COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

CPE 169 Experiment 3

Introduction to Function Reduction, Implementation, and Function Forms

Learning Objectives

1. Digital Logic
   - To analyze a digital circuit implemented on a programmable logic device
   - To reduce a typical digital function
   - To use DeMorgan's theorem to generate several forms of a function
   - To implement two equivalent forms of the reduced circuit using discrete logic gates

Introduction and Overview

One of the more interesting characteristics of digital circuits is that there is rarely only a single solution to a given circuit problem. It's generally true that you'll need your circuit to perform a certain function (have a specified output for a given set of inputs), but there are a seemingly infinite number of different digital circuits that can be designed to implement that same function. You may at this point be asking yourself, "What is the right or best circuit that I should use to implement my function?"

The "right" or "best" circuit implementation is dependent upon how exactly these criteria are defined. There is no one absolute definition that can be used to pinpoint the best circuitry to use to implement a function in all situations. The best circuit is dependent upon the context in which the circuit is used. Some of the more common definitions of the words "best" and "right" in the context of digital circuits are the most inexpensive, the smallest, the fastest, the easiest to implement, the one that can be implemented with the parts on hand, the one that the teacher wants you to implement, *etc*. The more creative you are, the longer this list becomes.

In this experiment, you'll *analyze* a circuit that you've downloaded into the programmable logic device on the development board. From there, you'll *realize* this same function using two different circuits which have been generated from clever applications of DeMorgan's theorem and Boolean algebra. Building the two circuits then allows you to verify that they are *functionally equivalent* despite the fact that they contain two distinctively different circuit implementations. This experiment highlights some of the more common methods used to implement functions using discrete logic gates. The approaches presented here are just two of the many different methods that *could* be used. As you further develop your digital circuit design skills, you'll be able to flow freely between the myriad of alternative circuit forms used to implement functions.

Function Reduction

Representing a function in its "smallest" form is probably the most common driving force behind the method you use to implement a function. Once again, this approach revolves around the definition of "smallest". In the context of digital logic, the word "smallest" has many meanings. In this experiment, you'll be implementing circuits using the logic gates contained in ICs, so the definition you'll want to use for "small" is the one that has the fewest number of gates. The indirect benefit from using the fewest gates is that you'll most likely also be using the fewest number of ICs. As you'll soon find out, implementing functions using discrete gates is tedious. The most effective method to control this tedium is by first reducing the equation describing the original function. Finding the circuit implementation that fits this definition is referred to as *function reduction* and is the underlying goal in most digital design applications.

There are three primary methods used to reduce functions: 1) software tools, 2) Boolean algebra theorems, and 3) Karnaugh mapping techniques. Once you graduate and start making the big bucks, you'll most likely be using the first technique exclusively. But until then, your only viable options are the last two techniques. Boolean algebra is effective, but is tedious and error prone for all but the few certified Boolean Jedi Masters. Karnaugh mapping techniques are your main tool for function reduction throughout this and subsequent digital courses. You may as well start enjoying Karnaugh maps now.

The nice thing about Karnaugh maps is that they are designed specifically for use by humans (as opposed to computers and/or extra-terrestrials). Karnaugh maps, or *K-maps* as they are referred to by the digital pros, are based upon the *logical adjacency* theorem. Function reduction using K-maps is nothing more than a *visual application of the logical adjacency theorem*. Reduction via K-maps takes advantage of your brain's ability to recognize distinctive shapes and patterns in the K-map. There are many rules that can be followed that allow you to detect these patterns and properly reduce the function. However, applying only a few rules along with some down-home intuition is arguably the most effective method of doing K-map reduction. The more severely rule-based your techniques are, the more you should consider using a computer to perform you circuit reduction, since they'll do a much better and faster job at following rules.

DeMorgan's Theorem

DeMorgan's theorem is one of the more commonly applied theorems of Boolean algebra. This is especially true in the context of circuit forms because the theorem allows for the transformation of a circuit from one form to another. DeMorgan's theorem is also useful in other fields such as discrete mathematics, computer programming, and bass fishing (well...maybe not that last one!). Table 1 lists DeMorgan's theorem for both two-variable and generalized forms. The final form listed in Table 1 is provided not only for its pure shock value, but also to emphasize the fact that the "variables" shown in the previous equations are not restricted to single Boolean variables as is implied by the equations. The symbols shown in the first two equations can be either simple Boolean variables or more complex Boolean expressions. In either case, the "overbar" symbol implies a negating operation that applies to the entire expression that it covers.

$\overline{X+Y} = \overline{X} \cdot \overline{Y}$	$\overline{X \cdot Y} = \overline{X} + \overline{Y}$
$\overline{X_1 + X_2 + \ldots + X_n} = \overline{X_1} \cdot \overline{X_2} \cdot \ldots \cdot \overline{X_n}$	$\overline{X1 \cdot X2 \cdot \ldots \cdot Xn} = \overline{X1} + \overline{X2} + \ldots + \overline{Xn}$
$\overline{\text{☯} + \text{☜} + \ldots + \text{👓}} = \overline{\text{☯}} \cdot \overline{\text{☜}} \cdot \ldots \cdot \overline{\text{👓}}$	$\overline{\text{☯} \cdot \text{☜} \cdot \ldots \cdot \text{👓}} = \overline{\text{☯}} + \overline{\text{☜}} + \ldots + \overline{\text{👓}}$

Table 1: DeMorgan's theorem in two-variable and generalized forms.

DeMorgan's theorem underlies a method we can use to transform the implementation of a Boolean logic function between a basic set of alternative circuit forms. In this way, transferring between these basic circuit forms provides a good example of the utility of DeMorgan's theorem. An overview of this functionality is presented in the example provided in the next section; and you will get a chance to put this method to use in the experiment that follows.

Circuit Forms

The term "circuit forms" is a somewhat common term in digital logic design vernacular. This term generally refers to the multiple, different logic circuits (different combinations of logic gates and input signals) that can be used to implement any given Boolean logic function. In other words, although the final output of the circuit using different forms is the same, the input equations on which the circuits are based can look considerably different. Using this definition, the end result is that while the various alternative circuit forms *appear* to be different, but they are actually *functionally equivalent*. In the context of a digital system, the term “functionally equivalent” refers to the fact that the input/output relationship of the circuit is preserved while the implementation details are different.

There are various reasons why you would want to use one circuit form over another. Generally speaking, one form is often desired over another. The more desired form is usually decided in terms of *efficiency*, in that one form may require fewer logic gates than the other forms. The intent of this experiment is to show that implementing a logic function using two different forms produces a functionally equivalent circuit.

The more common circuit forms that are discussed in this experiment can be generated with successive applications of DeMorgan’s theorem. The process to generate these forms follows two different paths depending on whether you start with a SOP or POS form, as indicated by the two columns of Table 2.

Circuit Forms Example

Equations **1(a)** and **2(a)** of Table 2 show the compact minterm and compact maxterm forms of an arbitrary function, respectively. These two forms can be reduced using Karnaugh-mapping techniques to the logic expressions shown in **1(b)** and **2(b)**. These two expressions are generated from grouping the 1’s of the circuit in a K-map (as in the left column) or the 0’s of the circuit (right column). The resultant minimized equations serve as the starting points for generating all the equivalent circuit implementation forms, as described in the following steps. Table 3 provides an in-depth written description of this procedure.

1(a) $F = \sum(1,4,5,9,10,11,13,14,15)$	**2(a)** $F = \prod(0,2,3,6,7,8,12)$
AND/OR Form	**OR/AND Form**
1(b) $F = AC + \overline{C}D + \overline{A}B\overline{C}$	**2(b)** $\overline{F} = \overline{A}C + A\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D}$ **2(c)** $\overline{\overline{F}} = \overline{\left(\overline{A}C + A\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D}\right)}$ **2(d)** $F = \overline{\left(\overline{A}C\right)}\cdot\overline{\left(A\overline{C}\,\overline{D}\right)}\cdot\overline{\left(\overline{A}\,\overline{B}\,\overline{D}\right)}$ **2(e)** $F = \left(A+\overline{C}\right)\cdot\left(\overline{A}+C+D\right)\cdot\left(A+B+D\right)$
NAND/NAND Form	**NOR/NOR Form**
1(c) $\overline{\overline{F}} = \overline{\overline{AC + \overline{C}D + \overline{A}B\overline{C}}}$ **1(d)** $F = \overline{\overline{(AC)}\cdot\overline{(\overline{C}D)}\cdot\overline{(\overline{A}B\overline{C})}}$	**2(f)** $\overline{\overline{F}} = \overline{\overline{\left(A+\overline{C}\right)\cdot\left(\overline{A}+C+D\right)\cdot\left(A+B+D\right)}}$ **2(g)** $F = \overline{\overline{\left(A+\overline{C}\right)} + \overline{\left(\overline{A}+C+D\right)} + \overline{\left(A+B+D\right)}}$
OR/NAND Form	**AND/NOR Form**
1(e) $F = \overline{\left(\overline{A}+\overline{C}\right)\cdot\left(C+\overline{D}\right)\cdot\left(A+\overline{B}+C\right)}$	**2(h)** $F = \overline{\left(\overline{A}C\right)+\left(A\overline{C}\,\overline{D}\right)+\left(\overline{A}\,\overline{B}\,\overline{D}\right)}$
NOR/OR Form	**NAND/AND Form**
1(f) $F = \overline{\left(\overline{A}+\overline{C}\right)}+\overline{\left(C+\overline{D}\right)}+\overline{\left(A+\overline{B}+C\right)}$	**2(i)** $F = \overline{\left(\overline{A}C\right)}\cdot\overline{\left(A\overline{C}\,\overline{D}\right)}\cdot\overline{\left(\overline{A}\,\overline{B}\,\overline{D}\right)}$

Table 2: The generation of standard circuit forms by using DeMorgan's law.

AND/OR Form	OR/AND Form
The form in 1(b) is the AND/OR form and is referred to as the Sum of Products (SOP) form. This form is obtained from a K-map by grouping the 1's of the circuit's output and applying K-map reduction techniques. The individual groupings in the K-map form what are referred to as the *product terms* (*AND* combinations). The final function represents a logical *summing* (*OR* operation) of the associated product terms.	The form in 2(b) is obtained by applying K-map reduction techniques to the 0's of the circuits output. In this case, since the K-mapping was based on the 0's of the circuit, the complement of the function ($\overline{F}$) is obtained. The expression is in AND/OR form but we'll massage it into a different form by determining an expression for F rather than $\overline{F}$ as is listed in 2(b). This is done by complementing the expressions on both sides of the equals sign, which preserves the equality and produces the equation shown in 2(c). The equation in 2(d) is obtained by dropping the double complement on the left side of equality. The right side of the equality is obtained by an application of DeMorgan's theorem. The equation in 2(e) shows the final OR/AND form which is also referred to as the Product of Sums (POS) form.
NAND/NAND Form	**NOR/NOR Form**
The form in 1(c) is obtained from the AND/OR form by double complementing both sides of the equation in 1(b). Double complementing each side of the equation preserves the equality of the expression. The double complement on the left side of the equation 1(c) drops out. On the right side of equation 1(c), one of the compleiments is used to *DeMorganize* the expression. The NAND/NAND form of the expression is shown in 1(d). This is referred to as NAND/NAND form because each of the individual product terms are complemented product terms (a NAND function). These individual terms are ANDed together and complemented, which effectively changes it from an AND function to an NAND function.	The form in 2(f) is obtained from the OR/AND form by double complementing both sides of the equation in 2(e). The double complement on the left side of the equation 2(f) drops out. On the right side of equation 2(f), one of the complements is used to *DeMorganize* the expression. The NOR/NOR form of the expression is shown in 2(g). This is referred to as NOR/NOR form because each of the individual sum terms are complemented (a NOR function). These individual terms are ORed together and compleimented which changes it from an OR function to an NOR function.
OR/NAND Form	**AND/NOR Form**
The OR/NAND form shown in 1(e) is obtained by DeMorganizing the individual terms from 1(d) to change them from product terms to sum terms. The overbar over the entire term is not altered.	The AND/NOR form shown in 2(h) is obtained by DeMorganizing the individual terms from in 2(g) to change them from sum terms to product terms. The overbar over the entire term is not altered.
NOR/OR Form	**NAND/AND Form**
The NOR/OR form shown in 1(f) is obtained by DeMorganizing the entire OR/NAND form shown in 1(e). In this way, the overbar on the right side of the equals sign is distributed to the individual terms in the equation.	The NAND/AND form shown in 2(i) is obtained by DeMorganizing the AND/NOR form shown in 2(h). In this way, the overbar on the right side of the equals sign is distributed to the individual terms in the equation.

Table 3: Written description of the circuit forms and derivations shown in Table 2.

Procedures

Procedure Overview: In this experiment, you are provided with a programming file that contains the implementation of a function which is downloaded to the development board. Your task is to analyze this circuit in order to generate a truth table representation of this circuit. You'll then implement this circuit using both the NAND/NAND and NOR/ NOR circuit forms to prove that these two forms are functionally equivalent. These two circuits can be implemented using the discrete logic gates contained on the ICs provided in the CPE 169 parts kit.

Procedure 1: Circuit Analysis and NAND/NAND Implementation

1. Download the appropriate programming file *exp3_function.bit* for your Nexys board from the CPE 169 website.

2. Figure 1 shows a black-box diagram of the circuit implemented in the FPGA. Table 4 shows the switch to signal mappings for the four input signals on the development board. Using the switches and LED, analyze the input / output behavior of the circuit programmed in the FPGA. Enter your results into a truth table and include the table with your final report.
   - The up and down positions of the switch represent 1's and 0's, respectively.
   - The lit and unlit conditions of the left-most LED on the development board represent 1's and 0's of the function's output, respectively.

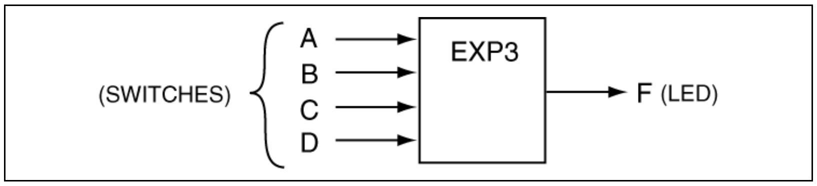

Figure 1: Black-box diagram of circuit represented by the provided programming file.

Development Board	A	B	C	D	F
Nexys & Nexys-2	SW7	SW6	SW5	SW4	LD7

Table 4: Switch and output mappings for the development boards.

3. Reduce the function described in the truth table to a minimum SOP expression using a K-map. Include your work from this step in the final lab report, including the minimized SOP expression.

4. Download the appropriate *exp23_signals.bit* configuration file from the course website to the FPGA on your Nexys board using the Digilent ExPort program. This program will again provide connections to the LEDs and the eight switch outputs on the Nexys board through the expansion connectors JA, JB, and JC as was shown in Experiment 2.

5. Using the ICs contained in the CPE 169 parts kit, implement the reduced SOP function using a NAND/NAND circuit form on the breadboard. Include a detailed schematic diagram of your circuit implementation in your lab report.

6. Test the circuit implemented in the previous step to verify that your circuit is functionally equivalent to the circuit provided in Step 1 on the FPGA. Show your results in a truth table and have the lab instructor or lab assistant verify proper operation of the circuit.

Procedure 2: NOR/NOR Circuit Implementation

1. Reduce the function described in the truth table generated in Step 2 from the previous procedure to a minimum POS expression using a K-map. Include your work from this step in the final lab report, including the minimized POS expression.

2. Using the ICs contained in the CPE 169 parts kit, implement the reduced POS function using a NOR/NOR circuit form on the breadboard. Include a detailed schematic diagram of this circuit implementation in your lab report.

3. Test the circuit implemented in the previous step. Show your results in a truth table and have the lab instructor or lab assistant verify proper operation of the circuit. Verify that your circuit is again functionally equivalent to the circuit originally provided in the FPGA.

4. Construct a truth table similar to the one shown in Table 5 and include it with your lab report.

A	B	C	D	F_{FPGA}	$F_{NAND/NAND}$	$F_{NOR/NOR}$
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

Table 5: The big truth table to be included with your report.

Questions

1. In an effort to implement circuits as cheaply as possible, you often want to find the *minimum cost* implementation of a circuit. The minimum cost of a circuit can be based on various factors such as the criteria listed below. Comment on which of the NAND/NAND or NOR/NOR circuit forms implemented in this experiment would be the cheapest based on the following minimum cost criteria:

   a) total gate count
   b) input count
   c) IC count
   d) time required to implement the circuit
   e) purchase price of ICs used in this experiment (do a search at www.digikey.com or www.jameco.com)

2. Show the minimized equations in each of the eight circuit forms (like those shown in Table 2) for the Boolean logic function that you analyzed and implemented in this Experiment.

3. Based on the function equations generated in the Question 2, which circuit form can be implemented using the fewest number of individual logic gates in this case? For this question, assume that the inputs to the system are all non-inverted; therefore, include the number of inverters required in your gate count. Assume you have the ability to use any type of gate with any number of inputs (such as 2,3,4-input NANDs, NORs, ANDs, ORs, *etc.*).

4. For the function equations generated in Question 2, which of the implemented function forms contains the fewest number of inputs to the gates used in the circuit? Why would the total number of gate inputs potentially be a design consideration?

5. Speculate on why IC manufacturers don't generally make items such as 7-input NAND gates.