Digital Design

A Datapath and Control Approach

Chris Coulston

September 26, 2024

This document was prepared with LaTeX.

Contents

List of Processes

Digital Design Body Of Knowledge

The focus of this text is very much on the datapath and control approach. To achieve this end in a single semester, sacrifices in coverage must be made.

```
Digital Design
  Numbering Systems
    Positional Numbering Systems
      Base 10 - Decimal
      Base 2 - Binary
      Base 16 - Hexadecimal
    Conversion Between Bases
    Word Size
    2's Complement
  Representation of Logical Function
    Elementary Logical Functions
    Word Statement
    Truth Table
    Symbolic
    Circuit Diagram
    Hardware Description Languages
    Conversion Between Representations
    Timing Diagrams
  Logic Minimization
    Karnaugh Maps (Kmaps)
    Kmaps for circuits with multiple outputs
    Kmaps to find POSmin
    Logic Minimization Software
  Combination Logic Building Blocks
    Decoder
    Multiplexers
    Adders
    Comparators
    Three-State Buffers
    Wire Logic
    Combination
      Arithmetic Statements
      Conditional Statements
  Primitive Sequential Circuits
    Characteristics
    Timing
    Asynchronous set/reset
  Sequential Logic Building Blocks
```

Chapter 1

Numbering Systems

The goal of this textbook is to teach students how to design digital systems. To understand what this means, it is necessary to understand the behavior of a digital system. A digital system is a device which receives binary numbers as input and generates binary numbers as output. A binary number is a number composed of bits, or binary digits. A bit is equal to 0 or 1. Generally, bits are represented by voltages, a 0 by a ground potential and a 1 by a 3.3v or 5v potential. However, for most of this text, the physical representation of bits will take a back seat to the the logical representation. Figure 1.1 shows a digital system with three bits of input and two bits of output.
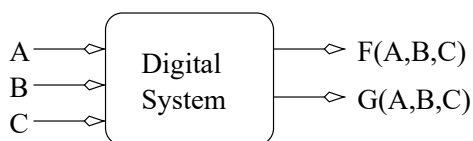
Figure 1.1: An abstract digital system with three bits of input labeled A,B,C and two bits of output, F(A,B,C) and G(A,B,C).

The letters A, B, C in Figure 1.1 represent *Boolean variables*, variables which are only allowed to assume the values 0 or 1. The notation $F(A, B, C)$ means that the output depends on the values of A, B, C.

Unfortunately, there is a disconnect between the normal way of describing quantities using the digits $0 \ldots 9$ and that used by digital systems using the bits $0, 1$. The study of digital systems starts by describing the numbering system used to describe decimal numbers and binary numbers. These systems are *positional numbering systems* - the position of a digit in a number determines its significance. The familiar decimal numbering system is used to illustrate the main concepts in positional numbering system.

1.1 Decimal

The goal of any numbering system (this includes both decimal and binary) is to represent quantities using a fixed set of symbols. One feature which distinguishes numbering systems is

the number of symbols used to represent quantities, the numbering system's *base* .

Decimal is a base-10 numbering system where quantities are expressed using 10 symbols $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. To explicitly denote a number's base, include the base as a subscript after the number. For example, the number of days in a year could be represented as 365_{10}.

Each digit in a number represents a quantity which depends on the digit, the base, and the position of the digit. The value of a digit is determined by the formula $d * b^i$ where d is the digit, i is the position of the digit and b is the base. The position of the digit immediately to the left of the decimal point is 0, every other digit is indexed starting from this position. The value of a multidigit number is the sum of the values of the digits, $\sum_{i=0}^{N} d_i * b^i$, where N is the number of digits in the number. Thus 365_{10} is interpreted as

$$3 * 10^2 + 6 * 10^1 + 5 * 10^0$$

The leftmost digit is called the most significant digit and the rightmost digit is called the least significant digit.

1.2   Binary

Binary is a base-2 numbering system where quantities are expressed using two symbols $\{0, 1\}$. When verbally communicating a binary value like 101_2, do not say "one hundred and one base two". The term *hundred* is a decimal concept and implies the quantity being described is decimal, contradicting the "base two". Instead, say "one zero one base two", with the "base two" part being optional. Enough about the nomenclature used to communicate binary numbers, how to represent values in binary? Since binary is a positional numbering system, then apply the formula $\sum_{i=0}^{N} d_i * b^i$.

Binary to Decimal

The value of 101_2 is interpreted as

$$101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 4_{10} + 0 * 2_{10} + 1 * 1_{10} = 4_{10} + 0_{10} + 1_{10} = 5_{10}$$

Application of the positional numbering formula to a binary number converts it into a decimal number. As in the decimal case there are special names for two of the bits. The leftmost bit is called the most significant bit (MSB) and the rightmost bit is called the least significant bit (LSB). As a final example, convert 1101_2 to decimal.

$$1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 8_{10} + 1 * 4_{10} + 0 * 2_{10} + 1 * 1_{10} = 8_{10} + 4_{10} + 0_{10} + 1_{10} = 13_{10}$$

Decimal to Binary

In order to provide inputs to a digital system, real world values represented in decimal need to be converted into binary. While there are several ways to perform this conversion, it makes sense to adapt a familiar procedure, the binary to decimal conversion. The key idea is to represent the decimal number as the sum of distinct powers-of-two. To assist in this procedure, use a power-of-two table.

i	0	1	2	3	4	5	6	7	8	9
2^i	1	2	4	8	16	32	64	128	256	512

The power-of-two table lists indices, i, and its value when it is raised to the exponent of 2. This table is used in the following 3-step decimal to binary conversion procedure.

Step 1 Find the largest power of two less than or equal to the number to convert.

Step 2 Subtract this power of two from the number being converting. This is the remaining number to convert.

Step 3 If the reminder is not equal to 0, go to step 1; otherwise stop.

The set of values found in Step 1 are the distinct powers-of-two that when added together equal the number to be converted. The conversion is completed by putting the sum into the positional numbering notation. The procedure is now applied to convert 13_{10} into binary.

Step	Action
1	The largest power-of-two less than or equal to 13 is 8.
2	The remaining number to convert is 13-8=5.
3	The new remainder is not 0.
1	The largest power-of-two less than or equal to 5 is 4.
2	The remaining number to convert is 5-4=1.
3	The new remainder is not 0.
1	The largest power-of-two less than or equal to 1 is 1.
2	The remaining number to convert is 1-1=0.
3	The new remainder is 0 so the conversion is complete.

Thus, $13_{10} = 8 + 4 + 1$. This derivation can also be expressed in the positional numbering notation as follows.

$$13_{10} = 8_{10} + 4_{10} + 1_{10} = 1*2^3 + 1*2^2 + 1*2^0$$

At this point the "missing" powers of two are included in the summation by setting their coefficients to 0. In the final step, all the coefficients are stripped off to form the binary number. Continuing with the previous example,

$$13_{10} = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 1101_2$$

Notice that this derivation is the exact reverse of the binary to decimal conversion shown on page 2.

What range of values can be described by N bits? Clearly, the smallest number to be represented is 0. To determine the largest number calculate how many different binary numbers can be formed with N bits. Each bit can be written in two different ways, 0 or 1. Since these choices are independent events, then the total number of possible outcomes is the product of the individual events. Hence, the number of ways to arrange N bits is equal to $2*2*\ldots*2$ (N times) which is equal to 2^N. Thus, N bits can be arranged in 2^N different ways. Since 0 is the smallest binary number then the maximum binary number is $2^N - 1$. The range of an N-bit binary number can be represented as $[0, 2^N - 1]$, where the [and] symbols mean that values 0 and $2^N - 1$ are included in the range.

1.3 Hexadecimal

Hexadecimal is a base-16 numbering system where quantities are expressed using 16 symbols $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. The symbols $\{A \ldots F\}$ represent quantities just like

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 1.1: The first 16 counting numbers represented in decimal, binary, and hexadecimal.

the symbols $\{0 \ldots 9\}$. The symbol A represents the quantity 10, B 11, C 12, D 13, E 14, and F 15. So if humans had adopted the hexadecimal numbering system instead of decimal you might say to a friend, "I had A friends over for a party last night" meaning that 10 people showed up. To show how hexadecimal is used as a shorthand for binary, consider the conversion of hexadecimal numbers into binary.

Hexadecimal to Binary

To convert a number from hexadecimal to binary, *unpack* it. That is, each hexadecimal digit is converted into a 4-bit binary representation. This conversion is possible because four bits exactly represents every hexadecimal digit as shown in Table 1.1.

In order to convert $1DAD_{16}$ to binary, replace each hexadecimal digit with its binary counterpart by consulting Table 1.1. For example $1DAD_{16} = 0001\ 1101\ 1010\ 1101_2$. The spaces between the binary numbers are included to make reading the number easier.

Binary to Hexadecimal

The above procedure can be reversed to convert binary numbers into hexadecimal. Group the bits into sets of four, starting at the least significant bit, then convert each set of four bits into its corresponding hexadecimal digit in Table 1.1. If there are not four digits in the most significant grouping, then just add 0s to make a grouping of three – that is pad the number with zeros. For example, convert 1110101011_2 into hexadecimal. $1110101011_2 = 0011\ 1010\ 1011 = 3AB_{16}$.

To understand why this conversion works, consider the binary number 1110101011_2. Start by writing down this number as the sum of powers of two, group the powers of two into sets of 4. The remaining steps are shown in the derivation below.

$1110101011_2 =$
$1 * 2^9 + 1 * 2^8 + 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 =$
$2^8 (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0) + 2^4 (1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0) + 2^0 * (1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0) =$
$2^8 (0011_2) + 2^4 (1010_2) + 2^0 (1011_2) =$
$2^{4*2} (0011_2) + 2^{4*1} (1010_2) + 2^{4*0} (1011_2) =$
$16^2 (0011_2) + 16^1 (1010_2) + 16^0 * (1011_2) =$
$16^2 (3_{16}) + 16^1 (A_{16}) + 16^0 * (B_{16}) =$
$3AB_{16}$

1.4   Binary Addition

The fact that the addition of binary numbers is similar to the addition of decimal numbers should not come as a surprise as both are positional numbering systems. However, when adding binary numbers with a digital system, it is possible that the result may exceed the digital system's capacity because the digital system can only accommodate a finite number of bit positions. The number of bits to be simultaneously manipulated by a digital system is refereed to as its *word size* . A digital system with a word size of N-bits can represent binary numbers in the range $[0, 2^N - 1]$.   *Overflow* occurs when a digital system is forced to represent a value outside the range of its word size.

The process of adding binary numbers is exactly the same as adding decimal numbers: Start in the LSB and work towards the MSB. Each of the bit-positions is called a *bit-slice* . At each bit-slice, the two bits of the sum are added together along with the carry-in generated in the previous bit-slice. This addition in a bit-slice will generate one bit of sum and possibly one bit of carry to the next bit-slice. The addition of bits must be performed in base-2, hence the results may look strange. For example, $1_2 + 1_2 = 10_2$. In this case the sum-bit equals 0 and the carry-bit equals 1. Now, consider a more complex problem, adding $3 + 2$ in binary, assuming a word size of four bits.

$$
\begin{array}{r|cccc}
 & & 1 & & \\
3 & 0 & 0 & 1 & 1 \\
+2 & 0 & 0 & 1 & 0 \\
\hline
5 & 0 & 1 & 0 & 1 \\
\end{array}
$$

Notice, one of the additions produced a carry which is propagated to the next significant bit position. The next example demonstrates an addition where the result is larger than the word size can accommodate.

$$
\begin{array}{r|ccccc}
 & 1 & 1 & 1 & 1 & 1 \\
13 & & 1 & 1 & 0 & 1 \\
+7 & & 0 & 1 & 1 & 1 \\
\hline
20 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
$$

The decimal result, 20, cannot be represented in four bits, hence the addition produced overflow. When adding binary numbers, overflow occurs whenever there is a carry-out from the most significant bit-slice. That is, the result requires more bits than are available in the word size.

After being introduced to binary numbering it might be tempting to look at all collections of bits as being binary numbers. In truth, a collection of bits has no implicit meaning. The sequence of bits, 0101, could just as easily represent the value 5 as it could represent the intensity of red on a display. A collection of bits gets its meaning from the interpretation used. When bits are used to represent integer quantities, two main interpretations, binary numbering and 2's complement, predominate.

1.5   Negative Numbers

The binary numbering systems is often called an *unsigned* numbering representation. The term unsigned arises from the fact that there is no need to write a sign symbol in front of a binary number because all binary numbers are positive – the positive sign is implicit. A *signed* numbering representation, 2's complement, is capable of representing both positive and negative numbers.

Like binary numbering, 2's-complement numbers exist within the confines of a word size. One way to determine the 2's-complement representation of a decimal number x, is to write down the binary representation for the quantity $2^N + x$ using N bits, where N is the word size. For example, assuming a word size of four bits, determine the 2's-complement representation for 6. To do this, compute $2^N + x = 2^4 + 6 = 16 + 6 = 22 = 10110_2$. Taking the least significant four bits yields 0110.

There are two points to note. First, this representation is the same as in binary numbering. Second, the 2's-complement value is written without a subscript 2, because it is not a binary number. Now consider the 2's-complement representation of a negative number.

Assuming a word size of four bits, determine the 2's-complement representation for -6. Compute $2^N + x = 2^4 - 6 = 16 - 6 = 10 = 01010_2$. Taking the least significant four bits yields 1010.

To determine the decimal value of a 2's-complement number, inspect its MSB. If the MSB is 0, then the number is positive, hence can be interpreted as a binary number. If the MSB is 1, then $2^N + x$ must be solved for x. There is, however an easier way to approach this problem.

Negating a 2's-complement number will mean changing the sign of the underlying decimal representation. The negation of a 2's-complement number, x, can be formed by flipping all the bits of x and then adding 1. For example, take the complement of the 4-bit 2's-complement number $x = 0110$ which equals 6. Flipping all the bits of x yields 1001. Adding 1 to this yields 1010, which was previously shown to equal -6.

This technique aids in interpreting negative 2's-complement numbers as follows. Given a 2's-complement number that is negative, form its negation, convert that to decimal, then stick a negative sign in front of the decimal representation. For example, determine the decimal representation for the 4-bit 2's-complement quantity 1010. Since the MSB is 1, this 2's-complement number represents a negative quantity. Flipping the bits, 0101, then adding 1, results in 0110. This is the representation for 6, so the original 2's-complement number 1010 represent -6.

Figure 1.2 shows every combination of four bits and their associated 2's-complement representation.

Clearly, half of the numbers in Figure 1.2 have a leading 0 as their MSB, and are positive; 0 is considered a positive number. The other half of the numbers have their MSB equal to 1 and are negative. Since 0 is considered a positive number, the largest negative number is 1 larger than the largest positive number. Given a word size of N bits the range of 2's-complement numbers is $[-2^{N-1}, 2^{N-1} - 1]$.

2's-complement numbers are added in the same way that binary numbers are added as shown in the following three problems.

```
                  |               | 1  1  1          |             | 1  1
      6    | 0  1  1  0           3 | 0  0  1  1          6 | 0  1  1  0
   + -7    | 1  0  0  1        + -2 | 1  1  1  0       + 6 | 0  1  1  0
   = -1    | 1  1  1  1         = 1 | 0  0  0  1      = 12 | 1  1  0  0
```

The first example, $6 + -7$, shows the addition process working correctly when the operands have different signs. The other two problems illustrate the need for a new definition of overflow
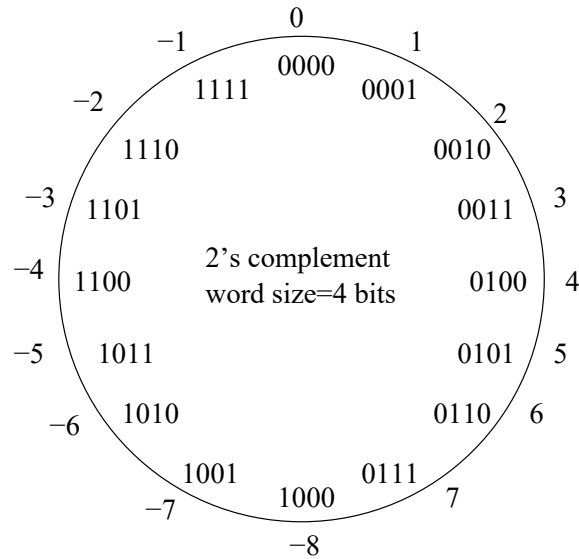
Figure 1.2: All possible combinations of four bits and their 2's-complement interpretations.

for 2's-complement numbers. The general rule for overflow in 2's complement is, if the carry-in and carry-out into and out from the MSB are not equal, then overflow has occurred. The carry-out from the MSB is always thrown away. For example, the carry-in and carry-out of the MSB in second problem are both 1, hence the result is valid. In the third problem, the carry in to the MSB is 1 and the carry-out from the MSB is 0, hence overflow has occurred. This result should not be a surprise because the expected result, 12, cannot be represented as a 4-bit 2's-complement number.

Taking the complement of a 2's-complement number allows subtraction problems to be converted into addition problems. For example, the subtraction problem $3-2$ can be rewritten as $3+(-2)$.

$$
\begin{array}{rcccc}
 & & & 1 & 1 & 1 \\
3 & 0 & 0 & 1 & 1 \\
-\ +2 & 0 & 0 & 1 & 0 \\
\hline
\end{array}
\quad \text{Becomes} \quad
\begin{array}{rcccc}
 & & 1 & 1 & 1 & \\
3 & 0 & 0 & 1 & 1 \\
+\ \text{-2} & 1 & 1 & 1 & 0 \\
\hline
=\ 1 & 0 & 0 & 0 & 1 \\
\end{array}
$$

Situations will arise which require increasing the number of bits required to represent a number while retaining the value of the number. For example, imagine having a 4-bit binary number that must be stored in a device that holds eight bits. How can this be done while preserving the magnitude of the number? For binary numbers, the answer is easy, add 4 leading zeros, an operation called *padding with 0s*.

For 2's-complement numbers this solution will not work. For example, consider the 4-bit 2's-complement representation for -1 (1111) that needs to be store in a 8-bit device. Adding 4 leading 0s will change the value of the number to 00001111, the value 15. The solution, in this case, is to pad with 1s, yielding 11111111, which represents -1 in 8-bit 2's complement. As with the binary numbering example, positive 2's-complement numbers can be padded with 0s and still retain their value. The general rule for padding in 2's complement is called *sign extension*; and involves copying the MSB to fill in the needed space. Table 1.2 shows five

4-bit 2's complement	8-bit 2's complement
1110=-2	11111110=-2
1010=-6	11111110=-6
1000=-8	11111000=-8
0011=3	00000011=3
0111=7	00000111=7

Table 1.2: Five examples showing how to sign-extend a 4-bit 2's-complement number.

examples of sign extension on 2-bit 2's-complement numbers.

1.6 Exercises

1. **(1 pt. each)** Syllabus:

   a) What is the late penalty for homework?

   b) True or False: Calculators can be used during exams.

   c) True of False: University ID is required during exams.

   d) What is my thesis regarding grades?

   e) Bob L. Student has the following grades. Determine his final overall course percentage and grade.

Component	Percentage
Homework	60%
Exam 1	90%
Exam 2	80%
Final	70%

   f) How should you prepare for the 43rd lecture?

2. **(1 pt. each)** Convert the following numbers to decimal. Show work, or receive $1/2$ credit.

   a) 100_2

   b) 1000_2

   c) 10000_2

   d) 100000_2

   e) 111111_2

   f) 1000100101000101_2

   g) $3EA_{16}$

3. **(1 pt. each)** Convert the following number to binary. Show work, or receive $1/2$ credit.

   a) 44_{16}

   b) 44_{10}

   c) 1023_{10}

4. **(1 pt. each)** Convert the following number to hex. Show work, or receive $1/2$ credit.

   a) 101011101_2

   b) 77_{10}

5. **(2 pts. each)** Toughies:

   a) Convert 123_5 to base-12

   b) Convert 789_{12} to base-5

   c) What is the largest base-10 quantity that can be represented using 5 digits in base 12?

6. **(1 pt. each)** Perform the following additions, assume a word size of four bits. Determine if overflow occurs.

a) $0110_2 + 0101_2$

b) $0010_2 + 0110_2$

c) $0111_2 + 0011_2$

d) $0010_2 + 0101_2$

e) $0010_2 + 1010_2$

f) $0101_2 + 1011_2$

g) $0011_2 + 1001_2$

Chapter 2

Representations of Logical Functions

Wo

A digital system starts its life deep inside an engineer's mind. In the beginning it is an abstract device; its very far removed from reality. In order to realize the digital system, the engineer undertakes the design process; a series of refinements to bring the digital system closer and closer to a real working system. This iterative approach is necessary in complex designs because going straight to a final implementation is overwhelming, error-prone, difficult to modify, and difficult to debug and test. The goal in breaking the design process into steps is to allow the proper specification of the digital system using a "language" which is easy to understand and modify, and then to show how to implement this specification using real circuit elements.

Four refinements in the design of digital systems are considered. Each of these refinements define the input, output, and behavior of the digital system.

Word Statement - A written description of the expected input, output, and behavior of the digital system.

Truth Table - A listing of every possible input to the digital system and the corresponding output.

Symbolic - A symbolic (math-like) description of the output(s) as a function of the input variable(s).

Circuit Diagram - A pictorial representation of the physical interconnections of the circuit elements.

When designing a digital system each of these representations should describe the same system, just in different ways. The most abstract representation of a digital system is the word statement. Word statements are notoriously ambiguous and have no standardized structure. However, these shortcomings are the strengths of this representation. The great expressibility of language allows very complex processes to be captured in brief statements. Hence, ideas can be quickly refined without having to expend much engineering effort. Also, because word statements have no prescribed form, there is a great deal of flexibility in structuring a word

statements. Because of its potential ambiguity and unclear boundaries, the word statement's step in the design process diagram shown in Figure 2.1 is drawn inside a puffy cloud.

Most Abstract   Most Real

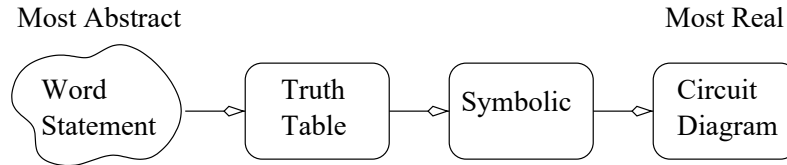Word Statement → Truth Table → Symbolic → Circuit Diagram

Figure 2.1: The steps in the design of a digital system.

Each of the four design stages shown in Figure 2.1 is a description of a digital system. The arrows in Figure 2.1 are transformations described in this chapter.

After the word statement, all subsequent stages of the design process are unambiguous, they have one clearly understood interpretation. The first refinement of a word statement is a truth table. The truth table strikingly illustrates the difference between digital and analog circuits. When working with analog phenomena, like the AC voltage from a wall outlet, it is impossible to list all possible values of the voltage, because an infinite number of potential values exists. However, when working with digital phenomena, each signal can have only one of two possible values, making it quite easy to enumerate them all. When working with a digital system with three bits of inputs, like that shown in Figure 1.1, there are only eight different combinations of the inputs. For each of these input combinations, a truth table specifies what the output should be.
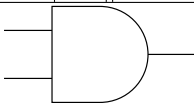
The symbolic form is a set of equations written in Boolean Algebra. These equations look similar to those encountered in an algebra class. However, instead of operations like addition and multiplication, Boolean Algebra uses the operations AND, OR, and NOT. These operations have hardware counterparts - real physical circuits which "compute" their values. A circuit diagram shows how these hardware components are interconnected to realize the digital system. As real devices, these circuits represent the bit values, 0 and 1 as voltages. Typically, logic 1 is represented by a high voltage level (5v) and logic 0 is represented by a low voltage level (0v).
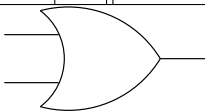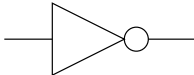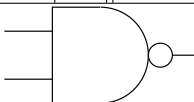
This text takes a bottom-up approach in designing digital systems. The design process starts with the most fundamental digital hardware components and shows how they are interconnected to form basic building blocks. These building blocks are then arranged into datapath and control circuit. Thus, all the digital circuits studies in this text will be built from a small pallet of fundamental digital hardware components, called the elementary logical functions.
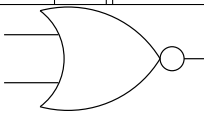
2.1 Elementary Logical Functions
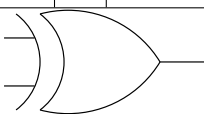
Elementary logical functions get their name because they are the fundamental (elementary) building blocks of digital design, they use the logical operators like AND, OR, and NOT (logical), and they describe a transformation (function) from input to output. For each elementary logical function, its word statement, truth table, symbolic and circuit diagram are presented.

AND

Word Statement	AND is a logical function with two inputs and one output. The output equals 1 only when all the inputs are equal to 1, otherwise the output equals 0.
Symbolic	A*B

Truth Table	A	B	A*B
	0	0	0
	0	1	0
	1	0	0
	1	1	1

Circuit Diagram

OR	Word Statement	OR is a logical function with two inputs and one output. The output equals 1 when any input is equal to 1, otherwise the output equals 0.
	Symbolic	A+B
	Truth Table	<table><tr><td>A</td><td>B</td><td>A+B</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>
	Circuit Diagram	
NOT	Word Statement	NOT is a logical function with one input and one output. The output is not equal to the input.
	Symbolic	A'
	Truth Table	<table><tr><td>A</td><td>A'</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>
	Circuit Diagram	
NAND	Word Statement	NAND is a logical function with two inputs and one output. The output equals 1 when any input is equal to 0, otherwise the output equals 0.
	Symbolic	(A*B)'
	Truth Table	<table><tr><td>A</td><td>B</td><td>(A*B)'</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>
	Circuit Diagram	

NOR	Word Statement	NOR is a logical function with two inputs and one output. The output equals 0 when any input is equal to 1, otherwise the output equals 0.
	Symbolic	(A+B)'
	Truth Table	

A	B	(A+B)'
0	0	1
0	1	0
1	0	0
1	1	0

Circuit Diagram	

XOR	Word Statement	XOR is a logical function with two inputs and one output. The output equals 1 when the inputs are different, otherwise the output is equal to 0.
	Symbolic	$A \oplus B$
	Truth Table	

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Circuit Diagram	

The physical elements for the elementary logical functions are typically called *gates*. Thus, the physical circuit for the AND logical function is called an AND gate.

The AND/OR functions can be enlarged to handle more than two inputs because their word statements use words ALL/ANY respectively. Thus a 4-input AND gate will output 1 only when all four inputs equal 1. The circuit diagram for this AND gate would look just like a 2-input AND gate with two additional inputs.

2.2 Conversion Between Representations

Figure 2.2 shows the four representations of a digital system along with the different transformations covered in this section. The arrows are numbered by the process in which the transformation is covered. The design process shown in Figure **??** is captured in Steps 7, 6, and 4. The other transformations introduced in this section enable designs to be optimized as well as to explore the relationships between the representations.

2.3 Circuit Diagram to Truth Table

The first transformation is the circuit diagram to truth table. Where the circuit diagram came from is unimportant for now. Figure 2.3 is a circuit diagram with three bits of input labeled A, B, C and one bit of output $F(A, B, C)$ that will be transformed into a truth table.
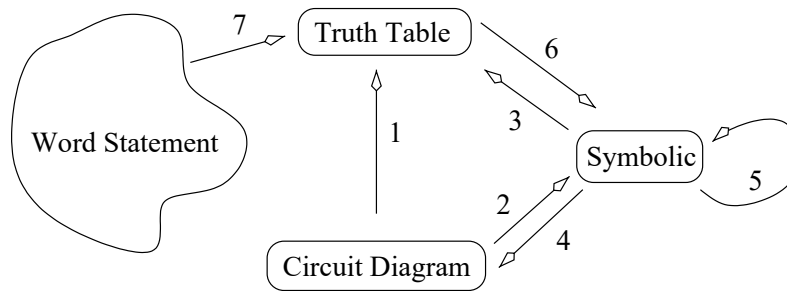
Figure 2.2: The four representations of a logical function. The numbered arrows are the transformations examined in this chapter.
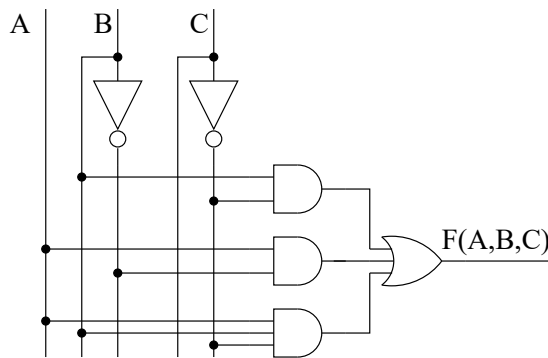

Figure 2.3: A simple 3-input, 1-output circuit.

Before starting with the transformation there are some important observations to make about Figure 2.3.

1. The gates in a circuit may have different numbers of inputs. For example, this circuit contains two AND gates with two bits of input and an AND gate with three bits of input.

2. The lines drawn in the figure represent physical wire.

3. When a voltage is applied to one end of the wire by an external source or by one of the gates, it is assumed to propagate instantaneously to all points on the wire.

4. The black dots on intersecting wires means that the two wires are connected. For example, the vertical wire labeled "A" is connected to the second and third AND gates.

5. Wires that cross one another and do not have a dot are assumed not to be connected. For example, the C signal does not go into the second AND gates.

6. Outputs are never connected directly together, because they could create a short-circuit destroying the participating gates.

7. Inputs are always tied to some signal source, usually the output of a gate.

8. A gate output may drive the inputs of many different gates.

Since the three external inputs in Figure 2.3 do not have sources shown, they must be provided by some external user. The output from a gate can provide input to multiple, different

inputs. For example, the output of the NOT gate associated with the C input feeds the first and third AND gates.

2.4 Circuit Diagram to Truth Table

Process 2.2: Circuit Diagram to Truth Table

To lend context to the discussion of the conversion process, let's describe this process by converting the circuit diagram in Figure 2.3 into a truth table.

Step 1: Identify the inputs and outputs of the circuit diagram. Any wire which is not being driven by some source is an input and any output which is not driving a source is an output. In the case of Figure 2.3, A, B, C are inputs and $F(A, B, C)$ is an output.
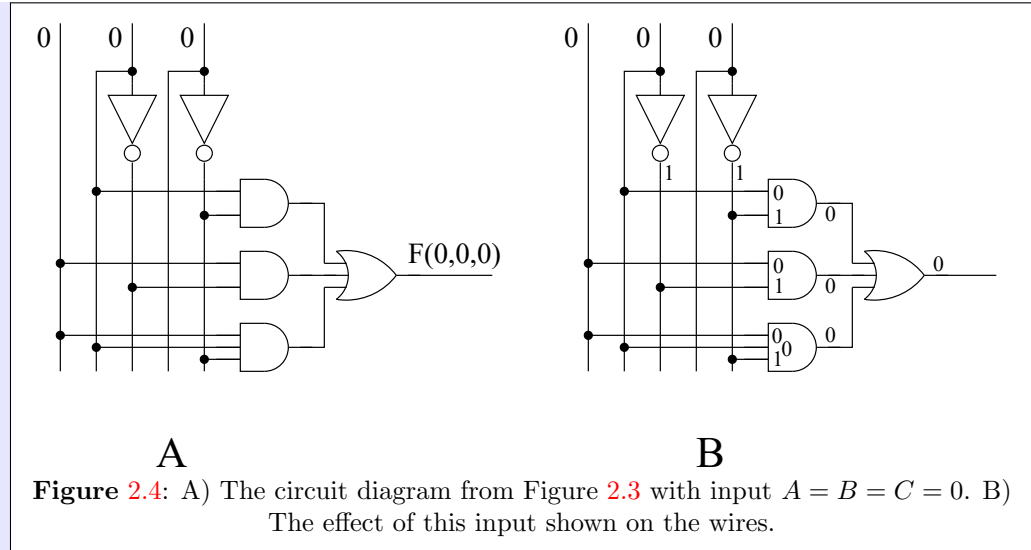
Step 2: Build the shell of the truth table to hold the solution. Remember that, "A truth table is a listing of every possible input to the digital system and the corresponding output." For example $A = 1$, $B = 0$ and $C = 1$ is a possible input to the circuit. All the potential inputs to the circuit could be listed in a *ad-hoc* manner, but a much more organized approach is shown below.

A	B	C	$F(A, B, C)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Two good reasons drive this choice of row ordering. First, the columns of the truth table have a strong pattern. The bits of the C-variable column repeat from top to bottom, $0101 \ldots$. The bits of the B-variable column repeat from top to bottom, two 0s followed by two 1s. The bits of the A-variable column repeat from top to bottom, four 0s followed by four 1s. Using this pattern makes filling a truth table easier. Second, when the A, B, C variables in each row are interpreted as a 3-bit binary number, they form consecutive integers. This approach makes finding a particular input/output much easier.

Step 3: Determine the values in the output column. In order to do this each input is applied, one at a time, to the circuit and determine the output of the circuit. This step is also referred to as evaluating the output of the circuit.

Imagine that the bits are flowing down hill from the inputs to the outputs. A gate computes its output only when the values of all its inputs are known. The output of a gate is determined from the truth tables given on pages 12 through 15. For example, in Figure 2.4A, the input $A = B = C = 0$ is applied to the input. The third AND gate cannot compute its output until it receives the output from the NOT gate associated with the C input. The outputs from each gate are shown on the signals in Figure ??B. Due to size constraints, the AND gate inputs are written inside the gate.

Figure 2.4: A) The circuit diagram from Figure 2.3 with input $A = B = C = 0$. B)
The effect of this input shown on the wires.

Figures 2.3 and 2.4 illustrate a subtle notational convention. Why is the output of
the circuit shown in Figure 2.3 labeled $F(A, B, C)$ and in Figure 2.4A this same output
is labeled $F(0, 0, 0)$? It is because the variables A, B, C in $F(A, B, C)$ are placeholders
for the actual values of the variables. Since these variables are given the values $A, B, C =
0, 0, 0$ in Figure 2.4A, this notational change is acknowledged by labeling the output
$F(0, 0, 0)$.

Since Figure 2.4B shows that $F(0, 0, 0) = 0$, then a 0 is placed in the top row of the
truth table for this function. The remaining seven rows of the truth table are computed
using this same method. The resulting truth table is shown below.

A	B	C	$F(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

While this transformation provides an excellent introduction to a variety of important
concepts, it is not a practical way to derive the truth table entries. First, it is tedious because
the same process must be performed eight times. Second, it is prone to errors because the bits
on the figure must be overwritten seven times, once for every row of the table. It turns out
that determining the symbolic form for this circuit's output and then determining the truth
table is a much less tedious and less error-prone method to determine the truth table from a
circuit diagram.
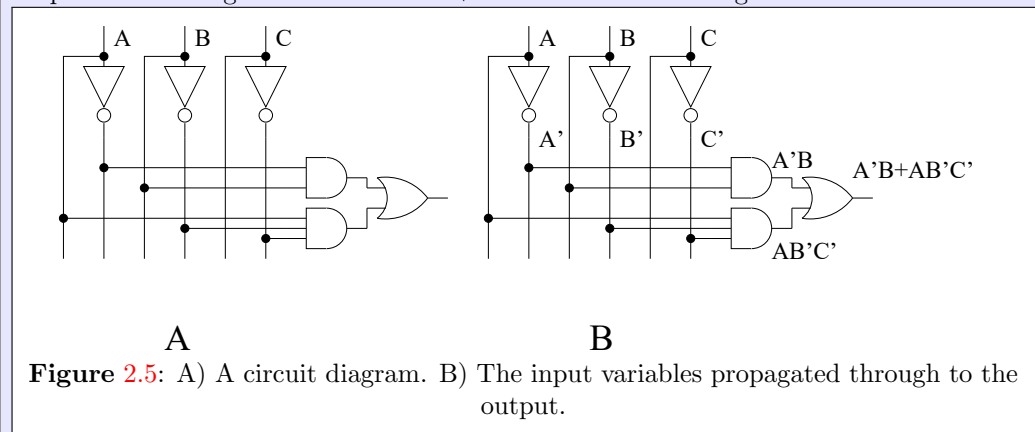
2.5   Circuit Diagram to Symbolic

Process 2.4: Circuit Diagram to Symbolic

The transformation of a circuit diagram into a symbolic expression is very similar to evaluating the output of the circuit, except symbols instead of bits flow through the circuit. To lend context to the discussion of the conversion process, let's describe this process by converting the circuit diagram in Figure 2.5into a symbolic form.

Step 1: Identify the inputs and outputs of the circuit diagram. See Step 1 of Process 2.

Step 2: Build the shell of the truth table to hold the solution. See Step of Process 2.

Step 3: Determine the values in the output column. The symbolic output of a gate is written out only when symbolic inputs are present on all of the gate's input. The output of a gate is determined from the symbolic forms given on pages 12 and 15. For example, in order to determine the output of the AND gates in Figure 2.5A, the symbolic outputs of the NOT gates must first be determined. These are A', B', C' as shown in Figure 2.5B. Since the inputs to the top AND gate are A' and B, the output of the AND gate is labeled $A'B$. The output was not labeled "A'*B" because the "*" is often dropped from the symbolic expressions involving AND in order to save space. This is similar to the convention of dropping the "*" symbol when it is used as the multiplication operator in regular algebra. When two Boolean variables appear next to one another it is assumed that they are ANDed together. After labeling the output of the other AND gate, the output of the OR gate is labeled $A'B + AB'C'$ as shown in Figure 2.5B.

Figure 2.5: A) A circuit diagram. B) The input variables propagated through to the output.

It was earlier suggested that in order to determine the truth table for a circuit diagram, it is easier to first determine the symbolic form for the circuit diagram, and then determine the truth table from this symbolic form. Determining the symbolic form of a circuit's output is eight times easier than determining all eight rows of the truth table. The next section will show how to transform a symbolic form into a truth table.

2.6   Symbolic to Truth Table

Before we get started with this transformation you will need to understand and apply the concept of evaluation.

Evaluation is the process of substituting each variable's value into the equation and applying the operators to the values. For example, evaluate $F(A, B) = (A + B)'$ for $(A, B) = (0, 1)$.

Substituting in the values for the variables yields $F(0,1) = (0+1)'$. Applying the OR operation to 0 and 1 gives the result 1. Applying the NOT operator to the result of the OR operator yields 0. Thus, $F(A,B)$ evaluated at $(A,B) = (0,1)$ equals 0. In other words, $F(0,1) = 0$. The evaluation of more complex expressions requires you to apply the rules of operator precedence in Boolean Algebra.

Operator precedence specifies the order in which the operations of an expression are evaluated. Operations with higher precedence are performed before operations with lower precedence. The rules of operator precedence are listed for both algebras in the following table.

	Regular Algebra	Boolean Algebra
High precedence	Parenthesis	Parenthesis
	Exponents	Not
	Multiplication/Division	And
Low precedence	Addition/Subtraction	Or

For example, if you given the Boolean function F(A,B,C,D) = (A + B(C' + D))' and asked to evaluate $F(0,1,0,0)$ you would first substitute the values of the variables into the expression. To do this look at the left to right order of the Boolean variables in the $F(A,B,C,D)$ expression and match them to the left to right order of the bit values in $F(0,1,0,0)$. Thus everywhere you see a A you will substitute the value 0. B will get replaced with 1, C with 0 and D with 0. This will produce the expression:

$$(0 + 1(0' + 0))'$$

You need to apply operator precedence to determine which operation to perform first. Since parenthesis are the highest precedence operation, we to "look" inside the outer most parenthesis and evaluate:

$$0 + 1(0' + 0)$$

Since parenthesis are the highest precedence operator, we must evaluate what is inside the parenthesis first:

$$0' + 0$$

Since NOT has a higher precedence than OR, we perform the NOT before performing OR.

$$0' + 0 = 1$$

We then go back and put this value into expression where the original parenthesis expression. This gives us

$$0 + 1 * 1$$

Since AND has a higher precedence than OR, we get

$$0 + 1 * 1 = 1$$

Substituting this value into expression where the original parenthesis expression gives us:

$$1' = 0$$

Thus, $F(0,1,0,0) = 0$

Process 2.6: Symbolic to Truth Table

To lend context to the discussion of the conversion process, let's describe this process by converting the symbolic expression $F(A, B, C) = A'B + AB'C'$ into a truth table.

Step 1: Identify the inputs and outputs of the circuit diagram. Like the transformation of a circuit diagram to a truth table, the first step in this transformation is to identify the inputs and outputs of the circuit. Normally, the variable on the left-hand side of the equal sign in an equation is the output and any variables which appear on the right-hand side are inputs. In the symbolic expression $F(A, B, C) = A'B + AB'C'$, F is the output and A, B, C are the inputs.

Step 2: Build the shell of the truth table to hold the solution. Write down the shell of the truth table using the ordering given on page 17. The basic structure of the truth table is the same no matter how many variables the symbolic expression has. For example, if a symbolic expression has four input variables A, B, C, D, then start by listing the variables across the top of the truth table. Then write the bit values for the D variable by alternating 0 and 1 on every row, from top to bottom. The C variable would alternate every other row, from top to bottom. The B variable every fourth row, and the A variable every eighth row. The resulting truth table has 16 rows. This should not come as a surprise since, from page 3, there are $2^4 = 16$ different combinations of four input bits.

Step 3: Evaluate the symbolic expression for each combination of inputs. We will evaluate $F(A, B, C) = A'B + AB'C'$ for every combination of (A, B, C) Let's start with the first row of the truth table and determine $F(0, 0, 0)$ by substituting $0, 0, 0$ for every occurrence of A, B, C in the symbolic expression. This yields $0' * 0 + 0 * 0' * 0'$. In order to evaluate this expression, perform the highest precedence operator first. In this case, evaluate the three NOTs yielding, $1 * 0 + 0 * 1 * 1$. The next highest precedence operator is AND, evaluating the two ANDs in the expression yields, $0 + 0$. Evaluating the remaining OR yields 0. Hence, $F(0, 0, 0) = 0$. Now continue this process for the remaining 7 row to produce the finished truth table.

A	B	C	$F(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

This process of evaluating the function for large numbers of inputs is tedious and error prone. In order to make good on the promise to produce an efficient transformation of a circuit diagram to a truth table, a better method is needed to complete the truth table from the symbolic form.

Two conditions make evaluating a symbolic expression difficult: its size and the evaluation process itself. Each difficulty will be addressed in turn. Instead of treating the symbolic expression as a single monolithic entity, break the expression into smaller subexpressions, evaluate each subexpression for all the inputs, and then put the values of the subexpressions back together. Three criteria govern the decomposition of an expression into subexpressions.

First, break the the expression into as few pieces as possible. Second, break the expression into subexpressions which are easy to put back together. Third, break the expression into subexpressions which are easy to evaluate.

For example, look at the expression $A'B + AB'C'$. This expression splits nicely into two subexpressions $A'B$ and $AB'C'$. This division meets the three criteria, only two subexpressions are defined, the subexpressions can be put back together by ORing them, and, as shown next, there is a simple way to evaluate the two subexpression.

In order to efficiently evaluate an expression for all of the rows of a truth table the number of questions asked about the expression must be reduced. This can be done for product terms. A *product term* is a group of variables (or variables negated) which are ANDed together. For example, $A'B$ and $AB'C'$ are both product terms. Instead of evaluating a product term for each row of a truth table, ask, "what input(s) cause the product term to evaluate to 1?" Since AND evaluates to 1 when all its inputs are 1, identify the rows of the truth table where the inputs equal 1. For example the product term $AB'C'$ evaluates to 1 only when $A = 1$, $B = 0$, and $C = 0$. Note, the B and C variables are negated in the product term so the variable must equal 0, so that its negation equals 1 before being ANDed. On the row $(A, B, C) = (1, 0, 0)$ the product term $AB'C'$ equals 1. What does the product term equal for all the other rows of the truth table? Since the product term does not equal 1 for these rows, the only alternative is for it to equal 0. Consequently, the remaining seven rows of the truth table equal 0 as shown in Table 2.1.

A	B	C	$A'B$	$AB'C'$	$F(A, B, C)$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0

Table 2.1: The truth table for $F(A, B, C) = A'B + AB'C'$ and its two subexpressions.

What inputs cause the product term $A'B$ to equal 1? Clearly, $A = 0$ and $B = 1$. While filling out the truth table for this product term there may be some confusion how to handle the C variable. Since the C variable is not present in the product term then its value is irrelevant. Consequently, the product term $A'B$ evaluates to 1 for $(A, B, C) = (0, 1, 0)$ and $(A, B, C) = (0, 1, 1)$, and evaluates to 0 for all other inputs as shown in Table 2.1.

Finally, combine these two subexpressions and compute the value of the function $F(A, B, C) = A'B + AB'C'$ for all rows of the truth table. In order to do this, look at each row of the truth table and ORing the value of the two subexpressions together. Since OR evaluates to 1 when any of its inputs equals 1, then it suffices to look for rows where either of the two subexpressions equal 1 and put a 1 for the output for F. All other rows will equal 0. The result is shown in Table 2.1.

The words *realize* and *implement* are used somewhat interchangeably to mean executing the design process in order to build a circuit – to make the digital system real. Since this text focuses on the logical behavior of circuits, not their physical behavior, the design process ends with a circuit diagram, see Figure **??**. In general, when asked to realize or implement a circuit make it as real as possible. Along similar lines, the realization or implementation of a circuit is its physical manifestation. In order to answer questions about a digital systems realization or

implementation, reason about the behavior of its physical implementation. In the next section the last step in the realization of of digital systems, symbolic to circuit diagram, is examined.

Process 2.8: Symbolic to Circuit Diagram

The transformation of a symbolic expression into a circuit diagram is a recursive journey from the output to the input parsing the expressions along the way. To (loosely) paraphrasing Meriam Webster, parsing is the process of dividing an expression into subexpressions and identifying the relationship between the expression and subexpresions. An *expression* is a collection of boolean variables connected properly to one another by elementary logical functions.

To lend context to the discussion of the conversion process, let's parse $F(A, B, C) = A * (B + C') + B'C$ into a circuit diagram.

Step 1: Indentify the lowest precedence operator in the expression. When a symbolic expression is evaluated, some operation is always performed last, yielding the single-bit output of the function. This operation is the lowest precedence operator of the expression. In the expression for $F(A, B, C)$, the lowest precedence operation is the OR between the subexpressions $A * (B + C')$ and $B'C$.

Step 2: Draw the gate of the lowest precedence operator in the expression. This pretty simple, we draw the OR gate, labeled "1" in Figure **??**.

Step 3: Connect the lowest precedence operator's output to the parent. Paraphrasing Mariam Webster, a parent is an entity from which other subordinates (children) arise.
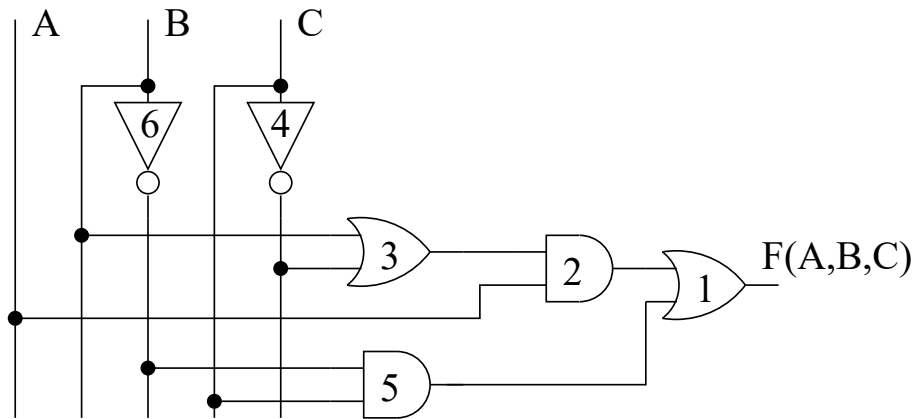
In this parsing process a *parent expression* is broken down into zero or more *child expressions* in Step 2 by removing gate from the parent expression.

In this example we started **Step 1** by parsing $F(A, B, C)$ so this is the parent expression. So we will connect the output of the OR gate we removed from the circuit in **Step 2** to $F(A, B, C)$

Step 4: Remove the lowest precedence operation from the expression creating zero or more child subexpressions. Removing the OR operator from $A * (B + C') + B'C$ yields two subexpressions, $A * (B + C')$, and $B'C$.

Step 5: Parse each child subexpression independently. Each of the subexpressions created in **Step 4** need to be parsed independently. This means that we must parse $A*(B+C')$, and $B'C$. This means that we will use $A*(B+C')$ as the starting expression at **Step 1**. The output of the (child) expression $A * (B + C')$ is an input to the parent expression of $A*(B+C')$ which is OR gate formed when we parsed $A*(B+C')+B'C$. We now proceed to complete the parsing process below starting with the two subexpression $A * (B + C')$ and $B'C$.

Parse $A * (B + C')$		Parse $B'C$	
Step 1: The lowest precedence operation in this expression, AND between A and $(B + C')$.		**Step 1:** The lowest precedence operation in this expression, AND between B' and C.	
Step 2: Draw the AND gate labeled 2 in Figure 2.6.		**Step 2:** Draw the AND gate labeled 5 in Figure 2.6.	
Step 3: Connect the output of AND gate labeled 2 to the input of the OR gate labeled 1.		**Step 3:** Connect the output of AND gate labeled 5 to the input of the OR gate labeled 1.	
Step 4: Remove the AND from $A * (B + C')$ producing two child subexpressions A and $(B + C')$.		**Step 4:** Remove the AND from $B'C$ producing two child subexpressions B' and C.	
Parse A	Parse $(B + C')$	Parse B'	Parse C
leaf	**1:** OR $B + C'$	**1:** NOT B'	leaf
	2: OR gate 3	NOT gate 6	
	3: Connect 3 to 2	**3:** Connect 6 to 5	
	4: B and C'	**4:** B	
Parse B	Parse C'	Parse B	
leaf	**1:** NOT C'	leaf	
	2: NOT gate 4		
	3: Connect 4 to 3		
	4: C		
	Parse C		
	leaf		

Figure 2.6: The circuit diagram for $F(A, B, C) = A(B + C') + B'C$.

Two simplification were made during the parsing process which deserve some attention leaves and *parenthesis*. When there the parsing process can go no further, there are no more operators in the expression, that expression is called a *leaf* . Examples of leafs include input variables and logic 0 and logic 1. You can think of these as leaves on a metaphorical parse tree.

Parenthesis are special operators which have no circuit representations. They allow us

a notation that enables low precedence operators to be evaluated before higher precedence operators. consequently, when an expression consists of a single set of parenthesis containing a expression, the parenthesis can be removed and parsing resumed with the expression.

During the parsing process, it is critically important to keep track of which parts of the expressions have been parsed and which have not. To do this, add lines under each of the subexpressions as they are created. After correctly parsing the expression, each variable should have a little line underneath it. This parsing-management can be a dizzying process jumping between different levels of the parsing process to resume parsing pieces of the expression left unrealized.

2.7 Symbolic to Symbolic

The laws of "regular" algebra are used to discover relationships between expressions, and to manipulate expressions into more convenient forms. Boolean Algebra has a set of laws which can be employed to manipulate symbolic expressions. The nine laws of Boolean Algebra are listed in Table 2.2.

Law	Primary	Dual
1.	x+0=x	x*1=x
2.	x+1=1	x*0=0
3.	x+x=x	x*x=x
4.	x''=x	
5.	x+x'=1	x*x'=0
6.	x+y=y+x	x*y=y*x
7.	x+(y+z)=(x+y)+z	x*(y*z)=(x*y)*z
8.	x*(y+z)=x*y+x*z	x+(y*z)=(x+y)*(x+z)
9.	(x+y)'=x'*y'	(x*y)'=x'+y'

Table 2.2: The laws of Boolean Algebra using the Boolean variables x, y, x.

Notice that all laws (except Number 4) have a dual. A dual of a symbolic expression is formed by swapping all ANDs and ORs and all 0s and 1s. Boolean Algebra expresses the *duality principle*: If a statement S is true, then the dual of S is also true. Later, it will be convenient to identify which law is used to manipulate a symbolic expression. In these cases, write down the law's number followed by D if it is a dual. For example, if the property $x * x = x$ is used to manipulate a symbolic expression, then indicate that Law 3D was used to manipulate the symbolic expression.

The laws in Table 2.2 can be verified as true by plugging in every possible combination of bits and checking if the two sides of the equation are equal. For example, to check Law 3D, set $x = 0$, yielding $0 * 0 = 0$ which is true, and then set $x = 1$ and verify that $1 * 1 = 1$.

The laws of Boolean Algebra can be used to prove two symbolic expressions are equal. This proof is performed by manipulating one of the expressions until it exactly equals the other side. For example, to prove that $A + A'B' = A + B'$, transform the more complex expression into the simpler expression. In the example, $A + A'B'$ is the more complex expression and $A + B'$ is the simpler expression. At each step of the transformation provide the law used to yield the next step.

$A + A'B' =$ Law 8D
$(A + A') * (A + B') =$ Law 5
$1 * (A + B') =$ Law 1D
$A + B'$

Working on these derivations is a little like solving a maze or playing a good game of chess. Both require a player to "look ahead" and see what the implications of a decision will be. Not looking far enough ahead may result in arriving at one of the preceding steps. For example, consider what happened in the following loopy-derivation of $A(A + B) = A$. The correct derivation is shown on the right.

Loopy		Correct	
$A(A + B) =$	Law 8	$A(A + B) =$	Law 8
$AA + AB =$	Law 3D	$A * A + A * B =$	Law 3
$A + AB =$	Law 8D	$A + A * B$	Law 1D
$(A + A)(A + B) =$	Law 3	$A * 1 + A * B$	Law 8
$A(A + B)$	huh?	$A(1 + B)$	Law 2
		$A(1)$	Law 1D
		A	

Process 2.10: Expansion Trick

The *expansion trick* can be used to show that two expressions are equal to one another. To do this we will make the expression on the left-hand side (LHS) of the "=" sign equal to the expression on the right-hand side (RHS).

To lend context to the discussion of the expansion trick process, let's prove that

$$B' + A'(B'C + C) = (AB + BC')'$$

.

Step 1: Convert the LHS and RHS expressions into SOP form. The LHS has a single set of parenthesis, so we distribute the A term to each term inside the parenthesis. We leave the B' term alone because it is already in a (degenerate) SOP form. Thus we are left with the SOP expression

$$B' + A'B'C + A'C$$

The RHS is more difficult because we will have to "distribute" the negation outside the parenthesis using **Law 9** using the following sequence of steps.

$(AB + BC')'$ Law 9
$(AB)'(BC')' =$ Law 9D
$(A' + B')(B' + C'') =$ Law 4
$(A' + B')(B' + C) =$ Law 8
$(A' + B')B' + (A' + B')C =$ Law 8
$A'B' + B'B' + A'C + B'C =$ Law 3
$A'B' + B' + A'C + B'C$

Thus after this step our original proof now looks like:

$$B' + A'B'C + A'C = A'B' + B' + A'C + B'C$$

Step 2: Identify the missing variables for each product term. A couple of definitions are needed to understand the term "missing variable"

Working Set The set of variables in an expression. The working set in our example is A, B, C.

Canonical Product Term A product term that contains all the variables (or their negation) in the working set. For example, the product term $A'B'C$ on the LHS is a canonical product term.

Missing variable The set of variable, that when ANDed to non-complete product term, makes the product term canonical. For example, the missing variable for the product term B' are A and C.

Putting all this together we get:

Term	Missing variable(s)
B'	A and C
$A'B'C$	None
$A'C$	B
$A'B'$	C
B'	A and C
$A'C$	B
$B'C$	A

Step 3: AND the missing variable and its complement to each product term. This step relies on Law 5 and La 1D from Boolean Algebra. Let's see how this work for the product term $A'C$ on the LHS.

Law 1D of Boolean Algebra states that $X * 1 = X$. Since the equality sign, "=" works both ways, we could also say that Law 1D of Boolean Algebra states that $X = X * 1$. In our case we will let X be the term $A'C$. Applying Law 1D, we have $A'C = A'C * 1$.

Law 5 of Boolean Algebra states that $X + X' = 1$. We could also interpret this as saying that $1 = X + X'$ In this case we will let X be the missing variable B. So we have $1 = B + B'$.

Combining these two ideas together, we get that $A'C = 1 * A'C = (B + B')A'C = A'BC + A'B'C$ The table below shows how each term is converted when it's missing variable(s) are inserted.

Term	Missing variable(s)	Add missing variable	Expand expression
B'	A and C	$B'(A + A')(C + C')$	$AB'C + AB'C' + A'B'C + A'B'C'$
$A'B'C$	None		
$A'C$	B	$A'C(B + B')$	$A'BC + A'B'C$
$A'B'$	C	$A'B'(C + C')$	$A'B'C + A'B'C'$
B'	A and C	$B'(A + A')(C + C')$	$AB'C + AB'C' + A'B'C + A'B'C'$
$A'C$	B	$A'C(B + B')$	$A'BC + A'B'C$
$B'C$	A	$B'C(A + A')$	$AB'C + A'B'C$

$$AB'C + AB'C' + A'B'C + A'B'C' + A'B'C + A'BC + A'B'C =$$

$$A'B'C + A'B'C' + AB'C + AB'C' + A'B'C + A'B'C'A'BC + A'B'C + AB'C + A'B'C$$

Step 4: Remove redundant product terms from each side. This step relies on Law 3 of Boolean Algebra which states that $X + X = X$. In our case, X will be a product term that is repeated on the LHS or RHS. For example the product term $A'B'C$ is repeated on the LHS as the 3rd and 5th terms. We could rearrange the terms on the LHS to look like $A'B'C + A'B'C$. Law 3 tells us that $A'B'C + A'B'C = A'B'C$. This may be unintuitive because in regular algebra $x + x = 2x$. But then again, in regular algebra "+" means addition not logical OR and regular algebra has the symbol 2 while Boolean Algebra has 0 and 1. Using Law 3 on both the LHS and RHS sides yields:

$$AB'C + AB'C' + A'B'C + A'B'C' + A'BC = A'B'C + A'B'C' + AB'C + AB'C' + A'BC$$

Step 5: Rearrange the terms so that LHS and RHS have the same order. This step relies on Law 6 of Boolean algebra which states that $X + Y = Y + X$. In our context X and Y are the term in the LHS and RHS. It is a good idea to leave the terms on the LHS alone and move the terms on the RHS to get the same order. This let's you check that you have in fact the same terms on both side and have not made an error in your derivation.

$$AB'C + AB'C' + A'B'C + A'B'C' + A'BC = AB'C + AB'C' + A'B'C + A'B'C' + A'BC$$

Step 6: Write out the proof. A formal proof requires us to manipulate one side of the equality into the other using the proper laws of Boolean algebra. The wqay that we will accomplish this is to transform the LHS into

$$AB'C + AB'C' + A'B'C + A'B'C' + A'BC$$

using the first 5 steps of this process. You must then continue the derrivation with the RHS by tracing your steps backwards from step 5 to step 1. This process is shown in the following proof that that $B' + A'(B'C + C) = (AB + BC')'$

$B' + A'(B'C + C) =$	Law 1D
$B' + A'B'C + A'C =$	Law 3
$B' * 1 * 1 + A'B'C + A'1C =$	Law 3
$(A + A')B'(C + C') + A'B'C + A'(B + B') =$	Law 3
$AB'C + AB'C' + A'B'C + A'B'C' + A'B'C + A'BC + A'B'C =$	Law 3
$AB'C + AB'C' + A'B'C + A'B'C' + A'BC =$	Law 3
$A'B'C + A'B'C' + AB'C + AB'C' + A'BC =$	Law 3
$A'B'C + A'B'C' + AB'C + AB'C' + A'B'C + A'B'C' + A'BC +$	
$A'B'C + AB'C + A'B'C =$	Law 3
$A'B'(C + C') + B'(A + A')(C + C') + A'C(B + B') + B'C(A + A') =$	Law 3
$A'B' + B' + A'C + B'C =$	Law 3
$A'B' + B'B' + A'C + B'C =$	Law 3
$(A' + B')(B' + C) =$	Law 3
$(AB + BC')'$	Q.E.D.

Since both expressions consist of product terms, Law 8 need not be applied. The second step of the expansion trick, adding the missing variable occurs when $B'C$ is expanded into $A'B'C + AB'C$. In this case, the $B'C$ product term is "missing" the A variable. It is included by first ANDing 1 with $B'C$, then replacing 1 with $(A + A')$, and finally distributing $B'C$.

The expansion trick runs into problems when there are a large number of terms in parenthesis. With a large number of terms in parenthesis, the distribution process becomes tedious and error prone. The following derivation utilizes Laws 4 and 9 to avoid this problem.

Show: $(A + B)(A' + B)(B + C) = B$

Derivation:

$(A + B)(A' + B)(B + C) =$	Law 4
$[(A + B)(A' + B)(B + C)]'' =$	Law 9D
$[(A + B)' + (A' + B)' + (B + C)']' =$	Law 9
$[A'B' + AB' + B'C']' =$	Law 8
$[B'(A' + A) + B'C']' =$	Law 5
$[[B'1 + B'C']' =$	Law 1D
$[B'(1 + C')]' =$	Law 2
$[B'1]' =$	Law 1
$[B']' =$	Law 4
B	

The first step in the derivation, double negating the expression, establishes the conditions for the application of Law 9D. One of the negations is pulled into the bracketed expression converting the product into the OR of each product term, negated. As shown, Law 9 can be applied to an expression with more than two terms. In the next step, Law 9 is applied to each term in parenthesis. What remains is a set of product terms inside the brackets. Over the next five steps this product term is simplified. In the final step, the second negation introduced back in the first step of the proof is combined with the B' inside the brackets, yielding the desired result.

2.8 Truth Table to Symbolic

The most technical transformation in the design process is transforming a truth table to symbolic expression. The idea behind this conversion is captured by the following analogy. Imagine eight members of the college cheer squad, each have a number between $0 \ldots 7$ pinned to their uniform, are gathered together. Whenever members of the squad hears the number pinned to their uniform they give a cheer. If a member hears any other number, they do nothing, even if someone else is cheering like crazy.

In order to get a cheer whenever 1, 4, 6, or 7 was announced over the public address system, which members of the squad should be selected? Clearly, four members of the squad would be selected, the member with number 1, the member with number 4, the member with number 6, and the member with number 7.

The squad members in this analogy are circuit elements called *minterms*. A minterm for a function is a Boolean expression evaluating to logic 1 for a single combination of the inputs and evaluates to logic 0 for every other input. A cheering squad member corresponds to a minterm with an output of 1. The number pinned to each squad member corresponds to the binary input which causes the minterm to output 1. In order to build a symbolic expression for a truth table, first select the minterms corresponding to the inputs where the function equals 1. Next, OR together these minterms and label the output of the OR gate with the function. In the cheer squad analogy, the ability of sound from any member of the cheer squad to be heard corresponds to the OR gates.

Minterms Up till now minterms have been abstract entities, albeit with well-defined properties. To change that, consider all the minterms for a function with three input variables

A, B, C. Since F has eight different inputs, it will have eight minterms. Next, build the minterm that "recognizes" the input $A = B = C = 1$. This task is accomplished by creating an expression which outputs 1 only when $A = B = C = 1$. In this case, the minterm is ABC because it equals 1 when all the inputs are 1, and for any other input, a 0. This minterm is denoted m_7, the lowercase m signifies that this is a minterm and the 7 denotes the decimal representation of the input which causes this minterm to equal 1.

The minterm for the input $A = 0, B = 1, C = 0$, m_2, is $A'BC'$. Negating the A and C variables allows a 0 input value for these variables to be inverted to 1 before being ANDed.

In order to make generating the remaining six minterms easier, formalize the observation just made and call it the *minterm trick*.   A minterm for a particular input needs to include all the variables. If a variable's value is equal to 0, the variable should be negated in the AND expression, if its value is 1, it appears as itself in the AND expression. The table below shows all the minterms for three variables A,B,C.

A	B	C	minterm	symbol
0	0	0	A'B'C'	m_0
0	0	1	A'B'C	m_1
0	1	0	A'BC'	m_2
0	1	1	A'BC	m_3
1	0	0	AB'C'	m_4
1	0	1	AB'C	m_5
1	1	0	ABC'	m_6
1	1	1	ABC	m_7

A minterm is said to "recognizes its input" when that input causes the minterm to evaluate to 1. Thus, minterm m_3 recognizes the input $(A, B, C) = (0, 1, 1)$ and ignores all other inputs. In order to construct the symbolic expression for a truth table, OR together the minterms for which the function equals 1. Thus, when any one of the minterms recognizes its input, one of the inputs to the OR gate will equal 1 causing the output of the OR gate to become 1. Note, that at most one of the OR gates inputs will equal 1, since only one of the minterms will recognize its input. If none of the minterms recognizes the input, all the inputs of the OR gate will equal 0, causing the output of the OR gate to equal 0.

Process 2.12: Truth Table to Symbolic

To lend context to the discussion of the conversion process, let's describe this process by converting the truth table shown below into a symbolic expression.

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Step 1: Write out the minterms for each input. While you only <u>need</u> to write out the minterms for the product terms where the output equals 1, it does us no harm in writing out the minterms for all the rows of the truth table as shown in the truth table

below.

A	B	C	F(A,B,C)	minterm
0	0	0	0	A'B'C'
0	0	1	1	A'B'C
0	1	0	1	A'BC'
0	1	1	1	A'BC
1	0	0	1	AB'C'
1	0	1	0	AB'C
1	1	0	0	ABC'
1	1	1	1	ABC

Step 2: OR together the minterms where the function equals 1. There are five inputs for which $F(A, B, C)$ equals 1, so the symbolic expression for F will include the five minterms from the rows where F equals 1. Hence,

$$F(A, B, C) = A'B'C + A'BC' + A'BC + AB'C' + ABC$$

Now, examine what happens to F for the input $(A, B, C) = (0, 0, 1)$. The minterm $m_1 = A'B'C$ evaluates to 1 while all the other minterms evaluate to 0. Since the OR function evaluates to 1 when any of the inputs are equal 1, F equals to 1. Hence, $F(0, 0, 1) = 1$ as expected. When $(A, B, C) = (1, 1, 0)$ is applied, all the minterms in the symbolic expression for F evaluate to 0 because none of them recognize this input. Thus, $F(1, 1, 0) = 0$ as expected.

When a function is constructed by ORing together minterms, it is said to be in *canonical sum of products* form or canonical SOP form. A symbolic expression is said to be in SOP form when it consists exclusively of a collection of product terms ORed together. The term sum/product is used to refer to the operators OR/AND respectively because these are what the operators look like in normal arithmetic. The term canonical is used because this symbolic form is the most elementary form possible to describe the function.

The canonical SOP representation for F defined above can be abbreviated as $F(A, B, C) = \sum m(1, 2, 3, 4, 7)$ where the $\sum$ symbol denotes the ORing together of minterms 1,2,3,4 and 7. In general, $\sum m(list)$ describes the inputs for which the function equals 1. When asked for a canonical SOP expression in the homework, write either the symbolic expression out, or use this abbreviated form.

Engineers are always striving to minimize the cost of their solutions. Towards this end, a second way to transform a truth table into a symbolic expression is introduced. It replaces the minterms with a different kind of building block, the maxterm.

Maxterms A *maxterm* for a function is a symbolic expression evaluating to 0 for a single combination of the inputs and evaluates to 1 for every other input. Before constructing symbolic expressions for a truth table, it is necessary to examine the structure of maxterms.

Since the function $F(A, B, C)$ has three inputs, it will have eight maxterms. Start by creating the maxterm for the input $(A, B, C) = (0, 0, 0)$. Many different expressions can be derived which equal 0 only for this input, but the simplest is $A + B + C$. This maxterm is abbreviated M_0, the uppercase M signifying a maxterm and the 0 denoting the decimal representation of the input which causes this maxterm to equal 0. The *maxterm trick* is used to simplify the process of creating maxterms.

A maxterm for a particular input needs to include all the variables. If a variable's value is equal to 1, the variable should be negated in the OR expression; if its value is 0, the expression

appears as itself in the OR expression. The table below lists all the maxterms for the three variables A, B, C.

A	B	C	maxterm	symbol
0	0	0	A+B+C	M_0
0	0	1	A+B+C'	M_1
0	1	0	A+B'+C	M_2
0	1	1	A+B'+C'	M_3
1	0	0	A'+B+C	M_4
1	0	1	A'+B+C'	M_5
1	1	0	A'+B'+C	M_6
1	1	1	A'+B'+C'	M_7

In order to construct the symbolic expression for a truth table, AND together the maxterms for which the function equals 0. Thus, when any one of the maxterms recognizes its input, one of the inputs to the AND gate will equal 0, causing the output of the AND gate to go to 1. Note, that at most one of the AND gate's inputs will equal 0, since only one of the maxterms will recognize the input. If none of the maxterms recognizes the input, all the inputs of the AND gate equal 1, causing the output of the AND gate to equal 1. These ideas are applied to the function $F(A, B, C)$ below.

A	B	C	F(A,B,C)	maxterm	symbol
0	0	0	0	A+B+C	M_0
0	0	1	1	A+B+C'	M_1
0	1	0	1	A+B'+C	M_2
0	1	1	1	A+B'+C'	M_3
1	0	0	1	A'+B+C	M_4
1	0	1	0	A'+B+C'	M_5
1	1	0	0	A'+B'+C	M_6
1	1	1	1	A'+B'+C'	M_7

There are three inputs of $F(A, B, C)$ equal to 0, so the symbolic expression for F includes the three maxterms from the rows where F equals 0. Hence,

$$F(A, B, C) = (A + B + C)(A' + B + C')(A' + B' + C)$$

What happens when the input $(A, B, C) = (1, 0, 1)$ is applied? The maxterm $M_5 = (A' + B + C')$ evaluates to 0 while all the other maxterms evaluate to 1. Since the AND function evaluates to 0 when any of the inputs are equal 0, F equals to 0. Hence, $F(1, 0, 1) = 0$ as expected. When $(A, B, C) = (1, 1, 1)$ is applied, all the maxterms in the symbolic expression for F evaluate to 1 because none of them recognize this input. Thus $F(1, 1, 1) = 1$ as expected.

This representation of F is called its *canonical product of sums* form or canonical POS form, because F is represented in its most elementary form as the AND (product) of a collection of OR (sum) terms. The canonical POS representation in the example can be abbreviated by the expression $F(A, B, C) = \prod M(0, 5, 6)$ where the $\prod$ symbol denotes the AND, the M symbol stands for maxterm, and the numbers in the list are the inputs for which $F(A, B, C)$ equals 0.

Notice, the functions in the two minterm and maxterm examples are the same. This was done to illustrate two points. First, the inputs present in the minterm list are those that do not appear in the maxterm list. In other words, $F(A, B, C) = \sum m(1, 2, 3, 4, 7) = \prod M(0, 5, 6)$. This should make sense. The minterm list denotes those inputs for which the function equals 1. If the function does not equal 1 for a particular input, this input is not in the minterm list, and then the function must equal 0 for this input, hence this input will appear in the maxterm list.

The second reason for using the same function is to illustrate how different realizations of the same function have different costs. A solution requiring fewer gates will cost less. The following table summarizes the number of gates required in the SOP and POS realization of $F(A, B, C)$.

	SOP	POS
number of NOTs	3	3
number of ANDs	5	1
number of ORs	1	3

Assuming that AND gates and OR gates have the same cost then the POS solution costs less. Thus a good engineer would recommend a canonical POS realization over a canonical SOP realization.

While, the transformation from a truth table to a symbolic expression may be the most technically challenging, it is certainly not the most difficult. This title belongs to the transformation of a word statement into a truth table. The difficulty of this transformation has its origin in the puffy cloud outline of a word statement shown in Figures **??** 2.2. Words can be ambiguous and the ability of authors to convey precisely what the digital system needs to accomplish depends on their skill with the English language.

2.9 Word Statement to Truth Table

There is no algorithmic approach guaranteed to transform a word statement into a truth table. That said, the following list outlines essential information to obtain from the word statement in order to have any hope of transforming it into a truth table.

> **Process 2.14: Word Statement to Truth Table**
>
> To lend context to the discussion of the conversion process, let's describe this process by converting the following word statement into a truth table.
>
> > Determine the truth table for a function with three bits of input and one bit of output. The output of the function should equal 1 when the 3-bit binary input represents a binary number greater than 4.
>
> **Step 1: Identify the inputs and outputs and their meaning**
> The function has three bits of input and one bit of output. Since the inputs and outputs were not given names, use the default labels a_2, a_1, a_0 for the inputs and F for the output. The input variables are given subscripts because they are working together to form a 3-bit binary number. The binary number is referred to as A and its individual bits as a_2, a_1, and a_0. Hence, when $(a_2, a_1, a_0) = (1, 0, 1)$ then $A = 5$.
> **Step 2: Write out the skeleton of the truth table.** Many 3-variable truth tables have been shown in this chapter. The notable point in this example is that the output is abbreviated as $F(A)$ to save space.

a_2	a_1	a_0	$F(a_2, a_1, a_0)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Step 3: Fill out the truth table. This example requires us to interpret all the inputs bits together as a single value. For example, we will look at the input $(a, b, c) = (1, 1, 0)$ not as three separate bits but as the binary number 110_2 which is equal to 6_{10}. With this in mind, let's try filling up the truth table.

Since the output should equals 0 for inputs less than or equal to $4 = 100_2$, we should list the output for inputs $(0, 0, 0)$ to $(1, 0, 0)$ with 0. Likewise we should fill in the output with 1 for inputs $(1, 0, 1)$ to $(1, 1, 1)$ as shown in the following table.

a_2	a_1	a_0	$F(a_2, a_1, a_0)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Before moving on, let's consider one final problem with two twists; the truth table will describe a function with four variables truth table has three bits of output. For the sake of brevity, the second and the third steps of the transformation process are combined.

Determine the truth table for a digital system having two, 2-bit inputs $A = a_1 a_0$ and $B = b_1 b_0$. Each 2-bit input represents a 2-bit binary number. The function has three bits of output called G, L, and E, which stand for greater, less, and equal, respectively. E equals 1 when $A = B$, $G = 1$ when $A > B$, and $L = 1$ when $A < B$.

Step 1 The function has four bits of input and three bits of output. The names of these variables are clear from the word statement.

Step 2 & 3 The truth table for this function has 16 rows because there are $2^4 = 16$ combinations of four bits (see page). In order to better visualize the solution to the problem, the values of A and B (derived from their values of a_1, a_0 and b_1, b_0) are included in the truth table.

a_1	a_0	b_1	b_0	A	B	G	L	E
0	0	0	0	0	0	0	0	1
0	0	0	1	0	1	0	1	0
0	0	1	0	0	2	0	1	0
0	0	1	1	0	3	0	1	0
0	1	0	0	1	0	1	0	0
0	1	0	1	1	1	0	0	1
0	1	1	0	1	2	0	1	0
0	1	1	1	1	3	0	1	0
1	0	0	0	2	0	1	0	0
1	0	0	1	2	1	1	0	0
1	0	1	0	2	2	0	0	1
1	0	1	1	2	3	0	1	0
1	1	0	0	3	0	1	0	0
1	1	0	1	3	1	1	0	0
1	1	1	0	3	2	1	0	0
1	1	1	1	3	3	0	0	1

In order to transform this truth table into a symbolic expression, realize each of the three outputs are independent of the others. For example, to determine the symbolic expression for G, cover-up the output columns corresponding to the L and E outputs and implement G. In order to realize L, just cover-up the columns associated with G and E and implement L as if the other two outputs did not exist. This yields the following three equations for the outputs.

$$G(a_1, a_0, b_1, b_0) = \sum m(4, 8, 9, 12, 13, 14)$$
$$L(a_1, a_0, b_1, b_0) = \sum m(1, 2, 3, 6, 7, 11)$$
$$E(a_1, a_0, b_1, b_0) = \sum m(0, 5, 10, 15)$$

Of course this leads to the question of how to build the circuit diagram for this 4-input, 3-output function. Students often want to OR together the G, L, and E outputs in order to have a single output, but this is would be incorrect because the word statement asked for three separate outputs. The correct way to build the circuit diagram is to build each of the circuits separately, give each its own output, and have them share the same inputs. A rough sketch of the circuit is shown in Figure 2.7.

Notice that the three output circuits share the same inputs, but otherwise are independent of one another. Sharing inputs allows each circuit to coordinate its output with the other circuits. When each circuit does what it is supposed to do for that input, the output looks like a unified whole even though it is generated from three separate circuits.

2.10 Timing Diagrams

After a circuit has been realized in hardware, the logical representation of 0s and 1s is replaced by physical voltages representing these logic levels. In order to observe the behavior of a physical digital system, an engineer typically uses a logic analyzer or oscilloscope. The waveforms drawn by the logic analyzer when applying inputs and examining the outputs is called a *timing diagram*. A timing diagram is a graph of the logic value of a signal versus time. Typically, several signals are combined on the same timing diagram to save space and in order to clearly show the relationship between the signals.
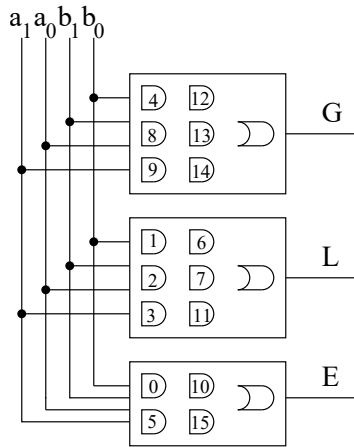
Figure 2.7: The circuit diagram for a circuit which compares the magnitude of the two inputs $A = a_1 a_0$ and $B = b_1 b_0$. Many of the internal details have been omitted.

It is imperative to check if the digital system implemented behaves according to its specification. Hence, an understanding of timing diagrams is essential. With a small circuit it is reasonable to apply every possible combination of inputs and examine the output of the circuit. For example, Figure 2.8 shows the timing diagram for $F(A, B, C) = A'B + AB'C$.
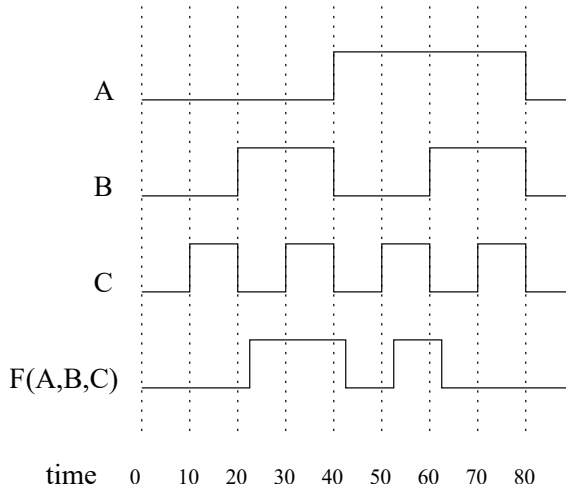
Figure 2.8: A timing diagram for $F(A, B, C) = A'B + AB'C$.

Observe that the time intervals are spaced 10 units apart. For now, the meaning of the units is unimportant. Most likely, they would be nanoseconds. In practice, the timing behavior of a circuit may be very important because circuits must not only have the correct output, but the circuit must meet strict timing constraints.

At time=5 the input $(A, B, C) = (0, 0, 0)$ and the output in response to this input is 0. At time=15 the input is $(0, 0, 1)$, and the output due to this input is 0. At time=75 the input is $(1, 1, 1)$ and the output is 0. Every combination of the inputs has been applied to the circuit allowing the test engineer to verify the functional behavior of the circuit.

2.11   Exercises

1. **(2 pts. each)** Given: $F(A, B, C, D) = (AB' + (C + (AD)')(BD))'$

   a) Determine the truth table for $F(A, B, C, D)$

   b) Draw a schematic of the logic circuit which realizes F as shown, i.e. do not use Boolean Algebra on F.

2. **(2 pts. each)** For the circuit in Figure 2.9

   a) Write a Boolean expression for the function.
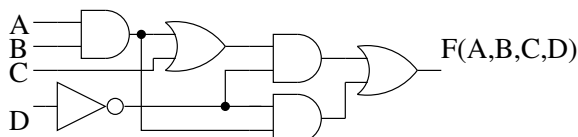
   b) Draw the truth table for the function.

Figure 2.9: The circuit for Problems 2 and 3.

3. **(2 pts. each)** For the functions F,G,H,I defined by the truth table shown below:

   a) Determine the canonical SOP and POS realization for F, G, H, I.

   b) Draw the circuit diagram for the canonical SOP and POS realization.

   Treat each output independently of the other. For example when working with function I, cover up the columns F, G and H.

 | A | B | C | F | G | H | I |
   |-----|-----|-----|-----|-----|-----|-----|
   | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
   | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
   | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
   | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
   | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
   | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
   | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
   | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

4. **(2 pts. each)** Prove the validity of the following statements using the laws of Boolean Algebra. For each step of the proof, identify which law was used.

   a) $X'Y' + XY + X'Y = X' + Y$

   b) $(X + Y')X'Y' = X'Y'$

   c) $(X + Y)(X' + Z) = XZ + X'Y$

   d) $X'Y' + (X + Y)Z = X'Y' + Z$

   e) $A'C + BC + AB = A'C + AB$

   f) $A(B + C) = AB + AB'C$

   g) $(A + B + C)(A + B + C')(A' + B + C')(A' + B' + C') = (A + B)(A' + C')$

5. **(4 pts.)** Design a circuit called MUX2. MUX2 has three bits of input S, y_0, y_1 and one bit of output F. If $S = 0$, then $F = y_0$; else if $S = 1$, then $F = y_1$.

   a) Write down the truth table for the MUX2 function.

   b) Determine the canonical SOP realization for MUX2; do not simplify.

6. **(6 pts.)** Design a circuit called MUX4. MUX4 has six bits of input $S_1 S_0, y_0, y_1, y_2, y_3$ and one bit of output F.
   If $S_1 S_0 = 00$ then $F = y_0$
   else if $S_1 S_0 = 01$ then $F = y_1$
   else if $S_1 S_0 = 10$ then $F = y_2$
   else if $S_1 S_0 = 11$ then $F = y_3$
   Without writing down the truth table determine a SOP expression to realize F by listing all possible inputs which will cause F to equal 1. Then try to simplify your expression using Boolean Algebra.

7. **(4 pts.)** Design a logic circuit called MAJ which has three inputs A, B, C and one output Z. The output equals 1 when a majority of the inputs are equal to 1, otherwise the output is 0.

   a) Write the truth table for the MAJ function.

   b) Determine the canonical SOP realization for the MAJ function, do not simplify.

8. **(4 pts.)** Let X and Y each be 2-bit signals whose elements are $x_1 x_0$ and $y_1 y_0$ respectively. Determine the $\sum m$ and $\prod M$ expression for a circuit whose 1-bit output z is defined by the following statement.

   ```
   if (X == Y) then z = 1 else z =0
   ```

9. **(4 pts.)** Let X and Y each be 2-bit signals whose elements are $x_1 x_0$ and $y_1 y_0$, respectively. Determine the $\sum m$ and $\prod M$ expressions for a circuit whose 1-bit output z is defined by the following statement.

   ```
   if (X + Y > 3) then z = 0 else z =1
   ```

10. **(3 pts.)** Determine the canonical SOP and POS expression for $F(A, B, C) = \prod M(0, 1, 4, 5)$ Hint, compose the truth table for F.

11. **(3 pts.)** Determine the canonical SOP and POS expression for $F(A, B, C, D) = \sum m(0, 4, 12, 15)$ Hint, write out the truth table for F.

12. **(4 pts.)** For the function $F(A, B, C) = BC + AB'C'$, draw a timing diagram for an input sequence that follows the same order as the rows of the truth table. Assume a propagation delay for NOT, AND and OR gate are all 10nS.

13. **(4 pts.)** Complete the timing diagram in Figure 2.10 for the functions $F(A, B, C) = AB' + BC + ABC'$ and $G(A, B, C) = (A + B')C + (BC')'$

14. **(16 pts.)** Design a circuit to control the water pump of a washing machine. The pump will not pump water if

    The lid is closed and the cycle is not fill
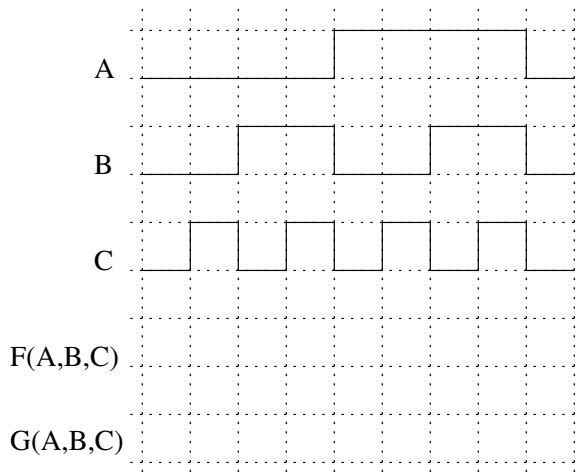
Figure 2.10: The timing diagram for two functions, F and G.

The cycle is fill and the detergent level is empty

The detergent is not empty and the lid is open

The variables for this problem are:

L = lid is closed

C = cycle is fill

D = detergent is empty

P = pump will pump water

Create a truth table which describes when the pump will not pump water. Call this output P'. Determine the canonical SOP expression for P'. Use this canonical SOP expression to generate a circuit diagram for P. This can be done by inserting an inverter onto the output of the circuit.

Take the P' column from truth table and invert all the entries to generate a new output column called P (because the negation of P' is P). Determine the canonical SOP realization for P using this new column.

Appendix A

74LS00 Data Sheets

Since the device documentation for the TI chips is covered by TI's copyright it was decided to leave these pages out of this text. The documents discussed in the text can be found onilne at: http://focus.ti.com/lit/ds/symlink/sn74ls00.pdf

Index