Tutorial 4

Combinational Logic and Memory Devices

THINGS TO LOOK FOR...

- The axioms and theorems of Boolean algebra.
- How to write and manipulate logical equations using Boolean algebra.
- How to implement logical equations using AND, OR, and Not gates.
- How to use the Karnaugh map to simplify logical equations.
- Don’t care conditions – what they are, when to use them, and how to use them to aid in simplifying logical equations.
- The basic memory devices: the SR latch, the JK latch and the D latch and their behavior.
- Extending latches to flip-flops.

4.0 INTRODUCTION

In this tutorial we review of the basics of Boolean algebra, combinational logic, and storage devices. We will use Boolean algebra to formulate and manipulate logical equations then implement the resulting equations using the AND, OR, and NOT gates. We will present the Karnaugh map as an easy to use graphical logic reduction tool for small problems. We will then introduce don’t care conditions and examine how they can be used to further simplify logic expressions.

The outputs of combinational logic devices are a function of inputs only; they are valid as long as the inputs are true. If the inputs change, the outputs change. Computers and other kinds of digital systems also need devices capable of storing data and information and of performing mathematical or logical operations on such data. Such devices are identified as memory devices – in the jargon of the field, they’re known as latches and flip-flops. In our review of these devices, we will begin with the basic storage device, add capabilities to support greater control over when and how the device changes state in response to its inputs, then develop the characteristic equations describing the behavior of the most common latch and flip-flop types.
4.1 Boolean Algebra

4.1.1 Relations

In the Introduction to the text, we presented a hierarchy that illustrated how we can move from abstract symbols to knowledge. We showed that the levels in the hierarchy are connected by relations. In this tutorial, we will review some of those relations. The simplest is unary – it involves only one object. Unary relations, however, are of limited utility. The rules of logic and arithmetic typically specify relations between two (binary relations) or more (n-ary relations) objects. From arithmetic, we’re quite familiar with the basic relationships of addition, subtraction, multiplication, and division.

To be able to build more complex systems of reasoning we can also include logical relationships. Like arithmetic, we start by talking about relations on or between one or two things; these, too, are unary and binary relations. From these we can go to n-ary relations. Based upon such relations, one can form a complete system for reasoning.

4.1.2 Algebra

We will now form what is called an algebra. Formally an algebra or algebraic system is defined as,

- Set A together with one or more n-ary operations on A
- Which satisfy a specified set of axioms

In our work, we will define a set objects and several binary relations on the objects in the set. The algebra that we will present is known as Boolean algebra and is based upon the work put forth by George Boole 1854 in the manuscript, An Investigation into the Laws of Thought. Boole’s work is an outgrowth of two key ideas presented by Aristotle: the Law of Excluded Middle and the Law of Non contradiction. The former states that everything must be either TRUE or FALSE and the latter that something cannot be simultaneously TRUE and FALSE.

In the Boolean algebra we will work with two valued variables and the binary relations AND and OR which are expressed by the accompanying operator symbols shown in Figure 4.0. Although we are restricting this work to two valued variables, it can easily be extended to multiple valued variables as initially put forth by Plato and later developed by the Polish mathematician, Lukasiewicz (also the inventor of Reverse Polish Notation, RPN).

An algebra is based upon a set of rules that are derived from a basic set of assumptions called axioms. These lead to the formulation of a set of theorems for operating on the variables in the algebra. We will work with the following axioms which were presented by E.V. Huntington in 1904.
Axioms

We define an algebraic system \( \{ A, \cdot, +, 0, 1 \} \) in which \( \cdot \) and \( + \) are the operations of AND and OR, 0 and 1 are elements of A, and an equivalence relation = on the elements in A. The operations \( \cdot \) and \( + \) are defined as shown in Figure 4.1.

The equivalence relation is defined such that,

I. For elements \( a, b, \) and \( c \) in \( A \) the equivalence relation is
   1. Reflexive
      \( a = a \) for all \( a \) in \( A \)
   2. Symmetric
      If \( a = b \) then \( b = a \) for all \( a, b \) in \( A \).
   3. Transitive
      If \( a = b \) and \( b = c \) then \( a = c \) for all \( a, b, \) and \( c \) in \( A \).
   4. Substitutive
      If \( a = b \) then substituting \( a \) for \( b \) in any expression will result in an equivalent relation.

II. Let operators \( \cdot \) and \( + \) be defined such that if \( a \) and \( b \) are elements of \( A \) then
   \( a \cdot b \) is in \( A \)
   \( a + b \) is in \( A \)

   This is called the closure property.

III. There exists elements 0 and 1 in \( A \) such that
   \( a \cdot 1 = a \)
   \( a + 0 = 0 \)

IV. The operators \( \cdot \) and \( + \) are commutative for all \( a \) and \( b \) in \( A \).
   \( a \cdot b = b \cdot a \)
   \( a + b = b + a \)

V. The operators \( \cdot \) and \( + \) are distributive for all \( a, b, \) and \( c \) in \( A \).
   \( a + (b \cdot c) = (a + b) \cdot (a + c) \)
   \( a \cdot (b + c) = a \cdot b + a \cdot c \)

VI. For every element \( a \) in \( A \) there exists an element \( \overline{a} \) such that
   \( a \cdot \overline{a} = 0 \)
   \( a + \overline{a} = 1 \)

VII. There are at least 2 elements, \( a \) and \( b \) in \( A \) such that
   \( a \neq b \)
Theorems

From the basic axioms, the following theorems can be derived. These theorems are the essential building blocks of digital logic design.

Th I The elements 0 and 1 are unique

Th II For every a in A
\[ a \cdot a = a \]
\[ a + a = a \]

Th III For every a in A
\[ a \cdot 0 = 0 \]
\[ a + 1 = 1 \]

Th IV The elements 0 and 1 are distinct and \( \bar{0} = 1 \).

Th V For all a and b in A
\[ a + a \cdot b = a \]
\[ a \cdot (a + b) = a \]

Th VI For all a in A
a is unique

Th VII For all a in A
a = not \( \bar{a} \)

Th VIII For all a, b, and c in A
\[ a \left((a + b) + c\right) = a(a + b) + ac = a \]

Th IX For all a, b, and c in A
\[ a + (b + c) = (a + b) + c \]
\[ a (bc) = (ab)c \]

Th X For all a and b in A
\[ a + a \cdot b = a + b \]
\[ a \cdot (a + b) = a \cdot b \]

Th XI For all a and b in A De Morgan’s Law
\[ (a \cdot b) = a + b \]
\[ (a + b) = a \cdot b \]

Note:
\[ (\overline{a \cdot b}) \neq \overline{a} \cdot \overline{b} \]

De Morgan’s Law extends to any number of variables. For 3 variables for example, we have
\[ (\overline{a \cdot b \cdot c}) = \overline{a} + \overline{b} + \overline{c} \]
\[ (a + b + c) = \overline{a} \cdot \overline{b} \cdot \overline{c} \]

4.1.3 Getting Some Practice

When designing digital circuits and systems, one of major objectives is simplicity. Why? Some answers include, lowering the cost of the system, reducing failure rates, sim-
ple systems are easier to build, and they are easier to test. One can use the axioms and theorems just presented to simplify logic expressions. Let’s try a few examples.

**Examples**

When writing logic equations, we tend not to explicitly write the AND operator. We treat the expressions $a \cdot b$ and $ab$ as equivalent. We state that the Boolean variable on the left hand side of the equation has a value of logical 1 under the conditions specified by the logical relations on the right hand side.

\[
F = 1 = a \cdot b + \overline{a} \cdot b \\
= \overline{a} \cdot (a + \overline{a}) \quad \text{axiom V} \\
= \overline{a} \cdot 1 \quad \text{axiom VI} \\
= \overline{a} \quad \text{axiom III}
\]

\[
G = 1 = \overline{a} \cdot \overline{b} + \overline{a} \cdot b + a \cdot \overline{b}
\]

Rewriting first

\[
G = 1 = \overline{a} \cdot \overline{b} + \overline{a} \cdot b + a \cdot \overline{b} \\
= \overline{a} \cdot (b + \overline{b}) + b \cdot (a + \overline{a}) \quad \text{axiom V} \\
= \overline{a} \cdot 1 + b \cdot 1 \quad \text{axiom VI} \\
= b \quad \text{axiom III}
\]

\[
H = 1 = \overline{a} + \overline{b} + b \cdot c + d \\
= \overline{a} \cdot (1 + b) + b \cdot (c + 1) \quad \text{axioms III and V} \\
= \overline{a} \cdot 1 + b \cdot 1 \quad \text{Th III} \\
= \overline{a} + b \cdot d \quad \text{axiom III}
\]

\[
I = 1 = a \cdot b \cdot c + a \cdot b \cdot c + b \cdot c
\]

Rewriting first

\[
I = 1 = a \cdot b \cdot c + a \cdot b \cdot c + b \cdot c \\
= c \cdot (a \cdot b + a \cdot b) \quad \text{axiom V} \\
= c \cdot (a \cdot b + a \cdot b) \quad \text{Th X} \\
= c \cdot (a + 1) \quad \text{axiom V} \\
= c \cdot 1 \quad \text{axiom III} \\
= c \quad \text{axiom III}
\]

\[
J = 1 = a \cdot b + a \cdot c + b \cdot c
\]

Rewriting first

\[
J = 1 = a \cdot b \cdot (c + c) + a \cdot (b + b) \cdot c + (a + a) \cdot b \cdot c \\
= (a \cdot b \cdot c + a \cdot b \cdot c) + (a \cdot b \cdot c + a \cdot b \cdot c) + (a \cdot b \cdot c + a \cdot b \cdot c) \quad \text{Th II} \\
= b \cdot c \cdot (a + a) + a \cdot c \cdot (b + b) \quad \text{axiom V} \\
= a \cdot b \cdot 1 + a \cdot c \cdot 1 \quad \text{Th III} \\
= a \cdot b + a \cdot c \quad \text{axiom III}
\]

\[
K = 1 = a + \overline{a} \cdot b + (a + b) \cdot c + (a + b + c) \cdot d
\]

\[
K = 1 = a + \overline{a} \cdot b + (a + b) \cdot c + (a + b + c) \cdot d \\
= a + \overline{a} \cdot b \cdot c + a \cdot b \cdot c \cdot d \quad \text{Th XI} \\
= a + \overline{b} + b \cdot c + b \cdot c \cdot d \quad \text{Th X} \\
= a + \overline{b} \cdot c + \overline{c} \cdot d \quad \text{Th X} \\
= a + \overline{b} \cdot c + d \quad \text{Th X}
\]
4.2 Defining and Using Relations

Using operands and the basic binary operators, we can easily begin to represent real world objects as the operands and to express relations between and among those objects or operands. We’ve learned how to express those relations in the form of Boolean equations. Now, we’ll look at other means.

4.2.1 The Truth Table

An alternate method for expressing a Boolean relation is to use a truth table. A truth table is a list of all possible combinations of the values of the variables in the relation and value of that relation (true or false) for each such combination.

In the simple case, a single variable, A, it will have two values, 0 or 1, false or true. In a more complex case of two variables, A and B, each can be either true or false and thus yield four combinations: A, B ⇒ { 0,0; 0,1; 1,0; 1,1}. Continuing, the case of n variables will produce N = 2^n possible combinations.

Let’s assume that a logical expression based upon the two variables is true for two of the combinations and false for the remainder. One possible expression can be written in equation form as,

G = 1 = A \overline{B} + A B

Certainly there are others as well...can you write them?

In tabular form, we will have the truth table given in Figure 4.2.

A truth table can be used to demonstrate the truth (or falsity) of a proposition or Boolean relation. Each relation has two sides, a left hand side (LHS) and a right hand side (RHS). One can state that if the left and right hand sides have the same truth values for all combinations of variables, then they must be equivalent.

We can use a truth table to easily prove De Morgan’s Law. The theorem is restated for reference

\( (a \overline{b}) = \overline{a} + \overline{b} \)

To demonstrate the proof, we must show that for all combinations of inputs, the LHS has the same truth values as the RHS. The truth table in Figure 4.3 expresses the truth values for both sides of the equation for all input combinations. As the truth table illustrates, the two expressions are equivalent.
Let’s apply what we’ve learned in a slightly more complex logical relationship.

**Example 4.0**

Our company has just won a contract to design and build a simple tea vending machine as pictured in Figure 4.4. This is an unusual vending machine in that, in addition to dispensing tea, it offers a critique of whether or not the selection is proper - a vending machine with an attitude.

The customer has the option of selecting any one or a combination of three items: tea, lemon, and milk, \( \{T, L, M\} \).

To start the design, we begin with a high level view of the system. That view is from the customer’s point of view - the view outside of the system - and identifies all of the inputs and outputs.

For this system, there are three input variables, Tea, Lemon, and Milk \( \{T, L, \text{ and } M\} \) and two output variables Good Tea and Bad Tea \( \{\text{GT and BT}\} \). The top level diagram is given in Figure 4.5.

The three input variables will give a total of eight combinations. The truth table relating the input combinations to each of the two outputs is given in Figure 4.6. In discussions with the customer, we, as the designer, have decided which output values we want for each input combination.
Each of the rows is labeled with the decimal equivalent of the binary combination expressed by the three input variables.

We can now extract the following two equations from the truth table

\[ GT = 1 = T \bar{L} \bar{M} + T L \bar{M} + T \bar{L} M \]
\[ BT = 1 = T \bar{L} \bar{M} + T \bar{L} M + T L \bar{M} + T L M + T L M \]

These two equations capture the desired logical relationships between the inputs to the system and the corresponding outputs.

4.2.2 Minterms

The two output expressions in the previous example are written in what is called sum of products form. Each product is called a minterm; each is also known as an implicant. These two expressions can also be written as,

\[ GT = 1 = m_4 + m_5 + m_6 \]
\[ BT = 1 = m_0 + m_1 + m_2 + m_3 + m_7 \]

The small \( m \) indicates a minterm and the subscripts are the binary equivalent of the minterm variables. For example the expression, \( m_4 \) denotes a minterm. The subscript 4 is 100 in binary and expressed the truth values of the three variables, \( T, L, \) and \( M \). The variable \( T \) has the value 1 and the variables \( L \) and \( M \) both have the value 0. Thus, \( T \bar{L} \bar{M} \).

**Note:**

An expression written as a series of minterms is also written in sum of products form. An expression written in sum of products form is not necessarily composed of all minterms.

The two expressions can also be written as,

\[ GT = \Sigma 4, 5, 6 \]
\[ BT = \Sigma 0, 1, 2, 3, 7 \]

reflecting the subscripts of the corresponding minterms.

The two expressions can be simplified using the axioms and theorems from Boolean algebra,

\[ GT = T \bar{L} \bar{M} + T L \bar{M} + T \bar{L} M \quad \text{Th II} \]
\[ = T \bar{M}( L + L) + T \bar{L}(M + \bar{M}) \quad \text{axiom VI} \]
\[ = T \bar{M} + T \bar{L} \quad \text{axiom III} \]
\[ = T(\bar{M} + \bar{L}) \]

\[ BT = \bar{T} \bar{L} \bar{M} + \bar{T} \bar{L} M + \bar{T} L \bar{M} + \bar{T} L M + T L M \]
\[ = \bar{T} \bar{L}(M + M) + \bar{T} L(M + M) + TLM \quad \text{axiom VI} \]
\[ = \bar{T} L + TL + TLM \quad \text{axiom III} \]
\[ = T (\bar{L} + L) + TLM \]
\[ = \bar{T} + TLM \quad \text{axiom VI} \]
\[ = \bar{T} + LM \quad \text{Th X} \]
4.2.3 Maxterms

In the previous example, the two output equations were written for the cases in which the outputs had a truth value of logical 1. The equations could also be written for the cases in which the outputs had a truth value of logical 0 and then complemented since 1 = 0 and vice versa. Taking that approach, the two output equations are first written as,

\[
\begin{align*}
GT &= 0 = T \bar{L} \bar{M} + T \bar{L} M + \bar{T} L \bar{M} + T L \bar{M} \\
BT &= 0 = T L \bar{M} + T L M + T \bar{L} M
\end{align*}
\]

Then inverted to give,

\[
\begin{align*}
GT &= 0 \quad = \neg ( T \bar{L} \bar{M} + T \bar{L} M + \bar{T} L \bar{M} + T L \bar{M} ) \\
BT &= 0 \quad = \neg ( T L \bar{M} + T L M + T \bar{L} M)
\end{align*}
\]

In minterm format, we have

\[
\begin{align*}
GT &= 0 \quad = (m_0 + m_1 + m_2 + m_3 + m_7) \\
BT &= 0 \quad = (m_4 + m_5 + m_6)
\end{align*}
\]

Applying De Morgan’s Law,

\[
\begin{align*}
GT &= \bar{0} = \bar{m}_0 \cdot \bar{m}_1 \cdot \bar{m}_2 \cdot \bar{m}_3 \cdot \bar{m}_7 \\
BT &= \bar{0} = \bar{m}_4 \cdot \bar{m}_5 \cdot \bar{m}_6
\end{align*}
\]

If De Morgan’s Law is now applied to each element of these two equations we will have,

\[
\begin{align*}
GT &= \bar{0} = ( \bar{T} + \bar{L} + \bar{M} ) \cdot ( \bar{T} + \bar{L} + \bar{M} ) \cdot ( \bar{T} + L + \bar{M} ) \cdot ( T + L + M ) \\
BT &= \bar{0} = ( T + L + M ) \cdot ( T + L + M ) \cdot ( T + L + M )
\end{align*}
\]

The two output expressions are now written in what is called product of sums form. Each product is called a Maxterm.

\[
\begin{align*}
GT &= M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_7 \\
BT &= M_4 \cdot M_5 \cdot M_6
\end{align*}
\]

We see that maxterm \( M_i = \bar{m}_i \). These two expressions can also be written as,

\[
\begin{align*}
GT &= \Pi 0, 1, 2, 3, 7 \\
BT &= \Pi 4, 5, 6
\end{align*}
\]

**Note:**

An expression written as a series of maxterms is also written in product of sums form. An expression written in product of sums form is not necessarily composed of all maxterms.

4.3 Implementation

Let’s now look at building these functions. From the equations, it’s evident that three functions necessary:

- **AND**
- **OR**
- **NOT**
The physical pieces of hardware that are used to implement such expression are called gates. The nomenclature was probably derived from concept of gating a signal or allowing signal to pass.

We use 3 symbols to represent these basic logical functions as depicted in Figure 4.7. Although drawn with 2 inputs each, they can be drawn with any number. Practical values are typically 2, 3, or 4 inputs.

Let’s now look at several examples to see how we can illustrate a combinational logic function using logic symbols.

**Example 4.1**

Consider the function

\[ F = 1 = AB + CD \]

F can be rewritten as

\[ S = AB \]
\[ T = CD \]
\[ F = S + T \]

Thus, first we implement the OR part, then the two AND parts, and finally we bring them together as in Figure 4.8.

**Example 4.2**

Now let’s try

\[ F = A \bar{B} \]
The implementation is given in Figure 4.9.

Let’s now return to the design if the tea machine. We can write the equation expressing the output \( GT \) in three different ways,

1. \( GT = 1 = T \overline{M} + T \overline{L} \)
2. \( GT = 1 = T( \overline{M} + \overline{L} ) \)
3. \( GT = 1 = T ( M \overline{L} ) \)

which give rise to the following three implementations as presented in Figure 4.10.

### 4.4 Karnaugh Maps

When properly applied, the axioms and theorems introduced at the opening of this tutorial can serve as a rather effective means for simplifying combinational logic equations. The key words here are *properly applied*. As human beings, we tend to see patterns *applied* in a graphical or acoustic medium before seeing a similar pattern in a collection of num-
bers or text. One such graphical tool that we have available to us is called a **Karnaugh map** or **K Map**.

The underlying theory behind the K Map is based upon axioms III and VI that state:

\[
\begin{align*}
    a \cdot 1 &= a \\
    a + \overline{a} &= 1
\end{align*}
\]

Let’s now look at the following equation:

\[ F = 1 = A \overline{B} + AB = m_2 + m_3 \]

The equation reduces to simply \( A \) using the two axioms III and VI.

Using a similar reasoning, the following equations can also be reduced:

\[
\begin{align*}
    G &= 1 = A B \overline{C} + A B C = m_6 + m_7 \\
    &= AB \\
    H &= 1 = A \overline{B} C + A B C = m_5 + m_7 \\
    &= AC \\
    I &= 1 = \overline{A} B C + A B C = m_5 + m_7 \\
    &= BC
\end{align*}
\]

Each of the previous equations can be simplified because each contains two minterms that differ only in the polarity of a single variable. Thus, each can be simplified using the afore noted pair of axioms.

As a first step in developing a K Map, let’s examine the input portion of several truth tables. Here-to-fore, we have been writing the input side of a truth table as shown in Figure 4.11,

![Figure 4.11](image)

If the order of the rows is slightly rearranged, the total number of combinations will remain unchanged, however, following the re-order, we will have only a single variable
change between adjacent rows as we see in Figure 4.12 below. That new ordering is called a *Gray sequence*.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>m₀</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m₁</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>m₃</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>m₂</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>m₀</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m₁</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>m₃</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>m₂</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>m₆</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>m₇</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>m₅</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>m₄</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 4.12
Rewriting the Two and Three Variable Truth Tables using a Gray Sequence*

The modified ordering retains the same number of variable combinations; however, it’s quickly evident that if minterms in any two adjacent rows are combined that one of the variables drops out.

Consider the truth table with two input variables,

Case I

\[ m₀ + m₁ = \overline{A} \]
\[ m₁ + m₃ = B \]
\[ m₃ + m₂ = \overline{A} \]

and a tricky one,
\[ m₂ + m₀ = \overline{B} \]

Now consider the truth table with three input variables,

Case II

\[ m₀ + m₁ = \overline{A} \cdot \overline{B} \]
\[ m₁ + m₃ = \overline{A} \cdot C \]

etc.

One additional rewriting of the truth table brings us to the K Map.

### 4.4.1 Going Forward

**Two Variable K Maps**

To create a two variable K Map, we reorganize the minterms into the matrix like map we see in Figure 4.13. We have numbered each of the cells with the corresponding minterms. Observe that each cell has two adjacent cells and that those cells (that is, the minterms represented by those cells) differ by single variable. Once again, this is a Gray sequence.
Returning to Case I from above, let minterms 0 and 1 be true and minterms 2 and 3 be false. This information is entered into the map as 1’s in the cells for \( m_0 \) and \( m_1 \) and 0’s in the cells for \( m_2 \) and \( m_3 \), as we see in the K Map in Figure 4.14.

\[
\begin{array}{c|c|c}
A & 0 & 1 \\
\hline
0 & m_0 & m_1 \\
1 & m_2 & m_3 \\
\end{array}
\]

**Figure 4.13**
Minterm Cells in a Two Variable K Map

The algebraic simplification,

\[
F = \overline{A} \overline{B} + \overline{A} B = \overline{A}
\]

is the same as looking at map and combining the two 2 true minterms marked by the cartouche in the next K Map in Figure 4.15.

\[
\begin{array}{c|c|c}
A & 0 & 1 \\
\hline
0 & 1 & 1 \\
1 & 0 & 0 \\
\end{array}
\]

**Figure 4.14**
Entering Values into the Minterm Cells in a Two Variable K Map

\[
\begin{array}{c|c|c}
A & 0 & 1 \\
\hline
0 & 1 & 1 \\
1 & 0 & 0 \\
\end{array}
\]

**Figure 4.15**
Graphically Grouping Minterm Cells in a Two Variable K Map

Now let’s reverse the roles of minterms 0 and 1 with 2 and 3 as we see in Figure 4.16.

\[
\begin{array}{c|c|c}
A & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

**Figure 4.16**
Entering Values into the Minterm Cells in a Two Variable K Map
The simplification
\[ F = A \overline{B} + A B = A \]
is the same as looking at map and combining the two true minterms as we see in Figure 4.17.

These next two maps, in Figure 4.18, contain the true minterms \( m_0 \) and \( m_3 \) and \( m_1 \) and \( m_2 \).

In the left hand map, the function is given by
\[ F = 1 = \overline{A} B + A \overline{B} \]
and in the right hand map, it is given by
\[ F = 1 = \overline{A} B + A \overline{B} \]
and we see that no logic simplification possible. In general to be able to combine minterms into a reduced expression, they must be in adjacent cells; that is differ by a single variable change. Cells on a diagonal cannot be combined.

Three Variable K Maps

The K Map can easily be extended to three variables. In making the extension, the cell ordering still follows the Gray code pattern. The diagram in Figure 4.19 illustrates the three variable K Map and identifies the placement of the minterms.
Returning to Case II from above, let minterms 0 and 1 be true and the remaining minterms be false. This information is entered into the map as 1’s in the cells for \( m_0 \) and \( m_1 \) and 0’s in the remainder as Figure 4.20 illustrates.

The reduction

\[
F = 1 = m_0 + m_1 = \overline{A} \overline{B} C + \overline{A} B C = \overline{A} \cdot B
\]

is the same as looking at map and combining the two true minterms marked above.

The reduction of the expression,

\[
G = 1 = m_1 + m_3 = \overline{A} B C + \overline{A} B C = \overline{A} \cdot C
\]

follows in a similar manner as we see in Figure 4.21. Once again, we are combining or grouping adjacent minterms.

Let’s try a more complex pattern as in Figure 4.22.

\[
H = 1 = \sum 4,5,7
\]
From the K Map,
\[ H = 1 = B \overline{C} + A \overline{B} \]

Observe that the minterm pair comprised of \( m_4 \) and \( m_6 \) was not considered. Since all of the cells containing logical 1’s had been covered, adding this pair would be redundant. As a point to note, in general, the 0 terms are not entered into the map.

The following function can be reduced in several different ways.
\[ J = 1 = \sum 0,4,5,7 \]

The expression and a reduction are shown in Figure 4.23. The most efficient cover is shown below and takes advantage of the fact that the top and bottom row in the map are adjacent; that is, they differ only in one variable. The reduced function is given as
\[ J = 1 = \overline{B} \overline{C} + A \overline{C} \]

In general on a three variable K Map, adjacent cells can be collected into groups of 2, 4, or 8. Several possible groupings are illustrated by the shaded areas in the maps in Figure 4.24.
4 Variables

Adding one additional variable is not much more complex. As we’ve seen in Figure 4.25.

<table>
<thead>
<tr>
<th>A B</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>m0</td>
<td>m1</td>
<td>m3</td>
<td>m2</td>
</tr>
<tr>
<td>01</td>
<td>m4</td>
<td>m5</td>
<td>m7</td>
<td>m6</td>
</tr>
<tr>
<td>11</td>
<td>m12</td>
<td>m13</td>
<td>m15</td>
<td>m14</td>
</tr>
<tr>
<td>10</td>
<td>m8</td>
<td>m9</td>
<td>m11</td>
<td>m10</td>
</tr>
</tbody>
</table>

Figure 4.24
Some Possible Groupings of Minterms in a Three Variable K Map

Figure 4.25
Minterm Cells in a Four Variable K Map
The resulting map adds more adjacencies. There are now several new adjacencies.

Corners
- \( m_0, m_2, m_8, \text{ and } m_{10} \)

Edges
- Top and bottom
- Left and right

**Example 4.3**

Given the logical relation

\[ F = 1 = \sum 0, 1, 4, 5, 10, 11, 14, 15 \]

We have the K Map in Figure 4.26. Grouping the minterms into two groups of four as illustrated gives the following reduced expression

\[ F = 1 = \overline{A} \overline{C} + AC \]

**Example 4.4**

Given the logical relation

\[ G = 1 = \sum 0, 2, 5, 7, 8, 10, 14, 15 \]

We have the K Map in Figure 4.27. The cover takes advantage of the corner adjacencies to give the following reduced expression,

\[ G = 1 = \overline{B} \overline{D} + BD \]
From the logical relation

\[ H = 1 = \sum 0, 2, 5, 8, 10, 15 \]

we have the K Map in Figure 4.28. The cover can still take advantage of the corner adjacencies; however, the two center terms cannot be combined with any other minterms. The reduced expression is given as,

\[ H = 1 = BD + \overline{A} BC D + AB CD \]

4.4.2 Going Backward

We have seen how to use an existing K Map to reduce a logical expression. The next step is to use the K Map to try to reduce an existing function to a simpler form.

2 Variables

Let’s begin with a 2 variable function

\[ F = 1 = A + \overline{A}B + AB \]

\( \overline{A}B \) is minterm 1

\( AB \) is minterm 3

A is the entire bottom row, as we see in the left hand K Map in Figure 4.29.

The function can be reduced to

\[ F = 1 = A + B \]
To reduce the following maxterm expression, we first convert to minterm form,

$$H = 1 = \prod(0, 1, 3) = M_0 \cdot M_1 \cdot M_3$$

$$H = 0 = (M_0 \cdot M_1 \cdot M_3) = M_0 + M_1 + M_3 = m_0 + m_1 + m_3$$

The expression is shown and reduced in the K Map in Figure 4.30.

Grouping and extracting,

$$H = 0 = A + B$$

Inverting,

$$H = 1 = A \cdot B$$

Observe that we could have arrived at the same answer by covering the single logical 1 in the map.

3 Variables

Working with a greater number of variables is more interesting. We start with the function,

$$F = 1 = A \cdot B + A(\overline{B} + C)$$

Expanding first,

$$F = 1 = A \cdot B + A \cdot \overline{B} + A \cdot C$$

The first term is independent of C. As a result, logical 1’s must be entered into both cells in the third row of the K Map. For a similar reason, logical 1’s must be entered into the fourth row of the map. The third expression is independent of B thus requires logical 1’s in last two entries of the second column.

The resulting K Map now appears as we see in Figure 4.31,
From the map, the reduced expression is given as,
\[ F = 1 = A \]

The logical expression,
\[ G = 1 = A \overline{B} \overline{C} + A C + \overline{B} C \]
when entered on a K Map as in Figure 4.32,

\[
\begin{array}{c|c|c}
A & B & 0 & 1 & C \\
00 & 1 \\
01 & \\
11 & 1 & 1 \\
10 & 1 \\
\end{array}
\]

reduces to
\[ G = 1 = A B + \overline{B} C \]

4 Variables

Expressions with four variables follow naturally.

**Example 4.6**

The logical equation
\[ H = 1 = A \overline{C} + A \overline{B} D + \overline{B} C D \]
when entered on a K Map as in Figure 4.33

\[
\begin{array}{c|c|c|c|c}
A & B & C & D \\
00 & 0 & 1 & 1 & 1 \\
01 & 1 & 1 & 1 & 1 \\
11 & 1 & 1 & 1 & 1 \\
10 & 1 & 1 & 1 & 1 \\
\end{array}
\]

and simplified gives,
\[ H = 1 = A \overline{C} + A \overline{B} D + \overline{B} C D \]
Example 4.7

\[ J = \overline{A} \overline{B} \overline{C} + \overline{A} \overline{C} D + B C D + \overline{A} \overline{B} C \overline{D} \] when entered on the K Map in Figure 4.34

<table>
<thead>
<tr>
<th>A B</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.34
Reducing a Logical Equation Using a Four Variable K Map

reduces to
\[ J = 1 = \overline{A} \overline{C} D + B C D + \overline{A} \overline{B} \overline{D}. \]

4.5 Don’t Care Variables

If we have a system with three inputs such as the one in Figure 4.35,

![Generic System with Three Inputs and a Single Output](image)

those inputs can potentially take on up to 8 different values. Assume that it is the case that, based upon physical constraints placed upon those inputs some of those combinations may not be possible. We can use such information to our advantage to simplify the design of the system.

Minterms that are so constrained are called don’t care conditions because we don’t care what values they take on in a logic equation since they cannot occur. That is, we can give them values of 0 or 1 if such an assignment helps to simplify the design.

Consider the function F given by the following logical expression,
\[ F = 1 = \Sigma 0, 1, 5 \]
We enter these as 1’s as usual in the K Map in Figure 4.36.

\[ F = 1 = A \overline{B} + \overline{B} C \]

As the map stands, the expression reduces to,

Assume combinations given by minterms 4 and 6 can never occur

Don’t Care = \( \Sigma 4, 6 \)

These are entered as X’s as we see in Figure 4.37 below.

If we assign a value of 0 for don’t care minterm 6 and 1 for don’t care minterm 4, the expression for \( F \) now reduces to

\[ F = 1 = B \]

We now have a much simpler expression. We can take advantage of don’t care conditions in a system design only under the following circumstances.

1. If it can be guaranteed that such minterms will never occur in the input set.
2. If such minterm combinations can occur, then it must be guaranteed that, no matter how small their probability of occurrence, they can never produce a risk to life or property.

4.6 Memory Devices

Today’s digital systems utilize a variety of memory devices both as SSI components and as building blocks in Very Large Scale Integrated (VLSI) circuits, microprocessors,
array logics or complex logic devices. Such devices fall into two major categories, latches and flip-flops, and are distinguished by how and when data is entered and held.

4.6.1 Set-Reset Latch

The set-reset, SR, latch is a simple memory device that can be built from two inverting combinational logic devices; one can use either NAND or NOR gates. Such a device is given in the circuit in Figure 4.38 utilizing two cross-coupled NOR gates.

If the two inputs (Reset and Set) are both in their quiescent state at logical 0, the state (output) of the latch will not change. If the line labeled Set is put at logical 0 and that labeled Reset is put at logical 1, the output labeled Q will go to 0 and Q will go to 1.

The state of the device, generally specified by the value of the Q output, is now a logical 0. If both the Set and Reset are returned to their quiescent condition (both having a value of logical 0) the state of the latch will not change. It is holding or storing the value of logical 0.

If the Set input is now put at logical 1 and the Reset input put at logical 0, the output labeled Q will go to 1 and Q will transition to 0. The state of the latch has changed to logical 1. Once again, if both inputs are returned to logical 0, the state of the latch will not change; it is storing the value of logical 1.

We express sequence of input changes and subsequent state changes in the state diagram in Figure 4.39. The diagram illustrates that we have two states: {0, 1} and two inputs {Set, Reset}. The arcs are labeled with the Set and Reset values effecting the state change or ensuring no change.

Summarizing the results; by setting either the Set or Reset input to a logical 1, the latch will make a transition to a different state. When both inputs are at logical 0, the device remembers what state it is in and does not change. If both inputs are logical 1, both latch outputs, Q and Q will transition to logical 0. The state of device depends upon which one removed first; this condition is meaningless, violates the fundamental rule for latches and flip-flops, and is thus strongly discouraged.
The behavior of the device can also be described in tabular form in the state table in Figure 4.40. The left two columns express the inputs to the latch and the right two columns give the state of the device at time \( t = n+1 \) in response to the inputs at time \( t = n \). With both the Set and Reset inputs in the logical 0 state, the state of the device does not change from its value at time \( t = n \).

Such a latch is said to exhibit asynchronous behavior. The state changes are not associated with any timing element; they follow any changes in the Set and Reset inputs.

4.6.2 The Gated R-S Latch

We gain greater control over when state changes occur with a slight modification to the basic latch. We add a specialized control signal called a gate or strobe. The logic diagram in Figure 4.41 reflects that change.

In the gated latch, when the Gate is in the logical 0 state, the two inputs to the latch portion, Set' and Reset', are both held at logical 0 as well. The latch cannot change state. When the Gate is in the logical 1 state, the latch will follow the Set and Reset, inputs as before.

We use either of the logic symbols in Figure 4.42 for the latch to reflect the different ways that the device is gated. For the circuit presented above, the output follows the input when the Gate is a logical 1. Alternatively, by adding an inverter in the Gate path, the output will follow the inputs when the Gate is a logical 0. The output for the device on the left responds to the Set and Reset inputs when the Gate is in the logical 1 state and that on the right responds when the Gate is in the logical 0 state as indicated by the bubble on the Gate.
4.6.3 The Gated J-K Latch

Adding the gate to the SR latch does not solve the problem of simultaneous logical 1 values on both inputs, however. To address that problem, we modify the circuit further by adding a third signal to each input gate and also re-label the input signals as J and K. The new circuit is shown in the logic diagram in Figure 4.43.

By connecting the output signals back to the input (the Set’ and Reset’ inputs to the NOR gates in the latch) the Q and Q outputs must have opposite values. We cannot have the same situation as with either SR device.

When the Gate is low, the state of the latch remains unchanged as before. When the Gate is high, the states of J and K input signals are propagated to output as shown in the accompanying state table in Figure 4.44. When both inputs have logical 1 values, the latch will continuously alternate states.

The logic symbols for the gated JK latch are given in Figure 4.45.

In practice, one rarely finds the JK latch. We have introduced the concept at this point so that we can use it later when we discuss flip-flop implementations.

4.6.4 The Gated D Latch

Another modification to the basic SR latch is given in the circuit in Figure 4.46. The modification also ensures that the Set’ and Reset’ can never both have a value of logical 1. As in the earlier gated latch designs, with the Gate low,
the SR latch cannot change states. With the Gate high, the state of D input signal is propagated to output as shown in the state table,

<table>
<thead>
<tr>
<th>D</th>
<th>Q_{n+1}</th>
<th>Q_{n+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.47
A State Table for a D Latch

The logic symbols for the gated D latch are given in Figure 4.48.

**4.7 Flip-Flops**

When the Gate input is asserted on any of the gated latches, the output state will track changes in the input signals. Based upon such behavior, the latch is said to be transparent; input changes are directly visible on the outputs. In some applications, such behavior is useful; in general, however, greater control over the device output state is preferred.

The term flip-flop applies to a device that accepts input signals and only affects a state change at its outputs at specific times - times that are established by the state of a clocking signal. The flip-flop, none-the-less, still utilizes a latch as an internal storage element.

Such increased control is typically achieved through two different schemes. One approach constrains the output of the device to change only when the clocking signal changes state. Such devices are called edge triggered flip-flops. The second scheme accepts input data into a temporary storage latch when the clocking signal is in one logical state (logical 1 or logical 0 based upon the design). The data being temporarily stored may be repeatedly changed as long as the clock signal remains unchanged. The contents of the temporary storage are then transferred to a second storage element when the clock makes a transition to the opposite state. On such a transition, further input changes are temporarily not accepted. Such a device is called a master-slave flip-flop.

**4.7.1 Master-Slave Flip-Flops**

The master-slave flip-flop utilizes two cascaded latches that are enabled on opposite polarities of the Gate input as illustrated in the drawing in Figure 4.49 for an SR master-slave flip-flop.

The components within the dashed box make up the flip-flop. When the Clock input (which controls the internal Gate signals) is in the logical 1 state, the Set and Reset inputs to the master flip-flop are able to affect the outputs of that device. Those changes are not seen on the output of the slave device because the gate for that device is disabled by the state of the gating signals.
When the state of the clock input changes to the logical 0 state, the gated input to the slave device is enabled and the contents of the master are transferred to the slave and thus to the output of the device. In addition, the state of the master gate blocks inputs from entering that device.

The behavior of the master-slave SR device is illustrated in the timing diagram in Figure 4.50.

The diagram illustrates that changes in the input signals during intervals in which the clock input is in the low state are ignored and any transients that occur when the master is enabled are accepted by the master latch. Those stored by the master device are subsequently transferred to the slave device.

Observe that the output of the slave appears to change on the negative going clock edge of the clock signal. In reality the state change is effected on the changing level of the gate to the slave device. With the addition of an inverter on the input to the master latch (as we did earlier) one can cause the output of the slave to appear to change on the rising edge of the clock signal. The logic symbols for the SR device are presented as illustrated in Figure 4.51.

The Verilog module in Figure 4.52 gives an RTL level implementation of the basic SR device. The included delays model two of the more important delays that must be considered when designing any such device into an application. The first, delay (delay1) models the delay from a change in the clock input to the corresponding change in the state of
the Q output. The second models the difference between state changes in the Q and the \( \overline{Q} \) outputs of the device; these two values are typically not symmetric. The timing diagram in Figure 4.53 expresses the modeled timing.

In the diagram, a change in the state of the Q output is slightly faster than a similar change in the \( \overline{Q} \) output. Such will not always be the case. The vendor’s data sheets or the process specifications for a proprietary design will provide the specific range of parameter variation that must be taken into consideration. When performing a constraints analysis of a system to ensure that a hard real-time deadline can be met, such differences can be significant.

The D and JK master-slave flip-flops are implemented as illustrated in the diagrams in Figure 4.54 and Figure 4.55 respectively.
Their logic diagrams for the three different flip-flop types are given in Figure 4.56. The diagrams include a small triangle on the clock input. The triangle symbol indicates that the output changes on a transition.

RTL Verilog modules for the D and JK devices are given in code fragments in Figure 4.57. The same modeling parameters as introduced in the SR model are again used in each of the flip-flop models here.
4.7.2 Edge Triggered Devices

Edge triggered devices change state on the rising or falling edge of the input clock. A master-slave flip-flop, implemented as illustrated above, is one form of edge sensitive device. Most commercial implementations, however, typically don’t use such a scheme. From a functional point of view, their behavior remains the same.

4.7.3 Preset and Clear

We now add one final piece to the flip-flop. As we do with software variables, we always want to initialize hardware variables to a known state. There are also occasions during operation when the ability to force an embedded system into a known state is desirable or necessary. This capability is supported by incorporating two additional signals into the design of the device. These signals, designated as Preset and Clear, are asynchronous to and override any other input signals. The affect of the Preset is to force the flip-flop into the logical 1 state; that of the Clear is to force the device into the logical 0 state. These are added to the circuit in Figure 4.58.
We now extend the D flip-flop Verilog model, as shown in Figure 4.59, to reflect the two asynchronous inputs. Observe that an additional delay from the reset to the q output has been added to the model.

```verilog
// D Flip-Flop Module
module DFF(q, qBar, D, clk, set, rst);
    input         D, clk, set, rst;
    output        q, qBar;
    parameter     delay0 = 2;     // delay reset to q
    parameter     delay1 = 3;     // delay clock to q
    parameter     delay2 = 2;     // delay for qBar with respect to q
    reg q;

    not #delay2 n1(qBar, q);
    always@ (negedge rst or negedge set or posedge clk)
    begin
        if(rst==0)
            #delay0 q = 0;
        else if (set==0)
            q = 1;
        else
            #delay1 q = D;
    end
endmodule
```

**Good Design Practice:**
Always ensure that all hardware state variables are initialized to a known state after power is first applied.

We can extend the design to each of the other types of latches and flip-flops in a similar manner. The logic symbols for the three flip-flop types so modified appear in the diagram in Figure 4.60.
4.7.4 Characteristic Equations

The functional behavior of each different type of flip-flop or latch is described by its characteristic equation. Such an equation formally relates the next state output of the device to its current state output and input values. These are summarized in Table 4.0 for the devices we've studied exclusive of the JK latch.

Table 4.0
Characteristic Equations for the Basic Flip-Flop Types

<table>
<thead>
<tr>
<th>Device Type</th>
<th>Characteristic Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR Latch</td>
<td>$Q_{n+1} = S + R Q_n$</td>
</tr>
<tr>
<td>D Latch</td>
<td>$Q_{n+1} = D$</td>
</tr>
<tr>
<td>Gated SR Latch</td>
<td>$Q_{n+1} = G (S + \overline{R} Q_n) + \overline{G} Q_n$</td>
</tr>
<tr>
<td>Gated D Latch</td>
<td>$Q_{n+1} = G D + \overline{G} Q_n$</td>
</tr>
<tr>
<td>SR Flip-Flop</td>
<td>$Q_{n+1} = S + \overline{R} Q_n$</td>
</tr>
<tr>
<td>JK Flip-Flop</td>
<td>$Q_{n+1} = J \overline{Q}_n + K Q_n$</td>
</tr>
</tbody>
</table>

4.8 Summary

In this tutorial we began with a brief review of the basics of Boolean algebra. We learned how to use Boolean algebra to formulate and manipulate logical equations then implement the resulting equations using the AND, OR, and NOT gates. The Karnaugh map was introduced as an easy to use graphical logic reduction tool for small problems. We concluded our review of combinational logic with don’t care conditions and examined how they can be used to further simplify logic expressions.

We then introduced and studied the basic memory devices known as latches and flip-flops. In our studies, we began with the basic storage device, added capabilities to support greater control over when and how the device changes state in response to its inputs, then developed the characteristic equations describing the behavior of the RS, D, and JK latch and flip-flop types.
References


