ECE241: Digital Systems

Arnav Patil

University of Toronto

Contents

1 Introduction

1.1 Digital Hardware

Until the 1960s, individual transistors and registers had to assembled to form logic circuits. The invention of the integrated circuit allowed for multiple transistors to be placed together; the number of transistors has roughly doubled every two years in a phenomenon called *Moore's Law*. Integrated circuits are manufactured on a silicon wafer, which is cut to make individual chips. For many digital hardware products, it is necessary to build logic circuits; we can use three main types of chips: standard chips, programmable logic devices, and custom chips.

1.1.1 Standard Chips

Each standard chip contains a small amount of circuitry (usually <100 transistors) and performs a very simple function. Smaller chips can be interconnected to create larger logic circuits. These chips are no longer popular as they take up valuable printed circuit board (PCB) space for more capable chips and their functionality is fixed.

1.1.2 Programmable Logic Devices

We may also construct chips with programmable switches that allow the user to implement their own logic circuits. This way, the designer can achieve the functionality they require; these chips are known as *programmable logic devices (PLDs)*. The most common type of PLD is called a *field-programmable gate array, or FPGA*. Because of their capability and flexibility, FPGAs are very popular and widely in use today.

1.1.3 Custom-Designed Chips

FPGAs may not be able to achieve performance or cost objectives because they consume valuable chip area, which further reduces operating speed. Thus, some designers may choose to build a custom chip that meets their requirements, which is then fabricated and put into use. These chips are known as *application-specific integrated circuits (ASICs)*.

1.2 The Design Process

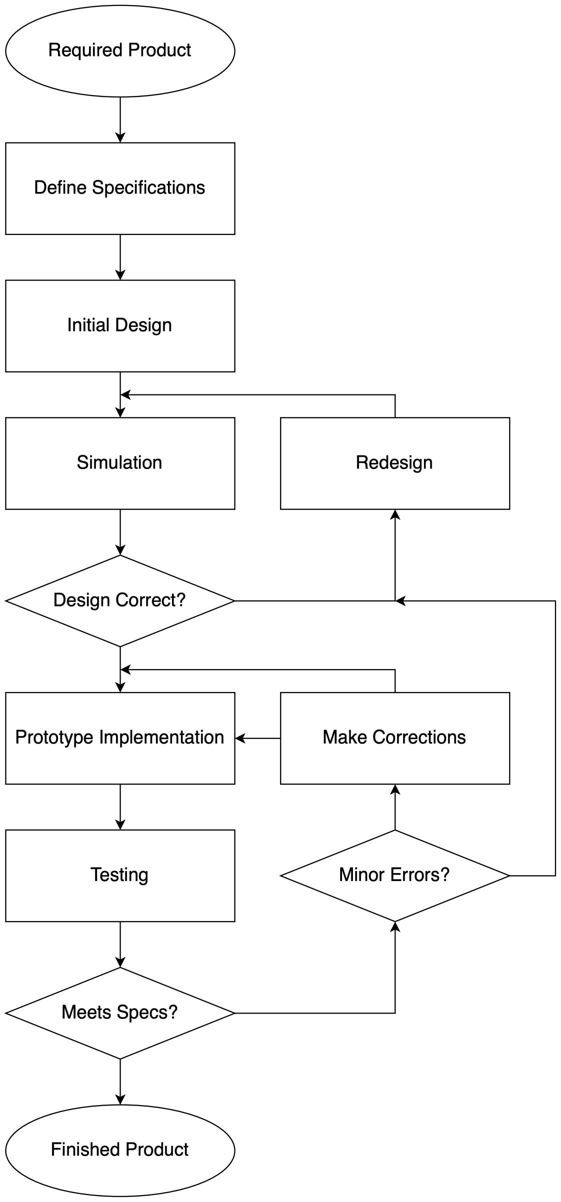


Figure 1: The Design Process.

1.3 Structure of a Computer

A computer consits of a number of PCBs, all of which are connected to a 'main PCB' called a motherboard. The motherboard holds several integrated circuit chips and provides slots for connections to other PCBs.

Each chip contains a number of subcircuits, which are interconnected and are logic circuits. Each logic circuit consists of a number of interconnected logic gates, which performs a very simple function. The text on which these notes are based is concerned with the design of these logic circuits. We will focus on building these circuits for speed or cost, and how they can be tested. We will also explore how transistors work and how they are constructed on silicon chips.

1.4 Logic Circuit Design in This Book

Section discussing Verilog HDL and Quartus II software.

1.5 Digital Representation of Information

Each digit is allowed to take on only one of two values: 0 or 1. 0 usually represents 0 V or ground, while 1 represents the logic circuit voltage (somewhere between 1-5 V).

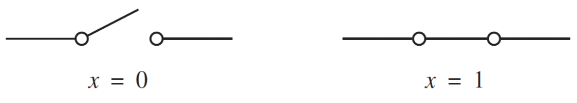
To convert a decimal number into a binary number, the easiest method is to procedurally divide by two (note down the reminders); though keep in mind this method gives the digits from the least significant bit (right most digit) to the most significant bit (left most digit).

2 Introduction to Logic Circuits

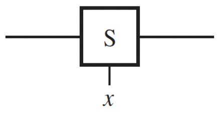
Any circuit in which signals are limited to some number of discrete values is called a **logic circuit**.

2.1 Variables and Functions

One of the simplest binary elements is the switch which has only two states: on and off.



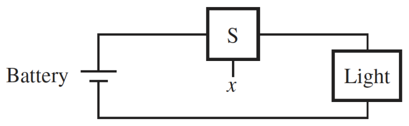
(a) Two states of a switch



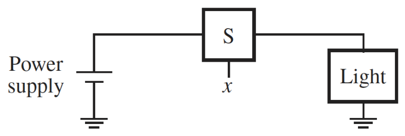
(b) Symbol for a switch

Figure 2: A simple binary switch

We can attach these switches into circuits to introduce control.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 3: Simple circuit with switch

In the above example, let us denote the light using the letter L. If the light is on, we say $L = 1$, and if it's off, we say $L = 0$. Since the light is controlled by the switch, we can denote L as a function of x, which we would write as $L(x) = x$. This simple logic expression describes the output as a function of the input. We can say that $L(x) = x$ is a logic function, and that x is a logic input.

Now we will consider what happens if we use two switches to control the light. If these switches are placed in series, then both must be turned on in order for the light to turn on. This is a logical AND function. The

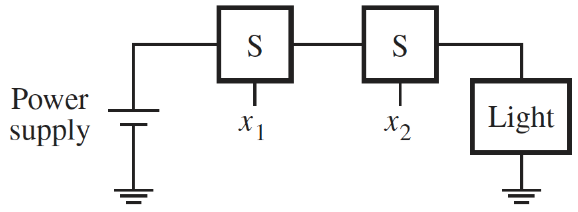


Figure 4: Logical AND function

behaviour of the AND function is described by:

$$L(x_1, x_2) = x_1 \cdot x_2$$

although we more commonly ignore the dot and instead write x_1x_2.

If, on the other hand, the switches were placed in parallel, then only one needs to be turned on in order for the light to turn on (note that the light will be on if both switches are turned on as well). This describes a logical OR function.

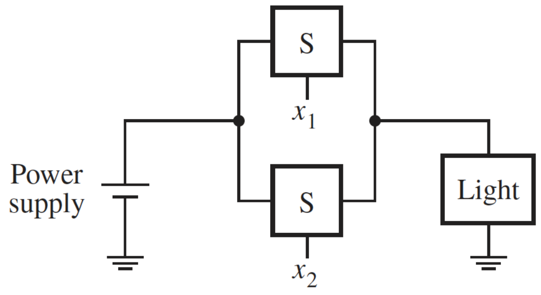


Figure 5: Logical OR function

The behaviour of the OR function is given by:

$$L(x_1, x_2) = x_1 + x_2$$

2.2 Inversion

Now we will consider a circuit where a positive action takes place when a switch is opened rather than closed. Consider the following inverting circuit, which described a NOT function. The reason this gate works is because of the fundamental circuit property that current will take the path of least resistance. When the switch is closed, it acts as a short circuit, meaning it will travel through the switch instead of the light (which has a given resistance), not turning it on. When the switch is opened, the current will be forced to go through the light, thereby turning it on. We describe the behaviour of the NOT function by:

$$L(x) = \overline{x}$$

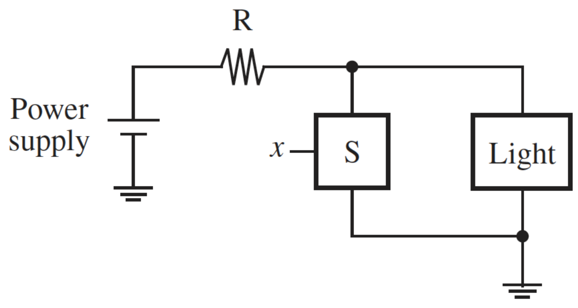


Figure 6: Logical NOT function

2.3 Truth Tables

We can visualize the behaviour of logic functions using a truth table, which gives the outcome for every possible input provided to the logic circuit. For example, here is a truth table visualizing both the AND and OR operations for two inputs.

x_1 x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0 0	0	0
0 1	0	1
1 0	0	1
1 1	1	1
	AND	OR

2.4 Logic Gates and Networks

Each logic operation can be implemented using transistors to form what's known as a logic gate. We often draw schematics to describe a logic function's inputs and outputs. The basic gates are given below:

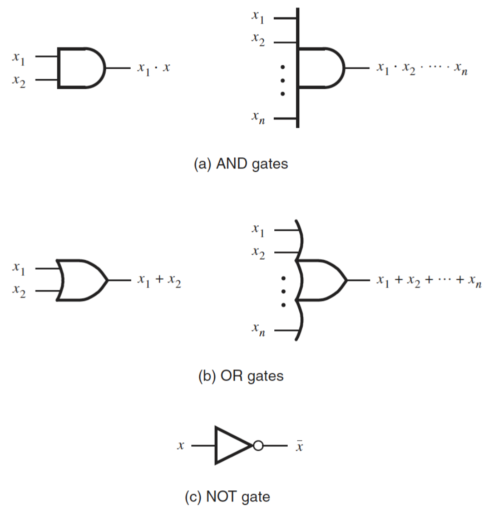


Figure 7: Basic Logic Gates

We can connect gates in a network to create a larger circuit. Since cost is a direct function of the complexity of any network, we try to find ways to describe a given logic function as simply as possible.

2.4.1 Analysis of a Logic Network

There are two parts to the digital systems design process: analysis (figuring out what the logic network does) and synthesis (building the actual circuit that does the function). The simplest method remains to build a truth table for the logic function, though that can make it difficult to visualize how exactly a given output is produced.

Timing Diagram

Timing diagrams are a way to visualize the changes in signal at different points in the network in a graphical form. Idealized waveforms rely on the assumption that logic gates can respond to changes to their inputs instantly, something that practical logic circuits do not do. However, we will study this behaviour in later chapters.

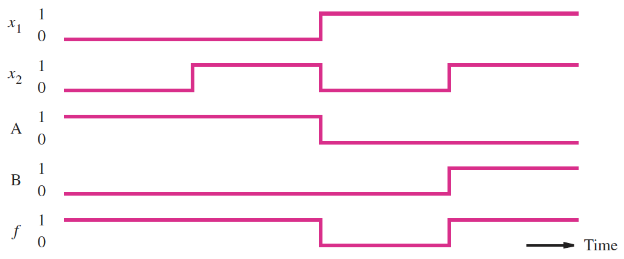


Figure 8: Timing Diagram for $f = \overline{x}_1 + x_1 \cdot x_2$

Functionally Equivalent Networks

Consider the function from the above timing diagram, $f = \overline{x}_1 + x_1 \cdot x_2$. The function $g = \overline{x}_1 + x_2$ will have the same diagram, as if we look closely, $f(x_1, x_2) = g(x_1, x_2)$. However, g is much simpler to implement, and it follows that one should implement g over f.

Example: Hallway Light Switch
Let us consider an example of a unique circuit, say one that controls the light in a hallway using two switches. When one is turned off and the other is turned on, the light is on. However, if both switches are on or off, the light is turned off. Essentially, we can define a function $L(x_1, x_2)$ where $L = 0$ when $x_1 = x_2$. The truth table for this function can be given by:

x y	L
0 0	0
0 1	1
0 0	1
1 1	0

Table 1: Truth Table for $L(x)$

This function is actually very common, so much so that it has a name: it's called the exclusive-OR (XOR) function. We write it as $L = \overline{x} \cdot y + x \cdot \overline{y} = x \bigoplus y$. The symbol is given by:

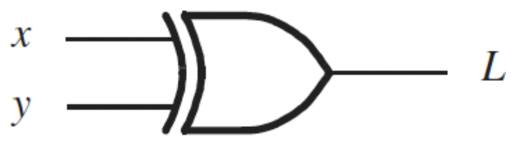


Figure 9: XOR gate symbol

Example: Two-Bit Adder

The XOR gate allows us to build a circuit to add binary numbers, known as an adder circuit. Since we are considering only two digits in the sum, $S = s_1 s_0$, we more specifically call it a two-bit adder.

a	b	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

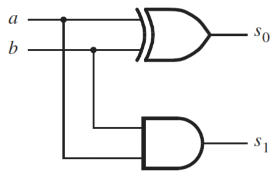


Figure 10: Truth Table and Logic Network for a Two-Bit Adder

2.5 Boolean Algebra

George Boole published the algebraic description of operations involved in logical thought and reasoning in 1849; this became known as Boolean Algebra. In the late 1930s, Claude Shannon showed that Boolean Algebra can describe circuits built with switches.

Axioms of Boolean Algebra

Like any other algebra, Boolean algebra is founded on axioms, or rules that we derive from basic assumptions. First we assume that Boolean algebra involves elements that take on only two values: 0 and 1. Then, we assume the following axioms are true:

1. $0 \cdot 0 = 0$
2. $1 + 1 = 1$
3. $1 \cdot 1 = 1$
4. $0 + 0 = 0$
5. $0 \cdot 1 = 1 \cdot 0 = 0$
6. $1 + 0 = 0 + 1 = 1$
7. If $x = 0$, then $\overline{x} = 1$
8. If $x = 1$, then $\overline{x} = 0$

Single-Variable Theorems

From the above axioms we can set some rules to deal with single variable expressions. These rules are more commonly called theorems. Therefore, if x is a Boolean variable, then the following theorems will hold true:

1. $x \cdot 0 = 0$
2. $x + 1 = 1$
3. $x \cdot 1 = x$
4. $x + 0 = x$
5. $x \cdot x = x$
6. $x + x = x$
7. $x \cdot \overline{x} = 0$
8. $x + \overline{x} = 1$
9. $\overline{\overline{x}} = x$

These theorems may be easily proved by showing that they hold true for $x = 1$ and $x = 0$.

Duality

Given a logic expression, its dual is obtained by replacing all AND operators with OR operators and vice versa. The importance of this concept will become apparent later in this chapter, when we show that every logic function can be expressed in at least two different ways.

Two- and Three-Variable Properties

1. $x \cdot y = y \cdot x$
2. $x + y = y + x$
3. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
4. $x + y \cdot z = (x + y) \cdot (x + z)$
5. $x + x \cdot y = x$ – Absorption
6. $x \cdot (x + y) = x$
7. $x \cdot y + x \cdot \overline{y} = x$ – Combining
8. $(x + y) \cdot (x + \overline{y}) = x$
9. $\overline{x \cdot y} = \overline{x} + \overline{y}$ – DeMorgan's Theorem
10. $\overline{x + y} = \overline{x} \cdot \overline{y}$
11. $x + \overline{x} \cdot y = x + y$
12. $x \cdot (\overline{x} + y) = x \cdot y$
13. $x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} + z$ – Consensus
14. $(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$

Example

We will prove the validity of the equation:

$$(x_1 + x_3) \cdot (\overline{x_1} + x\overline{x_3}) = x_1 \cdot \overline{x_3} + \overline{x_1} \cdot x_3$$

We can manipulate the left side using the distributive property as follows:

$$LHS = (x_1 + x_3) \cdot x_1 + (x_1 + x_3) \cdot \overline{x_3}$$

Applying the distributive property again:

$$LHS = x_1 \cdot \overline{x_1} + x_3 \cdot \overline{x_1} + x_1 \cdot \overline{x_3} + x_3 \cdot \overline{x_3}$$

By SVT 7, we have:

$$LHS = 0 + x_3 \cdot \overline{x_1} + x_1 \cdot \overline{x_3} + 0$$

Now we have:

$$LHS = x_1 \cdot \overline{x_3} + \overline{x_1} \cdot x_3 = RHS$$

2.5.1 Precedence of Operations

We use parentheses to indicate the order in which operations are performed. To limit the use of parentheses we establish a precedence of operations: NOT, AND, then OR.

2.6 Synthesis Using AND, OR, and NOT Gates

We will not try to implement some functions using the three gates. Suppose we want to synthesize a function that follows this truth table:

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Table 2: A Function to be Synthesize

We can find an AND expression for each line of the table, and connect them using OR operators. Thus, we can realize f as:

$$\begin{aligned} f(x_1, x_2) &= x_1x_2 + \overline{x_1}\,\overline{x_2} + \overline{x_1}x_2 \\ &= x_1x_2 + \overline{x_1}\,\overline{x_2} + \overline{x_1}x_2 + +\overline{x_1}x_2 \\ &= (x_1 + \overline{x_1})x_2 + \overline{x_1}(\overline{x_2} + x_2) \\ &= 1 \cdot x_2 + \overline{x_1} \cdot 1 \\ &= x_2 + \overline{x_1} \end{aligned}$$

This example highlights two things. First, a straightforward implementation is obtained by using the product term for each row of the truth table where the function equals 1. Second, there are multiple networks that can realize a given function, and some are much simpler and easier to implement than others.

2.6.1 Sum-of-Products and Products-of-Sums Forms

With an introduction into the synthesis process, we can now present it in more formal terminology. This is also where we'll explore how duality applies in the synthesis process.

Minterms

For a function of n variables, a product term in which each of the n variables appears once is called a minterm. These variables may be either complemented or uncomplemented, it only matters that they're present in the minterm.

Row	$x_1\ x_2\ x_3$	Minterm	Maxterm
0	0 0 0	$m_0 = \overline{x_1}\,\overline{x_2}\,\overline{x_3}$	$M_0 = x_1 + x_2 + x_3$
1	0 0 1	$m_1 = \overline{x_1}\,\overline{x_2}x_3$	$M_1 = x_1 + x_2 + \overline{x_3}$
2	0 1 0	$m_2 = \overline{x_1}x_2\overline{x_3}$	$M_2 = x_1 + \overline{x_2} + x_3$
3	0 1 1	$m_3 = \overline{x_1}x_2x_3$	$M_3 = x_1 + \overline{x_2} + \overline{x_3}$
4	1 0 0	$m_4 = x_1\overline{x_2}\,\overline{x_3}$	$M_4 = \overline{x_1} + x_2 + x_3$
5	1 0 1	$m_5 = x_1\overline{x_2}x_3$	$M_5 = \overline{x_1} + x_2 + \overline{x_3}$
6	1 1 0	$m_6 = x_1x_2\overline{x_3}$	$M_6 = \overline{x_1} + \overline{x_2} + x_3$
7	1 1 1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x_1} + \overline{x_2} + \overline{x_3}$

Table 3: Three-variable minterms and maxterms

Sum-of-Products Form

A function f can be represented as a sum of minterms that correspond to each row in its truth table for which $f = 1$. The resulting implementation is functionally correct, but is not necessarily the lowest-cost implementation. A logic expression consisting of product (AND) terms that are summed (OR'd) together is in its sum-of-products form. If each of the OR'd terms is a minterm, then the expression is in a **canonical sum-of-products form**.

Minterms with row-number scripts can be used to specify a given function in a more concise form. For example, we can describe a function as:

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

or even more simply as:

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

Maxterms

The principle of duality says that it is possible to synthesize a function f by considering the rows of a truth table for which $f = 0$. This approach considers maxterms, which are the complement of minterms.

Product-of-Sums Form

If a function f is specified by a truth table, then it's complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, or, where $f = 0$.For example, consider:

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2 \\ &= x_1\overline{x_2}\end{aligned}$$

We can use De Morgan's theorem to complement this expression, the result is:

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x_2}} \\ &= \overline{x_1} + x_2\end{aligned}$$

A logic expression consisting of sum (OR) terms that are factors of a product (AND'd together) is called a product-of-sum form. If each of the OR terms is a maxterm, then the expression is called a canonical product-of-sums.

We can use a shorthand notation as an alternative way of specifying a sample function:

$$f(x_1, x_2, x_3) = \Pi(M_0, M_1, M_2, M_7)$$

or more simply:

$$f(x_1, x_2, x_3) = \Pi M(0, 1, 2, 7)$$

The Π sign denotes a logical product operation.

2.7 NAND and NOT Logic Networks

The NAND and NOR functions are obtained by complementing the output of the AND and OR functions respectively. We place a bubble on the output side of the AND and OR gate symbols to get the new gate symbols, see below. Below is DeMorgan's theorem in terms of logic gates.

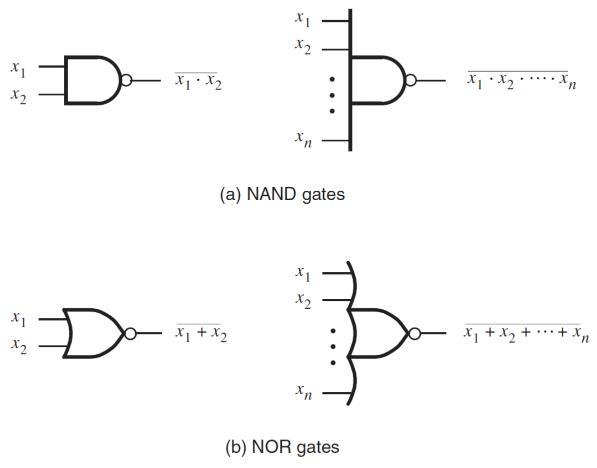

x_1
x_2
$\overline{x_1 \cdot x_2}$
x_n
$\overline{x_1 \cdot x_2 \cdot \cdots \cdot x_n}$
(a) NAND gates
$\overline{x_1 + x_2}$
$\overline{x_1 + x_2 + \cdots + x_n}$
(b) NOR gates

Figure 11: NAND and NOR gates

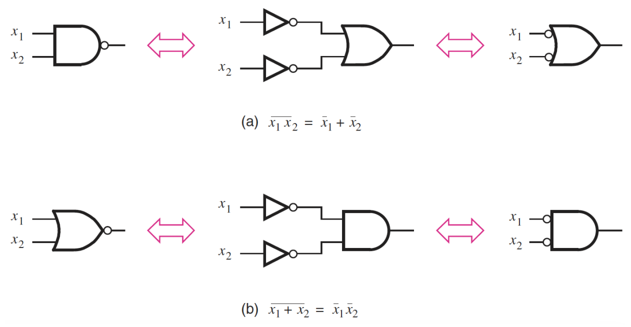

x_1
x_2
(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$
(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Figure 12: DeMorgan's theorem in terms of logic gates

2.8 Design Examples

2.8.1 Three-Way Light Control

We have a large room with three doors and a switch by each door. Flipping any one switch will turn the light on and off. We'll first define x_1, x_2, x_3 as our input variables that denote the state of each light switch. The light will be on if exactly one switch is closed, and off if two (or zero) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. The canonical sum-of-products expression for the specified function is:

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \overline{x_1}\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1\overline{x_2}\overline{x_3} + x_1x_2x_3 \end{aligned}$$

Unfortunately, this circuit cannot be simplified into a lower-cost SOP expression. However, we can also write this function in product-of-sums form. The expression would be:

$$\begin{aligned} f &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x_2} + \overline{x_3})(\overline{x_1} + x_2 + \overline{x_3})(\overline{x_1} + \overline{x_2} + x_3) \end{aligned}$$

2.8.2 Multiplexer Circuit

Sometimes, we want to design a system that chooses to output data from one of multiple sources; this is a circuit that outputs the same as either input x_1 or x_2, dependent on a third input s. We will assume that the circuit should output x_1 when $s = 0$ and x_2 when $s = 1$. We can derive the canonical sum of products:

$$f(s, x_1, x_2) = \overline{s}x_1\overline{x_2} + \overline{s}x_1x_2 + s\overline{x_1}x_2 + sx_1x_2$$

This equation may simplified to:

$$\begin{aligned} f &= \overline{s}x_1(\overline{x_2} + x_2 + s(\overline{x_1} + x_1)x_2 \\ &= \overline{s}x_2 \cdot 1 + s \cdot 1x_2 \\ &= \overline{s}x_1 + sx_2 \end{aligned}$$

2.8.3 Number Display

In this example, we want to design a logic circuit to drive a seven-segment display, that will allow us to show S as a decimal number, 0, 1, or 2.

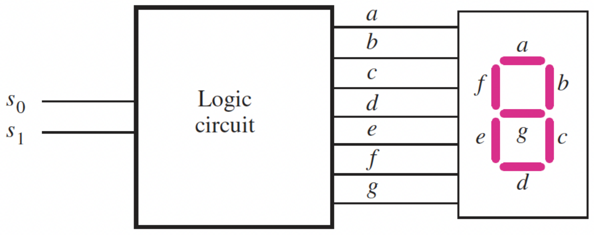


Figure 13: Logic Circuit to Seven-Segment Display

	s_1	s_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1

Figure 14: Truth Table for Display

Logic expressions for each of the seven functions:

$$\begin{aligned} &a = d = e = \overline{s_0} \\ &b = 1 \\ &c = \overline{s_1} \\ &f = \overline{s_1 s_0} \\ &g = s_1\overline{s_0} \end{aligned}$$

2.9 Introduction to CAD Tools

Logic circuits found in complex systems cannot be designed manually – they are designed using CAD tools that implement synthesis techniques.

2.9.1 Schematic Capture

A CAD tool used to enter a designed circuit this way is called a schematic capture tool; a schematic refers to a diagram of a circuit in which circuit elements are depicted as graphical symbols and connections are drawn as lines.

A schematic capture tool allows the user to draw a schematic diagram. The tool provides a collection of symbols, called a library, to represent various gates and elements. Designers often employ hierarchical design, which means creating a circuit that includes smaller circuits inside of it as a method of design.

2.9.2 Hardware Description Language

An HDL tool is similar to a computer programming language except that HDLs are used to describe hardware rather than a program to be executed on a computer. There are two HDLs that are endorsed as IEEE standards, Verilog HDL, and VHDL (Very High Speed Integrated Circuit Hardware Description Language). We will use Verilog in these notes.

2.10 Introduction to Verilog

Verilog was created as part of an effort to develop standard design practices for digital circuits. It was originally intended for simulation and verification of digital circuits.

2.10.1 Structural Specification of Logic Circuits

A gate is represented by indicating its functional name, outputs, and inputs. For example, a two-input AND gate with output y and inputs x_1 and x_2 is denoted as:

$$\textbf{and}\ (y, x_1, x_2);$$

A four-input OR gate is denoted by:

$$\textbf{or}\ (y, x_1, x_2, x_3, x_4);$$

Keywords **nand** and **nor** define the NAND and NOR gates respectively. The NOT gate is given by:

$$\textbf{not}\ (y, x);$$

A logic circuit is defined as a **module**, which contains the statements that define the circuit.

```
    module example1 (x1, x2, s, f);
        input x1, x2, s;
        output f;

        not (k,s);
        and (g, k, x1);
        and (h, s, x2);
        or (f, g, h);

    endmodule
```

Verilog Syntax

Names of modules and signals in Verilog must start with a letter, and can contain any letter and number, including the "$" and "_" characters, and Verilog is case sensitive. We use semicolons to indicate end of statements, and whitespaces are ignored. Comments begin with // and continue to the end of the line.

2.10.2 Behavioural Specification of Logic Circuits

We can also use “&” and “—” as AND and OR operators. The tilde “˜” negates the signal immediately following it. The **assign** keyword provides a continuous assignment for a signal. Whenever any signal on the right side changes, the value of the signal on the left will be re-evaluated and will change as well. We also have **if-else** statements:

```
    if (s==0)
        f=x1;
    else
        f=x2;
```

Verilog syntax requires that procedural statements (like if-else statements) be contained within **always** blocks. Always blocks evaluate statements in the order specified in the code, in contrast to the continuous assignment statements. If a signal is being assigned a value using procedural statements, we must declare it as a variable using the **reg** keyword.

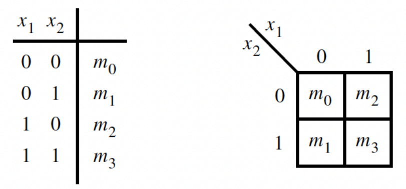
```
    module example5 (input x1, x2, s, output reg f);

        always@(x1, x2, s)
            if (s == 0)
                f = x1;
            else
                f = x2;

    endmodule
```

2.10.3 Hierarchical Verilog Code

For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a top-level module that includes multiple instances of lower-level modules.

2.11 Minimization and Karnaugh Maps

In general, it is not obvious how to go from a SoP or PoS implementation of a function to a simpler, reduced method. This section introduces Karnaugh maps, which are a more manageable approach to produce a minimum-cost logic expression. The key to this approach is that it allows the application of the Combining property as judiciously as possible.



(a) Truth table (b) Karnaugh map

Figure 15: Location of Two-Variable Minterms

The advantage of the Karnaugh map is that it allows us to easily recognize minterms that can be minimized using the Combining property. Minterms in any two adjacent cells can be combined. The Karnaugh map can be used to directly derive a minimum-cost circuit for a logic function.
The Karnaugh map is a simple mechanism to generate the product terms that are used to implement a given function. A product term must include only those variables that have the same value for all cells in the

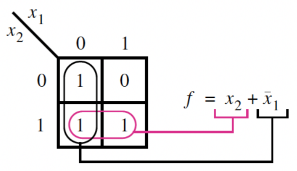
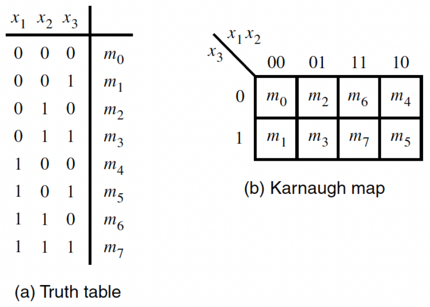


Figure 16: How to Read a Two-Variable Map



x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

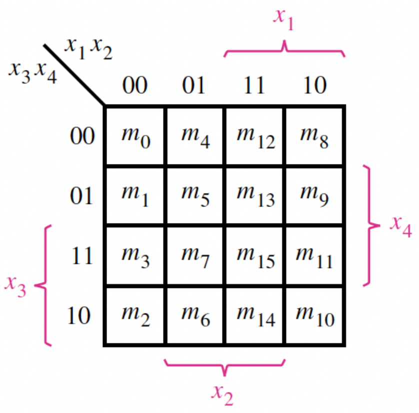
x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

Figure 17: How to Read a Three-Variable Map

group represented by that term. If that variable is 0, then it's complemented.

We can also build a four-variable Karnaugh map.



x_3x_4 \ x_1x_2	00	01	11	10
00	m_0	m_4	m_{12}	m_8
01	m_1	m_5	m_{13}	m_9
11	m_3	m_7	m_{15}	m_{11}
10	m_2	m_6	m_{14}	m_{10}

Figure 18: How to Read a Four-Variable Karnaugh Map

2.12 Strategy for Minimization

Our strategy in the previous section was to find few and large groups of variables that cover all cases where the function equals 1. For larger logic circuits, however, we should have a more organized method to derive a minimum-cost implementation.

2.12.1 Terminology

Lots of research has gone into developing techniques for synthesis of logic functions. Certain terminology has evolved to address the need to repeatedly describe highly-used phrases.

Literal – Each appearance of a variable in a product term, either complemented or not, is called a literal.

Implicant – A product term that indicates the input which equals 1 for a function is called an implicant.

Prime Implicant – A prime implicant is one that cannot be combined into another implicant with fewer literals. Another way to put it is that we cannot get an equal implicant if we remove any literals from the minterm.

Cover – A collection of implicants that account for all possible inputs for which a function will equal 1. The cover consisting of all prime implicants will lead to the lowest-cost implementation.

Cost – The sum of the number of gates and the total number of inputs to all gates in the circuit.

2.12.2 Minimization Procedure

1. Generate all prime implicants for the given function.
2. Find the set of essential prime implicants.
3. If the set of prime implicants covers all inputs for which $f = 1$, then the set is a good cover for f. If not, add the non-essential prime implicants to make a complete cover.

2.12.3 Incompletely Specified Functions

In digital systems it oftentimes occurs that certain input conditions can never occur. If we have two interlocked switches, x_1 and x_2, then the input valuations $(x_1, x_2) = 00, 01, 10$ can occur, but 11 is guaranteed to never occur in real life. Then, we can say that $(x_1, x_2) = 11$ is a don't care condition, meaning that a circuit with these inputs can be designed by ignoring this condition. A function with don't care conditions is known as incompletely specified. We can represent these in shorthand by:

$$f(x_1, ..., x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where D is the set of don't cares.

Although don't care values can be assigned arbitrarily, such an assignment may not lead to a minimum-cost implementation of a given function. With k don't cares, there are 2^k ways to assign 0 or 1 values to them.

3 Number Representation and Arithmetic Circuits

3.1 Positional Number Representation

When dealing with arithmetic operations, we will revert back to the regular symbols for addition and subtraction, $+$ and $-$. In the context of ambiguity between the OR operator and addition, it will be made explicit.

3.1.1 Unsigned Numbers

Numbers that are only positive are called unsigned, and those that may be negative as well are called signed. An n-bit unsigned number, $B = b_{n-1}b_{n-2}...b_1b_0$ represents an integer with the value:

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-1} \times 2^{n-2} + ... + b_1 \times 2^1 + b_0 \times 2^0 = \sum_{i=0}^{n-1} b_i \times 2^i$$

3.1.2 Octal and Hexadecimal Representations

The positional number representation can be used for any radix. If the radix is r:

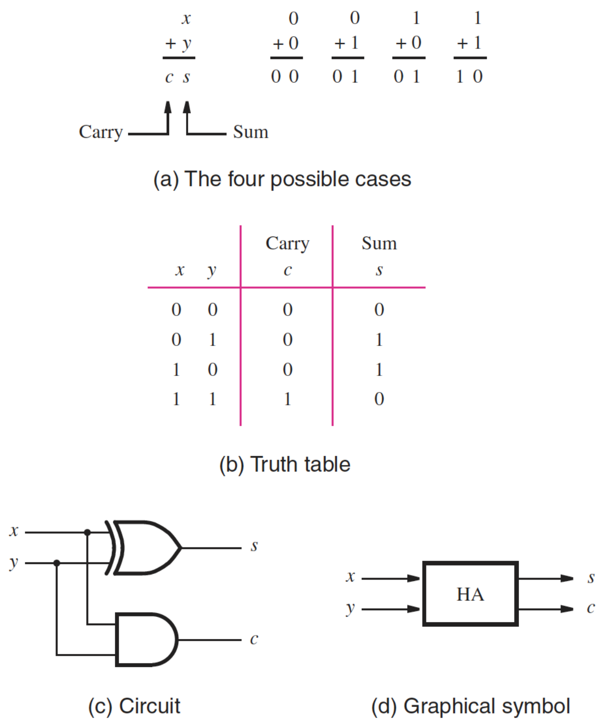
$$K = k_{n-1}k_{n-2}...k_1k_0$$

has the value:

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

3.2 Addition of Unsigned Numbers

Binary addition is performed the same way as decimal addition, except that individual digits may obviously only be 0 or 1. One-bit addition entails four possible combinations, and we need two bits to represent the result of the addition. The right-most bit is called the sum, s, while the left-most bit is called the carry, c. The sum bit s is realized by the XOR function, and the carry c is realized by the AND function. A circuit that implements the addition of two bits is called a half-adder.



x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 19: Half-Adder

When we extend to larger numbers with multiple bits involved, the addition operation at bit position i may include a carry-in bit from the position $i-1$. The optimal sum-of-products realization for the carry-out function is:

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

Whereas for s_i function, the sum-of-products realization is:

$$d_i = \overline{x}_1y_i\overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_iy_ic_i$$

Use of XOR Gates

The XOR functions of two variables is defined as $x_1 \oplus x_2 = \overline{x}_1 x_2 + x_1 \overline{x}_2$. The above expression for the sum bit can be manipulated into:

$$\begin{aligned} s_i &= (\overline{x}_1 y_i + x_i \overline{y}_i)\overline{c}_i + (\overline{x}_i \overline{y}_i + x_i y_i) c_i \\ &= (x_i \oplus y_i)\overline{c}_i + \overline{(x_i \oplus y_i)} c_i \\ &= (x_i \oplus y_i) \oplus c_i \\ &= x_i \oplus y_i \oplus c_i \end{aligned}$$

3.2.1 Decomposed Full-Adder

We can construct a full-adder by adding half-adder circuits onto one another in a multilevel circuit.

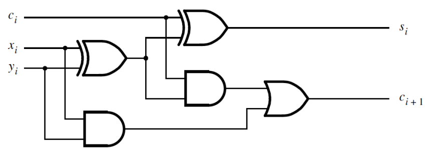


Figure 20: Implementation of the Full-Adder Circuit

3.2.2 Ripple-Carry Adder

We can implement a circuit that conducts the addition from the least-significant digit to the most-significant digit. If a carry is produced in position i, then it adds to the operands in position $i + 1$. We can use a full-adder circuit for each bit position, as given below:

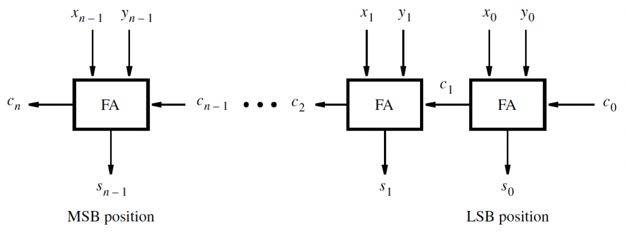


Figure 21: n-Bit Ripple-Carry Adder

The ripple-carry adder is named so because the signals "ripple" through the full-adder circuit.

3.3 Signed Numbers

In the binary system, the sign of a number is given by the left-most bit. If the left-most bit is 0, then the number is positive (or zero), otherwise it is negative. Since the left-most bit is the sign bit, the MSB in a signed number is the bit in position b_{n-2}.

3.3.1 Negative Numbers

We represent negative numbers in three different ways: sign-and-magnitude, 1's complement, and 2's complement.

Sign-and-Magnitude Representation

This is essentially what is described above. However, this representation is not well-suited for use in computers; this will be explained shortly.

1's Complement Representation

Negative numbers are defined according to a subtraction operation involving positive numbers. An n-bit number negative number, K, is obtained by subtracting its equivalent positive number, P, from $2^n - 1$. So, we have:

$$K = (2^n - 1) - P$$

With practice, we see that the 1's complement can be obtained by complementing each bit of the number, including the sign bit. These also have some drawbacks, which will be explained later.

2's Complement Representation

In this scheme, a negative number K is obtained by subtracting its equivalent positive number P from 2^n; namely, $K = 2^n - P$. We observe that:

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

We clearly see that $K_2 = K_1 + 1$.

3.3.2 Addition and Subtraction

Addition of positive numbers is the same for all three number representations. Things get more complicated when negative numbers are involved, and these difficulties become apparent when we consider operations with different combinations of signs.

Sign-and-Magnitude Addition

If both operands have the same sign, the addition is simple and the sum is given the sign of the operands. If they have opposite signs, then you have the subtract the smaller one from the larger one. We need logic circuits that compare and subtract needed; however, we don't need this circuitry to perform subtraction.

1's Complement Addition

An advantage of this scheme is that a negative number is the complement of all the bits in the corresponding positive number.

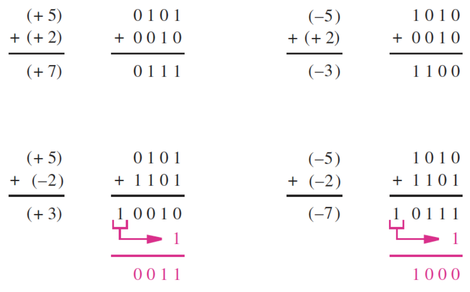


Figure 22: Examples of 1's Complement Addition

We see that in some cases, we need a correction, which is another addition operation that needs to be performed. So, we may need double the time compared to the time needed to add two unsigned numbers.

2's Complement Addition

When the numbers are added, the result is always correct, regardless of the operands. If there is an extra bit at the end, we can safely ignore it.

$$\begin{array}{rr} (+5) \\ +\ (+2) \\ \hline (+7) \end{array} \qquad \begin{array}{rr} 0101 \\ +\ 0010 \\ \hline 0111 \end{array} \qquad\qquad \begin{array}{rr} (-5) \\ +\ (+2) \\ \hline (-3) \end{array} \qquad \begin{array}{rr} 1011 \\ +\ 0010 \\ \hline 1101 \end{array}$$

$$\begin{array}{rr} (+5) \\ +\ (-2) \\ \hline (+3) \end{array} \qquad \begin{array}{rr} 0101 \\ +\ 1110 \\ \hline \underset{\substack{\uparrow \\ \text{ignore}}}{1}\,0011 \end{array} \qquad\qquad \begin{array}{rr} (-5) \\ +\ (-2) \\ \hline (-7) \end{array} \qquad \begin{array}{rr} 1011 \\ +\ 1110 \\ \hline \underset{\substack{\uparrow \\ \text{ignore}}}{1}\,1001 \end{array}$$

Figure 23: Examples of 2's Complement

2's Complement Subtraction