Chapter 3: Introduction to Asynchronous Logic

Objectives of Chapter 3

- Use the principles of Boolean algebra to simplify and manipulate logical equations and turn them into logical gate designs
- Create truth tables to describe the behavior of a digital system
- Use the Karnaugh Map to simplify your logical designs
- Describe the physical attributes of logic signals, such as rise time, fall time and pulse width
- Express system clock signals in terms of frequency and period
- Manipulate time and frequency in terms of commonly used engineering units such as Kilo, Mega, Giga, milli, micro, and nano

Introduction

Before we immerse ourselves in the rigors of logical analysis and design, it's fair to step back, take a breath and reflect on where all of this came from. We may have been given the erroneous impression that logic sprang forth from Silicon Valley with the invention of the transistor switch. However, this is not quite correct. The genesis of our industry can be traced back almost 2400 years to ancient Greece.

We generally attribute the birth of modern logical analysis to the Greek philosopher, Aristotle, who was born in 384 BC, and is generally acknowledged to be the father of modern logical thought. Thus, if we're to be completely accurate (and perhaps a bit generous) we might say that Aristotle is the father of the modern digital computer. However, if we were to look at the mathematics of the gates we've just been introduced to, we may find it difficult to make the leap to Aristotelian Logic.. However, one simple example might give us a hint.


Figure 3.1: Aristotle

The basis of Aristotle's logic revolves around the notion of ***deduction***. You may be more familiar with this concept from watching old Sherlock Holmes movies, but Sherlock didn't invent the idea of deductive reasoning, Aristotle did.

Aristotle proposes that a deductive argument has "things supposed," or a premise of the argument, and what "results of necessity", is the conclusion..[1]

One often cited example is:

1. Humans are mortal.
2. Greeks are human.
3. Therefore, Greeks are mortal.

We can convert this to a slightly different format:

1. Let A be the state of human mortality. It may be either TRUE or FALSE that human beings are mortal.
2. Let B be the state of Greeks. It may be TRUE or FALSE that natives of Greece are human beings.
3. Therefore, the only TRUE condition is that if A is TRUE AND B is TRUE then Greeks are mortal. Or, C (Greek mortality) = A * B.

We can trace our ability to manipulate logical expressions to the English mathematician and logician, George Boole (1816-1864).

"In 1854 Boole published an investigation into the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities. Boole approached logic in a new way reducing it to a simple algebra, incorporating logic into mathematics. He pointed out the analogy between algebraic symbols and those that represent logical forms. It began the algebra of logic called ***Boolean algebra*** which now finds application in computer construction, switching circuits etc."[2]

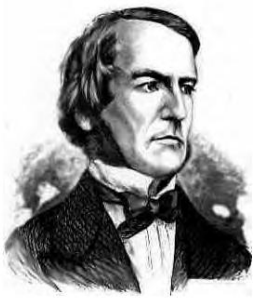


Figure 3.2: George Boole

Boolean algebra provides us with the toolset that we need to design complex logic systems with the certainty that the circuit will perform exactly as we intended. Also, as you'll soon see, the process of designing a digital circuit often leads to designs that are quite redundant and in need of simplification. Boolean algebra gives us the tools we need to simplify the circuit design and be confident that it will work as designed with the absolute minimum of hardware. As engineers, this is exactly the kind of analytical tool we need because we are most often tasked with producing the most cost-effective design possible.

In working with digital computing systems, there are two distinctly different binary systems with which we'll need to become familiar. Earlier in this course we were introduced to the binary number system. This is convenient, because we need our computer to be able to operate on numbers larger than one bit in width. The other component of this is the operation of the hardware as a digital system. In order to understand how the numbers are manipulated, we need to learn the principles of binary logical algebra, or Boolean algebra.

Therefore, the first step in the process is to state some of the laws of Boolean algebra. In most cases, the laws should be obvious to you. In other cases, you might have to scratch your head a bit until you see that we're dealing with logical operations on variables that can only have two possible values, and not the addition and multiplication of variables that can take on a continuum of values. Also, be aware that there is no fixed convention for representing a NOT condition, and several variant representations may apply. This is partially historical because the basic ASCII set of printable characters did not give us a simple way to represent a NOT variable, that is, a variable with a line over it, as shown in Chapter 2, figure 2.4. Thus, you may see several different representations of the NOT condition. For example, /A , ~A , *A and A* are all commonly used ways of representing the condition, NOT A. I will avoid using the *A

and A* notations because it is too easy to confuse this with the AND symbol. However, I will occasionally use ~A and /A interchangeably in the text for the NOT A condition if it makes the equation easier to understand. Most figures will continue to use the bar notation for the NOT condition. Sorry for the confusion.

Laws of Boolean Algebra

1- Laws of Complementation

• First law of complementation:	If A = 0 then /A = 1 and if A = 1 then /A = 0
• Second law of complementation:	A * /A = 0
• Third law of complementation:	A + /A = 1
• Law of double complementation:	/(/A) = //A = A

2- Laws of Commutation

• Commutation law for AND:	A * B = B * A
• Commutation law for OR:	A + B = B + A

3- Associative Laws

• Associative law for AND:	A * (B * C) = C * (A * B)
• Associative law for OR:	A + (B + C) = C + (A + B)

The associative laws enable us to combine three or more variables. The law tells us that we may combine the variables in any order without changing the result of the combinations. As we've seen in Chapter 2, it is this is the law that allows us, for example, to combine two, 2-input OR gates to obtain the logical equivalent of a single 3-input OR gate. It should be noted that when we say the “logical equivalent” we are neglecting any electrical differences. As we have seen, the timing behavior of a logic gate must be taken into account in any real system.

4- Distributive Laws

• First distributive law:	A * (B + C) = (A * B) + (A* C)
• Second distributive law:	A + (B * C) = (A + B) * (A + C)

The distributive laws look a lot like the algebraic operations of factoring and multiplying out.

5- Laws of Tautology

• First law of tautology: A * A = A	If A = 1 then A * A = 1. If A = 0, then A * A = 0. So the expression A * A reduced to A.
• Second law of tautology: A + A = A	If A = 1, then 1 + 1 = 1 , If A = 0, then 0 + 0 = 0. Again, the expression simply reduced to A.

6- Law of Tautology with Constants

- A + 1 = 1
- A * 1 = A
- A * 0 = 0
- A + 0 = A

7- Laws of Absorption

• First law of absorption:	A * (A + B) = A
• Second law of absorption:	A + (A * B) = A

This one is a bit trickier to see. Consider the expression

A * (A + B).

If A = 1, then this becomes 1 * (1 + B). By the law of tautology with constants, 1 + B = 1 so we are left with 1 + 1 = 1. If A = 0, the first expression now becomes 0 * (0 + B). Again, by the law of tautology with constants, this reduces to 0 * B, which has to be 0. Thus, in both cases, the expression reduced to the value of A, the value of B does not figure in the result.

8- DeMorgan's Theorems

- Case 1: /(A * B) = /A + /B
- Case 2: /(A + B) = /A * /B

DeMorgan's theorems are very important because they show the relationship between the AND and OR functions and the concepts of positive and negative logic. Also, DeMorgan's theorems show us that any logic function using AND gates and inverters (NOT gates) may be duplicated using OR gates and inverters. Also notice that the left side of both of the above equations are just the compound logic functions NAND and NOR, respectively.

Before we move on, we should discuss the relationship between DeMorgan's theorems and logic polarity. Up to now, we've adopted the convention that the more positive, or higher voltage signal was a 1, and the lower voltage, or more negative voltage was a 0. This is a good way to introduce the topic of logic because it's easier to think about TRUE/FALSE, 1/0 and HIGH/LOW in a consistent manner. However, while TRUE and FALSE have a logical significance, from an electrical circuit point of view it is rather arbitrary just how we define our logic.

This is not to say that the electrical circuit that is an AND gate would still be an AND gate if we swapped 1 and 0 in the electrical sense. It wouldn't. It would be an OR gate. Likewise, the OR gate would become an AND gate if we swapped 1 and 0 so that the lower voltage became a 1 and the higher voltage became a 0. You can verify this for yourself by reviewing the truth tables for the AND gate and OR gate in Chapter 2. You can see that if you take the truth table for the AND gate and swap all of the 1's with 0's and all of the 0's with 1's, you end up with the truth table for the OR gate (in negative logic). Try it again for the OR gate and you'll see that you now have an AND gate in negative logic.

An important point to keep in mind is that the same electronic circuit will give us the logical behavior of an AND gate if we adopt the convention that logical 1 is the more positive voltage. Thus, a logical 1 might be anything greater than about +3 volts and a logical 0 might be anything less than about 0.5V. We call this ***positive logic***. However, if we reverse our definition of what voltage represents a 1 and what voltage represents a 0, the same circuit element now gives us the logical OR function (***negative logic***).

Thus, in a digital system, we usually make TRUE and FALSE whatever we need it to be in order to implement the most efficient circuit design. Since we can't depend upon a consistent meaning for TRUE and FALSE, we generally use the term ***assert***. When a signal is asserted, it becomes active. It may become active by changing from a logic level LOW to a logic level HIGH, or vice versa. As you'll see shortly when we discuss memory systems, most of the memory control signals are asserted LOW, even though the address of the memory cells and the data stored in them are asserted HIGH. You saw this in Chapter 2 when we discussed the logical behavior of the tri-state gate. Recall that the tri-state gate's output goes into the low-Z state when the output enable (~OE) input is asserted low. This does not mean that the ~OE signal is FALSE, or TRUE in the negative logic sense, it simply means that the signal becomes active in the low state.

In figure 2.20 we considered the most general case of a logical system design. Each of 3 output variables is defined as a function of up to eight input variables, i.e., X = ***f***(a, b, c, d, e, f, g, h), and so on. Note that output variables X,Y and Z may each be a separate function of some or all of the input variables, a through h. The problem that we must now solve is in four parts:

1. How do we use the truth table to describe the intended behavior of our digital system? This is just specifying your design.
2. Once we have the behavior we want (as defined by the truth table), how do we use the laws of Boolean Algebra to transform the truth table into a Boolean Algebraic description of the system?
3. When we can describe the design as a system of equations can we use some of the rules of Boolean Algebra to simplify the equations?
4. Finally, when we can describe the equations of our system in the simplest algebraic form, then how can we convert the equations to a real circuit design?

In a moment, we'll see how to slog through this, and using the rules of Boolean algebra, reduce it to a simpler form. However, if we knew beforehand that output Y only depended upon inputs c, d, and g, then we could immediately simplify the design task by limiting the truth table for Y to one with only three input variables instead of eight. As we'll soon see, the general case can be reduced to the simplified case, so either path gets you to the end point. It is often the case that you won't know beforehand that there is no dependence of an output variable on certain input variables; you only learn this after you go through all of the algebraic simplifications.

Figure 3.3 is an example of some arbitrary truth table design. It doesn't describe a real system, at least not one that I'm aware of. I just made it up to go through the simplification process.

Outputs E and F are the two dependent variables that are functions of input variables A through D. Since there are four input variables, there are 2^4, or 16, possible combinations in the truth table, representing all the possible combinations of the input variables. Also, note how each of the input variables are written in each column. Using a method like this ensures that there are no missing or duplicate terms.

Since this is a made-up example, there is no real relationship between the output variables and the input variables. This means that I arbitrarily placed 1's and 0's in the E and F columns to make the example look interesting. If this exercise was part of a real digital design problem, you, the designer, would independently consider each row of the truth table and then decide what should be the logical state (1 or 0) of each of the dependent outputs as a function of the particular combination of input states represented on that row.

For example, suppose that we're designing a simple controller for a burglar alarm system. Let's say that output E controls a warning buzzer inside the house and output F controls a loud siren that can wake up the neighborhood if it goes off. Inputs A, B, and C are sensors that detect an intruder and input D is the button that controls whether or not the burglar alarm is active or not. If D = 0, the system is deactivated, if D = 1, the system is active and the sirens can go off.

A	B	C	D	E	F
0	0	0	0	0	1
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	0	1
1	1	1	0	0	0
0	0	0	1	0	0
1	0	0	1	0	0
0	1	0	1	0	0
1	1	0	1	0	0
0	0	1	1	0	0
1	0	1	1	0	0
0	1	1	1	0	0
1	1	1	1	1	0

$$E = \overline{A}*\overline{B}*C*\overline{D} + A*\overline{B}*C*\overline{D} + A*B*C*D$$

$$F = \overline{A}*\overline{B}*\overline{C}*\overline{D} + \overline{A}*B*C*\overline{D}$$

Figure 3.3: Truth table for an example digital system design. Outputs E and F are the SUM of Products (minterm) representation of the truth table.

In this case, for all the conditions in the truth table where D = 0, you don't want to allow the outputs to become asserted, no matter what the state of inputs A, B, or C. If D = 1, then you need to consider the effect of the other inputs. Thus, each row of the truth table gives you a new set of conditions for which you need to independently evaluate the behavior of the outputs. Sometimes, you can make some obvious decisions when certain variables (D = 0) have global effects on the system.

However, suppose for a moment that the example of figure 3.3 actually represented something real. Let's consider output variable F. The logical equation,

F = /A * /B * /C * /D + /A * B * C * /D

term 1 term 2

tells us that F will be TRUE under two different sets of input conditions. Either the condition that all the inputs are FALSE (term 1) or A and D are FALSE, while B and C are TRUE (term 2) will cause the output variable F to be TRUE. How did we know this to be the case? We designed it that way! As the engineers responsible for this digital design, these are the two particular sets of input conditions that can cause output F to be TRUE.

We call this form of the logical equation the ***sum of products form, or minterm form.*** There is an alternate form called the ***maxterm form***, which could be described as a ***product of sums.*** The two forms can be converted from on to the other through DeMorgan's Theorems. For our purposes, we'll restrict ourselves to the minterm form.

Remember, this truth table is an example of some digital system design. In a way, it represents a shorthand notation for the design specification for the system. At this point we don't know why the columns for the dependent variables, E and F, have 1s or 0s in some rows and not others. That came out of the design of the system. We're engineers and that's the engineering design

phase. What comes after the design phase is the implementation phase. So, let's tackle the implementation of the design.

Referring to figure 3.3, we see that that the output variable, E, is TRUE for three possible combinations of the input variables, A through D:

1. /A*/B*C*/D
2. A*/B*C*/D
3. A*B*C*D

We can express this relationship as a logical equation:

E= /A*/B*C*/D + A*/B*C*/D + A*B*C*D

The OR'ing of the three AND terms means that any one of the three AND terms will cause E to be TRUE. Thus, for the *combination*, we need to use AND. For the *aggregation*, we use OR. Likewise, we can express the value of F as

F = /A*/B*/C*/D + /A*B*C*/D

At this point it would be relatively easy to translate these equations to logic gates and we would have our digital logic system done. Right? Actually, we're sort of right. We still don't know if the terms have any redundancies in them that we could possibly eliminate and make the circuit easier to build. The redundancies are natural consequences of the way we build the system from the truth table. Each row is considered independently of the other, so it is natural to assume that certain duplications will creep into our equations.

Using the laws of Boolean algebra, we could manipulate the equations and do the simplifications. However, the most common form of redundant term is A*B + A*/B.

It is easy to show that A*B + A*/B = A. How?

1. First Distributive Law: A*B + A*/B = A*(B+/B)
2. Third Law of Complementation: B + /B = 1
3. Finally, A * 1 = A

Thus, if we can group the terms in a way that allows us to "factor out" a set of common AND terms and be left with an OR term appearing with its complement, then that term drops out.

The Karnaugh Map

In a classic paper, Karnaugh[3] (pronounced CAR NO) described a graphical method of simplifying the sum of products equation of the truth table without the need to resort to using Boolean Algebra directly. The Karnaugh Map is a graphical solution to the Boolean Algebraic simplification:

$$A*B + A*/B = A$$

This simplification is one of the most commonly occurring ones because of the redundancies that are inherent in the construction of a truth table.

There are a few simple rules to follow in building the Karnaugh map (K-Map). Figure 3.4 shows the construction of 3, 4 and 5 variable K-Maps.

Refer to the K-Map for 4 input variables. Note the vertical red edges and the horizontal green edges. These edges are considered to be adjacent to each other. In other words, the map can be considered to be a cylinder wrapped either horizontally or vertically.

The method used to identify the columns may look strange to you, but if you look carefully you'll see that as you move across the top of the map, the changes in the variables A and B are such that:

- Only one variable changes at a time,
- All possible combinations are represented,
- The variables in column 1 and column 4 are adjacent to each other.

The order for listing the variables shown in figure 3.4 is not the only way to do it, it is just the way that I am most comfortable using and, through experience, is the way that I know I am drawing the map correctly. It is easy to have the correct combinations of variables listed, but to err in writing down the correct order of those combinations across the top, or down the left side of the K-map. If the row or column headings are not in the correct order then the map will not provide the correct simplification of the logical equations.

Another point that should be noted is that the K-Map yields the simplest form of the equation when the number of variables is 4 or less. For maps of 5 or more variables, the correct procedure is to use a 3-dimensional map comprised of planes of multiple 4 variable maps. However, for the purposes of this discussion, we'll make it a point to note whenever the 5 variable maps require further simplification. In other words, it's easier to do a little Boolean Algebra then it is to draw a 3D K-Map.

We can summarize the steps necessary to simplify a truth table using the K-Map process as follows:

1- The number of cells in the K-map equals the number of possible combinations of input variable states. For example,

a. 3 input variables: A, B, C = 8 cells
b. 4 input variables: A, B, C, D = 16 cells
c. 5 input variables: A, B, C, D, E = 32 cells

Thus, the number of cells = $2^{(\text{NUMBER OF INPUT VARIABLES})}$

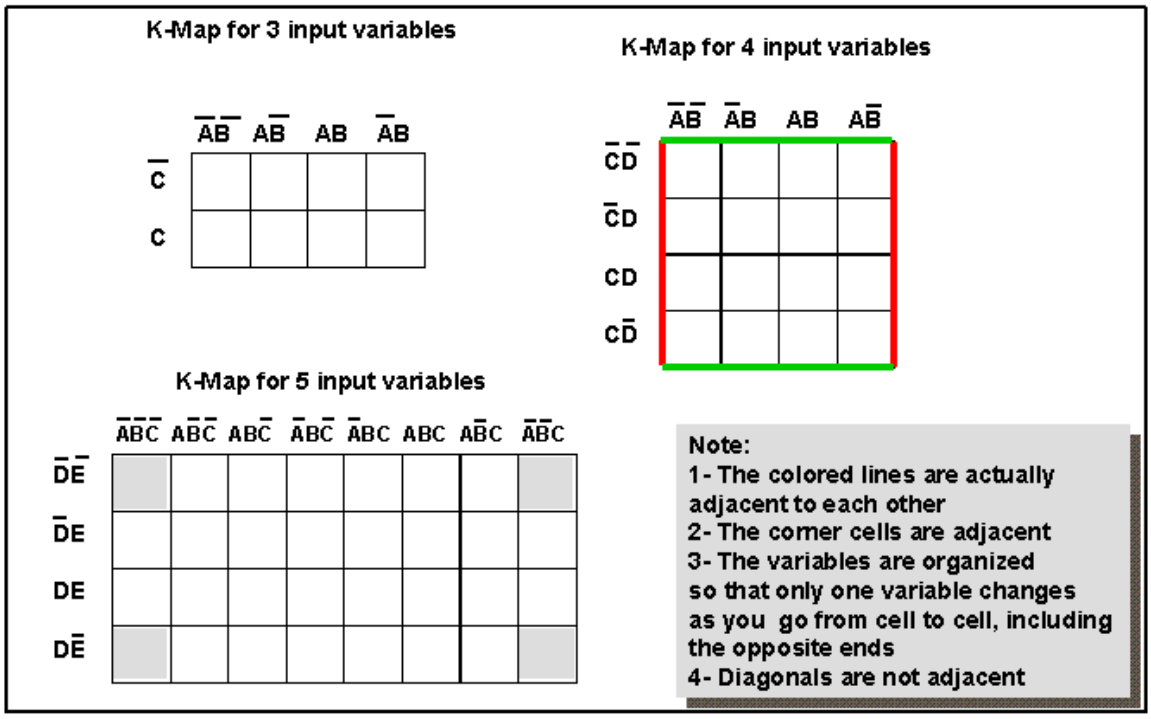


Figure 3.4: Format for building a Karnaugh map in 3, 4, and 5 variables.

2- Construct the K-map so that as you move across the row or down the column, only one variable changes. Referring to figure 3.5, note that the first and last columns are adjacent and the top and bottom rows are also adjacent. It is as if the map is actually wrapped on a cylinder. Note that the first and last cells along the diagonals are not considered to be adjacent.

3- Construct a separate K-Map for every output variable.

4- Place a "1" in every cell of the K-Map that has a "1" in the corresponding row of the truth table.

5- Draw a loop around the largest possible number of **adjacent** cells that contain a 1. You can form loops of 2, 4, 8, 16, 32, etc. adjacent cells.

6- You may form multiple loops and a cell in one loop may be in another loop, but each loop must contain at least one cell that is not contained in any other loop. Inspect the map for any loop whose terms are all enclosed in other loops and remove those loops.

7- Finally, simplify the loop by removing any variable that appears within that loop in both its complemented and un-complemented form. The simplified equation is now the ORing of the loops of the K-Map where each loop represents the simplified minterm form.

Perhaps an example would help. Figure 3.5 demonstrates the process. We have a truth table with three independent input variables—A,B, and C—and one dependent output variable, x. Three input variables imply eight rows for the truth table. In figure 3.5, there are four rows that have a 1 in the "x" column. We can thus write down the unsimplified logic equation for x:

$$x = \sim A*\sim B*\sim C + A*\sim B*\sim C + A*B*\sim C + A*\sim B*C$$

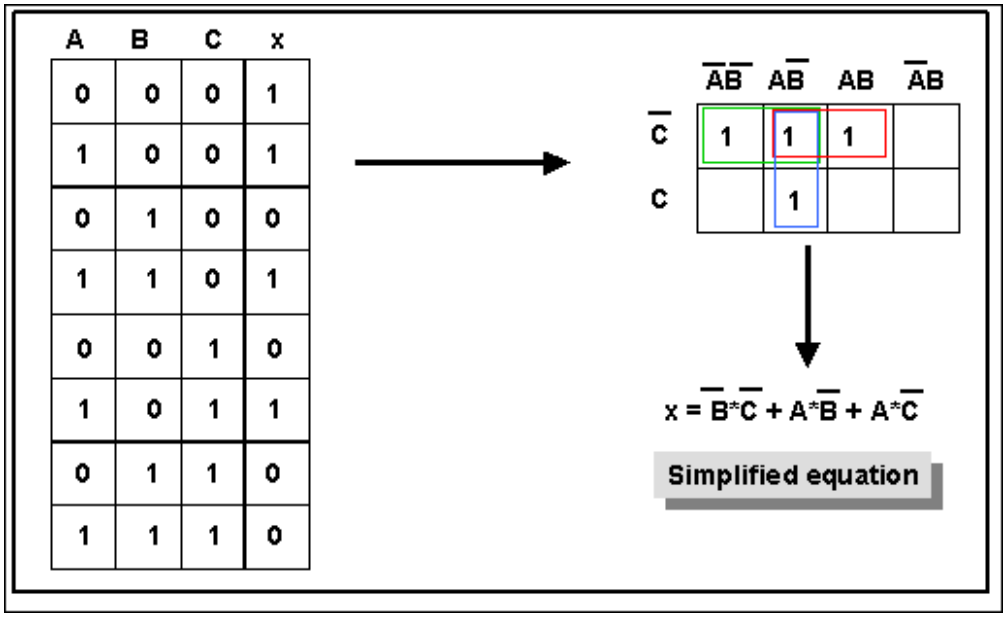


Figure 3.5: Translating a truth table to a K-Map. Each cell of the K-map represents one combination of the independent variables. A '1' is placed in each cell that contains a 1 in the corresponding row of the truth table.

- Now, refer to figure 3.5. The K-Map corresponding to the truth table is shown on the right. Four cells contain a '1' term for the output variable 'x', in agreement with the truth table. We can draw three loops as follows:
- Green loop around the term ~A*~B*~C, and the term A*~B*~C
- Red loop around the term A*~B*~C, and the term A*B*~C
- Blue loop around the term A*~B*~C, and the term A*~B*C

We can thus remove the variable A from the green loop, B from the red loop and C from the blue loop, respectively. The resulting equation: x = ~B*~C + A*~B + A*~C is the simplified version of out original equation.

Note that in this example, the cell corresponding to the state of the input variables A*~B*~C is common to the three loops and it appears in each loop. However, each of the loops also contains one variable not contained in any other loop, so we can draw these three loops.

Before we go any further, it is reasonable to see if we could have achieved the same simplified logical equation by just doing the algebra the way George Boole intended us to. Below are the algebraic steps to simplifying the logical equation:

Step 1	x = ~A*~B*~C + A*~B*~C + A*B*~C + A*~B* C	From the truth table
Step 2	x = ~A*~B*~C + A*B*~C + A*~B*(C + ~C)	First Law of Distribution
Step 3	x = ~A*~B*~C + A*B*~C + A*~B	First Law of Complementation
Step 4	x = ~A*~B*~C + A*(B*~C + ~B)	First Law of Distribution
Step 5	x = ~A*~B*~C + A*[(~B +~C)*(B + ~B)]	Second Law of Distribution
Step 6	x = ~A*~B*~C + A*(~B +~C)	Law of Complementation
Step 7	x = ~A*~B*~C + A*~B + A*~C	First Law of Distribution
Step 8	x = ~B*(~A*~C + A) + A*~C	First Law of Distribution
Step 9	x = ~B*[(~A + A)*(~C + A)] + A*~C	Second Law of Distribution
Step 10	x = ~B*(~C + A) + A*~C	First Law of Distribution
Step 11	x = ~B*~C + ~B*A + A*~C	First Law of Distribution

Let's consider a slightly more involved example. Figure 3.6 shows a 4 input variable problem with two dependent variables for the outputs. Again, this is a made-up example; as far as I know it doesn't represent a real system. If we were actually trying to build a digital system, then the systems requirements would dictate the state of each output variable for a given set of input variables.

If you consider the K-Map for variable 'X' you see that we are able to construct two simplifying loops. The red loop folds around the edges of the map because those edges are adjacent. In a similar way, the blue loop folds around the top and bottom edges of the map. You might wonder why we couldn't also make a loop with the terms A*B*~C*~D and ~A*B*~C*~D. We can't make another loop because both terms are already in other loops. In order to make a third loop we would need to have one or more terms not contained in any other loop.

Using the K-Map does not always result in the most simplification possible, but it comes pretty close. This is particularly true of K-Maps larger than 4 variables. In fact, the five variable K-Map should technically be represented as two four variable K-Maps, one on top of the other. Remember that you may always try to use Boolean algebra and DeMorgan's Theorems to simplify your equations to their final form.

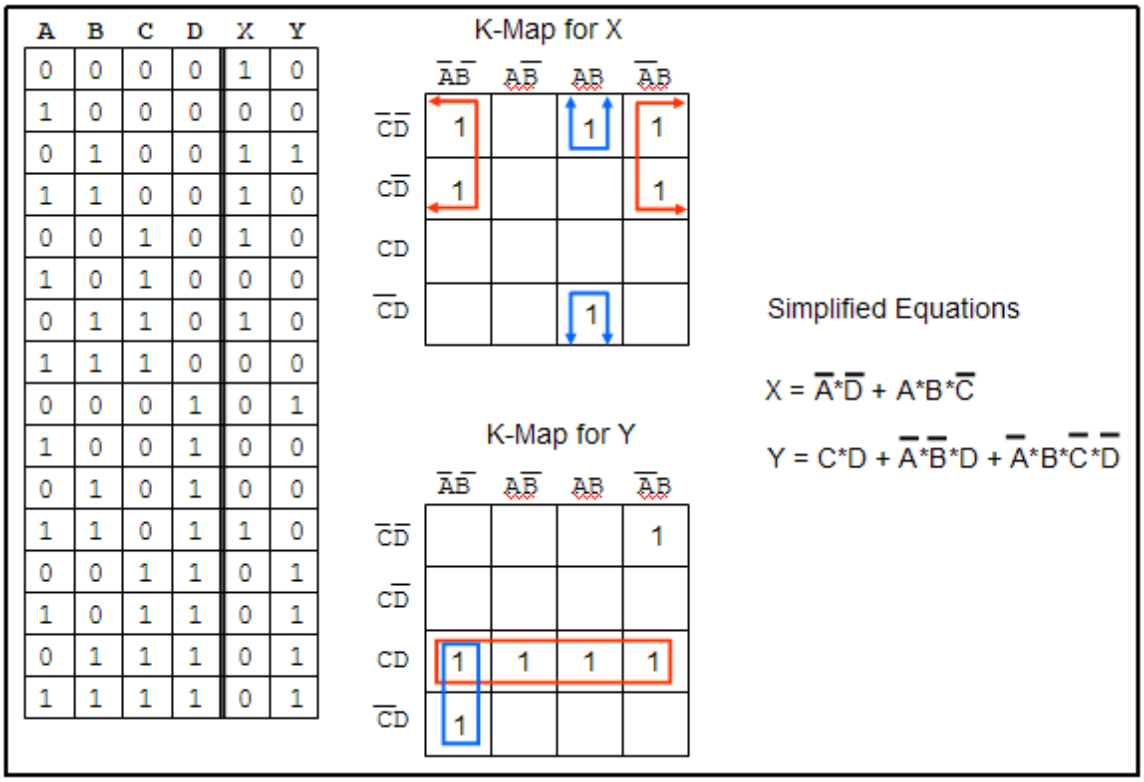


Figure 3.6: Simplifying a 4-variable truth table with two dependent variables

As a final step, let's convert our simplified logical equations to a real hardware gate implementation. We'll take the logical equation that we simplified in figure 3.5 and convert it to its gate equivalent circuit. Figure 3.7 shows the implementation of the design using NOT, AND and OR gates. This is not the way that you "must" design it. It is just a convenient way to show the hardware design. The three input signals are shown along the top left of the figure. For each input variable, we also add a NOT gate because it will be convenient to assume that we'll need to use either the variable or its complement in the circuit design.

Also note that we use a black dot to indicate a point where two separate wires are physically connected together. We need to do this so we can differentiate wires that cross each other in our drawing, **but are not connected together**, from those wires that are connected together. The black dot serves that purpose for us.

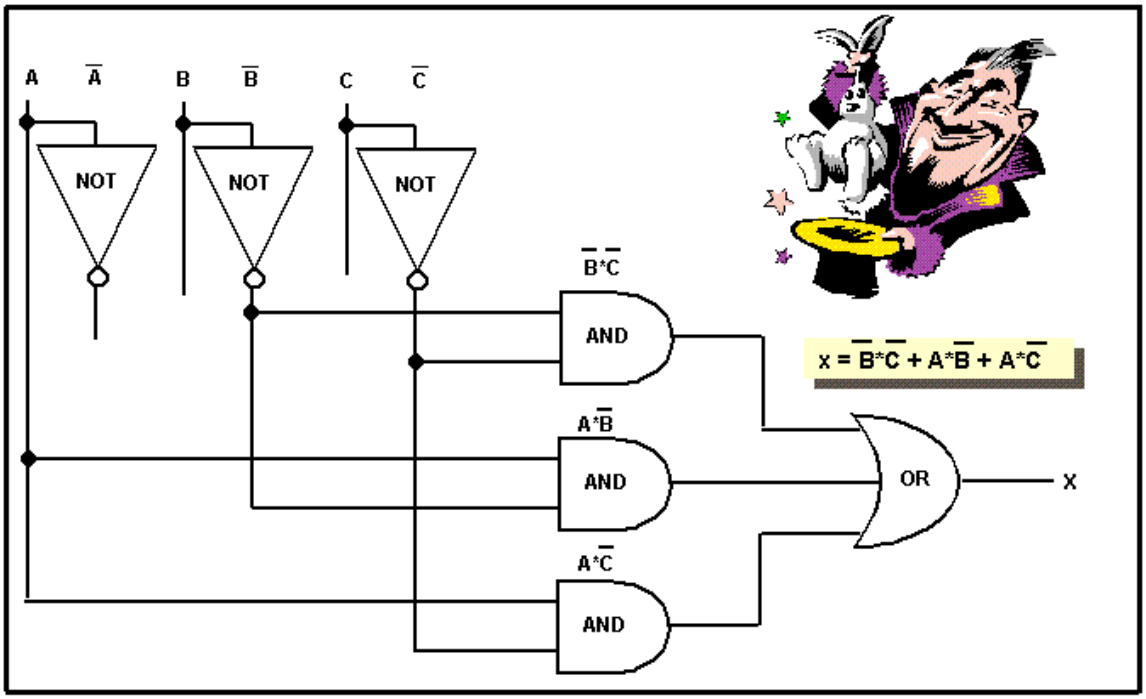


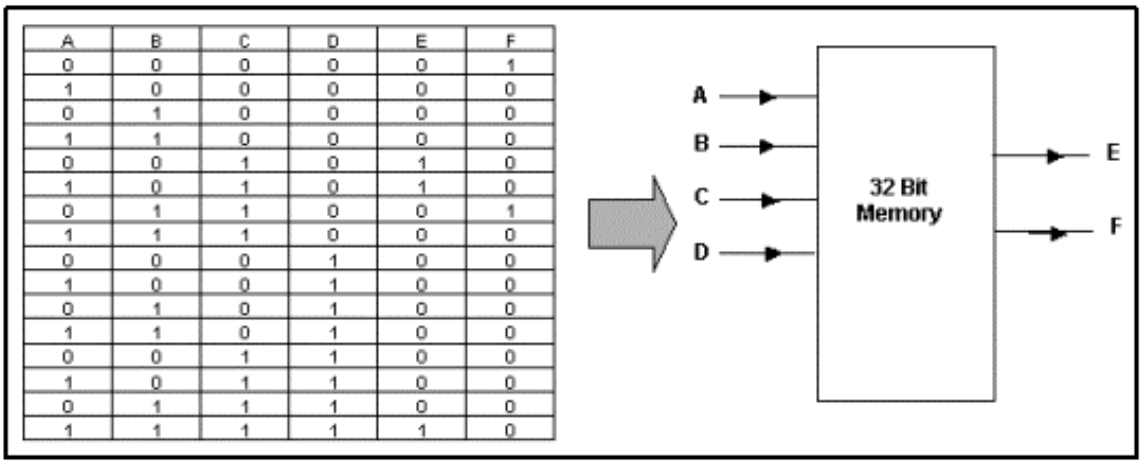
Figure 3.7: Circuit implementation of the logical equation X = /B*/C + A*/B + A*/C

The circuit in figure 3.7 also shows the input variables, their complements, the combinatorial terms in the AND gate and the aggregation of the combinatorial terms using the OR gate. Notice that we needed 3, 2-input AND gates and 1, 3-input OR gate to implement the design. If we had a more complex problem we might choose to use AND and OR gates with more available inputs or use several levels of gates with fewer inputs. Thus, we could create an equivalent 7-input AND gate from 3, 3-input AND gates.

Does the circuit of figure 3.7 actually agree with the original design of our truth table? It might be a good idea to do a quick check, just to build our confidence for the complexities to come. Let's do the first three terms of the truth table and we'll leave it as an exercise for you to do the remaining five terms.

1. Term 1: A = 0, B = 0, C = 0. Here the NOT gates for the variables B and C invert the inputs and feed the values B = 1 and C=1 to the first AND gate. Since both inputs are '1', the output of this AND gate is 1. The output of this AND gate is the input to the OR gate. Since one of the inputs is '1', the output is also '1' and x=1, just as required by the truth table.
2. Term 2: A = 1, B = 0, C = 0. According to the truth table, 'x' should also equal '1' for this situation. Since the first AND gate does not require variable A as an input, variable B and C are unchanged, so we also get x=1 for this situation.
3. Term 3: A = 0, B = 1, C = 0. The first AND gate now gives us a '0' because the complement of B = 1 is ~B = 0. Thus, 0 AND 1 = 0. The second AND gate has A and ~B as its inputs. Since this condition has A = 0 and ~B = 0 as its inputs, it is also '0'. The third AND gate has A = 0 and ~C = 1 as its inputs. Again, it results in an output of '0'. Since all three inputs to the OR gate are '0', x = 0.

Before we leave the topic of logic gates and begin to consider systems that depend upon a synchronization or clock signal, let's examine how else we might build a digital system. The truth table is an interesting format because it looks very close in form to how memory is organized. Figure 3.8 is the truth table example from figure 3.3 but shown as if it was a memory device. We have four independent variables and two dependent variables.



A	B	C	D	E	F
0	0	0	0	0	1
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	0	1
1	1	1	0	0	0
0	0	0	1	0	0
1	0	0	1	0	0
0	1	0	1	0	0
1	1	0	1	0	0
0	0	1	1	0	0
1	0	1	1	0	0
0	1	1	1	0	0
1	1	1	1	1	0

Figure 3.8: Converting a truth table to a memory image. The independent variables, A through D, provides the address to the memory. The 2 dependent variables, E and F, are represented by the data in the memory cell corresponding to the address of the memory (row of the truth table).

Let's consider the implications of what we just did. Here we can imagine that we use a real memory device and fill it so that when we give it the appropriate address bit values (in this case, the appropriate combination of input variables A through D) the data out from the memory (the dependent variables E and F) is the circuit behavior that we have assigned to the truth table. Thus, we can implement logical systems either by creating an electrical circuit design using logical gates, or we can create a logically equivalent design by using a memory device to directly implement the contents of the truth table. In the case of memory as logic, we don't do any logical simplification as we would with a gate circuit design. Also, we need to consider other factors, such as dynamic performance (speed), power consumption, parts costs, or the availability of space on the printed circuit board, before we can make a decision about which method to use for a particular application.

To make this point even stronger, let's redraw figure 3.8 as figure 3.9. The only difference is that we'll represent it as a real memory device. The independent variables (A through D in figure 3.3) are represented as address bits, A0-A3, to the memory device. The dependent variables (E and F in figure 3.3) are now the data bits stored in the various pairs of memory locations.

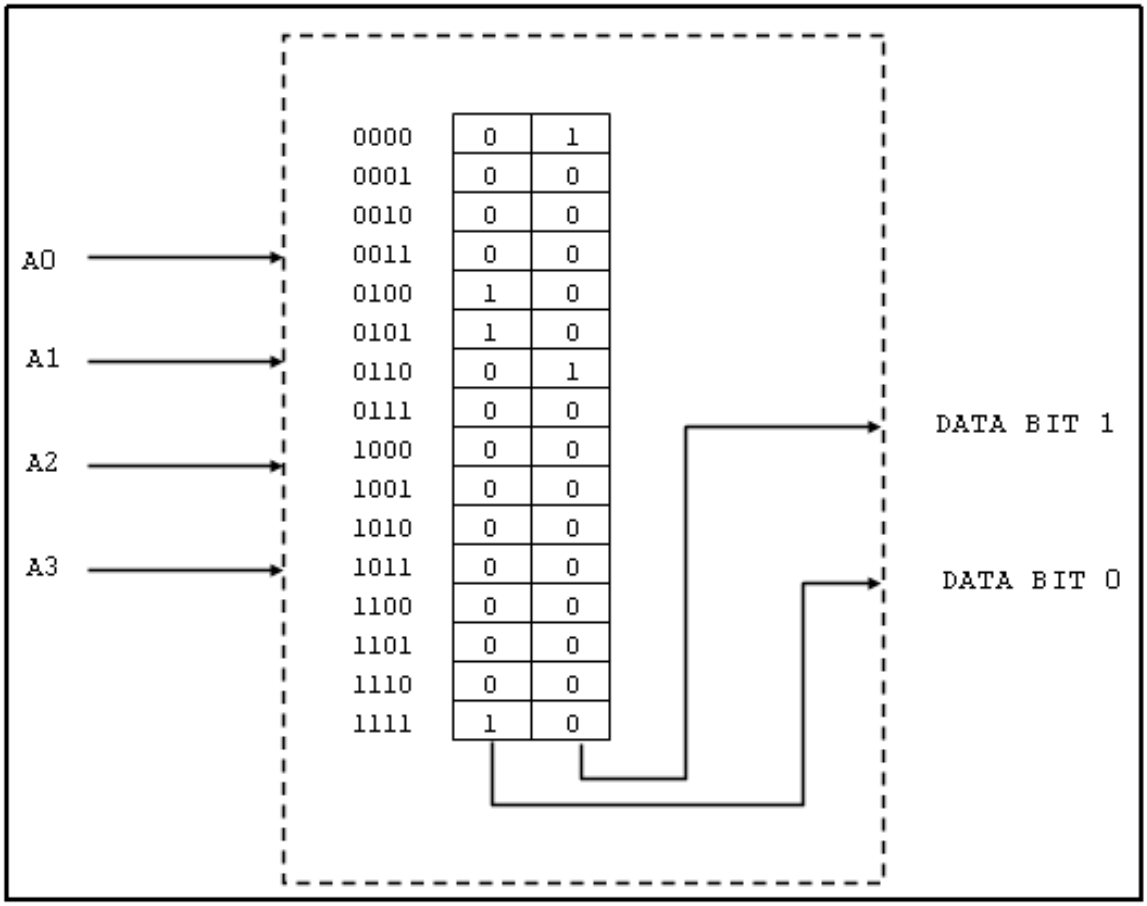


Figure 3.9: Converting a truth table to a memory image. The independent variables, A through D provides the address to the memory. The 2 dependent variables, E and F, are represented by the data in the memory cell corresponding to the address of the memory (row of the truth table).

Thus, our memory needs to be able to hold a total of 32 bits, 16 bits for each of the two dependent variables. Each bit represents the state of that variable for the combination of states of the input variables forming the address of the memory cell.

Figures 3.8 and 3.9 represent an alternative way of creating a hardware implementation of the logical design. In the prior example we used the laws of Boolean algebra and the K-Maps to build a simplified set of logical equations that we could then implement as a combination of logic gates (also called ***combinatorial logic***). Figure 3.8 shows that we could simply take the truth table, as is, fill up a memory chip with all the possibilities and be done with it. It turns out that both methods are equally valid and are used where it's most appropriate. The use of memory as logic, called ***microcode***, forms the basis for much of the control circuitry within a modern digital computer. We'll revisit this concept in a later chapter when we discuss ***state machines.***

Clocks and pulses

In Chapter 2 we first saw digital signals as waveforms. That is, we represented the logical signals as values that change over time. The waveform is a strip chart recorder view of the

digital signal. The simplest digital signal that we might want to consider is the simple positive pulse of figure 3.10, which schematically shows a single positive pulse with a pulse height of about 3 volts.

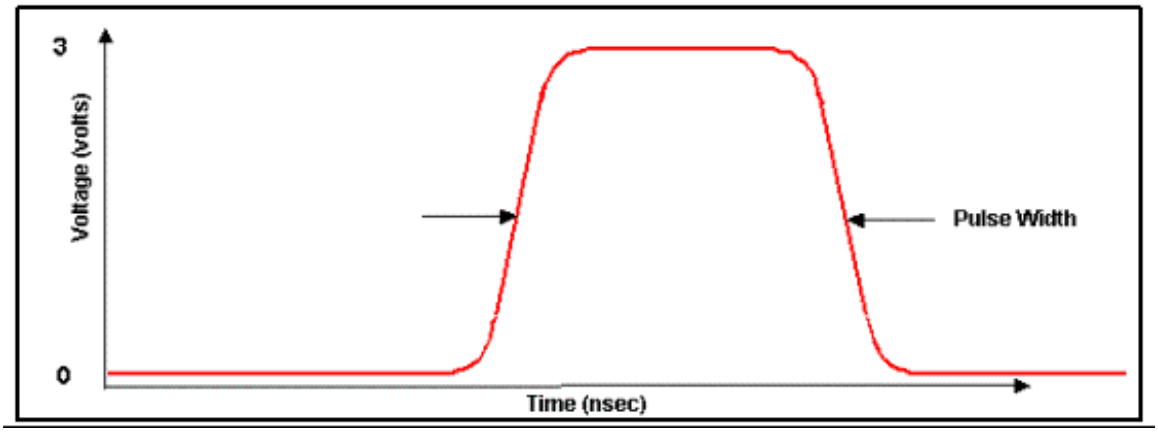


Figure 3.10: Typical waveform for a positive pulse, approximately 3 volts high.

We might stop for a moment and ask, "What's a pulse?" You probably know the medical term for your "pulse". You feel, or should feel a pressure surge in your veins every time your heart beats. What you are feeling is the blood creating a pressure pulse as it flows because the pumping is discontinuous, and happens in discrete pulses of blood. You also can see this when you get an EKG done on your heart and you see the characteristic spikes of the electrical signals around your heart (figure 3.11).

What is characteristic of any pulse is that the system goes from a relaxed state, such as low pressure, to an excited state (blood surge) and then back again. Electrically, we can describe a pulse as a signal going from low to high and then back to low, or vice versa. In other words, we could have a pulse such that the system goes from asserted, to non-asserted, and then back to an asserted state. We call a pulse that goes from low to high and back to low a ***positive pulse;*** a pulse that goes from high to low and back to high a ***negative pulse.***

The pulse in figure 3.10 is positive because it starts from the LOW state, goes HIGH, and then returns to the LOW state when it is completed. In this example, the width of the pulse, or ***pulse width***, measures the amount of time that the pulse exists. Since we don't have a timescale (x-axis) for this pulse, let's assume that the pulse is approximately 50 nanoseconds (often abbreviated as 50 ns), or 50 billionths of a second, wide. The pulse width is measured at a point mid way between the base level of the pulse and its nominal height. Thus, for a pulse that is 3 volts high, the pulse width would be measured at the 1.5 volts portion of the waveform.

Figure 3.11: Part of an EKG showing the characteristics pulses of the electrical signals around the heart.

Figure 3.12 shows a, more or less, "real pulse". The real pulse is what you might see if you had a really fast stopwatch, a really fast voltmeter and you could scribble like crazy (remember, this is a thought experiment). Real people use analytical equipment called ***oscilloscopes*** to see this waveform. Notice how the red line, which represents the change in voltage of the pulse over time, has some slope to it as it rises and falls. This is because the pulse can't change state infinitely rapidly. It takes some time for the voltage to rise to a 1, or fall back to 0. We call these times the ***rise time*** and ***fall time***, respectively. Technically, for reasons that we don't need to consider, the rise and fall times are measured at the 10% and 90% points of the voltage.

Figure 3.12 shows how the pulse might be seen on the screen of an oscilloscope. The horizontal axis displays the elapsed time in some convenient units (nanoseconds, in this example) and the vertical axis displays how the voltage signal changes with time.

Figure 3.13 shows an actual rise time measurement from an R&D laboratory oscilloscope. The oscilloscope can perform a number of automatic measurements, such as rise time, fall time, pulse width, pulse height, frequency and period. The oscilloscope circuitry automatically analyzes the shape of the pulse waveform and locates the 10% and 90% points on the pulse and then calculates the time difference between those two points.

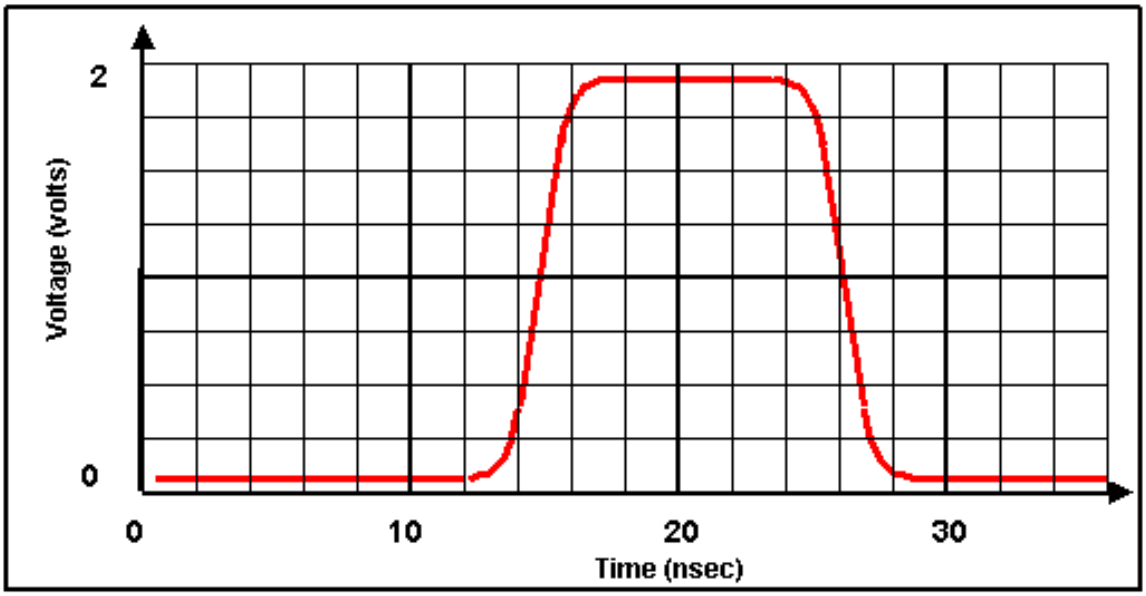


Figure 3.12: A positive pulse as it might appear on the display of an oscilloscope.

Before we move on to consider clocks we should discuss one last point about gates. We've previously defined the ***propagation delay*** of a gate as time delay from a change at the input of the gate to the corresponding change at the output of the gate. Let's see what an actual propagation delay measurement might look like on our lab oscilloscope. Figure 3.14 shows us what a propagation delay measurement might look like for a NOT gate. We connect our oscilloscope probes to the input and output of the NOT gate as shown in the figure. To make this measurement, we start our oscilloscope trace a few nanoseconds before the input signal to the gate (the falling edge) occurs. This oscilloscope can simultaneously display the logic state of both the input and output waveforms so we see the input signal going low and then slightly later in time, the output signal goes high. In fact, if you look closely at the oscilloscope display, you'll see that the time interval between the falling edge of the input and the rising edge of the output is 12.60 nanoseconds.

To this point we've considered pulses as single events. However, just like the EKG shows, your heart pumps continuously, so the EKG displays a string of pulses.

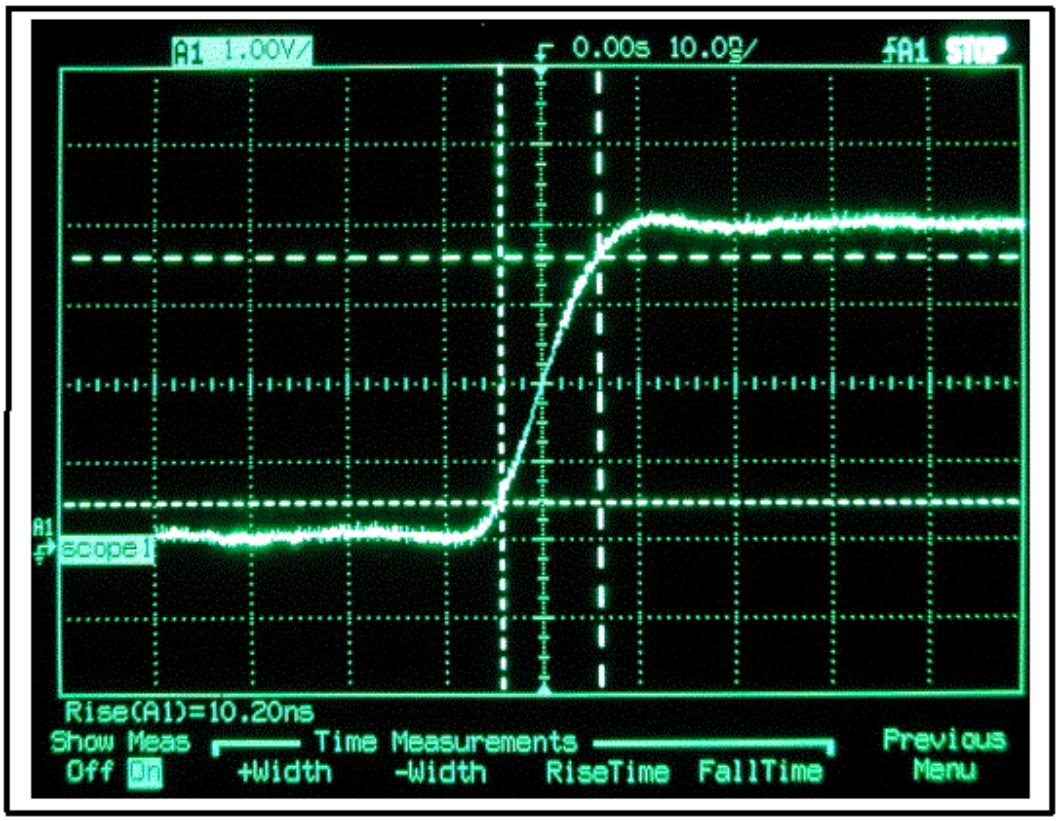


Figure 3.13: Oscilloscope rise time measurement. The vertical axis is 1 volt per division and the horizontal axis is 10 nanoseconds per division.

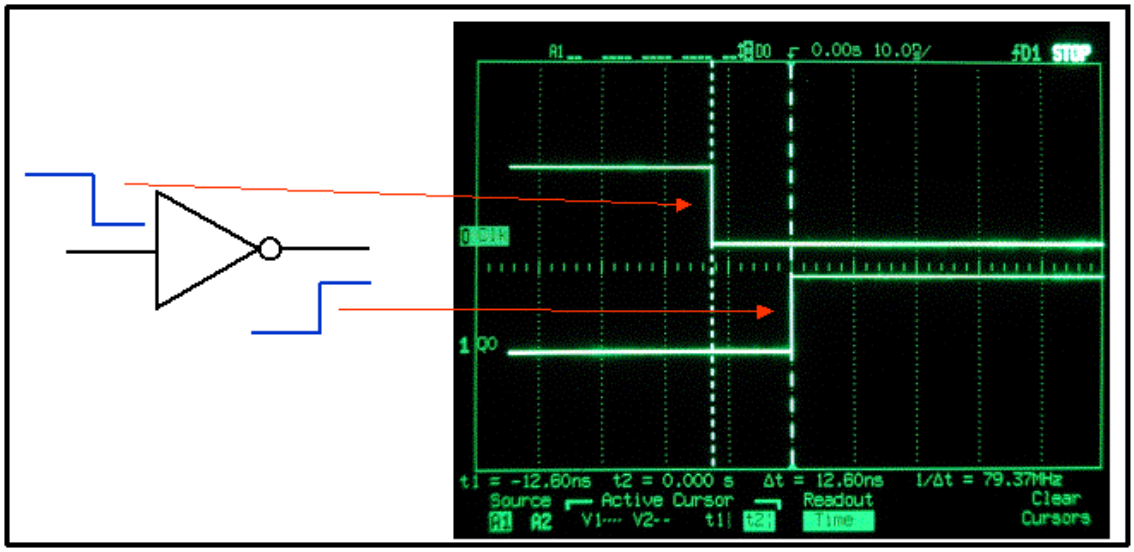


Figure 3.14: Oscilloscope trace of a propagation delay measurement. The upper trace represents a measurement probe on the input side of the NOT gate and the lower trace shows the output of the gate. The display shows that the output goes high 12.60 nanoseconds after the input goes low.

In figure 3.15, we once again go back to the idealized view of the waveform. Note the absence of any slope to the rising and falling edges of the waveform. We consider these pulses to have infinitely fast transitions from LOW to HIGH and HIGH to LOW. It doesn't hurt anything to make this assumption and it makes our diagrams easier to analyze and understand.

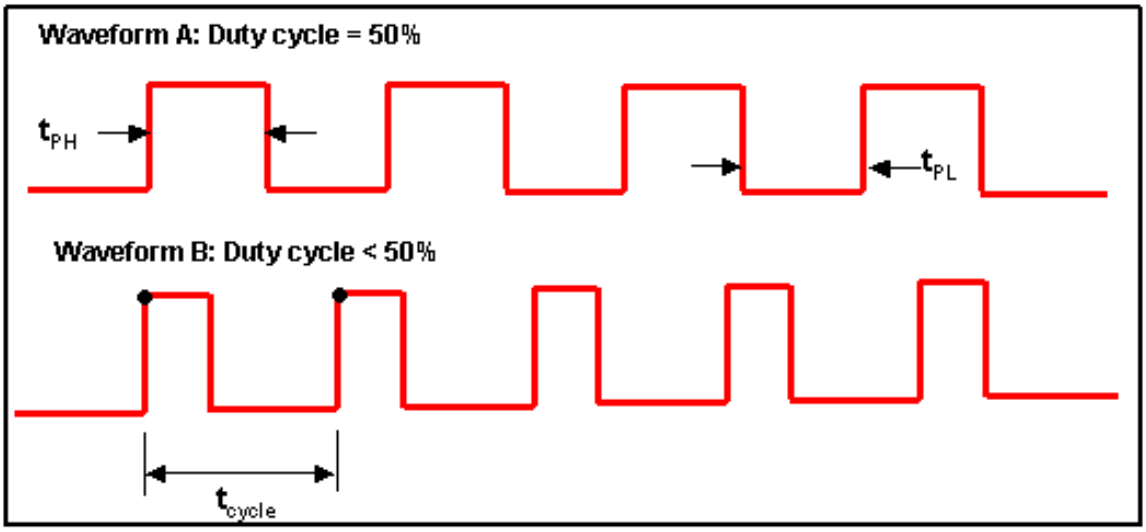


Figure 3.15: Examples of clock waveforms. When the pulse width of the high portion of the waveform equals the pulse width of the low portion of the waveform (waveform A), we have a 50% duty cycle. Waveform B has a duty cycle below 50%. The time difference between the two black dots on waveform B represent the ***period*** of the waveform.

We refer to this continuous stream of pulses, such as the waveform shown in figure 3.15, as a ***clock***. The clock is not the same as the clock in your computer that gives you the time of day. The clock that we are concerned with in this context is a continuous stream of tightly regulated pulses, usually controlled by crystal oscillators. The crystals have the property that they can create a tuned circuit that resonates, or oscillates, at a very predictable and stable frequency. For example, even the cheapest digital wristwatch can keep time to better than 1 minute per month because it uses a crystal that oscillates at approximately *32K Hz.* We use the term ***Hertz***, to represent cycles per second, or the oscillating frequency of a clock stream. The symbol is *Hz*. The unit, *Hertz*, was named in honor of the German physicist, Heinrich Rudolf Hertz (1857-1894).

It's easy to get lost in the minutia of clock and waveforms, but let's stop for a second and consider the question, *What really is a clock, anyhow?"*

A clock is a constant stream of pulses that is usually quite accurately spaced in time. Even an inexpensive digital watch only gains or loses a few seconds a month. Considering that its pulse is about 32 thousand beats per second, that's pretty impressive. We won't spend any time discussing how such accurate time signals are generated because we don't want to offend the Electrical Engineers, but we will try to understand what the clock is doing in our system.

Imagine that you are looking at the pendulum of an impressive old grandfather's clock. If you could start a stopwatch at the point where the pendulum just stops at one end of its travel and stop the watch when it travels to the side and returns, you would measure the period of the

pendulum. The grandfather's clock uses the fact that the period of the pendulum's swing varies very little from cycle to cycle to advance the clock's time through a mechanism of gears. The point was to show you that in real systems the pendulum provides us with an accurate synchronization mechanism that we can use to drive the clock's internal timekeeping mechanism. Also, the action of the pendulum provides the source of synchronization for the entire clock. Each time it swings back and forth it advances the gears that tell time.

Suppose it takes the pendulum 5 seconds to go from one extreme of its travel to the other and back again. This is 1 cycle every 5 seconds, so the period is 5 seconds. The number of cycles in one second is just the reciprocal of the period, or 0.2 Hz.

Within our digital systems, we use a clock signal to provide the same synchronization mechanism. In most computer systems a single clock signal is distributed throughout the circuitry to provide one accurate timing source that all internal operations may be synchronized with. When you go into your local computer store to buy the latest PC, the salesperson will try to convince you to buy a PC with a 3.2 Gigahertz clock in order for you to get the maximum gaming experience. What are you really being sold? Just that this super PC has a faster clock. This means more things happen in one second, so it runs faster.

You're probably already familiar with the term "Hertz" and the abbreviation "Hz" if you've ever purchased a PC. The salesperson told you that this PC or that PC was a 3.0 "gigahertz" machine, and it was clearly better than your old 500 "megahertz" boat anchor. What you now understand is that the clock frequency of your old computer was 500 million cycles per second, or 500 MHz. The computer that is only $1000 away from being in the trunk of your car has a clock frequency of 3 billion cycles per second, or 3 GHz. Since one billion is 1000 million, this is the same as saying that the new computer has a clock frequency of 3000 MHz, or roughly 6 times the clock speed of the old one.

What are we trying to show in figure 3.16? Let's try to imagine that we're sitting on the clock signal input on a typical computer or microprocessor. You've got a really fast Radio Shack® voltmeter and you're measuring the voltage (logic levels) at the clock input. The clock voltage is low for a while (logic level 0), then it goes high for a while (logic level 1) and then low again, over and over. Each low-time interval is exactly the same as the previous one. Each high-time interval is also the same as every other one. The transitions from low to high and high to low are very fast and also equal. Thus the time period for one complete cycle (low to high and back to low) is extremely repeatable.

Figure 3.16 is an actual oscilloscope display of a 2.5 MHz clock signal. Notice that the clock waveform of the figure is not as "clean" as our idealized waveforms, but it isn't too far off. The rising and falling edges are clearly visible in this view. We can also learn a valuable lesson from looking at the figure. Even though the clock waveform isn't as pretty as the idealized waveform, it functions correctly in our circuit. The reason is simple. In our digital world of 1's and 0's, nothing else counts. As long as the signal is less than the zero threshold or greater than the threshold for a 1, the voltage will be properly interpreted.

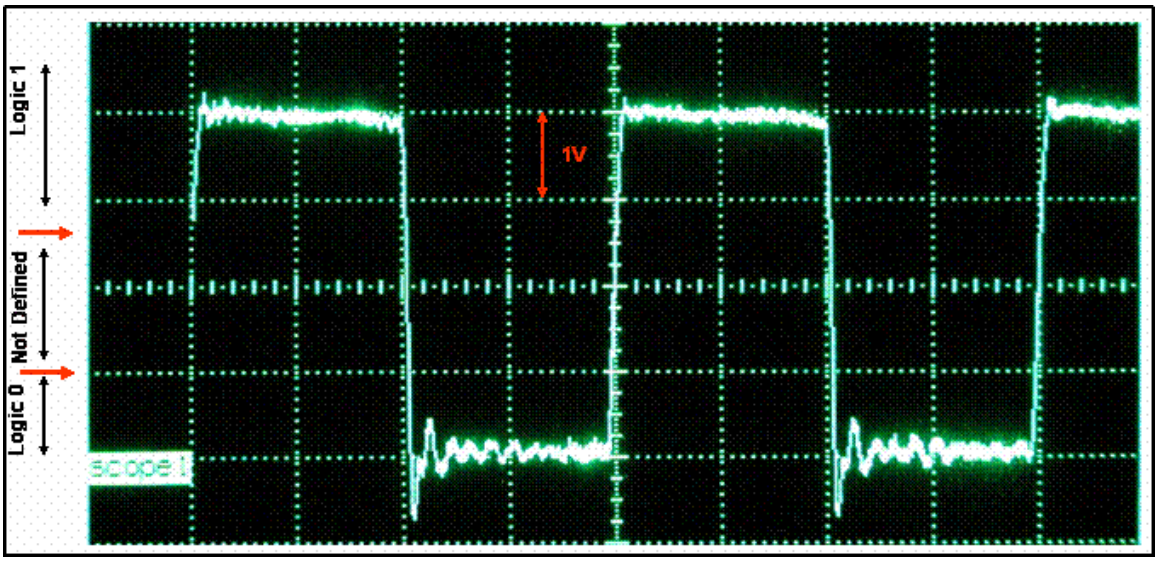


Figure 3.16: Oscilloscope image of a 2.5 MHz clock. Each vertical unit represents 1 volt and each horizontal unit represents 100 nanoseconds. Notice that as long as the signal is above the logic 1 threshold, or below the logic 0 threshold, it will be interpreted correctly.

Let's define some terms:

- **Frequency:** The number of clock pulses per unit time, usually seconds. The frequency is measured in Hertz, or Hz. One Hz is the same as one clock cycle per second.
- **Period:** The inverse of the frequency. It is the amount of time between two equal points on adjacent waveforms. The 2 black dots on waveform B of figure 3.15 represent the period of the waveform, or the time for one cycle of the wave, $t_{\mathbf{cycle}}$. The frequency of the clock equals the inverse of the period. A clock waveform with a period of one second has a frequency of 1 Hz.
- **Duty cycle:** Ratio of the amount of time the clock is HIGH to the period. A 50% duty cycle means that the clock is high for exactly ½ of the period, or the amount of time the clock is HIGH equals the amount of time the clock is LOW. We can also use the equation: Duty cycle = $(t_{\mathbf{PH}} / (t_{\mathbf{PH}} + t_{\mathbf{PL}}))$ x 100%

Figure 3.17, shows the relationship between the common units of time (in computer lingo) and units of frequency.

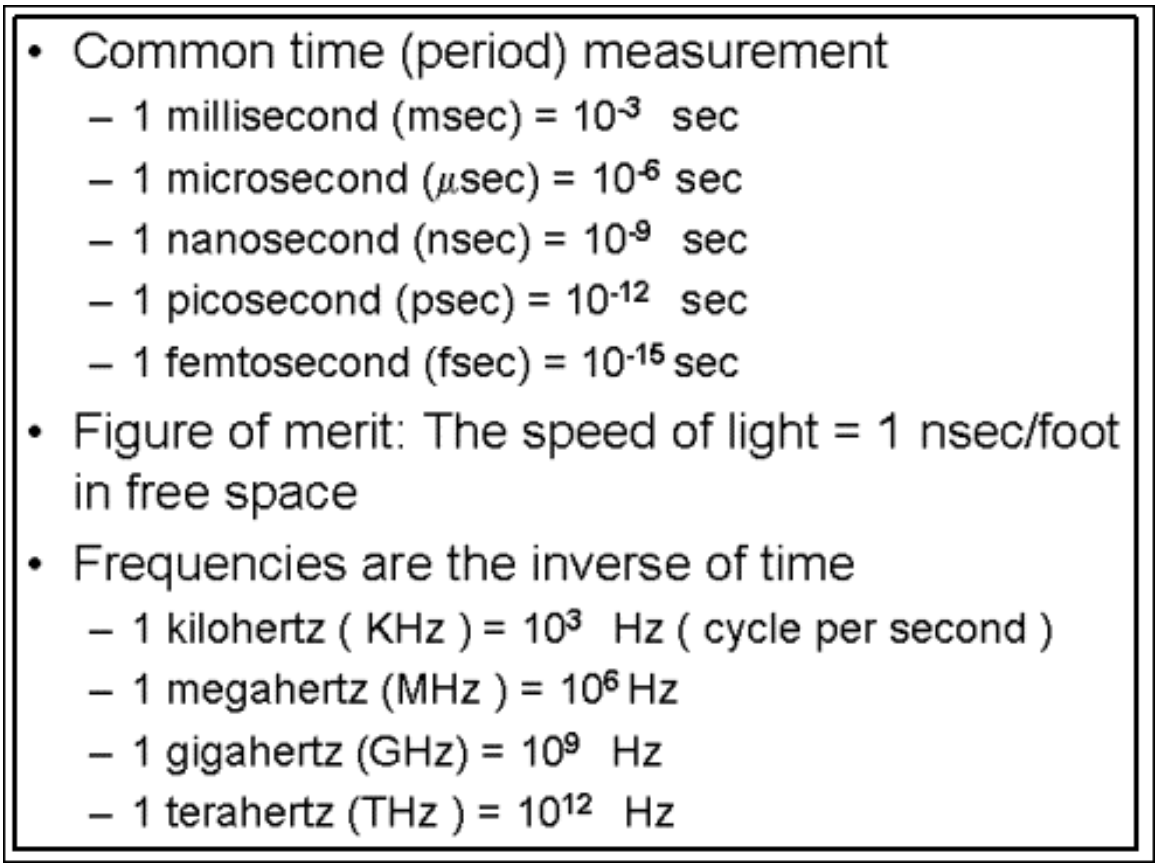


Figure 3.17: Common units of measurement of time and frequency. The speed of light in air is amazingly close to 1 nanosecond per foot. The speed of light through a conductor path on an integrated circuit is about half that, or about 6 inches per nanosecond.

Figure 3.17 also shows us that a clock frequency of 1 MHz has a clock period of 1 microsecond (μs) and a clock frequency of 1 GHz has a period of 1 nanosecond (1 ns). Thus, your 1 GHz Athlon computer has a clock that oscillates so quickly that light can only travel about 1 foot in the time it takes for one cycle of the clock to occur.

Let's review some useful relationships:

- A clock period of 1 ns has a frequency of 1 GHz.
- A clock frequency of 1 MHz has a period of 1 μsec (microsecond).
- A clock period of 1 msec (millisecond) has a frequency of 1 KHz.
- A clock period of 1 second has a frequency of 1 Hz.

Summary of Chapter 3

- Boolean algebra provides the rules for manipulating and simplifying logical equations
- Truth tables provide a convenient method to describe the behavior of an arbitrary digital system in terms of the state of input variables and the resulting state of the output variables.
- The Karnaugh Maps are a graphical method of simplifying the minterm form of truth tables
- Digital systems are driven by clocks which are a continuous stream of pulses and these pulses may be describes in terms of their width, height, rise time and fall time.
- Frequency and period have an inverse relationship to one another.
- We use the engineering number system to describe the commonly occurring frequencies and times that we will be dealing with in digital systems.

Bibliography and references

1- Smith, Robin, "Aristotle's Logic", The Stanford Encyclopedia of Philosophy (Fall 2003 Edition), Edward N. Zalta (ed.), URL = http://plato.stanford.edu/archives/fall2003/entries/aristotle-logic/
2- J J O'Connor and E F Robertson, http://www-gap.dcs.st-and.ac.uk/
3- M. Karnaugh, *The Map Method for Synthesis of Combinational Logic Circuits,* taken from, *Computer Design Development Principle Papers,* Edited by Earl E. Swartzlander, Jr., ISBN #0-8104-5988-4, Hayden Book Company, Rochelle Park, NJ, 1976, Pg. 25
4- Gerald Williams, *Digital Technology*, Second Edition, Science Research Associates, INC. ISBN 0-574-21555-7, 1982

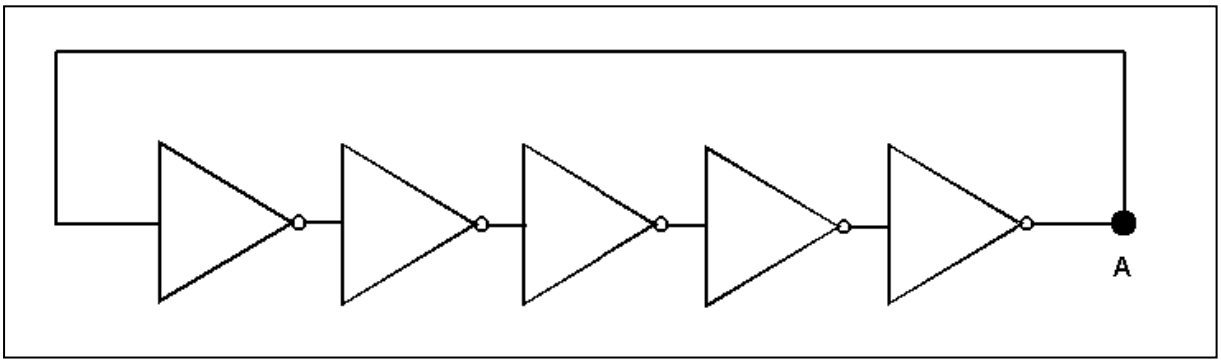
Exercises for chapter 3

1- Design a 1-bit full adder circuit. The full adder adds two input bits and a carry-in bit together and produces the sum and a carry-out bit. See the figure below. Create the truth table, Karnaugh maps, Simplified Boolean equations and a gate level diagram.

2- Prove the two cases of DeMorgan's Theorems using truth tables.

3- The circuit shown below is called a ***Ring Oscillator.*** It consists of 5 NOT gates connected as shown. Imagine that you are measuring the voltage level at point A. The ***propagation delay*** through each gate is exactly 10 ns. The propagation delay is defined as the time it takes the output to change when the input to the gate changes.. Assume that at time t=0 ns the voltage at point A goes from 0 to 1.

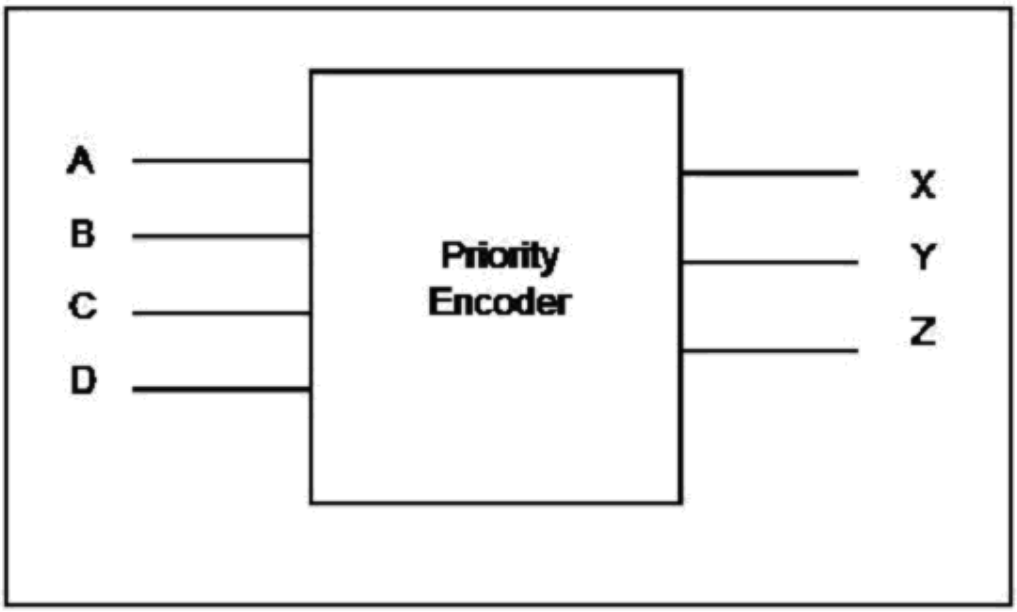
a- Sketch the waveform that you would expect to see at point A.
b- What is the period of oscillation for this circuit?
c- What is the frequency of oscillation for this circuit?



4- Design a Truth Table that has 4 address inputs, A,B,C and D and 1 output, x. Assume that A and B are the control inputs, C and D are arbitrary input variables. Fill in the Truth Table for x so that the design will implement different logic functions depending upon the state of the control inputs, A&B. The functions are as shown in the following table:

A	B	output logic function of C&D (x)
0	0	NAND
0	1	XOR
1	0	NOR
1	1	AND

5- A ***Priority Encoder*** is a circuit whose output is the binary code of the most significant input that is turned on. Suppose that you have the circuit diagram shown below:



Here, A is the least significant input bit and D is the most significant input bit. X is the least significant output bit (2^0) and Z is the most significant output bit (2^2).
If all inputs are 0, all outputs are 0. The priority is determined by the most significant input bit that is 1. Create the truth table for this circuit and then use the Karnaugh Map to simplify the truth table and draw the simplified circuit.

6- Assume a logic circuit that has as its inputs, two 4-bit binary numbers (A0 through A3 and B0 through B3) and as its output, a single binary output, Z. The output Z is TRUE (HIGH) if the two numbers are equal. Design the circuit that implements this equality tester.

7- You are the Chief Designer of the ***Road to Nirvana*** Hot Tub and Spa Company. Your assignment is to design a new digital spa controller to replace the old one that you adapted from a 1972 washing machine. You have the following input specification:

	Variable = 0	Variable = 1
Temperature indicator: A	Water temperature is below the desired hot tub temperature	Water temperature is equal to or above the desired hot tub temperature
Daily filter timer switch: B	Circulation pump is off	Circulation pump is on.
Air blower switch: D	Air blower is off	Air blower is on for soothing bubbles
Key switch: E	System is turned off	System is on
Manual pump switch: F	Circulation pump is off	Circulation pump is on.

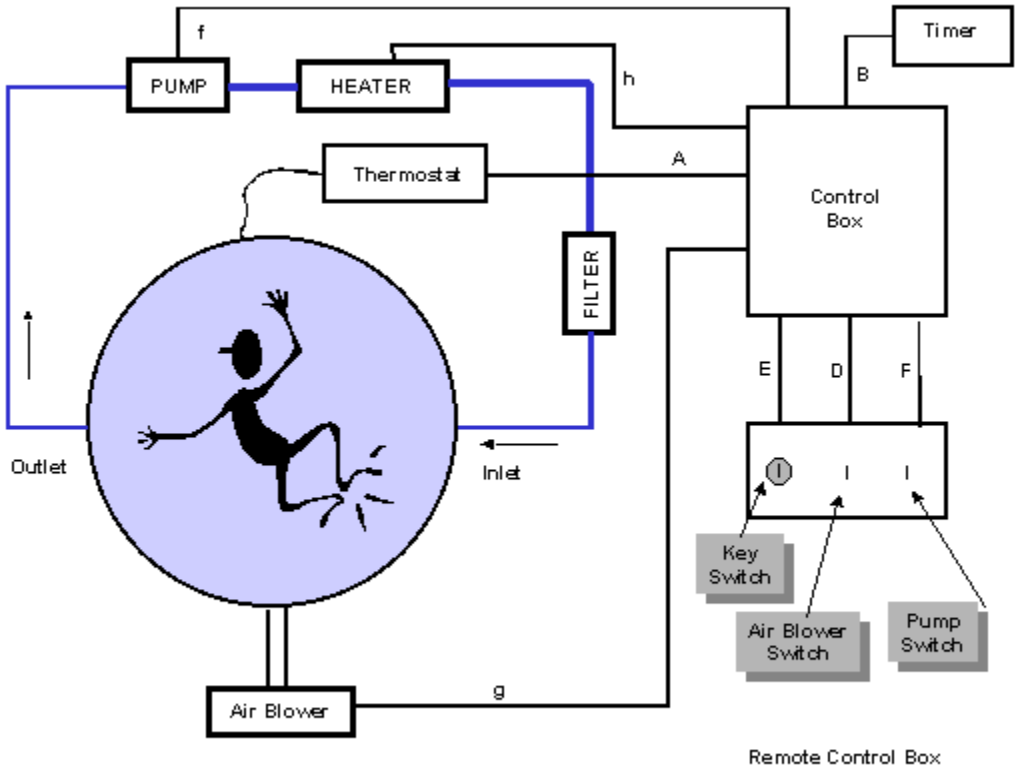
Your logic controls the following outputs:

f = Pump Motor: 1 = On

g = Air Blower: 1 = On

h = Heater: 1 = On

See the figure, below:



How to proceed:

1- Create the truth table for the hot tub. Keep in mind that there may be several alternative ways to define the operation of the hot tub.
2- Use Karnaugh Maps, DeMorgan's Theorems and the algebraic relationships to simplify your equations.
3- After you've simplified the equations as much as possible, draw the logic in terms of the gate structures.

8- Below is the truth table for a circuit called a ***3 input by 8 output decoder***. Basically, it is used to assert LOW only one of its outputs at a time, based upon the binary value (0 to 7) of the input variables. Using the Boolean Algebraic methods and Karnaugh Maps techniques you've learned, draw the simplest gate level diagram that you can to represent the circuit. Hint: notice that the "active outputs" are low, not high. You can use this to your advantage to greatly simplify the logic of this problem. Note that there is a reason for calling the outputs ~CS0 through ~CS7. We'll see the reason when we study memory system organization.

A0	A1	A2	~CS0	~CS1	~CS2	~CS3	~CS4	~CS5	~CS6	~CS7
0	0	0	0	1	1	1	1	1	1	1
1	0	0	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

9- Assume that you have four variables, A, B, C, and D defined as follows:

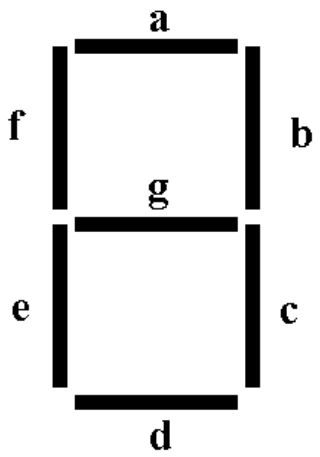
```
bool     A,B,C,D ;
```

Consider the C control statement,

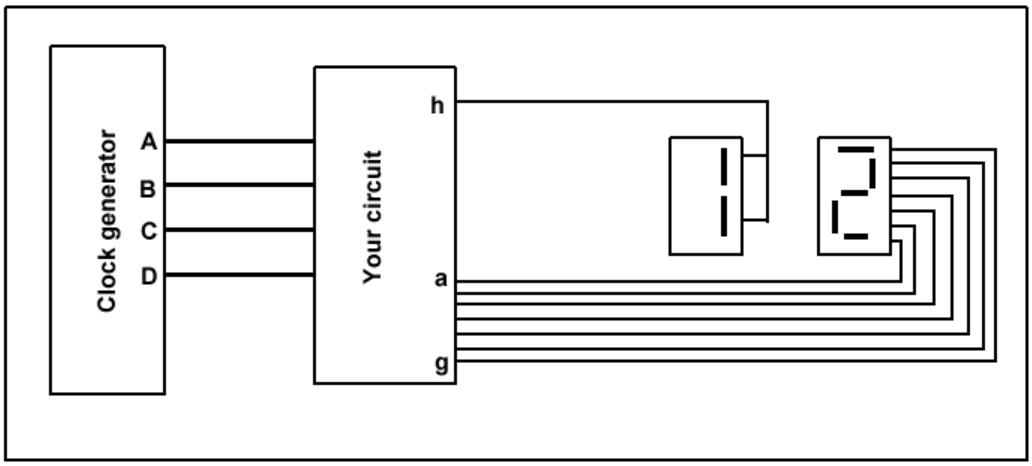
```
If (D)
    C = B;
else
    C = A;
```

Draw the gate equivalent circuit of this control statement

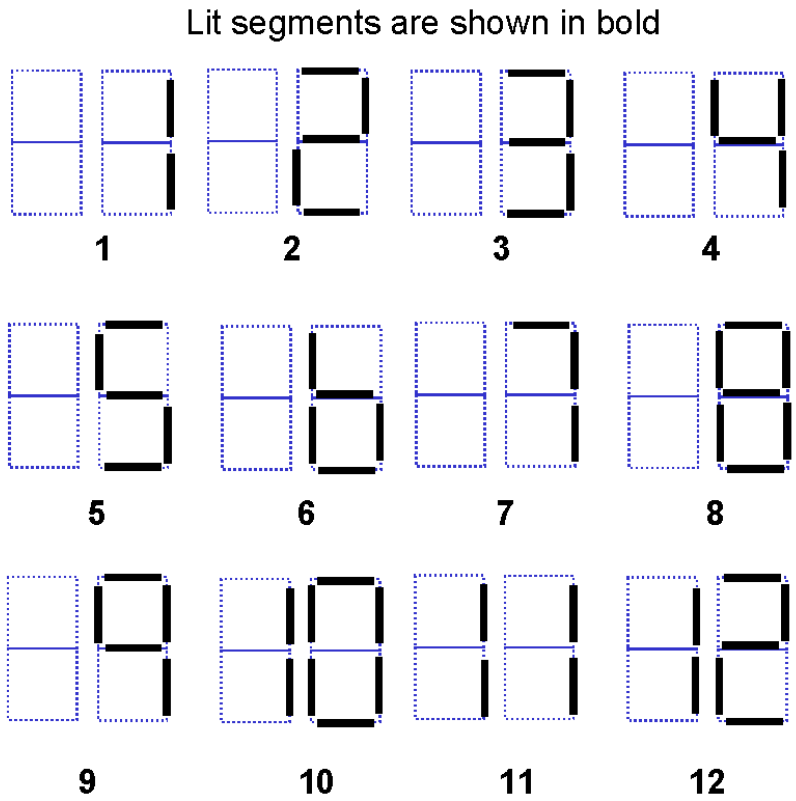
10- Four binary bits (one hexadecimal digit) can used to represent the numbers 0 through F. A seven-segment display, shown in the figure at the right, is typically used in many devices to display the numbers from 0 through 9. Note that in a pinch, it can also be used to represent the alpha portions of the hexadecimal numbers, A through F.



Each of the seven segments is lit up as a short bar segment, labeled a through g, by a light-emitting diode (LED). This shown schematically in the figure below. Consider a clock circuit, shown below.



Suppose that the clock generator circuit provides an output in the form of 4 binary digit lines A,B,C,D which taken together, provide an output with12 distinct hour values, encoded as 0000 through 1011. Assume that A is the most significant digit (MSD) and that D is the least significant digit (LSD) and that the output code 0000 represents 12 o'clock on the display. The figure, below, shows two digits of the display and how the segments are lit for each possible hour displayed.



Note that since the tens of hours display will only need to display nothing or the digit "1", we can save some circuitry by using one output line, "h", to drive both segments of the display.

a- Draw the truth table for each of the segments, "a" through "h", on the seven-segment display for the hours corresponding to the display of time that you would see on a clock.

b- Use the Karnaugh Map to simplify the logical equations for each of the segment outputs.

c- Draw a schematic diagram of the logic circuit for each segment. In order to simplify your drawing, you may assume that you have AND, OR, NAND and NOR gates available to you in 2, 3, or 4 inputs, as well as NOT gates. You also have XOR gates, but they are always 2-inputs gates, not more.

11- Assume that you have two 1-bit input variables, A and B and two one-bit control variables, C and D. There are two one-bit output variables, x and y. Create a truth table for each of the output variables and then simplify each truth table using the Karnaugh map. Then draw the simplified gate diagram for the following logical specification:

If C,D = 0,0 x = AND(A,B), y = NOR(A,B)

If C,D = 0,1 x = XOR(A,B), y = OR(A,B)

If C,D = 1,0 x = NOT(A), y = NAND(A,B)

If C,D = 1,1 x and y are not defined.

Hint: Since the case C,D = 1,1 is not defined you may find that using it in your truth table might help you to simplify the logical equations and the Karnaugh map.