplus I_0) \\
    f &= I_3 + I_1'I_0' + I_0(I_2 \oplus I_0) \\
    g &= I_3 + (I_2 \oplus I_1) + I_1I_0'
\end{align*}
\]

From these seven simplified equations, we can now implement the circuit as follows.

3.5 VHDL Code for Combinational Circuits

3.5.1 Dataflow BCD to 7-Segment Decoder

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity Seven_Segment_Display is
    port(I0: in std_logic;
         I1: in std_logic;
         I2: in std_logic;
         I3: in std_logic;
         SGMTA: out std_logic;
         SGMTB: out std_logic;
         SGMTC: out std_logic;
         SGMTD: out std_logic;
         SGMTF: out std_logic;
         SGMTG: out std_logic);
end Seven_Segment_Display;

architecture Dataflow of Seven_Segment_Display is
begin
```

Principles of Digital Logic Design     Enoch Hwang     Last updated 1/31/2002 11:32 AM
3.5.2 Behavioral BCD to 7-Segment Decoder

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY BCD2Seven_Segment_Decoder IS
  PORT ( BCD: IN std_logic_vector(3 DOWNTO 0);
         Segments: OUT std_logic_vector(1 TO 7));
END BCD2Seven_Segment_Decoder;

ARCHITECTURE Behavioral OF BCD2Seven_Segment_Decoder IS
BEGIN
  process(BCD)
  BEGIN
    CASE BCD IS
      WHEN "0000" => Segments <= "1111110";
      WHEN "0001" => Segments <= "1100000";
      WHEN "0010" => Segments <= "1011011";
      WHEN "0011" => Segments <= "1110011";
      WHEN "0100" => Segments <= "1100101";
      WHEN "0101" => Segments <= "0110111";
      WHEN "0110" => Segments <= "0111111";
      WHEN "0111" => Segments <= "1100010";
      WHEN "1000" => Segments <= "1111111";
      WHEN "1001" => Segments <= "1100111";
      WHEN OTHERS => Segments <= "0000000";
    END CASE;
  END PROCESS;
END Behavioral;