INTRODUCTION TO LOGIC CIRCUITS

By

Brock J. LaMeres, Ph.D.
Montana State University – Bozeman
Rev 08.26.13

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction – Analog vs. Digital

We often hear that we live in a digital age. This refers to the massive adoption of computer systems within every aspect of our lives from smart phones to automobiles to household appliances. This statement also refers to the transformation that has occurred to our telecommunications infrastructure that now transmits voice, video and data using 1's and 0's. There are a variety of reasons that digital systems have become so prevalent in our lives. In order to understand these reasons, it is good to start with an understanding of what a digital system is and how it compares to its counterpart, the analog system.

First, let's talk about signaling. In electrical systems, signals represent information that is transmitted between devices using an electrical quantity (voltage or current). An analog signal is defined as a continuous, time varying quantity that corresponds directly to the information it represents. An example of this would be a barometric pressure sensor that outputs an electrical voltage corresponding to the pressure being measured. As the pressure goes up, so does the voltage. While the range of the input (pressure) and output (voltage) will have different spans, there is a direct mapping between the pressure and voltage. Another example would be sound striking a traditional analog microphone. Sound is a pressure wave that travels through a medium such as air. As it strikes the diaphragm in the microphone, it moves it back and forth. Through the process of inductive coupling, this movement is converted to an electric current. The characteristics of the current signal produced (e.g., frequency and magnitude) correspond directly to the characteristics of the incoming sound wave. The current can travel down a wire and go through another system that works in the opposite manner by inductively coupling the current onto another diaphragm, which in turn moves back and forth thus forming a pressure wave and ultimately produces sound (e.g., a speaker). In both of these examples, the electrical signal represents the *actual* information that is being transmitted and is considered *analog*. Analog signals can be resented mathematically as a function with respect to time.

In digital signaling, the electrical signal itself is not directly the information it represents. Instead, the information is encoded. The most common type of encoding is binary (1's and 0's). The 1's and 0's are represented by the electrical signal. The simplest form of digital signaling is to define a threshold voltage directly in the middle of the range of the electrical signal. If the signal is above this threshold, the signal is representing a 1. If the signal is below this threshold, the signal is representing a 0. This type of signaling is not considered continuous as in analog signaling. Instead, it is considered to be *discrete* because the information is transmitted as a series of distinct values. The signal transitions between a 1 to 0 or 0 to 1 are assumed to occur instantaneously. While this is obviously impossible, for the purposes of information transmission, the values can be interpreted as a series of discrete values. This is a *digital* signal and is not the actual information, but rather the binary encoded representation of the original information. Digital signals are not represented using traditional mathematical functions. Instead the digital values are typically held in tables of 1's and 0's.

Figure 1.1 shows an example analog signal (left) and an example digital signal (right). While the digital signal is in reality continuous, it represents a series of discrete 1 and 0 values.

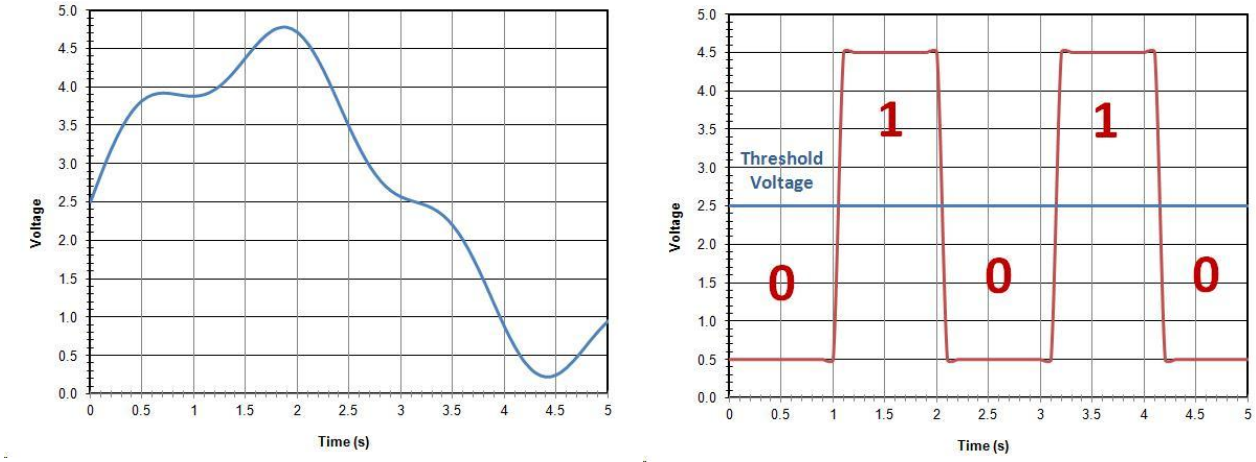


Figure 1.1
Example Analog (left) and Digital (right) Signals

There are a variety of reasons that digital systems are preferred over analog ones. First is their ability to operate within the presence of noise. Since an analog signal is a direct representation of the physical quantity it is transmitting, any noise that is coupled onto the electrical signal is interpreted as noise on the original physical quantity. An example of this is when you are listening to an AM/FM radio and you hear distortion of the sound coming out of the speaker. The distortion you hear is not due to actual distortion of the music as it was played at the radio station, but rather electrical noise that was coupled onto the analog signal transmitted to your radio prior to being converted back into sound by the speakers. Since the signal in this case is analog, the speaker simply converts it in its entirety (noise + music) into sound. In the case of digital signaling, a significant amount of noise can be added to the signal while still preserving the original 1's and 0's that are being transmitted. For example, if the signal is representing a 0, the receiver will still interpret the signal as a 0 as long as the noise doesn't cause the level to exceed the threshold. Once the receiver interprets the signal as a 0, it stores the encoded value as a 0 thus ignoring any noise present during the original transmission. **Figure** 1.2 shows the exact same noise added to the analog and digital signals from **Figure** 1.1. The analog signal is distorted; however, the digital signal is still able to transmit the 0's and 1's that represent the information.

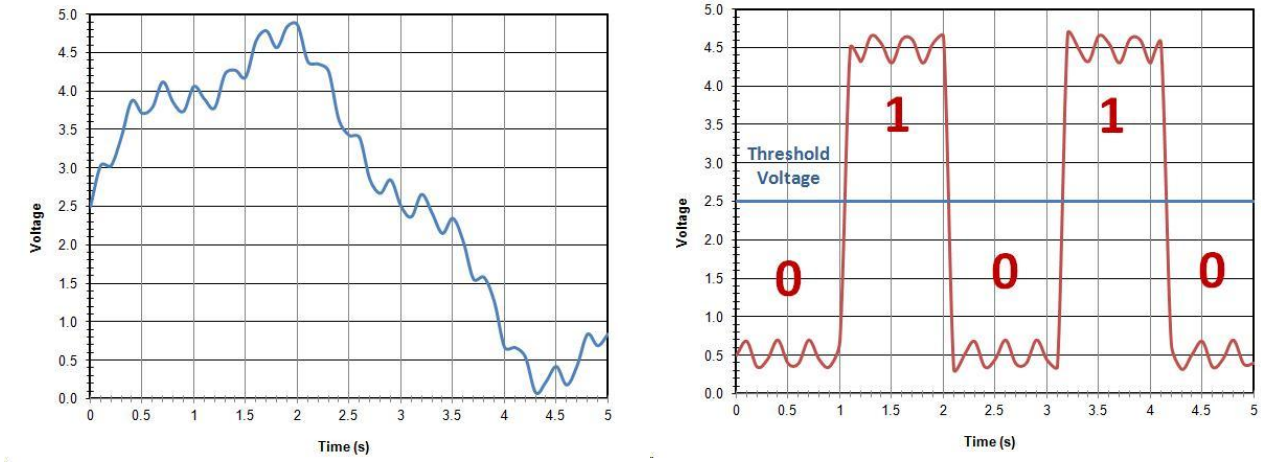


Figure 1.2
Noise on Analog (left) and Digital (right) Signals

Another reason that digital systems are preferred over analog ones is the simplicity of the circuitry. In order to produce a 1 and 0, you simply need an electrical switch. If the switch connects the output to a voltage below the threshold, then it produces a 0. If the switch connects the output to a voltage above the threshold, then it produces a 1. It is relatively simple to create such a switching circuit using modern transistors. Analog circuitry however needs to perform the conversion of the physical quantity it is representing (e.g., pressure, sound) into an electrical signal all the while maintaining a direct correspondence between the input and output. This type of circuit design is much more complicated and also extremely sensitive to operating conditions such as power supply variation, fabrication tolerances and temperature extremes. **Figure** 1.3 shows an analog inverting amplifier and a digital inverter. The analog amplifier uses dozens of transistors (inside the triangle) and two resistors to perform the inversion of the input. The digital inverter uses two transistors that act as switches to perform the inversion.

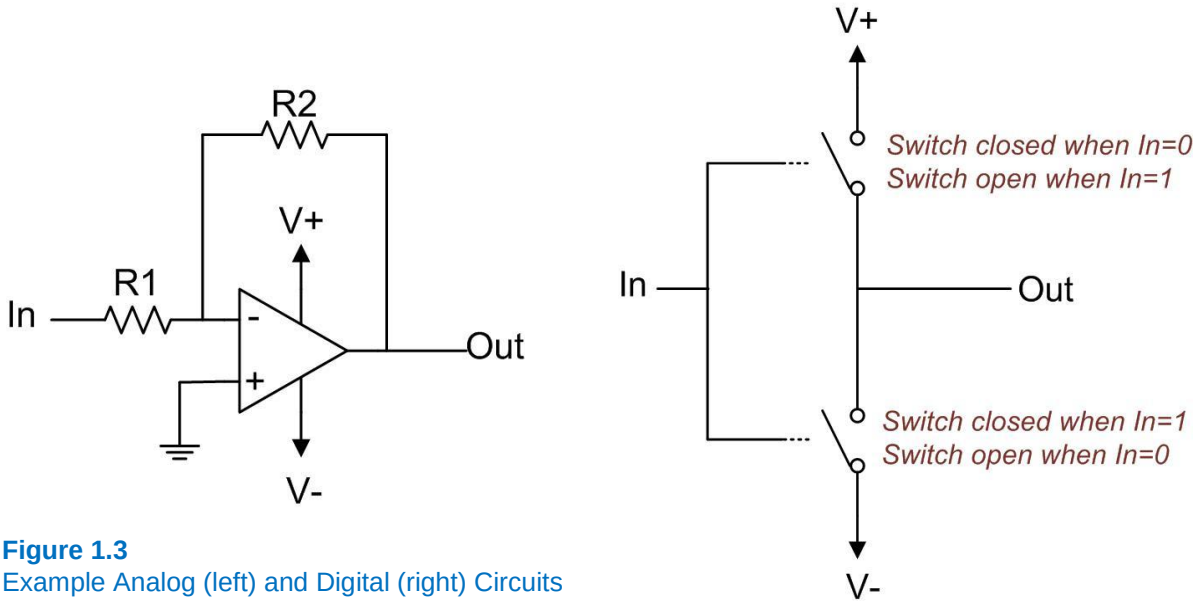


Figure 1.3
Example Analog (left) and Digital (right) Circuits

A final reason that digital systems have become so widely adopted is their reduced power consumption. With the advent of Complementary Metal Oxide Transistors (CMOS), electrical switches can be created that consume very little power to *turn* on or off and consume relatively negligible amounts of power to *keep* on or off. This has allowed digital circuitry to be fabricated on a large scale without reaching excessive levels of power consumption. For stationary digital systems such as servers and workstations, this means extremely large and complicated systems can be constructed that consume achievable amounts of power. For portable digital systems such as smart phones and tablets, this means sophisticated and useful tools can be designed that are able to run on portable power sources. Analog circuits on the other hand require continuous power in order to accurately convert and transmit the electrical signal representing the physical quantity. Circuit techniques that are required to compensate for variances in power supply, fabrication and temperature require additional power consumption. For these reasons, analog systems are being replaced with digital systems wherever possible to exploit their noise immunity, simplicity and reduced power they provide. While analog systems will always be needed at the transition between the physical (e.g., microphones, camera lenses, sensors, video displays) and the electrical world, it is anticipated the push toward digitization of everything in between (e.g., processing, transmission, storage) will continue.

Exercise Problems

1.1 Give three advantages of using digital systems over analog.

1.2 What part of any system will always require an analog system?

1.3 For the following part of an iPod, indicate whether they are analog or digital:

a) The sound coming out of the ear buds
b) The MP3 music file stored on the iPod
c) The circuitry that reads the MP3 file from memory
d) The electrical signal going down the ear phone wires to the ear buds
e) The voltage coming from the battery to power the iPod
f) The touch display

Chapter 2: Number Systems

Logic circuits are used to generate and transmit 1's and 0's to compute and convey information. This two-valued number system is called *binary*. As presented earlier, there are many advantages of using a binary system; however, the human brain has been taught to count, label and measure using the *decimal* number system. The decimal number system contains 10 unique symbols (0→9), commonly referred to as the *Arabic numerals*. Each of these symbols is assigned a relative magnitude to the other symbols. For example, 0 is less than 1, 1 is less than 2, etc… It is often conjectured that the 10 symbol number system that we humans use is due to the availability of our 10 fingers (or *digits*) to visualize counting up to 10. Regardless, our brains are trained to think of the real world in terms of a decimal system. In order to bridge the gap between the way our brains think (decimal) and how we build our computers (binary), we need to understand the basics of number systems. This includes the formal definition of a positional number system and how it can be extended to accommodate any arbitrarily large (or small) value. This also includes how to convert between different number systems that contain different numbers of symbols. In this chapter, we cover 4 different number systems: decimal (10 symbols), binary (2 symbols), octal (8 symbols) and hexadecimal (16 symbols). The study of decimal and binary is obvious as they represent how our brains interpret the physical world (decimal) and how our computers work (binary). Hexadecimal is studied because it is a useful means to represent large sets of binary values using a manageable number of symbols. Octal is rarely used but is studied as an example of how the formalization of the number systems can be applied to all systems regardless of the number of symbols they contain. This chapter will also discuss how to perform basic arithmetic in the binary number system and represent negative numbers.

2.1 Positional Number Systems

A positional number system allows the expansion of the original set of symbols so that they can be used to represents any arbitrarily large (or small) value. For example, if we use the 10 symbols in our decimal system, we can count from 0 to 9. Using just the individual symbols we do not have enough symbols to count beyond 9. To overcome this, we use the same set of symbols but assign a different value to the symbol based on its position within the number. The *position* of the symbol with respect to other symbols in the number allows an individual symbol to represent greater (or lower) values. We can use this approach to represent numbers larger than the original set of symbols. For example, let's say we want to count from 0 upward by 1. We begin counting 0, 1, 2, 3, 4, 5, 6, 7, 8 to 9. When we are out of symbols and wish to go higher, we bring on a symbol in a different position with that position being valued higher and then start counting over with our original symbols (e.g., …,9, 10, 11, 12,..19, 20, 21, 22,...). This is repeated each time a position runs out of symbols (e.g., …99, 100, 101, 102…999, 1000, 1001, 1002…).

To begin, let's look at the formation of a number system. The first thing that is needed is a set of symbols. The formal term for one of the symbols in a number system is a *numeral***.** One or more numerals are used to form a *number*. We define the number of numerals in the system using the terms *radix* or *base*.

Radix = Base ≡ the number of numerals in the number system

For example, our decimal number system is said to be *base 10*, or have a *radix of 10* because it consists of 10 unique numerals or symbols.

The next thing that is needed is the relative value of each numeral with respect to the other numerals in the set. We can say 0 < 1 < 2 < 3 etc… to define the relative magnitudes of the numerals in this set. The numerals are defined to be greater or less than their neighbors by a magnitude of 1. For example, in the decimal number system each of the subsequent numerals is greater than its predecessor by exactly 1. When we define this relative magnitude, we are defining that the numeral 1 is greater than the numeral 0 by a magnitude of 1, the numeral 2 is greater than the numeral 1 by a magnitude of 1, etc… At this point, we have the ability to count from 0 to 9 by 1's. We also have the basic structure for mathematical operations that have results that fall within the numeral set from 0 to 9 (e.g., 1+2=3). In order to expand the values that these numerals can represent, we need define the rules of a positional number system.

2.1.1 Generic Structure

In order to represent larger or smaller numbers than the lone numerals in a number system can represent, we adopt a positional system. In a positional number system, the relative position of the numeral within the overall number dictates its value. When we begin talking about the position of a numeral, we need to define a location to which all of the numerals are positioned with respect to. We define the *radix point* as the point within a number to which numerals to the left represent whole numbers and numerals to the right represent fractional numbers. The radix point is denoted with a period (e.g., "."). A particular number system often renames this radix point to reflect its base. For example, in the base 10 number system (e.g., decimal), the radix point is commonly called the *decimal point*; however the term *radix point* can be used across all number systems as a generic term. If the radix point is not present in a number, it is assumed to be to the right of number. **Figure** 2.1 shows an example number highlighting the radix point and the relative positions of the whole and fractional numerals.

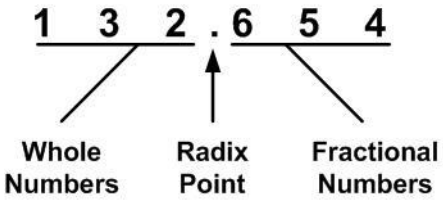


Figure 2.1
Definition of Radix Point

Next, we need to define the position of each numeral with respect to the radix point. The position of the numeral is assigned a whole number with the number to the left of the radix point having a position value of 0. The position number increases by 1 as numerals are added to the left (2, 3, 4….) and decreased by 1 as numerals are added to the right (-1, -2, -3). We will use the variable *p* to represent position. The position number will be used to calculate the value of each numeral in the number based on its relative position to the radix point. **Figure** 2.2 shows the example number with the position value of each numeral highlighted.

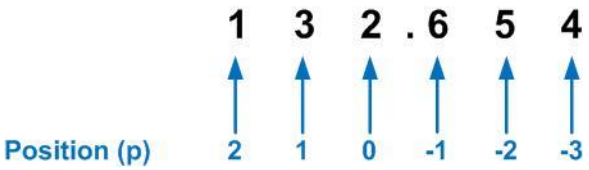


Figure 2.2
Definition of Position Number (p) within the Number

In order to create a generalized format of a number, we assign the term *digit* (d) to each of the numerals in the number. The term digit signifies that the numeral has a position. The position of the digit within the number is denoted as a subscript. The term *digit* can be used as a generic term to describe a numeral across all systems, although some number systems will use a unique term instead of digit which indicates its base. For example, the binary system uses the term *bit* instead of digit; however using the term digit to describe a generic numeral in any system is still acceptable. **Figure** 2.3 shows the generic subscript notation used to describe the position of each digit in the number.

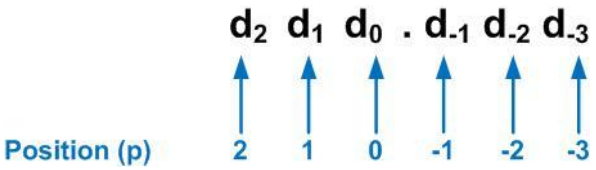


Figure 2.3
Digit Notation

We write a number from left to right starting with the highest position digit that is greater than 0 and end with the lowest position digit that is greater than 0. This reduces the amount of numerals that are written; however, a number can be represented with an arbitrary number of 0's to the left of the highest position digit greater than 0 and an arbitrary number of 0's to the right of the lowest position digit greater than 0 without effecting the value of the number. For example, the number 132.654 could be written as 0132.6540 without affecting the value of the number. The 0's to the left of the number are called *leading 0's* and the 0's to the right of the number are called *trailing 0's*. The reason this is being stated is because when a number is implemented in circuitry, the number of numerals is fixed and each numeral must have a value. The variable *n* is used to represent the number of numerals in a number. If a number is defined with n=4, that means 4 numerals are always used. The number 0 would be represented as 0000 with both representations having an equal value.

2.1.2 Decimal Number System (Base 10)

As mentioned earlier, the decimal number system contains 10 unique numerals (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9). This system is thus a base 10 or a radix 10 system. The relative magnitudes of the symbols are 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9.

2.1.3 Binary Number System (Base 2)

The binary number system contains 2 unique numerals (0 and 1). This system is thus a base 2 or a radix 2 system. The relative magnitudes of the symbols are 0 < 1. At first glance, this system looks very limited in its ability to represent large numbers due to the small number of numerals. When counting up, as soon as you count from 0 to 1, you are out of symbols and must increment the p+1 position in order to represent the next number (e.g., 0, 1, 10, 11, 100, 101, …); however, magnitudes of each position scale quickly so that circuits with a reasonable amount of digits can represent very large numbers. The term *bit* is used instead of *digit* in this system to describe the individual numerals and at the same time indicate the base of the number.

Due to the need for multiple bits to represent meaningful information, there are terms dedicated to describe the number of bits in a group. When 4 bits are grouped together, they are called a **nibble**. When 8 bits are grouped together, they are called a **byte**. Larger groupings of bits are called **words.** The size of the word can be stated as either an *n-bit word* or omitted if the size of the word is inherently

implied. For example, if you were using a 32-bit microprocessor, using the term *word* would be interpreted as a *32-bit word.* For example, if there was a 32 bit grouping, it would be referred to as a 32-bit word. The leftmost bit in a binary number is called the ***Most Significant Bit*** **(MSB)**. The rightmost bit in a binary number is called the **Least Significant Bit (LSB)**.

2.1.4 Octal Number System (Base 8)

The octal number system contains 8 unique numerals (0, 1, 2, 3, 4, 5, 6, 7). This system is thus a base 8 or a radix 8 system. The relative magnitudes of the symbols are 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7. We use the generic term *digit* to describe the numerals within an octal number.

2.1.5 Hexadecimal Number System (Base 16)

The hexadecimal number system contains 16 unique numerals. This system is most often referred to in spoken word as "hex" for short. Since we only have 10 Arabic numerals in our familiar decimal system, we need to use other symbols to represent the remaining 6 numerals. We use the alphabetic characters A-F in order to expand the system to 16 numerals. The 16 numerals in the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The relative magnitudes of the symbols are 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < A < B < C < D < E < F. We use the generic term digit to describe the numerals within a hexadecimal number.

At this point, it becomes necessary to indicate the base of a written number. The number 10 has an entirely different value if it is a decimal number or binary number. In order to handle this, a subscript is typically included at the end of the number to denote its base. For example, 10_{10} indicates that this number is decimal "ten". If the number was written as 10_2, this number would represent binary "one zero". **Error! Reference source not found.** lists the equivalent values in each of the 4 number systems ust described for counts from 0_{10} to 15_{10}. The left side of the table does not include leading 0's. The right side of the table contains the same information but includes the leading zero's. The equivalencies of decimal, binary and hexadecimal in this table are typically committed to memory.

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
0	0	0	0	00	0000	00	0
1	1	1	1	01	0001	01	1
2	10	2	2	02	0010	02	2
3	11	3	3	03	0011	03	3
4	100	4	4	04	0100	04	4
5	101	5	5	05	0101	05	5
6	110	6	6	06	0110	06	6
7	111	7	7	07	0111	07	7
8	1000	10	8	08	1000	10	8
9	1001	11	9	09	1001	11	9
10	1010	12	A	10	1010	12	A
11	1011	13	B	11	1011	13	B
12	1100	14	C	12	1100	14	C
13	1101	15	D	13	1101	15	D
14	1110	16	E	14	1110	16	E
15	1111	17	F	15	1111	17	F
(Without Leading 0's)				(With Leading 0's)			

Table 2.1
Number System Equivalency

2.2 Base Conversion

Now we look at converting between bases. There are distinct techniques for converting to and from decimal. There are also techniques for converting between bases that are powers of 2 (e.g., base 2, 4, 8, 16, etc…)

2.2.1 Converting to Decimal

The value of each digit within a number is based on the individual digit value and the digit's position. Each position in the number contains a different *weight* based on its relative location to the radix point. The weight of each position is based on the radix of the number system that is being used. The weight of each position in decimal is defined as:

$$\textbf{Weight} = (\textbf{Radix})^{p}$$

This expression gives the number system the ability to represent fractional numbers since an expression with a negative exponent (e.g., x^{-y}) is evaluated as one over the expression with the exponent change to positive (e.g., $1/x^{y}$). **Figure** 2.4 shows the generic structure of a number with its positional weight highlighted.

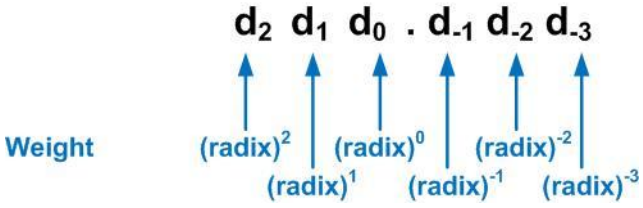


Figure 2.4
Weight Definition

In order to find the decimal value of each of the numerals in the number, its individual numeral value is multiplied by its positional weight. In order to find the value of the entire number, each value of the individual numeral-weight products is summed. The generalized format of this conversion is written as:

$$Total\ Decimal\ Value = \sum_{i=p_{min}}^{p_{max}} d_i \cdot (radix)^i$$

In this expression, p_{max} represents the highest position number that contains a numeral greater than 0. The variable p_{min} represents the lowest position number that contains a numeral greater than 0. These limits are used to simplify the hand calculations; however, these terms theoretically could be $+\infty$ to $-\infty$ with no effect on the result since the summation of every leading 0 and every trailing 0 contributes nothing to the result.

As an example, let's evaluate this expression for a decimal number. The result will yield the original number but will illustrate how positional weight is used. Let's take the number 132.654_{10}. To find the decimal value of this number, each numeral is multiplied by its positional weight and then all of the products are summed. The positional weight for the digit 1 is $(radix)^{p}$ or $(10)^{2}$. In decimal this is called the hundred's position. The positional weight for the digit 3 is $(10)^{1}$, referred to as the ten's position. The positional weight for digit 2 is $(10)^{0}$, referred to as the one's position. The positional weight for digit 6 is

$(10)^{-1}$, referred to as the tenth's position. The positional weight for digit 5 is $(10)^{-2}$, referred to as the hundredth's position. The positional weight for digit 4 is $(10)^{-3}$, referred to as the thousandth's position.

When these weights are multiplied by their respective digits and summed, the result is the original decimal number 132.654_{10}. **Figure** 2.5 shows this process step by step.

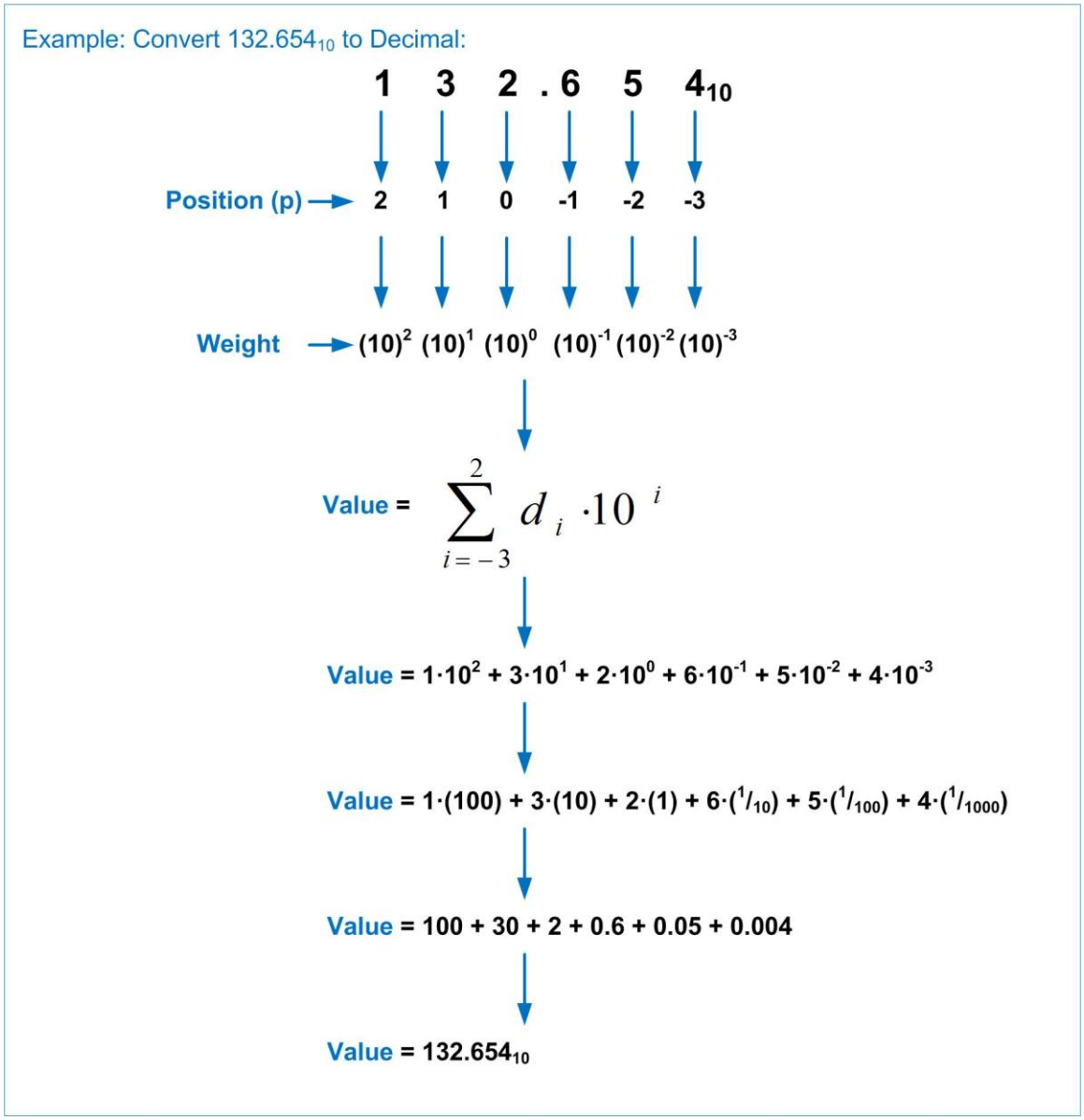


Figure 2.5
Example: Converting Decimal to Decimal

This process is used to convert between any other base to decimal.

2.2.1.1 Binary to Decimal

Let's convert 101.11_2 to decimal. The same process is followed with the exception that the base in the summation is changed to 2. Converting from binary to decimal can be accomplished quickly in your head due to the fact that the bit values in the products are either 1 or 0. That means any bit that is a 0 has no impact on the outcome and any bit that is a 1 simply yields the weight of its position. **Figure** 2.6 shows the step by step process converting a binary number to decimal.

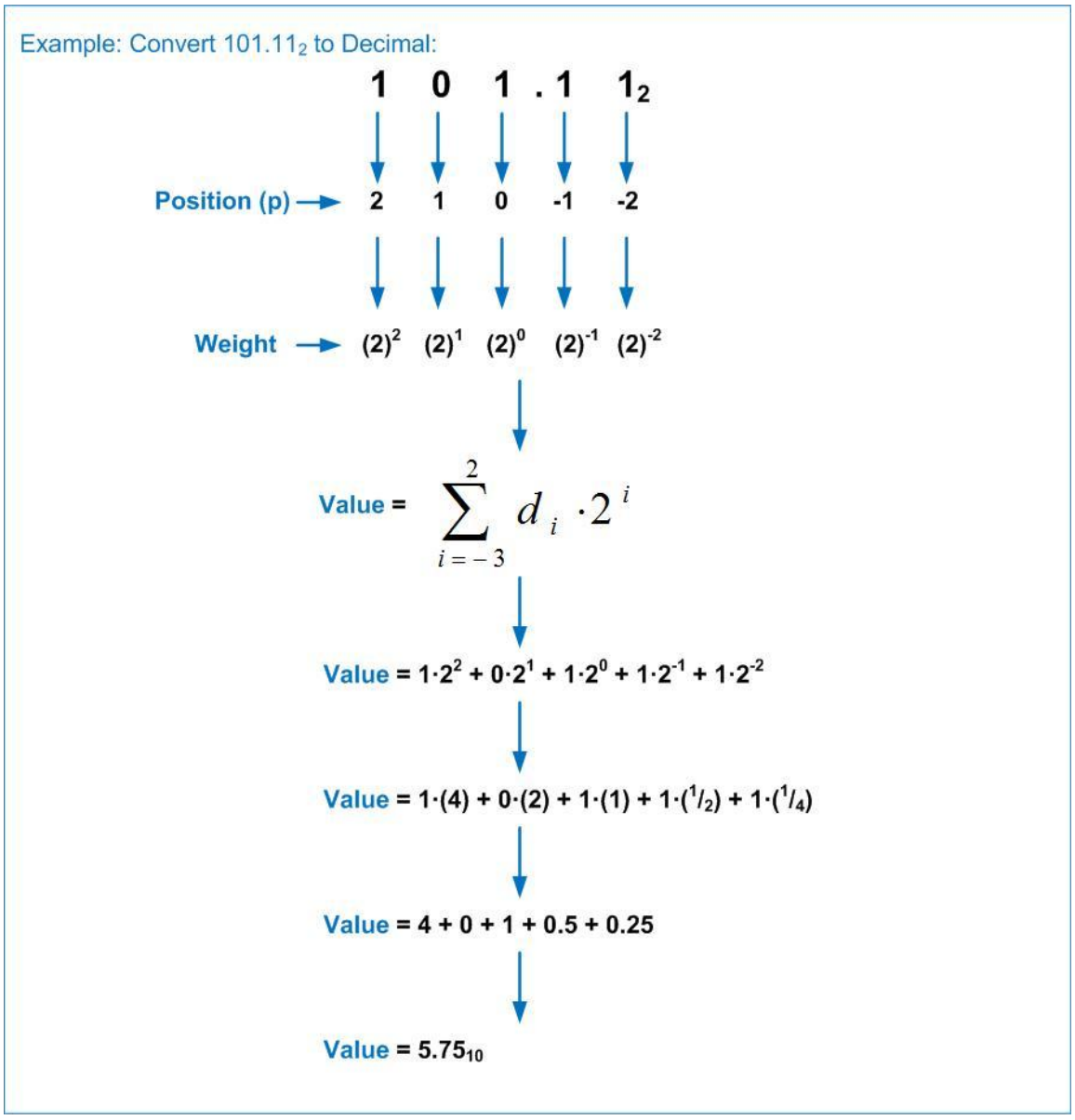


Figure 2.6
Example: Converting Binary to Decimal

2.2.1.2 Octal to Decimal

Let's convert 17.17_8 to decimal. The same process is followed with the exception that the base in the summation is changed to 8. **Figure** 2.7 shows the step by step process converting an octal number to decimal.

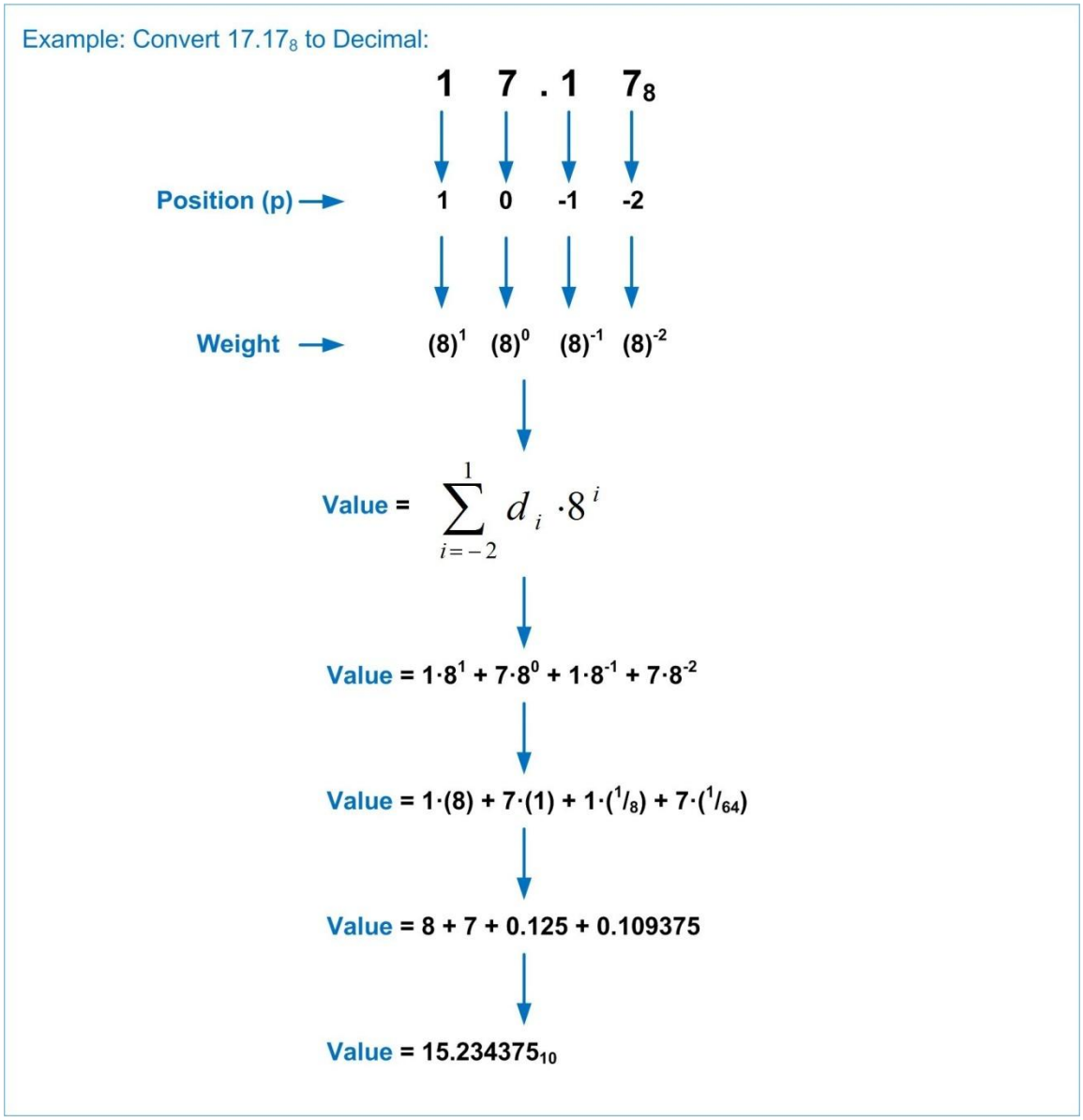


Figure 2.7
Example: Converting Octal to Decimal

2.2.1.3 Hexadecimal to Decimal

Let's convert $1AB.EF_{16}$ to decimal. The same process is followed with the exception that the base is changed to 16. When performing the conversion, the decimal equivalent of the numerals A-F need to be used. **Figure** 2.8 shows the step by step process converting a hexadecimal number to decimal.

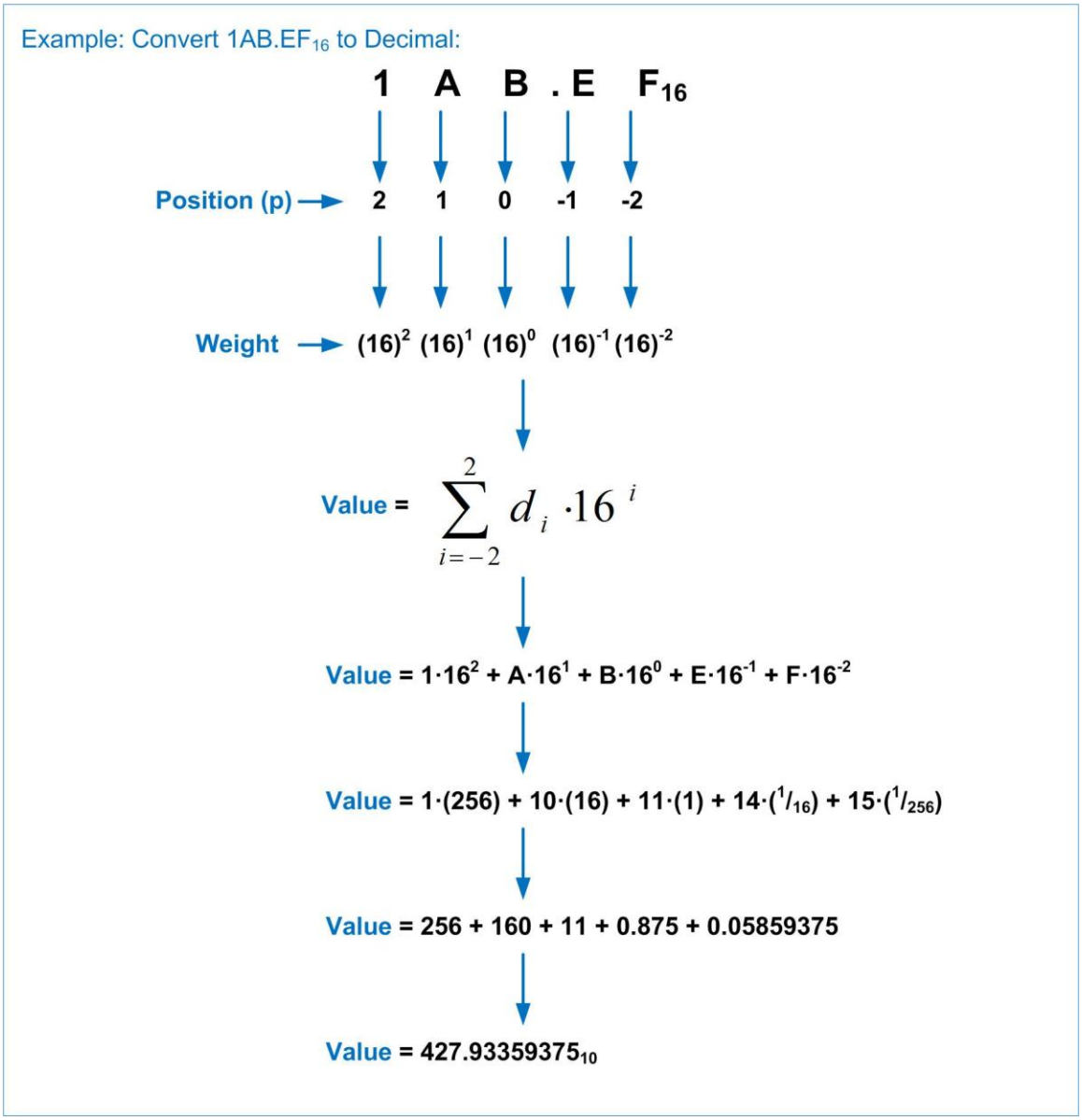


Figure 2.8
Example: Converting Hexadecimal to Decimal

2.2.2 Converting from Decimal

The process of converting from decimal to another base consists of two separate algorithms, one for the whole number portion of the number and one for the fractional portion of the number. The process for converting the whole number portion is to divide the decimal number by the base of the system you wish to convert to. The division will result in a quotient and a whole number remainder. The remainder is recorded as the *least significant numeral* in the converted number. The resulting quotient is then divided again by the base, which results in a new quotient and new remainder. The remainder is recorded as the next higher order numeral in the new number. This process is repeated until a quotient of 0 is achieved. At that point, the conversion is complete. The remainders will always be within the numeral set of the base being converted to.

The process for converting the factional portion is to multiply just the fractional component of the number by the base. This will result in a product that contains a whole number and a fraction. The whole number is recorded as the *most significant digit* of the new converted number. The new fractional portion is then multiplied again by the base with the whole number portion being recorded as the next lower order numeral. This process is repeated until the product yields a fractional component equal to zero or the desired level of accuracy has been achieved. The level of accuracy is specified by the number of numerals in the new converted number. For example, the conversion would be stated as “convert this decimal number to binary with a fractional accuracy of 4 bits”. This means the algorithm would stop once four bits of fraction had been achieved in the conversion.

2.2.2.1 Decimal to Binary

Let's convert 11.375_{10} to binary. **Figure** 2.9 shows the step by step process converting a decimal number to binary.

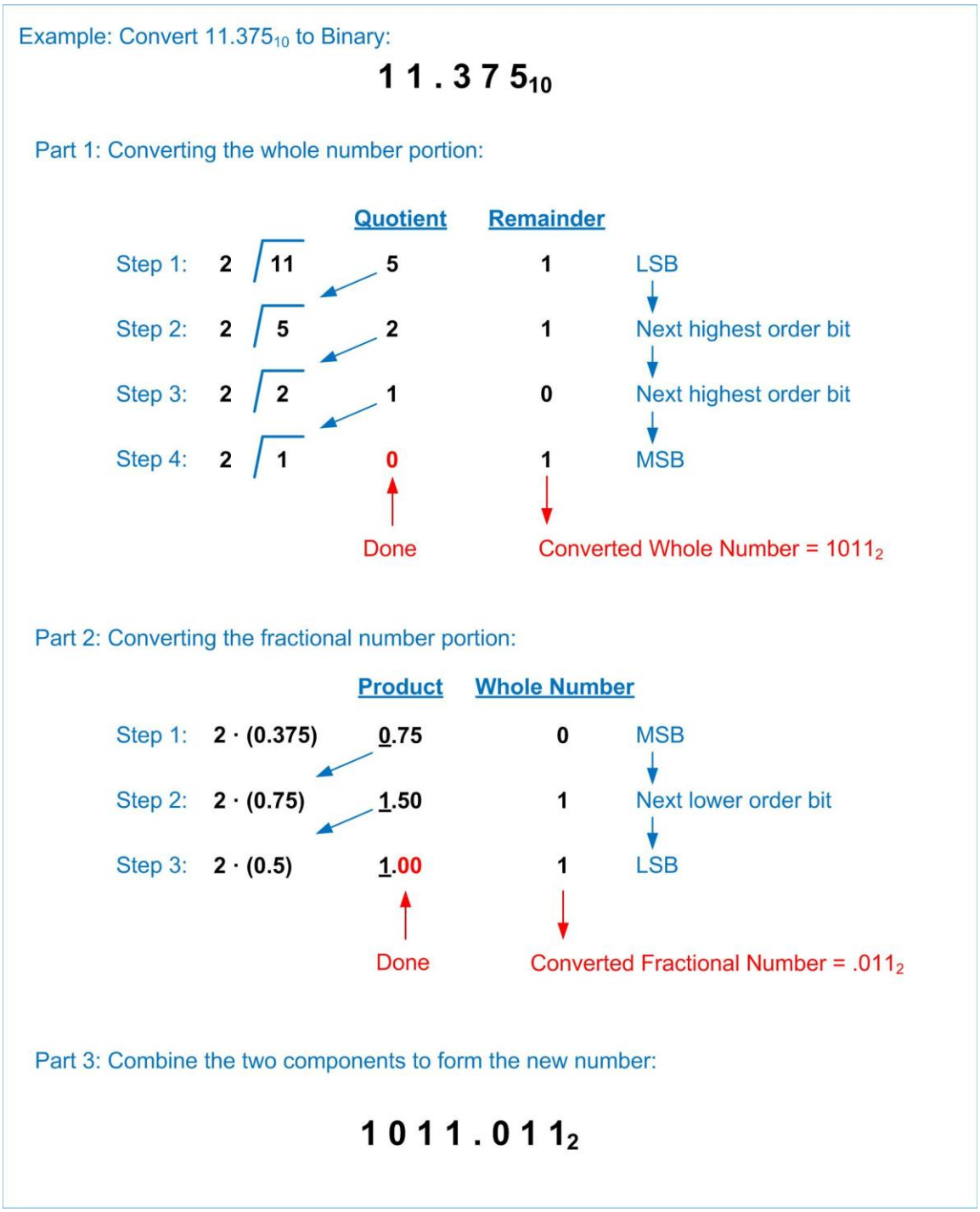


Figure 2.9
Example: Converting Decimal to Binary

2.2.2.2 Decimal to Octal

Let's convert 10.4_{10} to octal with an accuracy of 4 fractional digits. When converting the fractional component of the number, the algorithm is continued until 4 digits worth of fractional numerals has been achieved. Once the accuracy has been achieved, the conversion is finished even though a product with a zero fractional value has not been obtained. **Figure** 2.10 shows the step by step process converting a decimal number to octal with a fractional accuracy of 4 digits.

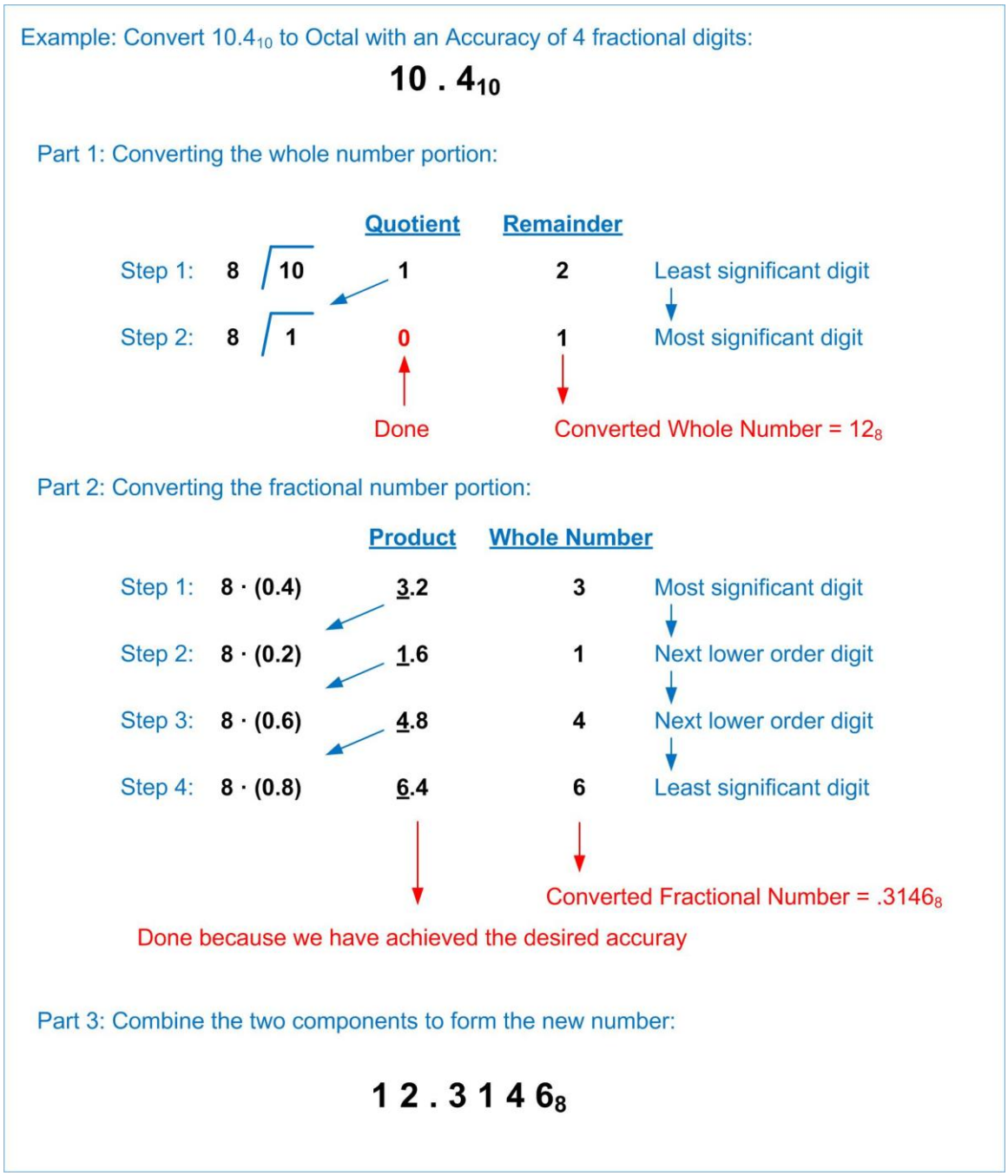


Figure 2.10
Example: Converting Decimal to Octal

2.2.2.3 Decimal to Hexadecimal

Let's convert 254.655_{10} to hexadecimal with an accuracy of 3 fractional digits. When doing this conversion, all of the divisions and multiplications are done using decimal. If the results end up between 10_{10} and $16_{10,}$ then the decimal numbers are substituted with their hex symbol equivalent (e.g., A to F). **Figure** 2.11 shows the step by step process of converting a decimal number to hex with a fractional accuracy of 3 digits.

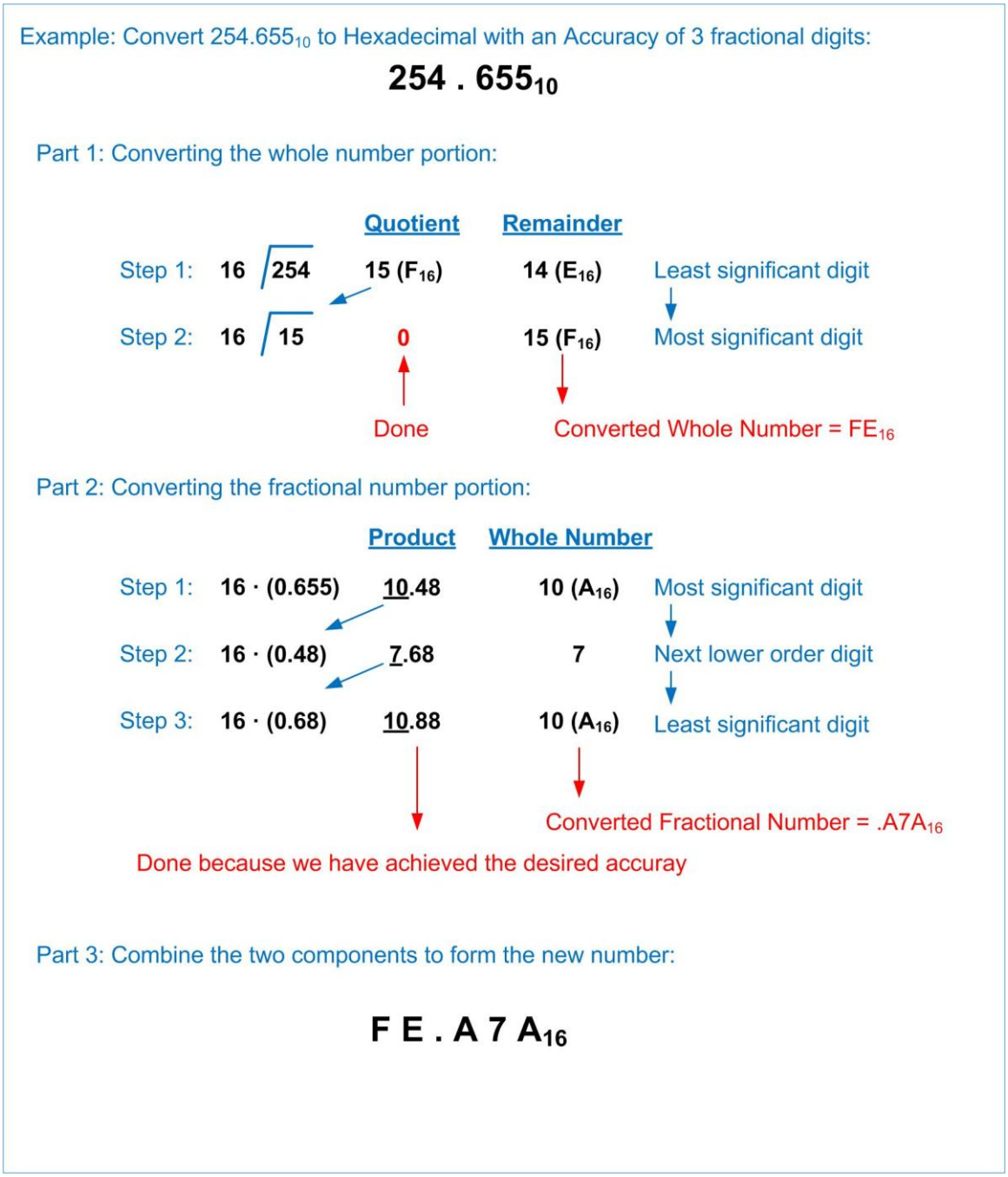


Figure 2.11
Example: Converting Decimal to Hexadecimal

2.2.3 Converting Between 2^n Bases

Converting between 2^n bases (e.g., 2, 4, 8, 16, etc…) takes advantage of the direct mapping that each of these bases has back to binary. Base 8 numbers take exactly 3 binary bits to represent all 8 symbols (e.g., $0_8 = 000_2$, $7_8 = 111_2$). Base 16 numbers take exactly 4 binary bits to represent all 16 symbols (e.g., $0_{16} = 0000_2$, $F_{16} = 1111_2$).

When converting *from* binary to any other 2^n base, the whole number bits are grouped into the appropriate sized sets starting from the radix point and working left. If the final leftmost grouping does not have enough symbols, it is simply padded on left with leading 0's. Each of these groups are then directly substituted with their 2^n base symbol. The fractional number bits are also grouped into the appropriate sized sets starting from the radix point, but this time working right. Again, if the final rightmost grouping does not have enough symbols, it is simply padded on the right with trailing 0's. Each of these groups are then directly substituted with their 2^n base symbol.

2.2.3.1 Binary to Octal

Figure 2.12 shows the step by step process of converting a binary number to octal.

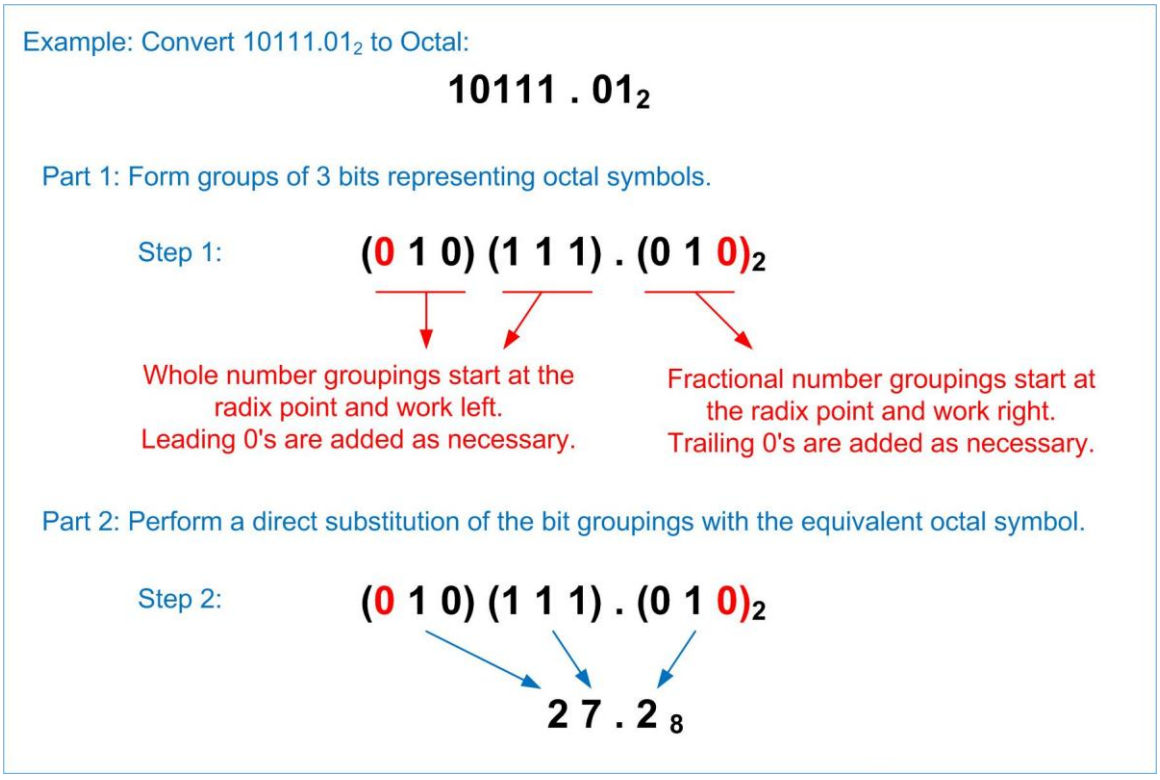


Figure 2.12
Example: Converting Binary to Octal

2.2.3.2 Binary to Hexadecimal

Figure 2.13 shows the step by step process of converting a binary number to hexadecimal.

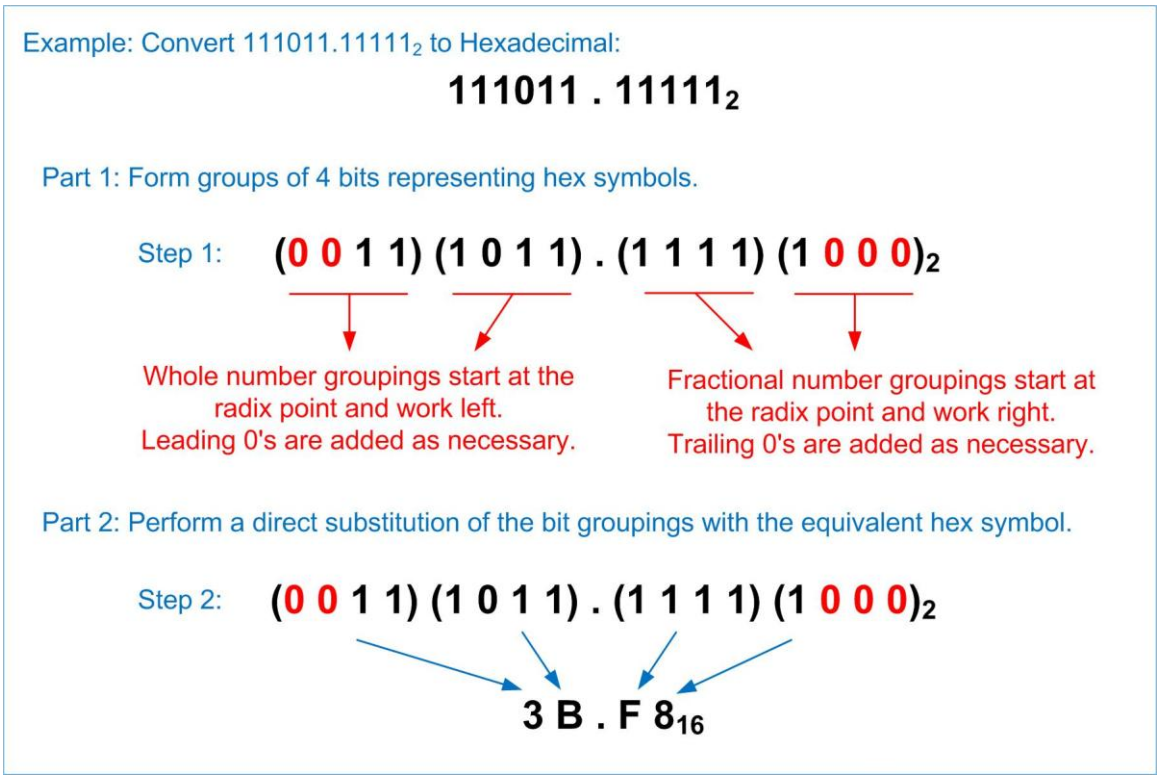


Figure 2.13
Example: Converting Binary to Hexadecimal

2.2.3.3 Octal to Binary

When converting *to* binary from any 2^n base, each of the symbols in the originating number are replaced with the appropriate sized number of bits. An octal symbol will be replaced with 3 binary bits while a hexadecimal symbol will be replaced with 4 binary bits. Any leading or trailing 0's can be removed from the converted number once complete. **Figure** 2.14 shows the step by step process of converting an octal number to binary.

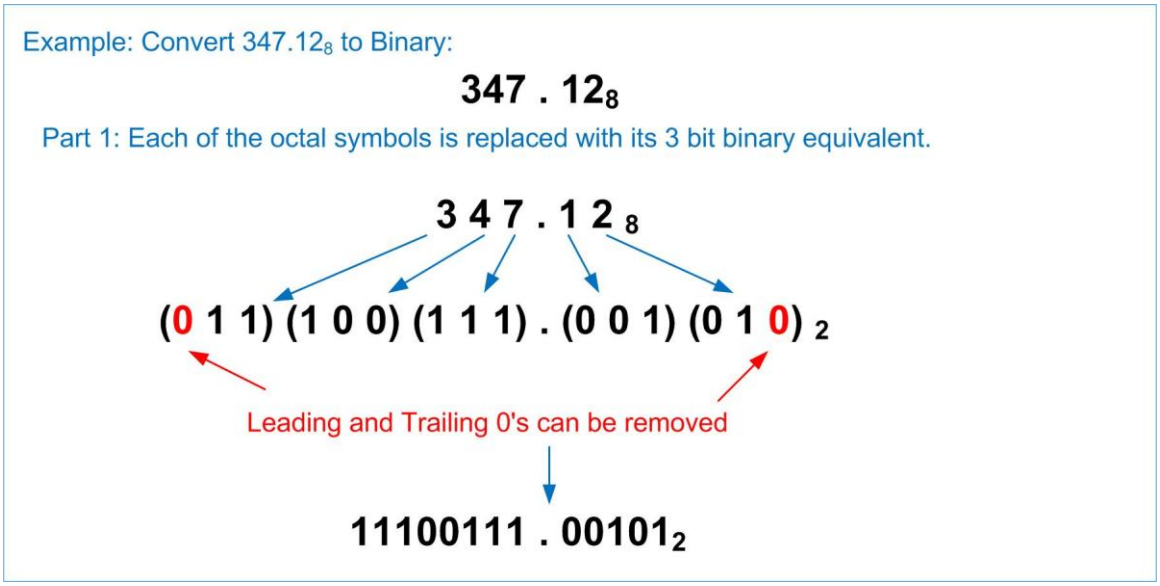


Figure 2.14
Example: Converting Octal to Binary

2.2.3.4 Hexadecimal to Binary

Figure 2.15 shows the step by step process of converting a hexadecimal number to binary.

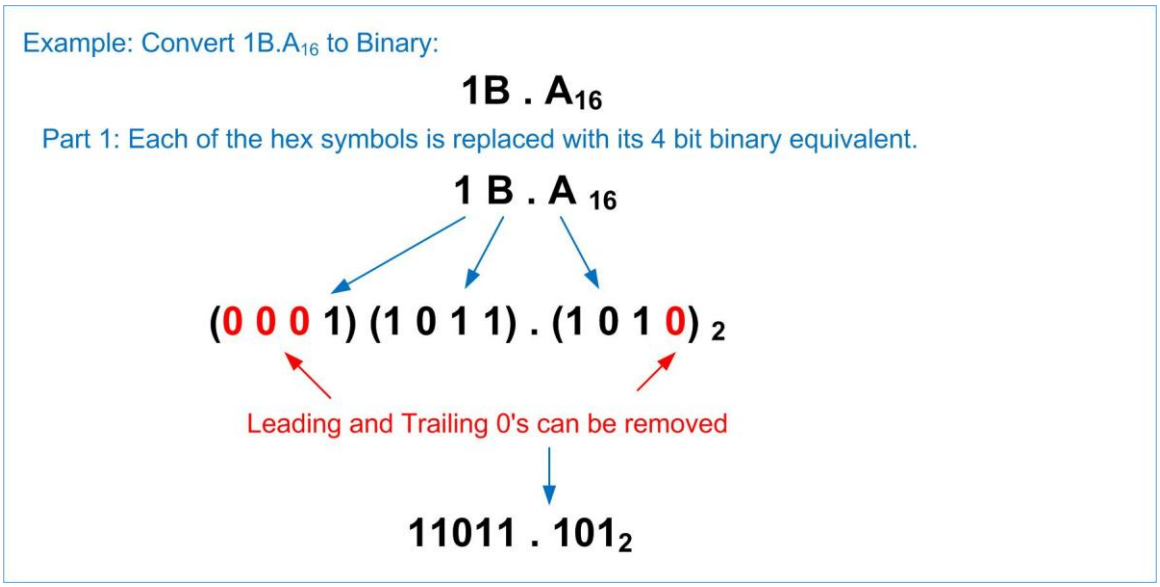


Figure 2.15
Example: Converting Hexadecimal to Binary

2.2.3.5 Octal to Hexadecimal

When converting between 2^n bases (excluding binary) the number is first converted into binary and then converted from binary into the final 2^n base using the algorithms described before. **Figure** 2.13 shows the step by step process of converting an octal number to hexadecimal.

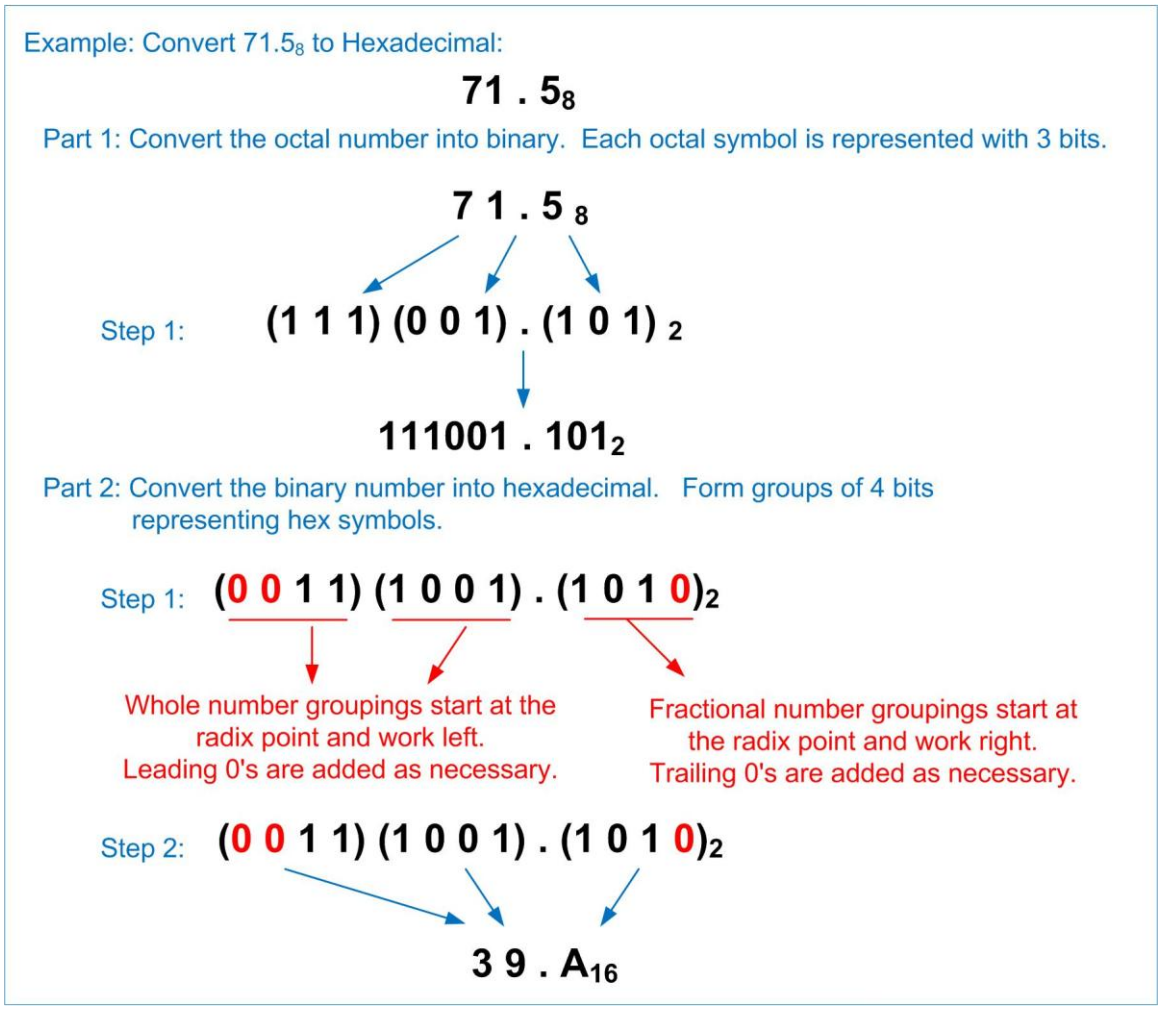


Figure 2.16
Example: Converting Octal to Hexadecimal

2.2.3.6 Hexadecimal to Octal

Figure 2.17 shows the step by step process of converting a hexadecimal number to octal.

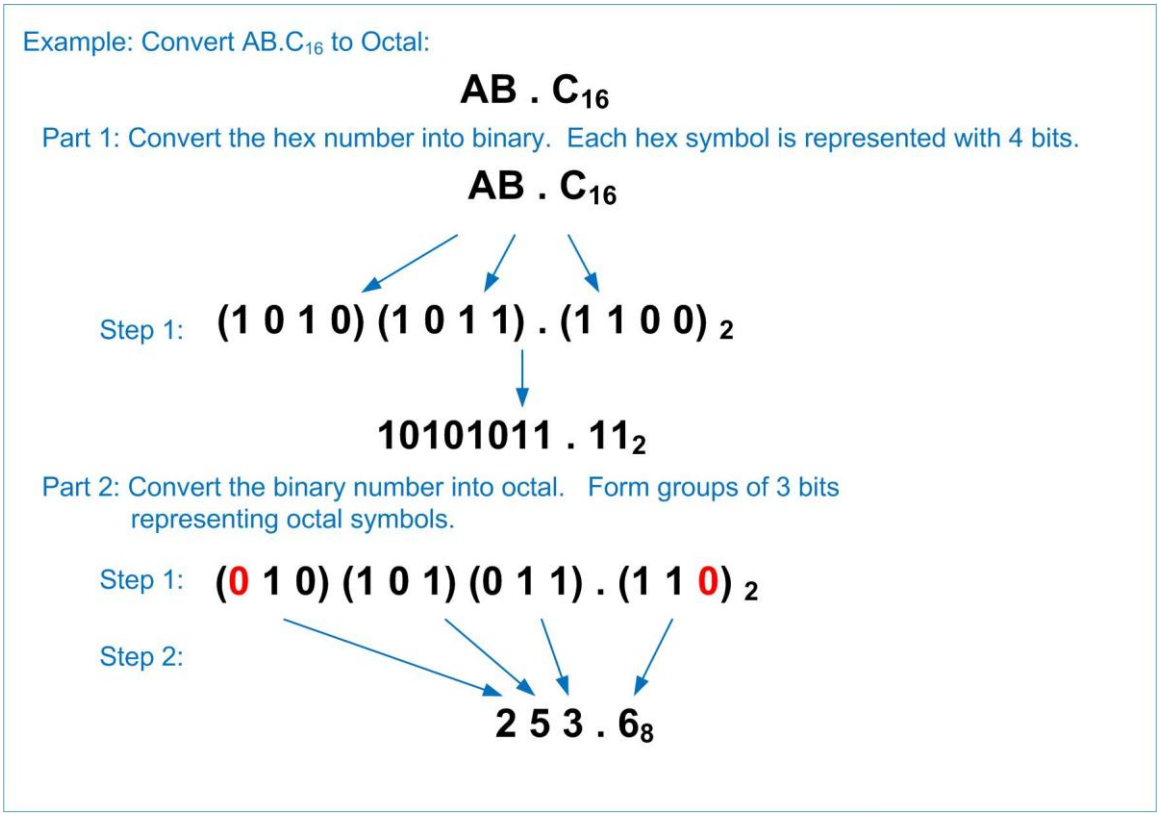


Figure 2.17
Example: Converting Hexadecimal to Octal

2.3 Binary Arithmetic

2.3.1 Addition (Carrys)

Binary addition is a straight forward process that mirrors the approach we have learned for longhand decimal addition. The two numbers (or terms) to be added are aligned at the radix point and addition begins at the least significant bit. If the sum of the least significant position yields a value with two bits (e.g., 10_2), then the least significant bit is recorded and the most significant bit is *carried* to the next higher position. The sum of the next higher position is then performed including the potential *carry bit* from the prior addition. This process continues from the least significant position to the most significant position.

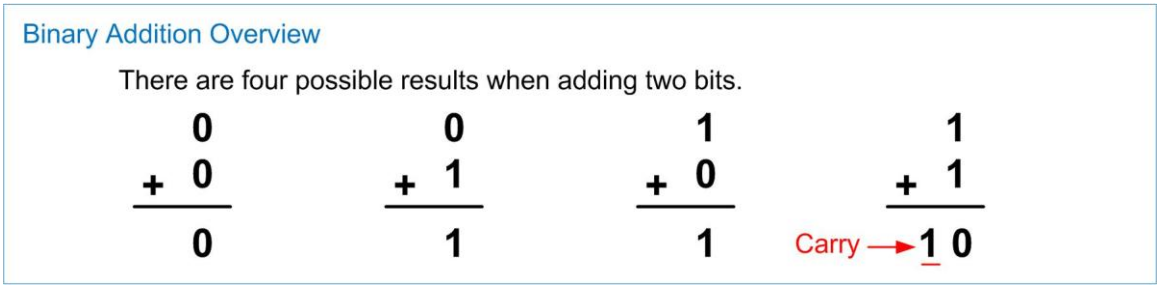


Figure 2.18
Binary Addition Overview

When performing binary addition, the width of the inputs and output is fixed (e.g., n-bits). Carrys that exist within the n-bits are treated in the normal fashion of including them in the next higher position sum; however, if the highest position summation produces a carry, this is a uniquely named event. This event is called a *carry out* or the sum is said to *generate a carry.* The reason this type of event is given special terminology is because in real circuitry, the number of bits of the inputs and output is fixed in hardware and the carry out is typically handled by a separate circuit.

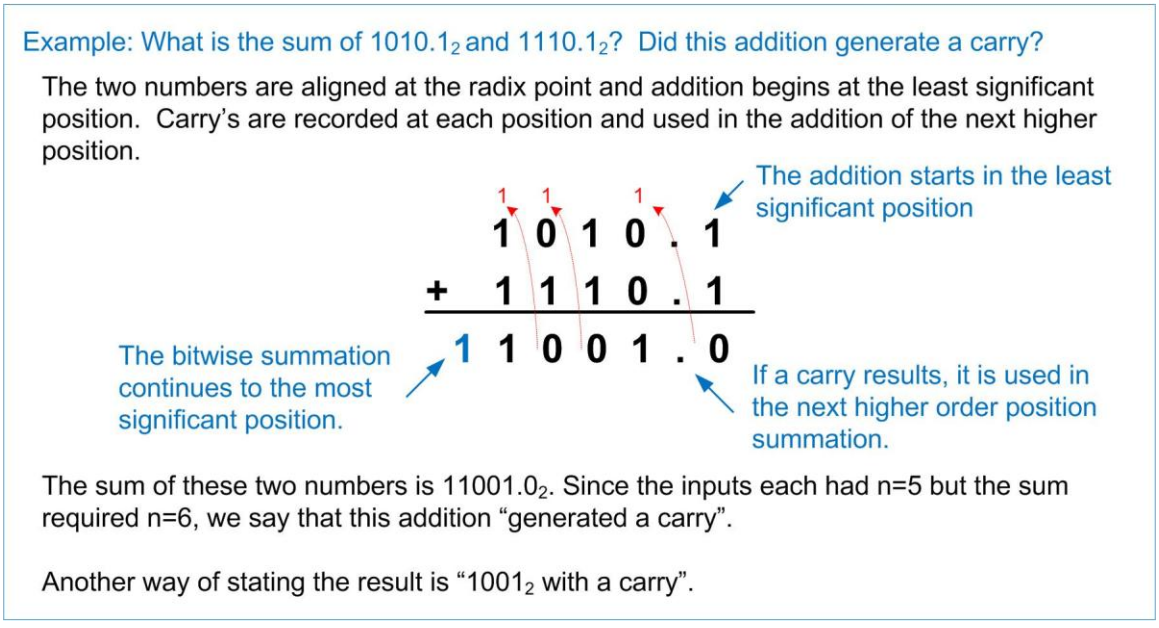


Figure 2.19
Example: Binary Addition

The largest decimal sum that can result from the addition of two binary numbers is given by $2 \cdot (2^n - 1)$. For example, two 8-bit numbers to be added could both represent their highest decimal value of $(2^n - 1)$ or 255_{10} (e.g., 1111 1111_2). The sum of this number would result in 510_{10} or (1 1111 1110_2). Notice that the largest sum achievable would only require one additional bit. This means that a single carry bit is sufficient to handle all possible magnitudes for binary addition.

2.3.2 Subtraction (Borrows)

Binary subtraction also mirrors longhand decimal subtraction. In subtraction, the formal terms for the two numbers being operated on are *minuend* and *subtrahend*. The subtrahend is subtracted from the minuend to find the *difference*. In longhand subtraction, the minuend is the top number and the subtrahend is the bottom number. For a given position, If the minuend is less than the subtrahend, it needs to *borrow* from the next higher order position to produce a difference that is positive. If the next higher position does not have a value that can be borrowed from (e.g., 0), then it in turn needs to borrow from the next higher position, and so forth.

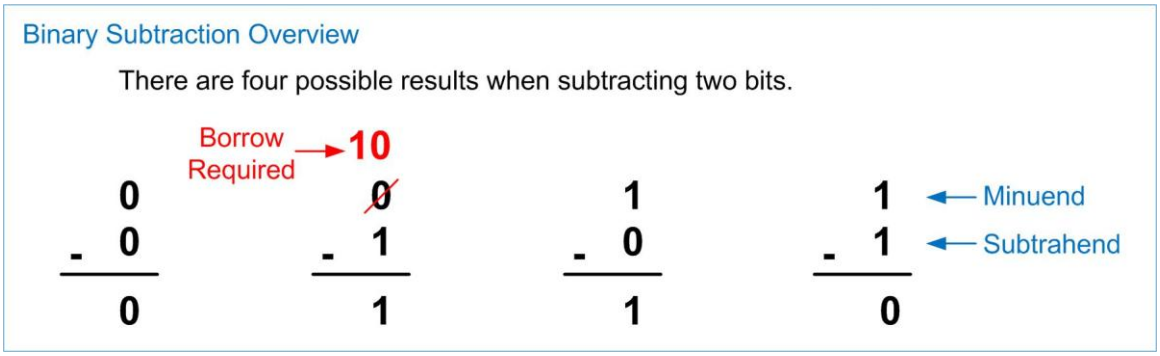
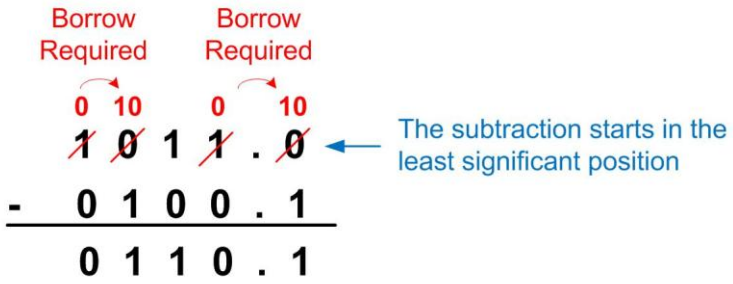


Figure 2.20
Binary Subtraction Overview

As with binary addition, binary subtraction is accomplished on fixed widths of inputs and output (e.g., n-bits). The minuend and subtrahend are aligned at the radix point and subtraction begins at the least significant bit position. Borrows are used as necessary as the subtractions moves from the least significant position to the most significant position. If the most significant position requires a borrow, this is a uniquely named event. This event is called a *borrow in* or the subtraction is said to *require a borrow*. Again, the reason this event is uniquely named is because in real circuitry, the number of bits of the input and output is fixed in hardware and the borrow in is typically handled by a separate circuit.

Example: What is the difference between 1011.0_2 and 0100.1_2? Did this subtraction require a borrow in?

The way this question is phased indicates that 1011.0_2 is the minuend and 0100.1_2 is the subtrahend. The two numbers are aligned at the radix point and subtraction begins at the least significant position. Borrows are taken as needed from the next higher order position.



The difference of these two numbers is 0110.1_2 and it did not require a borrow in. To double check if this subtraction worked, we can use look at the decimal equivalents of the numbers: 1011.0_2 (11_{10}) - 0100.1_2 (4.5_{10}) = 0110.1_2 (6.5_{10}), which verifies the subtraction was correct.

Figure 2.21
Example: Binary Subtraction

Notice that if the minuend is less than the subtrahend then the difference will be negative. At this point, we need a way to handle negative numbers.

2.4 Unsigned and Signed Numbers

All of the number systems presented in the prior sections were positive. We need to also have a mechanism to indicate negative numbers. When looking at negative numbers, we only focus on the mapping between decimal and binary since octal and hexadecimal are used as just another representation of a binary number. In decimal, we are able to use the negative *sign* in front of a number to indicate it is negative (e.g., -34_{10}). In binary, this notation works fine for writing numbers on paper (e.g., -1010_2), but we need a mechanism that can be implemented using real circuitry. In a real digital circuit, the circuits can only deal with 0's and 1's. There is no "-" in a digital circuit. Since we only have 0's and 1's in the hardware, we use a bit to represent whether a number is positive or negative. This is referred to as the *sign bit*. If a binary number is not going to have any negative values, then it is called an **unsigned** number and it can only represent positive numbers. If a binary number is going to allow negative numbers, it is called a **signed** number. It is important to always keep track of the type of number we are using as the same bit values can represent very different numbers depending on the coding mechanism that is being used.

2.4.1 Unsigned Numbers

An unsigned number is one that does not allow negative numbers. When talking about this type of code, the number of bits is fixed and stated up front. We use the variable *n* to represent the number of bits in the number. For example, if we had an 8-bit number, we would say "This is an 8-bit, unsigned number".

The number of unique codes in an unsigned number is given by 2^n. For example, if we had an 8-bit number, we would have 2^8 or 256 unique codes (e.g., 0000 0000_2 to 1111 1111_2).

The *range* of an unsigned number refers to the decimal values that the binary code can represent. If we use the notation $N_{unsigned}$ to represent any possible value that an n-bit unsigned number can take on, the range would be defined as:

$$\text{Range of an n-bit UNSIGNED number} \equiv \ 0 \ \leq N_{unsigned} \leq (2^n - 1)$$

For example, if we had an unsigned number with n=4, it could take on a range of values from $+0_{10}$ (0000_2) to $+15_{10}$ (1111_2). Notice that while this number has 16 unique possible codes, the highest decimal value it can represent is 15_{10}. This is because one of the unique codes represents 0_{10}. This is the reason that the highest decimal value that can be represented is given by (2^n-1).

Example: What is the range of decimal numbers that an unsigned 16-bit word can represent?

The term "16-bit word" means that the binary number has n=16. We can plug this into the equation for the range of an unsigned numbers directly.

$$0 \leq N_{unsigned} \leq (2^n - 1)$$

$$\downarrow$$

$$0 \leq N_{unsigned} \leq (2^{16} - 1)$$

$$\downarrow$$

$$0 \leq N_{unsigned} \leq (65{,}536 - 1)$$

$$\downarrow$$

$$0 \leq N_{unsigned} \leq 65{,}535$$

An unsigned 16-bit word can represent decimal numbers from 0 to 65,535.

Figure 2.22
Example: Range of an Unsigned Number

2.4.2 Signed Numbers

Signed numbers are able to represent both positive and negative numbers. The most significant bit of these numbers is always the *sign bit*, which represents whether the number is positive or negative. The sign bit is defined to be a **0 if the number is positive** and **1 if the number is negative**. When using signed numbers, the number of bits is fixed so that the sign bit is always in the same position. There are a variety of ways to encode negative numbers using a sign bit. The encoding method used exclusively in modern computers is called *2's complement*. There are two other encoding techniques called *signed magnitude* and *1's complement* that are rarely used, but are studied to motivate the power of 2's complement. When talking about a signed number, the number of bits and the type of encoding is always stated. For example, we would say "This is an 8-bit, 2's complement number".

2.4.2.1 Signed Magnitude

Signed Magnitude is the simplest way to encode a negative number. In this approach, the most significant bit (e.g., leftmost bit) of the binary number is considered the sign bit (0=positive, 1=negative). The rest of the bits to the right of the sign bit represent the magnitude or absolute value of the number. As an example of this approach, let's look at the decimal values that a 4-bit signed magnitude number can take on.

Decimal	4-bit Signed Magnitude
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Table 2.2
Example: Decimal Values that a 4-bit Signed Magnitude Code Can Represent

There are drawbacks of signed magnitude encoding that are apparent from this example. First, the value of 0_{10} has two signed magnitude codes (0000_2 and 1000_2). This is an inefficient use of the available codes and leads to complexity when building arithmetic circuitry since it must account for two codes representing the same number.

The second drawback is that addition using the negative numbers does not directly map to how decimal addition works. For example, in decimal if we added (-5) + (1), the result would be -4. In signed magnitude, adding these numbers using a traditional adder would produce (-5) + (1) = (-6). This is because the traditional addition would take place on the magnitude portion of the number. A 5_{10} is represented with 101_2. Adding 1 to this number would result in the next higher binary code 110_2 or 6_{10}. Since the sign portion is separate, the addition is performed on |5|, thus yielding 6. Once the sign bit is included, the resulting number is -6. It is certainly possible to build an adder that works on signed magnitude numbers, but it more complex than a traditional adder because it must perform a different addition operation for the negative numbers versus the positive numbers. It is advantageous to have a single adder that works across the entire set of numbers.

Due to the duplicate codes for 0, the range of decimal numbers that signed magnitude can represent is reduced by 1 compared to unsigned encoding. For an n-bit number, there are 2^n unique binary codes available but only 2^n-1 can be used to represent unique decimal numbers. If we use the notation N_{SM} to

represent any possible value that an n-bit signed magnitude number can take on, the range would be defined as:

$$\text{Range of an n-bit SIGNED MANGITUDE number} \equiv -(2^{n-1}-1) \leq N_{SM} \leq +(2^{n-1}-1)$$

Example: What is the range of decimal numbers that an 8-bit signed magnitude number can represent?

The term "8-bit" means that n=8. We can plug this into the equation for the range of a signed magnitude number directly.

$$-(2^{n-1}-1) \leq N_{SM} \leq +(2^{n-1}-1)$$

$$\downarrow$$

$$-(2^{8-1}-1) \leq N_{SM} \leq +(2^{8-1}-1)$$

$$\downarrow$$

$$-127 \leq N_{SM} \leq +127$$

$$\downarrow$$

An 8-bit, signed magnitude number can represent decimal numbers from -127 to +127.

Figure 2.23
Example: Range of a Signed Magnitude Number

Example: What is the decimal value of the 5-bit, signed magnitude code 11010_2

The most significant bit of this 5-bit number is a 1, which indicates that the number is negative.

The remaining 4-bits are the magnitude of the decimal number and are converted directly to decimal.

$$1\ 0\ 1\ 0_2$$

$$|Value| = \sum_{i=0}^{3} d_i \cdot 2^i$$

$$|Value| = 1\cdot2^3 + 0\cdot2^2 + 1\cdot2^1 + 0\cdot2^0$$

$$|Value| = 1\cdot(8) + 0\cdot(4) + 1\cdot(2) + 0\cdot(1)$$

$$|Value| = 8 + 0 + 2 + 0$$

$$|Value| = 10_{10}$$

The negative sign is then added back to the converted number giving a decimal value of -10_{10}

Figure 2.24
Example: Decimal Value of a Signed Magnitude Number

2.4.2.2 One's Complement

One's complement is another simple way to encode negative numbers. In this approach, the negative number is obtained by taking its positive equivalent, and flipping all of the 1's to 0's and 0's to 1's. This procedure of *flipping the bits* is called a **complement** (notice the two e's). In this way, the most significant bit of the number is still the sign bit (0=positive, 1=negative). The rest of the bits represent the value of the number, but in this encoding scheme, the negative number values are less intuitive. Let's look at the decimal values that a 4-bit 1's complement number can take on.

Decimal	4-bit 1's Complement
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Table 2.3
Example: Decimal Values that a 4-bit 1's Complement Code Can Represent

Again we notice that there are two different codes for 0_{10} (0000_2 and 1111_2). This is a drawback of 1's complement because it reduces the possible range of numbers that can be represented from 2^n to (2^n-1) and requires arithmetic operations that take into account the gap in the number system. If we use the notation N_{1comp} to represent any possible value that an n-bit 1's complement number can take on, the range is defined as:

$$\text{Range of an n-bit 1's COMPLEMENT number} \equiv -(2^{n-1}-1) \leq N_{1comp} \leq +(2^{n-1}-1)$$

There are advantages of 1's complement, however. First, the numbers are ordered such that traditional addition works on both positive and negative numbers (excluding the double 0 gap). Taking the example of (-5) + (1) again, in 1's complement the result yields -4, just as in a traditional decimal system. Notice in 1's complement, -5_{10} is represented with 1010_2. Adding 1 to this entire binary code would result in the next higher binary code 1011_2 or -4_{10} from the above table. This makes addition circuitry less complicated, but still not as simple as if the double 0 gap was eliminated.

Another advantage of 1's complement is that as the numbers are incremented beyond the largest value in the set, they *roll over* and start counting at the lowest number. For example, if you increment the number 0111_2 (7_{10}), it goes to the next higher binary code 1000_2, which is -7_{10}. The ability to have the numbers roll over is a useful feature for computer systems.

Example: What is the range of decimal numbers that a 24-bit 1's complement number can represent?

The term "24-bit" means that n=24. We can plug this into the equation for the range of a 1's complement number directly.

$$-(2^{n-1}-1) \leq N_{1comp} \leq +(2^{n-1}-1)$$

$$\downarrow$$

$$-(2^{24-1}-1) \leq N_{1comp} \leq +(2^{24-1}-1)$$

$$\downarrow$$

$$-8{,}388{,}607 \leq N_{1comp} \leq +8{,}388{,}607$$

$$\downarrow$$

A 24-bit, 1's complement number can represent decimal numbers from -8,388,607 to +8,388,607.

Figure 2.25
Example: Range of a 1's Complement Number

Example: What is the decimal value of the 5-bit, 1's complement code 11010_2

The most significant bit of this 5-bit number is a 1, which indicates that the number is negative.

11010

Sign Bit

To find the magnitude of the number, we first perform a complement on the entire number to find its positive equivalent.

$$1\ 1\ 0\ 1\ 0_2 \rightarrow 0\ 0\ 1\ 0\ 1_2$$

A complement operation turns all 1's to 0's and all 0's to 1's

The number can now be converted into decimal to find its magnitude.

$$|Value| = \sum_{i=0}^{4} d_i \cdot 2^i$$

$$\downarrow$$

$$|Value| = 0\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0$$

$$\downarrow$$

$$|Value| = 0\cdot(16) + 0\cdot(8) + 1\cdot(4) + 0\cdot(2) + 1\cdot(1)$$

$$\downarrow$$

$$|Value| = 0 + 0 + 4 + 0 + 1$$

$$\downarrow$$

$$|Value| = 5_{10}$$

The negative sign is then added back to the converted number giving a decimal value of -5_{10}

Figure 2.26
Example: Decimal Value of a 1's Complement Number

2.4.2.3 Two's Complement

Two's complement is an encoding scheme that addresses the double 0 issue in signed magnitude and 1's complement representations. In this approach, the negative number is obtained by subtracting its positive equivalent from 2^n. This is identical to performing a complement on the positive equivalent and then adding 1. If a carry is generated, it is discarded. This procedure is called "*taking the 2's complement of a number*". The procedure of complementing each bit and adding one is the most common technique to perform a 2's complement. In this way, the most significant bit of the number is still the sign bit (0=positive, 1=negative) but all of the negative numbers are in essence *shifted up* so that the double 0 gap is eliminated. Taking the 2's complement of a positive number will give its negative counterpart and vice versa. Let's look at the decimal values that a 4-bit 2's complement number can take on.

Decimal	4-bit 2's Complement
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Table 2.4
Example: Decimal Values that a 4-bit 2's Complement Code Can Represent

There are many advantages of 2's complement encoding. First, there is no double 0 gap, which means that all possible 2^n unique codes that can exist in an n-bit number are used. This gives the largest possible range of numbers that can be represented. If we use the notation N_{2comp} to represent any possible value that an n-bit 2's complement number can take on, the range is defined as:

$$\text{Range of an n-bit 2's COMPLEMENT number} \equiv -(2^{n-1}) \leq N_{2comp} \leq +(2^{n-1}-1)$$

Another advantage of 2's complement is that addition with negative numbers works exactly the same as decimal. In our example of (-5) + (1), the result 4. Arithmetic circuitry can be built to mimic the way our decimal arithmetic works without the need to consider the double 0 gap. Finally, the roll over characteristic is preserved from 1's complement. Incrementing +7 by +1 will result in -7.

Example: What is the range of decimal numbers that a 32-bit 2's complement number can represent?

The term "32-bit" means that n=32. We can plug this into the equation for the range of a 2's Complement number directly.

$$-(2^{n-1}) \leq N_{2comp} \leq +(2^{n-1} - 1)$$

$$\downarrow$$

$$-(2^{32-1}) \leq N_{2comp} \leq +(2^{32-1} - 1)$$

$$\downarrow$$

$$-2{,}147{,}483{,}648 \leq N_{2comp} \leq +2{,}147{,}483{,}647$$

$$\downarrow$$

A 32-bit, 2's complement number can represent decimal numbers from -2,147,483,648 to +2,147,483,647.

Figure 2.27
Example: Range of a 2's Complement Number

Example: What is the decimal value of the 5-bit, 2's complement code 11010_2

The most significant bit of this 5-bit number is a 1, which indicates that the number is negative.

$$\underline{1}1010$$

Sign Bit

To find the magnitude of the number, we take the two's complement of the entire number to find its positive equivalent.

Taking the 2's Complement

Step 1 – Complement the number

$$1\ 1\ 0\ 1\ 0_2$$

$$\downarrow$$

$$0\ 0\ 1\ 0\ 1_2$$

Step 2 – Add 1, ignore carry out if any

$$\begin{array}{r} {\scriptstyle 1} \\ 0\ 0\ 1\ 0\ 1 \\ + \qquad\quad 1 \\ \hline 0\ 0\ 1\ 1\ 0_2 \end{array}$$

The number can now be converted into decimal to find its magnitude. By inspection, this number is $00110_2 = 6_{10}$

The negative sign is then added back to the converted number giving a decimal value of -6_{10}

Figure 2.28
Example: Decimal Value of a 2's Complement Number

Example: What is the 8-bit, 2's complement code for -99_{10}?

Step 1 – Determine if -99_{10} can be represented within the 2's complement number range

An 8-bit 2's complement number has a range of:

$$-(2^{n-1}) \leq N_{2comp} \leq +(2^{n-1} - 1)$$

$$-(2^{8-1}) \leq N_{2comp} \leq +(2^{8-1} - 1)$$

$$-128 \leq N_{2comp} \leq +127$$

Yes, the number -99_{10} falls within the range that an 8-bit 2's complement number can represent, continue...

Step 2 – Find the positive binary code for -99_{10}

		Quotient	Remainder	
2	99	49	1	LSB
2	49	24	1	Next highest order bit
2	24	12	0	Next highest order bit
2	12	6	0	Next highest order bit
2	6	3	0	Next highest order bit
2	3	1	1	Next highest order bit
2	1	0	1	MSB

Done

Converted Whole Number = $0110\ 0011_2$

A leading 0 is added to the left to make the code 8-bits

Step 3 – Perform 2's Complement on the positive equivalent of 99_{10}

Complement the number

$$0\,1\,1\,0\ \ 0\,0\,1\,1_2$$

$$1\,0\,0\,1\ \ 1\,1\,0\,0_2$$

Step 2 – Add 1, ignore carry out if any

$$\begin{array}{r} 1\,0\,0\,1\ \ 1\,1\,0\,0 \\ +\qquad\qquad\ \ 1 \\ \hline 1\,0\,0\,1\ \ 1\,1\,0\,1_2 \end{array}$$

The 8-bit, 2's complement code for -99_{10} is $1001\ 1101_2$

Figure 2.29
Example: 2's Complement Code of a Decimal Number

2.4.2.4 Arithmetic with 2's Complement

Two's complement has a variety of arithmetic advantages. First, the operations of addition, subtraction and multiplication are handled exactly the same as when using unsigned numbers. This means that duplicate circuitry is not needed in a system that uses both number types. Second, the ability to convert a number from positive to its negative representation by performing a *2's complement* means that an adder circuit can be used for subtraction. For example, if we wanted to perform the subtraction $13_{10} - 4_{10} = 9_{10}$, this is the same as performing $13_{10} + (-4_{10}) = 9_{10}$. This allows us to use a single adder circuit to perform both addition and subtraction as long as we have the ability to take the 2's complement of a number. Creating a circuit to perform 2's complement can be simpler and faster than building a separate subtraction circuit, so this approach can sometimes be advantageous.

There are specific rules for performing 2's complement arithmetic that must be followed to ensure proper results. First, any carry or borrow that is generated is **ignored**. The second rule that must be followed is to always check if **2's complement overflow** occurred. Two's complement overflow refers to when the result of the operation falls outside of the range of values that can be represented by the number of bits being used. For example, if you are performing 8-bit 2's complement addition, the range of decimal values that can be represented is -128_{10} to $+127_{10}$. Having two input terms of 127_{10} ($0111\ 1111_2$) is perfectly legal because they can be represented by the 8-bits of the 2's complement number; however, the summation of $127_{10} + 127_{10} = 254_{10}$ ($1111\ 1110_2$). This number does *not* fit within the range of values that can be represented and is actually the 2's complement code for -2_{10}, which is obviously incorrect. Two's complement overflow occurs if any of the following occurs:

- The sum of like signs results in an answer with opposite sign
  (e.g., Positive + Positive = Negative or Negative + Negative = Positive)
- The subtraction of a positive number from a negative number results in a positive number
  (e.g., Negative – Positive = Positive)
- The subtraction of a negative number from a positive number results in a negative number
  (e.g., Positive – Negative = Negative)

Computer systems that use 2's complement have a dedicated logic circuit that monitors for any of these situations and lets the operator know that overflow has occurred. These circuits are straight forward since they simply monitor the sign bits of the input and output codes.

Example: Use 4-bit, 2's complement addition to find the differences between 6_{10} and 3_{10}.

The answer in decimal to this problem is $6_{10} - 3_{10} = 3_{10}$. Instead of using subtraction, we will use the 2's complement representation of -3_{10} and add the two numbers.

$$\begin{array}{r} 6_{10} \\ -\ 3_{10} \\ \hline 3_{10} \end{array} \quad = \quad \begin{array}{r} 6_{10} \\ +\ (-3_{10}) \\ \hline 3_{10} \end{array}$$

Step 1 – Find the 4-bit, 2's complement codes for $+6_{10}$ and -3_{10}.

Since 6 is positive, its code is simply its 4-bit binary equivalent ($+6_{10} = 0110_2$)

Since 3 is negative, we'll need to take the 2's complement of its 4-bit positive binary equivalent ($+3_{10} = 0011_2$)

1) Complement the number

$$0\,0\,1\,1_2 \rightarrow 1\,1\,0\,0_2$$

2) Add 1, ignore carry out if any

$$\begin{array}{r} 1\,1\,0\,0 \\ +\quad\ 1 \\ \hline 1\,1\,0\,1_2 \end{array}$$

Step 2 – Add the two codes, ignore carry out if any

$$\begin{array}{r} 6_{10} \\ +\ (-3_{10}) \\ \hline 3_{10} \end{array} \quad = \quad \begin{array}{r} {\scriptstyle 1} \\ 0\,1\,1\,0_2 \\ +\ 1\,1\,0\,1_2 \\ \hline 1\,0\,0\,1\,1_2 \end{array}$$

The sum resulted in a carry out, but in 2's complement addition, this bit is ignored.

The result of the addition was 0011_2 or $+3_{10}$, verifying that this approach was correct. Also, 2's complement overflow did not occur because the result of this operation was within the range of possible values that a 4-bit 2's complement number can represent (e.g., -8_{10} to $+7_{10}$).

Figure 2.30
Example: 2's Complement Addition

Exercise Problems

2.1 Convert the following unsigned numbers to decimal:

a) $1100\ 1100_2$
b) 1001.1001_2
c) 72_8
d) 12.57_8
e) $F3_{16}$
f) $15B.CEF_{16}$

2.2 Convert the following decimal numbers to the base indicated (use an accuracy of 4 digits if necessary and treat all numbers as unsigned):

a) 67_{10} to binary
b) 252.987_{10} to binary
c) 67_{10} to octal
d) 252.987_{10} to octal
e) 67_{10} to hexadecimal
f) 252.987_{10} to hexadecimal

2.3 Convert between the following base 2^n numbers (treat all numbers as unsigned):

a) $1\ 0000\ 1111_2$ to octal
b) $1\ 0000\ 1111.011_2$ to hexadecimal
c) 77_8 to binary
d) $F.A_{16}$ to binary
e) 66_8 to hexadecimal
f) $AB.D_{16}$ to octal

2.4 Perform addition on the following unsigned numbers giving the sum and indicating whether a *carry out* occurred:

a) $1010_2 + 1011_2$
b) $1111\ 1111_2 + 0000\ 0001_2$
c) $1010.1010_2 + 1011.1011_2$
d) $1111\ 1111.1011_2 + 0000\ 0001.1100_2$

2.5 Perform subtraction on the following unsigned numbers giving the difference and indicating whether a *borrow in* was required:

a) $1010_2 - 1011_2$
b) $1111\ 1111_2 - 0000\ 0001_2$
c) $1010.1010_2 - 1011.1011_2$
d) $1111\ 1111.1011_2 - 0000\ 0001.1100_2$

2.6 For the following sizes of 2's complement codes, give the range of decimal numbers that can be represented:

a) 8-bit
b) 16-bit
c) 32-bit
d) 64-bit

2.7 Give the 8-bit 2's complement code for the following decimal numbers:

a) $+88_{10}$
b) -88_{10}
c) -128_{10}
d) -1_{10}

2.8 Give the decimal value that the following <u>2's complement</u> codes represent:

a) 0010_2 (4-bit)
b) 1010_2 (4-bit)
c) $0111\ 1110_2$ (8-bit)
d) $1111\ 1110_2$ (8-bit)

2.9 Perform addition on the following <u>2's complement</u> numbers giving the sum and indicating whether a *2's complement overflow* occurred:

a) 1110_2 + 1011_2

b) $1101\ 1111_2$ + $0000\ 0001_2$

c) 1010.1010_2 + 1000.1011_2

d) $1110\ 1011.1001_2$ + $0010\ 0001.1101_2$

Chapter 3: Digital Circuitry & Interfacing

Now we turn our attention to the physical circuitry and electrical quantities that are used to represent and operate on the binary codes 1 and 0. In this chapter we begin by looking at how logic circuits are described and introduce the basic set of gates used for all digital logic operations. We then look at the underlying circuitry that implements the basic gates including digital signaling and how voltages are used to represent 1's and 0's. We then look at interfacing between two digital circuits and how to ensure that when one circuit sends a binary code, the receiving circuit is able to determine which code was sent. Logic families are then introduced and the details of how basic gates are implemented at the switch level are presented. Finally, interfacing considerations are covered for the most common types of digital loads (e.g., other gates, resistors, and LEDs).

3.1 Basic Gates

The term *gate* is used to describe a digital circuit that implements the most basic functions possible within the binary system. When discussing the operation of a logic gate, we ignore the details of how the 1's and 0's are represented with voltages and manipulated using transistors. We instead treat the inputs and output as simply ideal 1's and 0's. This allows us to design more complex logic circuits without going into the details of the underlying physical hardware.

3.1.1 Describing the Operation of a Logic Circuit

3.1.1.1 The Logic Symbol

A logic symbol is a graphical representation of the circuit that can be used in a schematic to show how circuits in a system interface to one another. For the set of basic logic gates, there are uniquely shaped symbols that graphically indicate their functionality. For more complex logic circuits that are implemented with multiple basic gates, a simple rectangular symbol is used. Inputs of the logic circuit are typically shown on the left of the symbol and outputs are on the right.

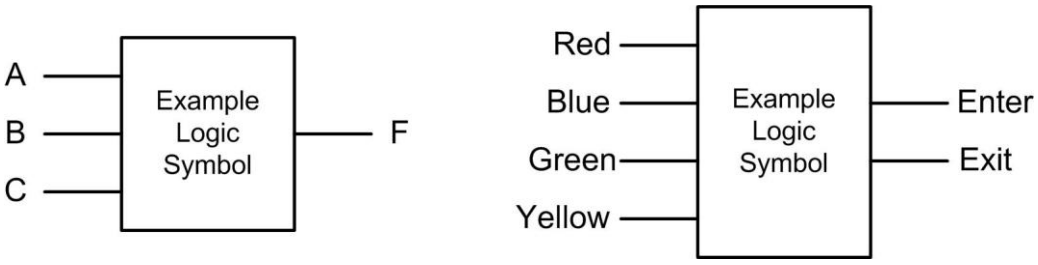


Figure 3.1
Example Logic Symbols

3.1.1.2 The Truth Table

We formally define the functionality of a logic circuit using a *truth table*. In a truth table, each and every possible input combination is listed and the corresponding output of the logic circuit is given. If a logic circuit has n inputs, then it will have 2^n possible input codes. The binary codes are listed in ascending order within the truth table mimicking a binary count starting at 0. By always listing the input codes in this way, we can assign a *row number* to each input that is the decimal equivalent of the binary input code. Row numbers can be used to simplify the notation for describing the functionality of larger circuits.

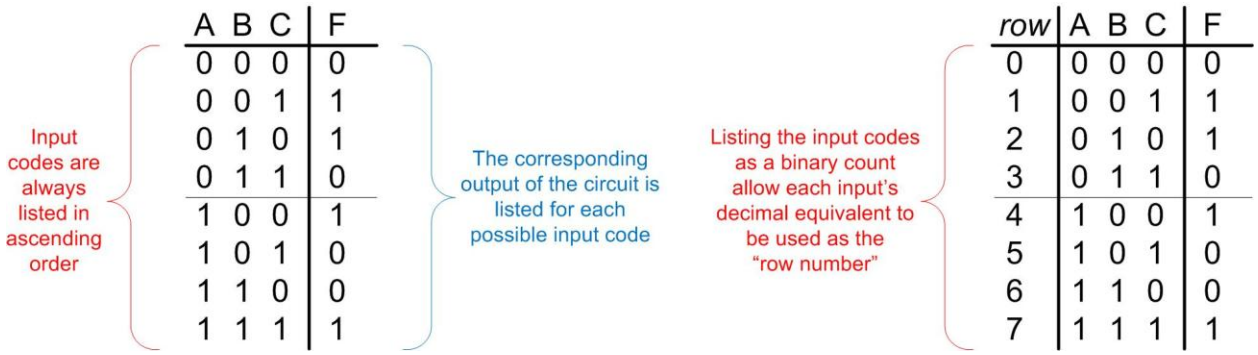


Figure 3.2
Truth Table Formation

3.1.1.3 The Logic Function

A logic *function*, (also called a *logic expression*), is an equation that provides the functionality of each output in the circuit as a function of the inputs. The logic operations for the basic gates are given a symbolic set of operators (e.g., +, ·, ⊕), the details of which will be given in the next sections. The logic function describes the operations that are necessary to produce the outputs listed in the truth table. A logic function is used to describe a single output that can take on only the values 1 and 0. If a circuit contains multiple outputs, then a logic function is needed for each output. The input variables can be included in the expression description just as in an analog function. For example, "F(A,B,C)=…" would state that "F is a function of the inputs A, B and C". This can also be written as "$F_{A,B,C}$=…". The input variables can also be excluded for brevity as in "F=…".

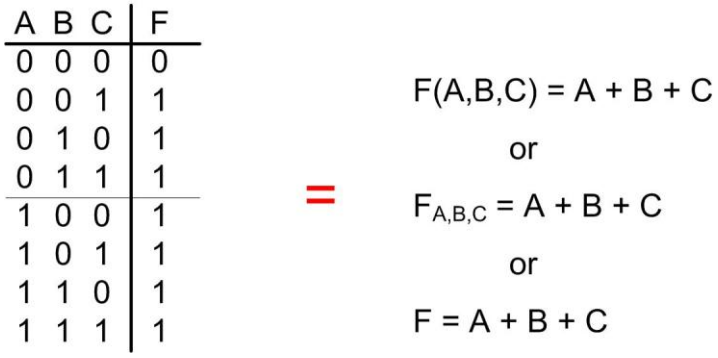


Figure 3.3
Logic Function Formation

3.1.1.4 The Logic Waveform

A logic *waveform* is a graphical depiction of the relationship of the output to the inputs with respect to time. This is often a useful description of behavior since it mimics the format that is typically observed when measuring a real digital circuit using test equipment such as an oscilloscope. In the waveform, each signal can only take on a value of 1 or 0. It is useful to write the logic values of the signal at each transition in the waveform for readability.

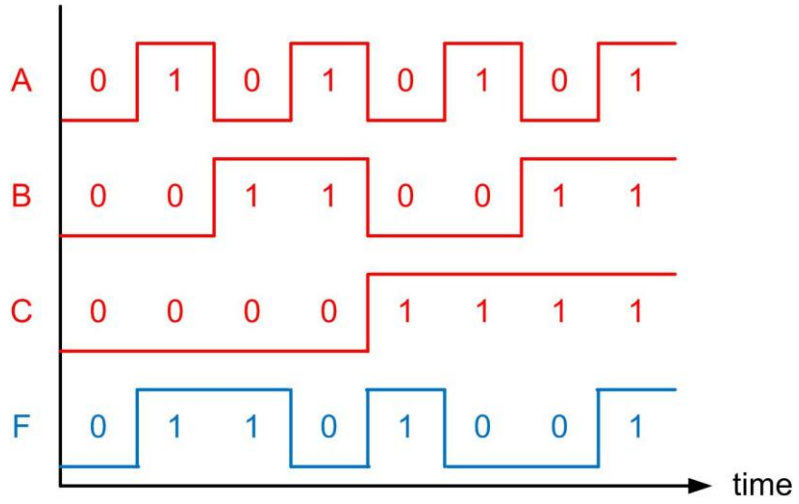


Figure 3.4
Example Logic Waveform

3.1.2 The Buffer

The first basic gate is the *buffer*. The output of a buffer is simply the input. The logic symbol, truth table, logic function and logic waveform for the buffer are given in **Figure** 3.5.

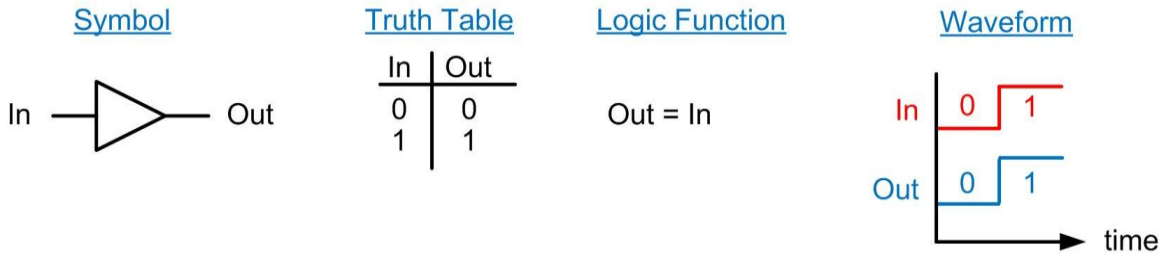


Figure 3.5
Buffer Symbol, Truth Table, Logic Function and Logic Waveform

3.1.3 The Inverter

The next basic gate is the *inverter*. The output of an inverter is the complement of the input. Inversion is also often called the *not* operation. In spoken word, we might say "A is equal to *not* B". This gate is also often called a *not* gate. The symbol for the inverter is the same as the buffer with the exception that an *inversion bubble* (e.g., a circle) is placed on the output. The inversion bubble is a common way to show inversions in schematics and will be used by many of the basic gates. In the logic function, there are three common ways to show this operation. The first way is by placing a prime (') after the input variable (e.g., Out = In'). This notation has the advantage that it is supported in all text editors but has the drawback that it can sometimes be difficult to see. The second way to indicate inversion in a logic function is by placing an *inversion bar* over the input variable (e.g., Out = $\overline{\text{In}}$). The advantage of this notation is that it is easy to see but has the drawback that it is not supported by many text editors. In this text, both conventions will be used to provide exposure to each. The logic symbol, truth table, logic function and logic waveform for the inverter are given in **Figure** 3.6

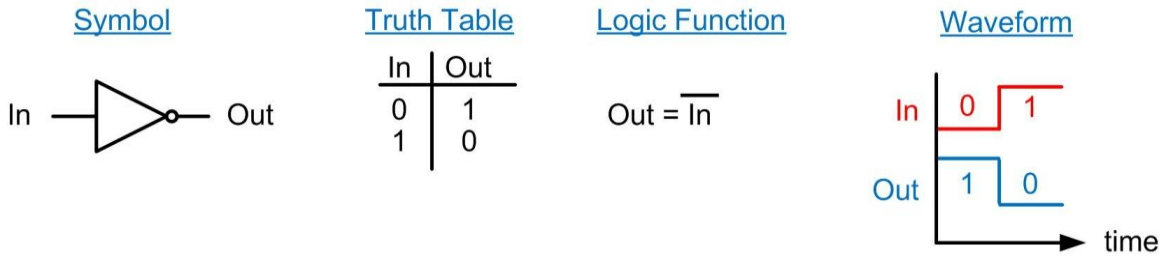


Figure 3.6
Inverter Symbol, Truth Table, Logic Function and Logic Waveform

3.1.4 The AND Gate

The next basic gate is the *AND gate*. The output of an AND gate will only be true (e.g., a logic 1) if **all of the inputs are true**. This operation is also called a *logical product* because if the inputs were multiplied together, the only time the output would be a 1 is if each and every input was a 1. As a result, the logic operator is the dot (·). Another notation that is often seen is the ampersand (&). The logic symbol, truth table, logic function and logic waveform for a 2-input AND gate are given in **Figure** 3.7.

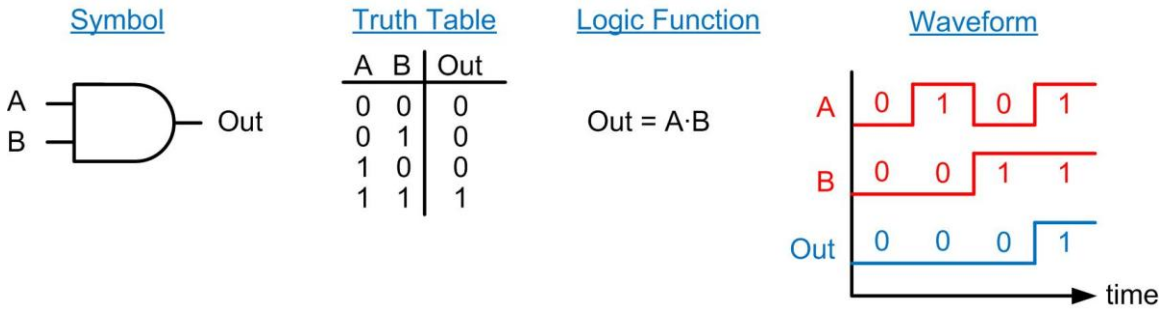


Figure 3.7
2-Input AND Gate Symbol, Truth Table, Logic Function and Logic Waveform

Ideal AND gates can have any number of inputs. The operation of an n-bit AND gates still follows the rule that the output will only be true when all of the inputs are true. Later sections will discuss the limitations on expanding the number of inputs of these basic gates indefinitely.

3.1.5 The NAND Gate

The NAND gate is identical to the AND gate with the exception that the output is inverted. The "N" in NAND stands for "NOT", which represents the inversion. The symbol for a NAND gate is an AND gate with an inversion bubble on the output. The logic expression for a NAND gate is the same as an AND gate but with an inversion bar over the entire operation. The logic symbol, truth table, logic function and logic waveform for a 2-input NAND gate are given in **Figure** 3.8. Ideal NAND gates can have any number of inputs with the operation of an n-bit NAND gate following the rule that the output is always the inversion of an n-bit AND operation.

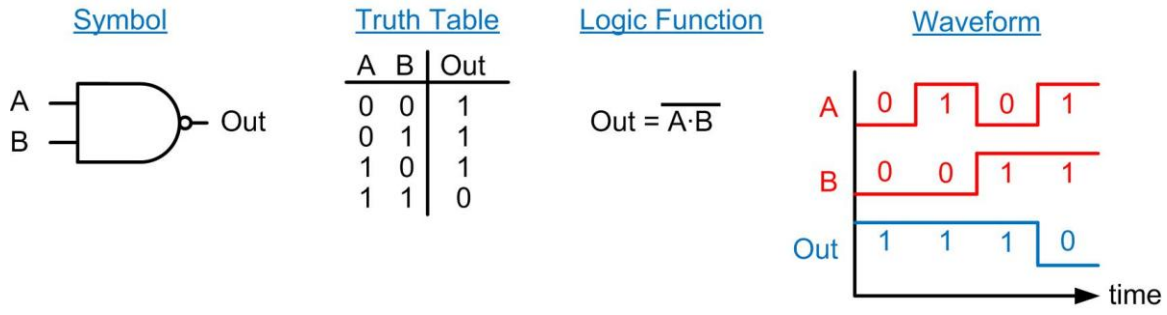


Figure 3.8
2-Input NAND Gate Symbol, Truth Table, Logic Function and Logic Waveform

3.1.6 The OR Gate

The next basic gate is the *OR gate*. The output of an OR gate will be true when **any of the inputs are true**. This operation is also called a *logical sum* because of its similarity to logical disjunction in which the output is true if *at least one* of the inputs is true. As a result, the logic operator is the plus sign (+). The logic symbol, truth table, logic function and logic waveform for a 2-input OR gate are given in **Figure** 3.9. Ideal OR gates can have any number of inputs. The operation of an n-bit OR gates still follows the rule that the output will be true if any of the inputs are true.

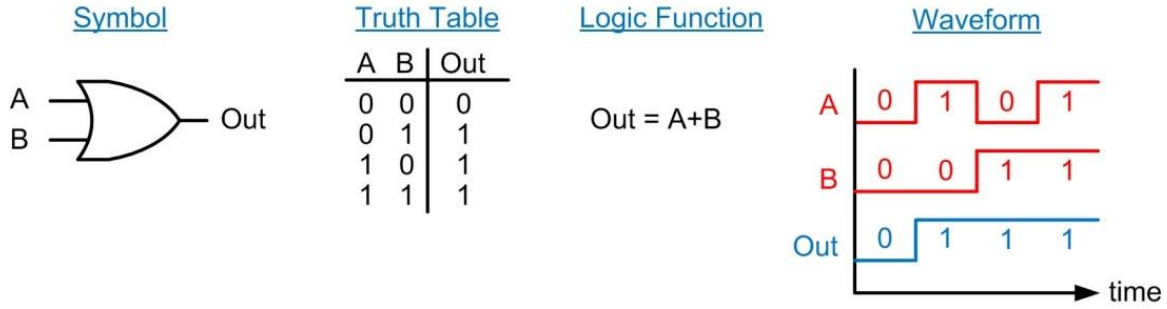


Figure 3.9
2-Input OR Gate Symbol, Truth Table, Logic Function and Logic Waveform

3.1.7 The NOR Gate

The NOR gate is identical to the OR gate with the exception that the output is inverted. The symbol for a NOR gate is an OR gate with an inversion bubble on the output. The logic expression for a NOR gate is the same as an OR gate but with an inversion bar over the entire operation. The logic symbol, truth table, logic function and logic waveform for a 2-input NOR gate are given in **Figure** 3.10. Ideal NOR gates can have any number of inputs with the operation of an n-bit NOR gate following the rule that the output is always the inversion of an n-bit OR operation.

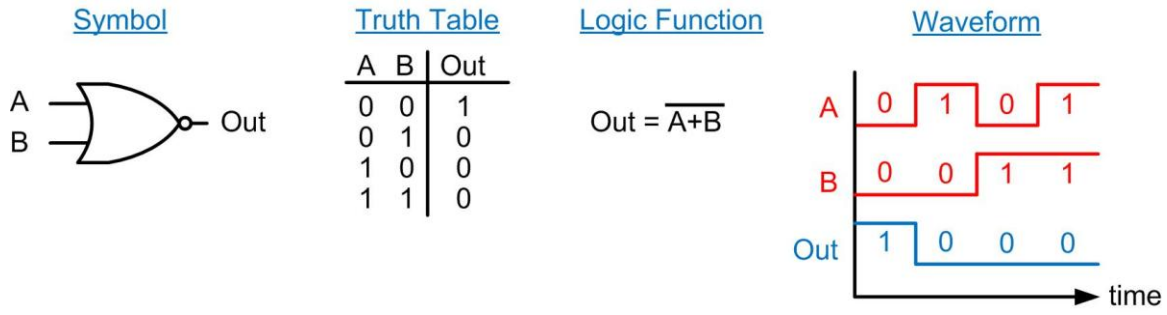


Figure 3.10
2-Input NOR Gate Symbol, Truth Table, Logic Function and Logic Waveform

3.1.8 The XOR Gate

The next basic gate is the *exclusive-OR gate*, or XOR gate for short. This gate is also called a *difference* gate because for the 2-input configuration, its output will be true when the **input codes are different from one another**. The logic operator is a circle around a plus sign (⊕). The logic symbol, truth table, logic function and logic waveform for a 2-input XOR gate are given in **Figure** 3.11.

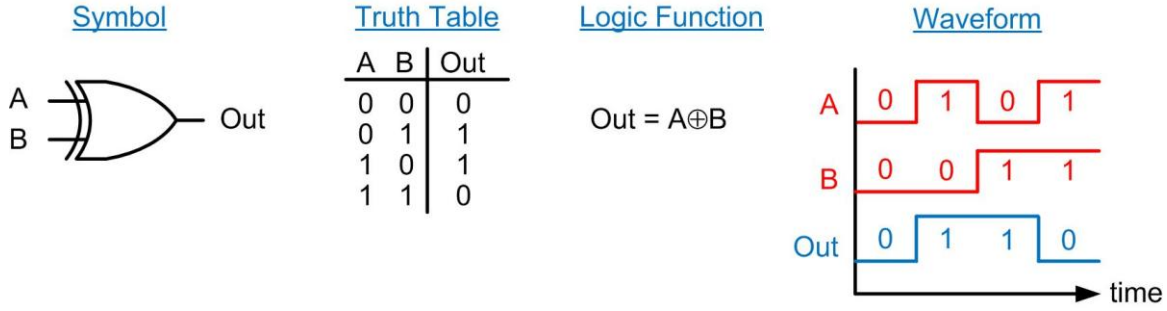


Figure 3.11
2-Input XOR Gate Symbol, Truth Table, Logic Function and Logic Waveform

Using the formal definition of an XOR gate (e.g., the output is true if any of the input codes are different from one another), an XOR gate with more than two inputs can be built. The truth table for a 3-bit XOR gate using this definition is shown in **Figure** 3.12. In modern electronics, this type of gate has found little use since it is much simpler to build this functionality using a combination of AND and OR gates. As such, XOR gates with greater than two inputs do not implement the *difference* function. Instead, a more useful functionality has been adopted in which the output of the n-bit XOR gate is the result of a cascade of 2-input XOR gates. This results in an ultimate output that is true when there are **an ODD number of 1's on the inputs**. This functionality is much more useful in modern electronics for error correction codes and arithmetic. As such, this is the functionality that is seen in modern n-bit XOR gates. This functionality is also shown in **Figure** 3.12.

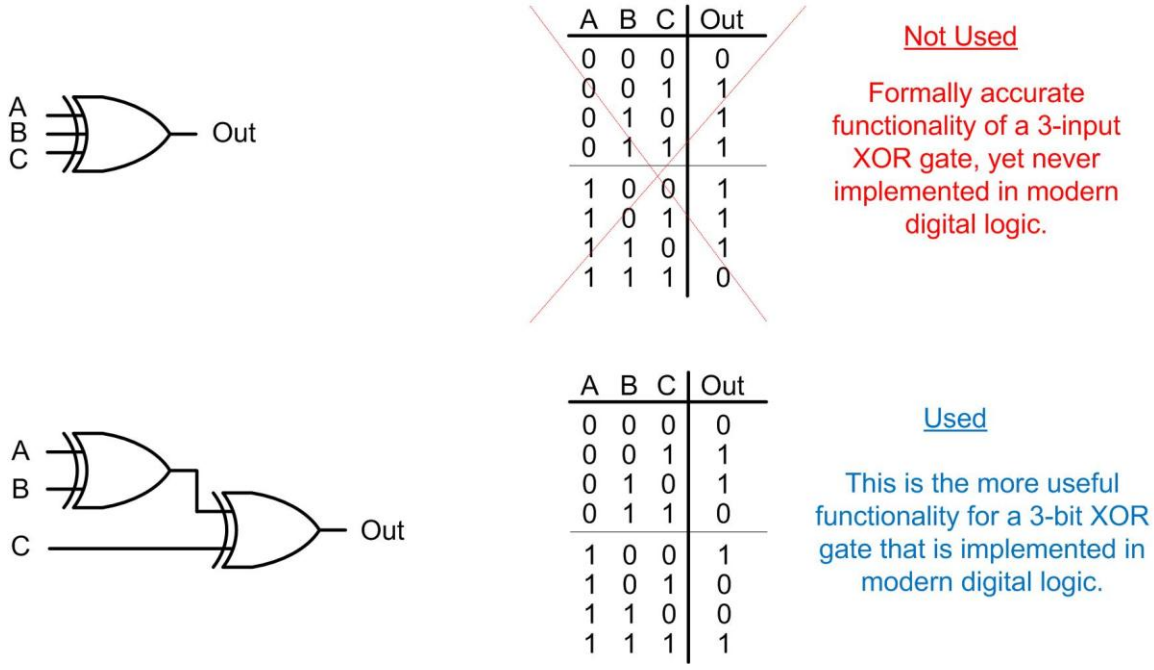


Figure 3.12
3-Input XOR Gate Functionality

3.1.9 The XNOR Gate

The exclusive-NOR gate is identical to the XOR gate with the exception that the output is inverted. This gate is also called an *equivalence* gate because for the 2-input configuration, its output will be true when the **input codes are equivalent to one another**. The symbol for an XNOR gate is an XOR gate with an inversion bubble on the output. The logic expression for an XNOR gate is the same as an XOR gate but with an inversion bar over the entire operation. The logic symbol, truth table, logic function and logic waveform for a 2-input XNOR gate are given in **Figure** 3.13. XNOR gates can have any number of inputs with the operation of an n-bit XNOR gate following the rule that the output is always the inversion of an n-bit XOR operation (e.g., the output is true if there are an ODD number of 1's on the inputs).

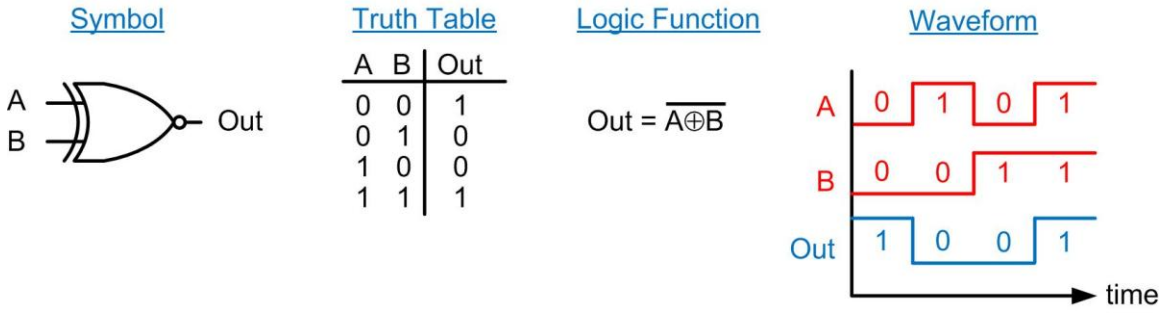


Figure 3.13
2-Input XNOR Gate Symbol, Truth Table, Logic Function and Logic Waveform

3.2 Digital Circuit Operation

Now we turn our attention to the physical hardware that is used to build the basic gates just described and how electrical quantities are used to represent and communicate the binary values 1 and 0. We begin by looking at digital signaling. Digital signaling refers to how binary codes are generated and transmitted successfully between two digital circuits using electrical quantities (e.g., voltage and current). Consider the digital system shown in **Figure** 3.14. In this system, the sending circuit generates a binary code. The sending circuit is called either the *transmitter* (Tx) or *driver*. The transmitter represents the binary code using an electrical quantity such as voltage. The receiving circuit (Rx) observes this voltage and is able to determine the value of the binary code. In this way, 1's and 0's can be communicated between the two digital circuits. The transmitter and receiver are both designed to use the same digital signaling scheme so that they are able to communicate with each other. It should be noted that all digital circuits contain both inputs (Rx) and outputs (Tx) but are not shown in this figure for simplicity.

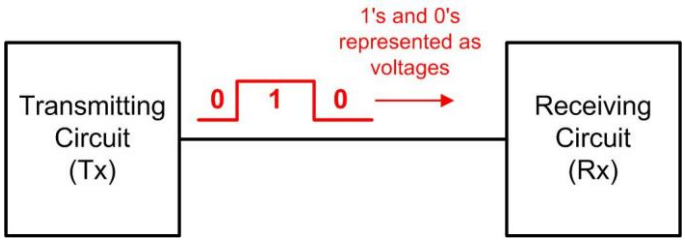


Figure 3.14
Generic Digital Transmitter / Receiver Circuit

3.2.1 Logic Levels

A *logic level* is the term to describe all possible states that a signal can have. We will focus explicitly on circuits that represent binary values so these will only have two finite states (1 and 0). To begin, we define a simplistic model of how to represent the binary codes using an electrical quantity. This model uses a voltage threshold (V_{th}) to represent the switching point between the binary codes. If the voltage of the signal (V_{sig}) is above this threshold, it is considered a *logic HIGH*. If the voltage is below this threshold, it is considered a *logic LOW*. A graphical depiction of this is shown in **Figure** 3.15. The terms HIGH and LOW are used to describe which logic level corresponds to the *higher* or *lower* voltage.

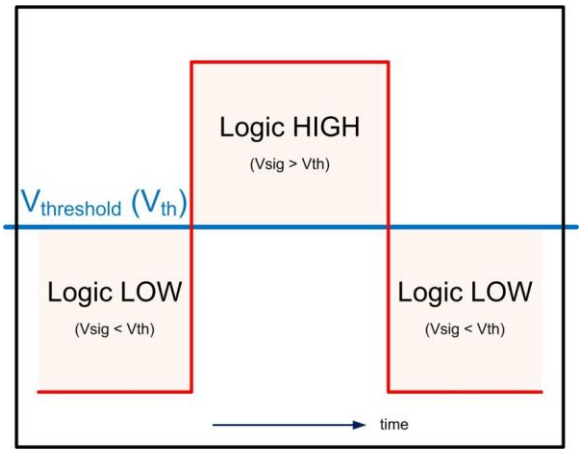


Figure 3.15
Definition of Logic HIGH and LOW

It is straight forward to have the HIGH level correspond to the binary code 1 and the LOW level correspond to the binary code 0; however, it is equally valid to have the HIGH level correspond to the binary code 0 and the LOW level correspond to the binary code 1. As such, we need to define how the logic levels HIGH and LOW map to the binary codes 1 and 0. We define two types of digital assignments, Positive Logic and Negative Logic. In **Positive Logic**, the logic HIGH level represents a binary 1 and the logic LOW level represents a binary 0. In **Negative Logic**, the logic HIGH level represents a binary 0 and the logic LOW level represents a binary 1. There are certain types of digital circuits that benefit from using negative logic; however, we will focus specifically on systems that use positive logic since it is more intuitive when learning digital design for the first time. The transformation between positive and negative logic is straight forward and will be covered in chapter 4.

Logic Level	Logic Value	
	Positive Logic	Negative Logic
LOW	0	1
HIGH	1	0

Table 3.1
Definition of Positive and Negative Logic

3.2.2 Output DC Specifications

Transmitting circuits provide specifications on the range of output voltages (V_O) that they are guaranteed to provide when outputting a logic 1 or 0. These are called the DC output specifications. There are four DC voltage specifications that specify this range: $V_{OH\text{-}max}$, $V_{OH\text{-}min}$, $V_{OL\text{-}max}$, and $V_{OL\text{-}min}$. The $V_{OH\text{-}max}$ and $V_{OH\text{-}min}$ specifications provide the range of voltages the transmitter is guaranteed to provide when outputting a logic HIGH (or logic 1 when using positive logic). The $V_{OL\text{-}max}$ and $V_{OL\text{-}min}$ specifications provide the range of voltages the transmitter is guaranteed to provide when outputting a logic LOW (or logic 0 when using positive logic). In the subscripts for these specifications, the "O" signifies "output" and the "L" or "H" signifies "LOW" or "HIGH" respectively.

The maximum amount of current that can flow through the transmitter's output (I_O) is also specified. The specification $I_{OH\text{-}max}$ is the maximum amount of current that can flow through the transmitter's output when sending a logic HIGH. The specification $I_{OL\text{-}max}$ is the maximum amount of current that can flow through transmitter's output when sending a logic LOW. When the maximum output currents are violated, it usually damages the part. Manufactures will also provide a *recommended* amount of current for I_O that will guarantee the specified operating parameters throughout the life of the part. **Figure** 3.16 shows a graphical depiction of these DC specifications. When the transmitter output is providing current to the receiving circuit (a.k.a., the *load*), it is said to be ***sourcing*** current. When the transmitter output is drawing current from the receiving circuit, it is said to be ***sinking*** current. In most cases, the transmitter sources current when driving a logic HIGH and sinks current when driving a logic LOW.

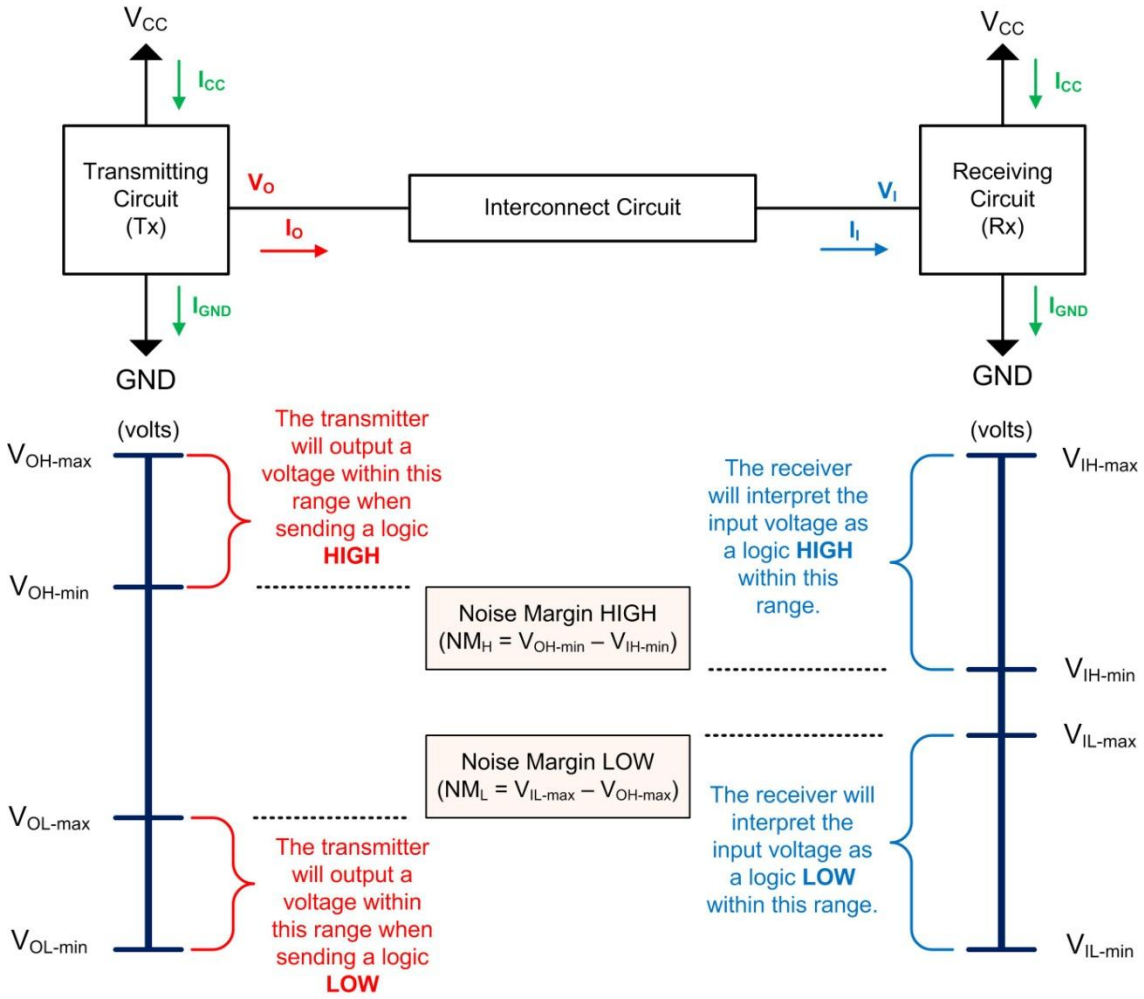


Figure 3.16
DC Specifications of a Digital Circuit

3.2.3 Input DC Specifications

Receiving circuits provide specifications on the range of input voltages (V_I) that they will interpret as either a logic HIGH or LOW. These are called the DC input specifications. There are four DC voltage specifications that specify this range: $V_{IH\text{-}max}$, $V_{IH\text{-}min}$, $V_{IL\text{-}max}$, and $V_{IL\text{-}min}$. The $V_{IH\text{-}max}$ and $V_{IH\text{-}min}$ specifications provide the range of voltages that the receiver will interpret as a logic HIGH (or logic 1 when using positive logic). The $V_{IL\text{-}max}$ and $V_{IL\text{-}min}$ specifications provide the range of voltages that the receiver will interpret as a logic LOW (or logic 0 when using positive logic). In the subscripts for these specifications, the "I" signifies "input".

The maximum amount of current that the receiver will draw (or take in) when connected is also specified. The specification $I_{IH\text{-}max}$ is the maximum amount of current that the receiver will draw when it is being driven with a logic HIGH. The specification $I_{IL\text{-}max}$ is the maximum amount of current that the receiver will draw when it is being driven with a logic LOW. Again, **Figure** 3.16 shows a graphical depiction of these DC specifications.

3.2.4 Noise Margins

For digital circuits that are designed to operate with each other, the $V_{OH\text{-}max}$ and $V_{IH\text{-}max}$ specifications have equal voltages. Similarly, the $V_{OL\text{-}min}$ and $V_{IL\text{-}min}$ specifications have equal voltages. The $V_{OH\text{-}max}$ and $V_{OL\text{-}min}$ output specifications represent the *best case* scenario for digital signaling as the transmitter is sending the largest (or smallest) signal possible. If there is no loss in the interconnect between the transmitter and receiver, the full voltage levels will arrive at the receiver and be interpreted as the correct logic states (HIGH or LOW).

The *worst case* scenario for digital signaling is when the transmitter outputs its levels at $V_{OH\text{-}min}$ and $V_{OL\text{-}max}$. These levels represent the furthest away from an *ideal* voltage level that the transmitter can send to the receiver and are susceptible to loss and noise that may occur in the interconnect system. In order to compensate for potential loss or noise, digital circuits have a pre-defined amount of margin built into their worst-case specifications. Let's take the worst case example of a transmitter sending a logic HIGH at the level $V_{OH\text{-}min}$. If the receiver was designed to have $V_{IH\text{-}min}$ (e.g., the lowest voltage that would still be interpreted as a logic 1) equal to $V_{OH\text{-}min}$, then if even the smallest amount of the output signal was attenuated as it traveled through the interconnect, it would arrive at the receiver below $V_{IH\text{-}min}$ and would not be interpreted as a logic 1. Since there will always be some amount of loss in any interconnect system, the specifications for $V_{IH\text{-}min}$ is always less than $V_{OH\text{-}min}$. The difference between these two quantities is called the **Noise Margin**. More specifically, it is called the Noise Margin HIGH (or NM_H) to signify how much margin is built into the Tx/Rx circuit when communicating a logic 1. Similarly, the V_{IL}-max specification is always higher than the $V_{OL\text{-}max}$ specification to account for any voltage added to the signal in the interconnect. The difference between these two quantities is called the Noise Margin LOW (or NM_L) to signify how much margin is built into the Tx/Rx circuit when communicating a logic 0. Noise margins are always specified as positive quantities, thus the order of the subtrahend and minuend in these differences.

$$\mathbf{NM_H = V_{OH\text{-}min} - V_{IH\text{-}min}}$$
$$\mathbf{NM_L = V_{IL\text{-}max} - V_{OL\text{-}max}}$$

Figure 3.16 shows the graphical depiction of the noise margins. Notice in this figure that there is a region of voltages that the receiver will not interpret as either a HIGH or LOW. This region lies between the $V_{IH\text{-}min}$ and $V_{IL\text{-}max}$ specifications. This is the **uncertainly region** and should be avoided. Signals in this region will cause the receiver's output to go to an unknown voltage. Digital transmitters are designed to transition between the LOW and HIGH states quickly enough so that the receiver does not have time to react to the input being in the uncertainty region.

3.2.5 Power Supplies

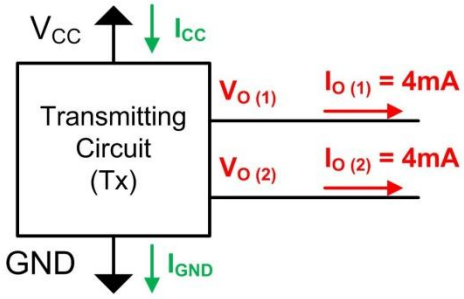
All digital circuits require a power supply voltage and a ground. There are some types of digital circuits that may require multiple power supplies. For simplicity, we will focus on digital circuits that only require a single power supply voltage and ground. The power supply voltage is commonly given the abbreviations of either V_{CC} or V_{DD}. The "CC" or "DD" have to do with how the terminals of the transistors inside of the digital circuit are connected (e.g., "collector to collector" or "drain to drain"). Digital circuits will specify the required power supply voltage. Ground is considered an ideal 0v. Digital circuits will also specify the maximum amount of DC current that can flow through the V_{CC} (I_{CC}) and GND (I_{GND}) pins before damaging the part.

There are two components of power supply current. The first is the current that is required for the functional operation of the device. This is called the *quiescent current* (I_q). The second component of the power supply current is the output currents (I_O). Any current that flows out of a digital circuit must also flow into it. When a transmitting circuit sources current to a load on its output pin, it must bring in that same amount of current on another pin. This is accomplished using the power supply pin (V_{CC}). Conversely, when a transmitting circuit sinks current from a load on its output pin, an equal amount of current must exit the circuit on a different pin. This is accomplished using the GND pin. This means that the amount of current flowing through the V_{CC} and GND pins will vary depending on the logic states that are being driven on the outputs. Since a digital circuit may contain numerous output pins, the maximum amount of current flowing through the V_{CC} and GND pins can scale quickly and care must be taken not to damage the device.

Figure 3.17 gives an example of calculating the I_{CC} and I_{GND} currents when sourcing multiple loads. **Figure** 3.18 gives an example of calculating the I_{CC} and I_{GND} currents when both sourcing and sinking loads.

Example: Find I_{CC} and I_{GND} for the following driver configuration:

Given: The driver is specified to have a quiescent current of 1mA and is driving a logic HIGH on two of its output pins. Each of the two loads on the output pins is being sourced with 4mA of current from the driver.

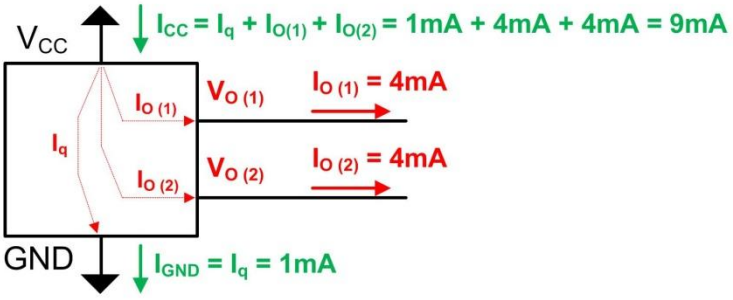


Solution: The current into the device must equal the current out of the device. The quiescent current of 1mA is used for the functional operation of the transistors within the transmitter and will flow into the device through the V_{CC} pin and out of the device on the GND pin. The output currents that are being sourced by the driver exit the circuit on the two output pins $V_{O(1)}$ and $V_{O(2)}$. An equal amount of current must also flow into the device ($I_{O(1)} + I_{O(2)} = 8mA$), this current enters the device on the V_{CC} pin. This means the total amount of current flowing into the circuit on the V_{CC} pin is:

$$I_{CC} = I_q + I_{O(1)} + I_{O(2)} = 1mA + 4mA + 4mA = 9mA$$

The total amount of current flowing out of the circuit on the GND pin is simply the quiescent current I_q.

$$I_{GND} = I_q = 1mA$$



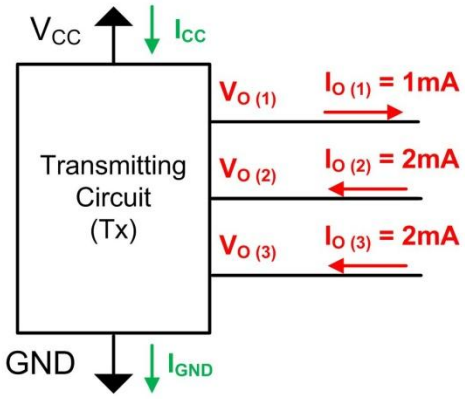
Check: Does the total amount of current entering the circuit equal the total amount of current existing the circuit?

Yes, there is 9mA entering the circuit through the V_{CC} pin. There is also 9mA exiting the circuit using the $V_{O(1)}$, $V_{O(2)}$ and GND pins.

Figure 3.17
Example: Supply Current when Sourcing Multiple Loads

Example: Find I_{CC} and I_{GND} for the following driver configuration:

Given: The driver is specified to have a quiescent current of 0.5mA and is driving a logic HIGH on one of its output pins and a logic LOW on two of its output pins. The driver is sourcing 1mA when driving a HIGH and sinking 2mA when driving a LOW.



Solution: The current into the device must equal the current out of the device. The quiescent current of 0.5mA enters the circuit on the V_{CC} pin and exits on the GND pin. The output current for $V_{O(1)}$ enters the circuit on the V_{CC} pin and exits the circuit on the $V_{O(1)}$ pin. The output current for $V_{O(2)}$ and $V_{O(3)}$ enters the circuit on the $V_{O(2)}$ and $V_{O(3)}$ pins and exits the circuit on the GND pin. This means the total amount of current flowing into the circuit on the V_{CC} pin is:

$$I_{CC} = I_q + I_{O(1)} = 0.5mA + 1mA = 1.5mA$$

The total amount of current flowing out of the circuit on the GND pin is the quiescent current I_q plus the current being sunk from the pins $V_{O(2)}$ and $V_{O(3)}$:

$$I_{GND} = I_q + I_{O(2)} + I_{O(3)} = 0.5mA + 2mA + 2mA = 4.5mA$$

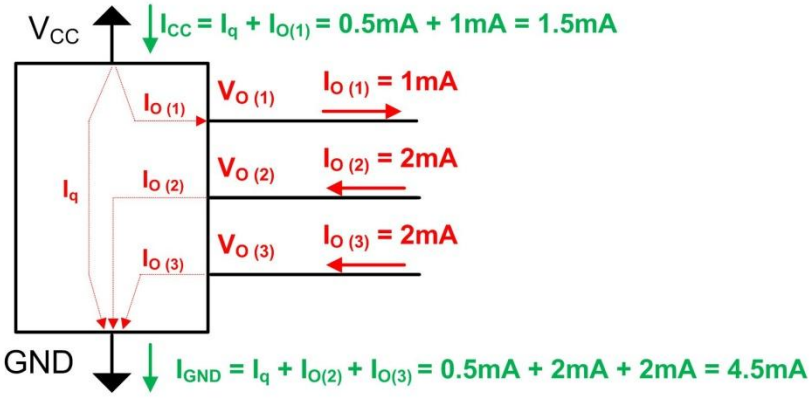


Figure 3.18
Example: Supply Current when Both Sourcing and Sinking Loads

3.2.6 Switching Characteristics

Switching characteristics refer to the transient behavior of the logic circuits. The first group of switching specifications characterize the *propagation delay* of the gate. The propagation delay is the time it takes for the output to respond to a change on the input. The propagation delay is formally defined as the time it takes from the point at which the input has transitioned to 50% of its final value to the point at which the output as transitioned to 50% of its final value. The initial and final voltages for the input are defined to be GND and V_{CC} while the output initial and final voltages are defined to be V_{OL} and V_{OH}. Specifications are given for the propagation delay when transitioning from a LOW to HIGH (t_{PLH}) and from a HIGH to LOW (t_{PHL}). When these specifications are equal, the values are often given as a single specification of t_{pd}. These specifications are shown graphically in **Figure** 3.19.

The second group of switching specifications characterize how quickly the output switches between states. The *transition time* is defined as the time it takes for the output to transition from 10% to 90% of the output voltage range. The *rise time* (t_r) is the time it takes for this transition when going from a LOW to HIGH and the *fall time* (t_f) is the time it takes for this transition when going from a HIGH to LOW. When these specifications are equal, the values are often given as a single specification of t_t. These specifications are shown graphically in **Figure** 3.19.

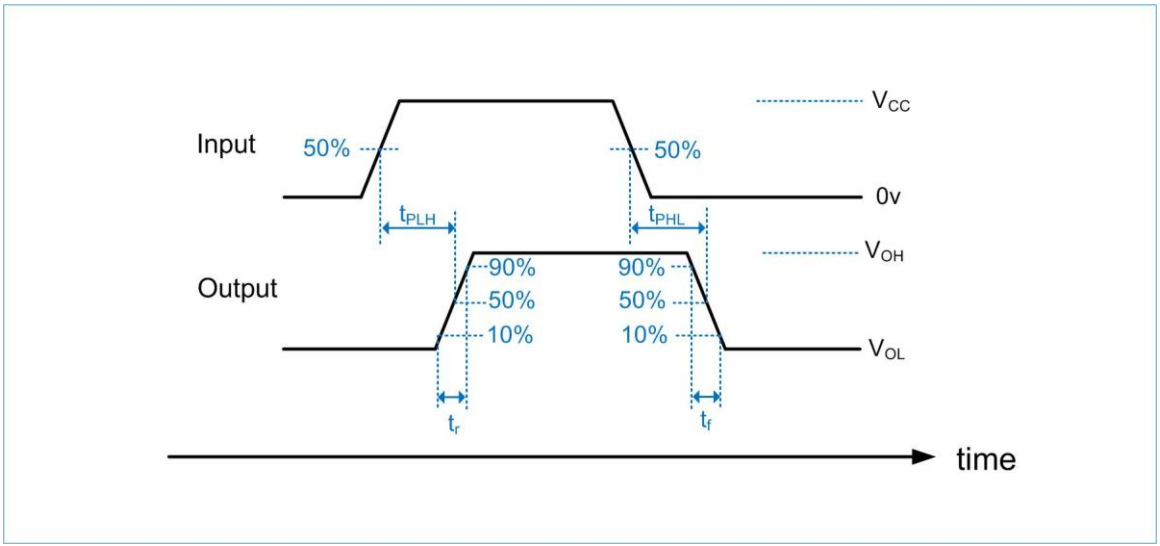
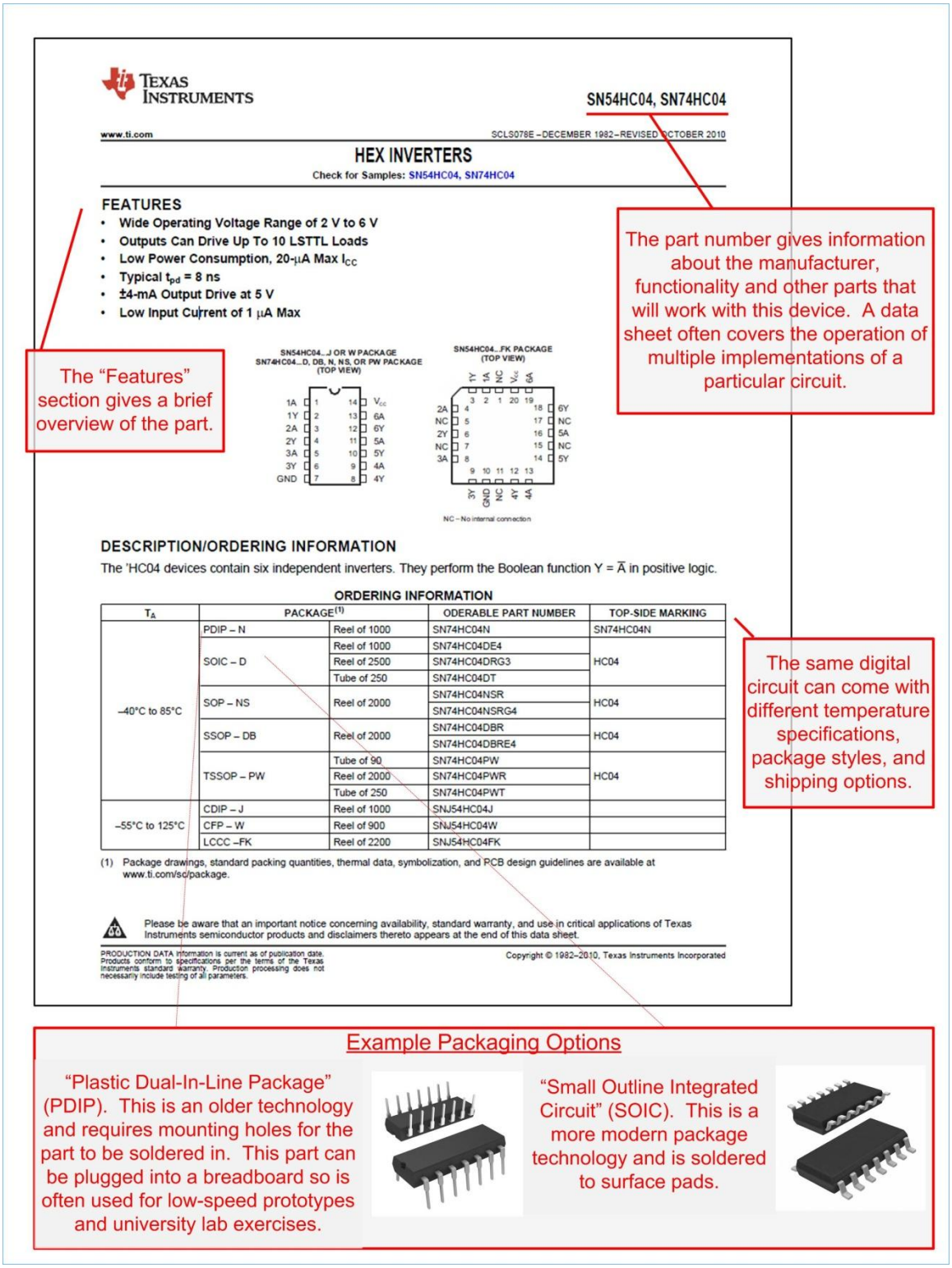


Figure 3.19
Definition of Switching Characteristics

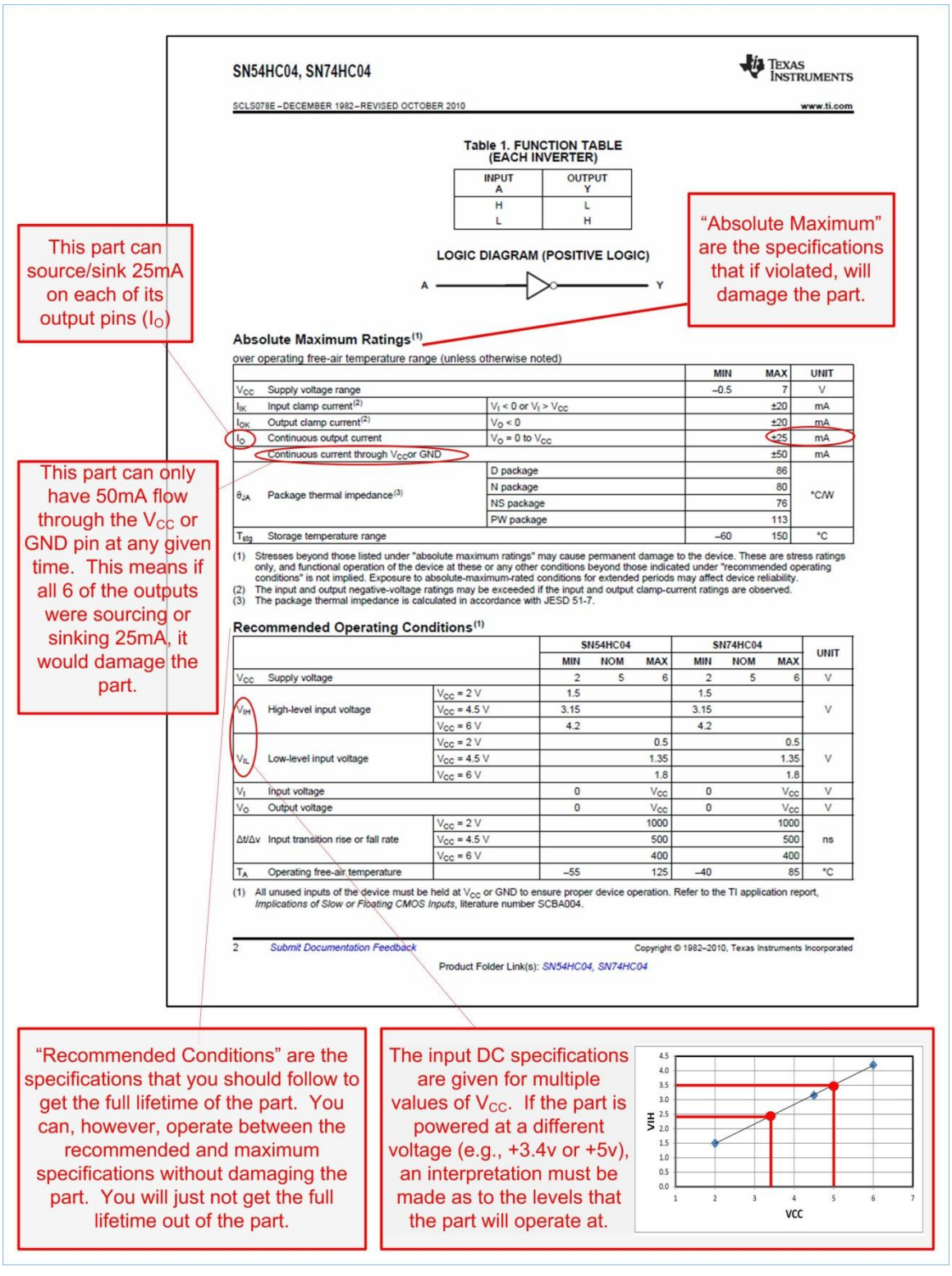
3.2.7 Datasheets

The specifications for a particular part are given in its **datasheet**. The datasheet contains all of the operating conditions about a part in addition to functional information such as package geometries and pin assignments. The datasheet is usually the first place a designer will look when selecting a part. Figure 3.20**,** Figure 3.21 and Figure 3.22 show excerpts from an example data sheet highlighting some of the specifications just covered.



T_A	PACKAGE(1)		ODERABLE PART NUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Reel of 1000	SN74HC04N	SN74HC04N
	SOIC – D	Reel of 1000	SN74HC04DE4	HC04
		Reel of 2500	SN74HC04DRG3	
		Tube of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
			SN74HC04NSRG4	
	SSOP – DB	Reel of 2000	SN74HC04DBR	HC04
			SN74HC04DBRE4	
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Tube of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Reel of 1000	SNJ54HC04J	
	CFP – W	Reel of 900	SNJ54HC04W	
	LCCC –FK	Reel of 2200	SNJ54HC04FK	

Figure 3.20
Example Datasheet Excerpt (1)



SN54HC04, SN74HC04

TEXAS INSTRUMENTS

SCLS078E – DECEMBER 1982 – REVISED OCTOBER 2010

www.ti.com

Table 1. FUNCTION TABLE (EACH INVERTER)

INPUT A	OUTPUT Y
H	L
L	H

LOGIC DIAGRAM (POSITIVE LOGIC)

A → Y

Absolute Maximum Ratings(1)

over operating free-air temperature range (unless otherwise noted)

			MIN	MAX	UNIT
V_{CC}	Supply voltage range		–0.5	7	V
I_{IK}	Input clamp current(2)	$V_I < 0$ or $V_I > V_{CC}$		±20	mA
I_{OK}	Output clamp current(2)	$V_O < 0$		±20	mA
I_O	Continuous output current	$V_O = 0$ to V_{CC}		±25	mA
	Continuous current through V_{CC} or GND			±50	mA
θ_{JA}	Package thermal impedance(3)	D package		86	°C/W
		N package		80	
		NS package		76	
		PW package		113	
T_{stg}	Storage temperature range		–60	150	°C

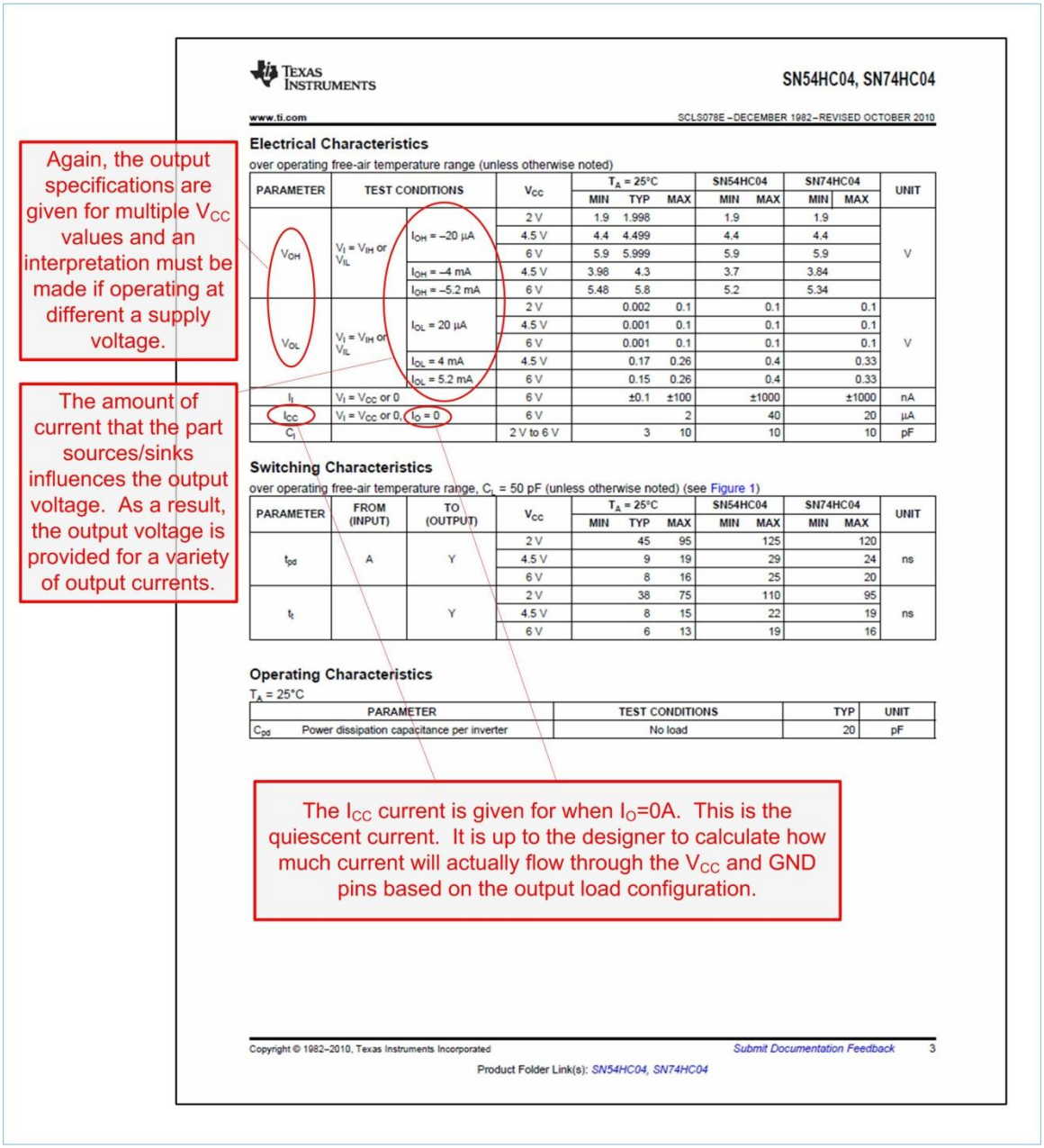
(1) Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.
(2) The input and output negative-voltage ratings may be exceeded if the input and output clamp-current ratings are observed.
(3) The package thermal impedance is calculated in accordance with JESD 51-7.

Recommended Operating Conditions(1)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise or fall rate	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

(1) All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

Figure 3.21
Example Datasheet Excerpt (2)



TEXAS INSTRUMENTS

SN54HC04, SN74HC04

www.ti.com

SCLS078E – DECEMBER 1982 – REVISED OCTOBER 2010

Again, the output specifications are given for multiple V_{CC} values and an interpretation must be made if operating at different a supply voltage.

The amount of current that the part sources/sinks influences the output voltage. As a result, the output voltage is provided for a variety of output currents.

Electrical Characteristics

over operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A = 25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	V_I = V_{IH} or V_{IL}	I_{OH} = –20 µA	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		I_{OH} = –4 mA	4.5 V	3.98	4.3		3.7		3.84		
		I_{OH} = –5.2 mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	V_I = V_{IH} or V_{IL}	I_{OL} = 20 µA	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		I_{OL} = 4 mA	4.5 V		0.17	0.26		0.4		0.33	
		I_{OL} = 5.2 mA	6 V		0.15	0.26		0.4		0.33	
I_I	V_I = V_{CC} or 0		6 V			±0.1		±1000		±1000	nA
I_{CC}	V_I = V_{CC} or 0,	I_O = 0	6 V			2		40		20	µA
C_i			2 V to 6 V		3	10		10		10	pF

Switching Characteristics

over operating free-air temperature range, C_L = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		125		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

Operating Characteristics

T_A = 25°C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

The I_{CC} current is given for when I_O=0A. This is the quiescent current. It is up to the designer to calculate how much current will actually flow through the V_{CC} and GND pins based on the output load configuration.

Figure 3.22
Example Datasheet Excerpt (3)

3.3 Logic Families

It is apparent from the prior discussion of operating conditions that digital circuits need to have comparable input and output specifications in order to successfully communicate with each other. If a transmitter outputs a logic HIGH as +3.4v and the receiver needs a logic HIGH to be above +4v to be successfully interpreted as a logic HIGH, then these two circuits will not be able to communicate. In order to address this interoperability issue, digital circuits are grouped into *Logic Families*. A logic family is a group of parts that all adhere to a common set of specifications so that they work together. The logic family is given a specific name and once the specifications are agreed upon, different manufactures produce parts that work within the particular family. Within a logic family, parts will all have the same power supply requirements and DC input/output specifications such that if connected *directly*, they will be able to successfully communicate with each other. The phrase "connected directly" is emphasized because it is very possible to insert an interconnect circuit between two circuits within the same logic family and alter the output voltage enough so that the receiver will not be able to interpret the correct logic level. Analyzing the effect of the interconnect circuit is part of the digital design process. There are many logic families that exist (up to a 100 different types!) and more emerge each year as improvements are made to circuit fabrication processes that create smaller, faster and lower power circuits.

3.3.1 Complementary Metal Oxide Semiconductors (CMOS)

The first group of logic families we will discuss is called Complementary Metal Oxide Semiconductors, or CMOS. This is currently the most popular group of logic families for digital circuits implemented on the same integrated circuit (IC). An **integrated circuit** is where the entire circuit is implemented on a single piece of semiconductor material (or chip). The IC can contain transistors, resistors, capacitors, inductors, wires and insulators. Modern integrated circuits can contain billions of devices and meters of interconnect. The opposite of implementing the circuit on an integrated circuit is to use **discrete components**. Using discrete components refers to where every device (transistor, resistor, etc…) is its own part and is wired together externally using either a printed circuit board (PCB) or jumper wires as on a breadboard. The line between ICs and discrete parts has blurred in the past decades because modern discrete parts are actually fabricated as an IC and regularly contain multiple devices (e.g., 4 logic gates per chip). Regardless, the term *discrete* is still used to describe components that only contain a *few* components where the term IC typically refers to a much larger system that is custom designed.

The term CMOS comes from the use of particular types of transistors to implement the digital circuits. The transistors are created using a Metal Oxide Semiconductor (MOS) structure. These transistors are turned on or off based on an electric field, so they are given the name Metal Oxide Semiconductor *Field Effect* Transistors, or MOSFETs. There are two transistors that can be built using this approach that operate *complementary* to each other, thus the term *Complementary Metal Oxide Semiconductors.* To understand the basic operation of CMOS logic, we begin by treating the MOSFET transistors as ideal switches. This allows us to understand the basic functionality without diving into the detailed electronic analysis of the transistors.

3.3.1.1 CMOS Operation

In CMOS, there is a single power supply (V_{CC} or V_{DD}) and a single ground (GND or sometimes called V_{SS}). The maximum input and output DC specifications are equal to the power supply ($V_{CC} = V_{OH\text{-}max} = V_{IH\text{-}max}$). The minimum input and output DC specification are equal to ground (GND = 0v = $V_{OL\text{-}min} = V_{IL\text{-}min}$). In this way, using CMOS simplifies many of the specifications. If you state that you are using "CMOS with a +3.4v power supply", you are inherently stating that $V_{CC} = V_{OH\text{-}max} = V_{IH\text{-}max}$ = +3.4v and that

$V_{OL\text{-}min}$ = $V_{IL\text{-}min}$ = 0v. Many times the name of the logic family will be associated with the power supply voltage. For example, a logic family may go by the name "+3.3v CMOS" or "+2.5v CMOS". These names give a first level description of the logic family operation, but more details about the operation must be looked up in the datasheet.

These are two types of transistors used in CMOS. The transistors will be closed or open based on an input logic level. The first transistor is called an N-type MOSFET, or **NMOS**. This transistor will be closed when it receives a logic 1 on its input and open when it receives a logic 0. The second transistor is called a P-type MOSFET, or **PMOS**. This transistor will be closed when it receives a logic 0 on its input and open when it receives a logic 1. The type of transistor (e.g., P-type or N-type) has to do with the type of semiconductor material used to fabricate the device but a description of device physics is beyond the scope of this material at this point. **Figure** 3.23 shows the symbols for the PMOS and NMOS transistors and their switch level equivalents.

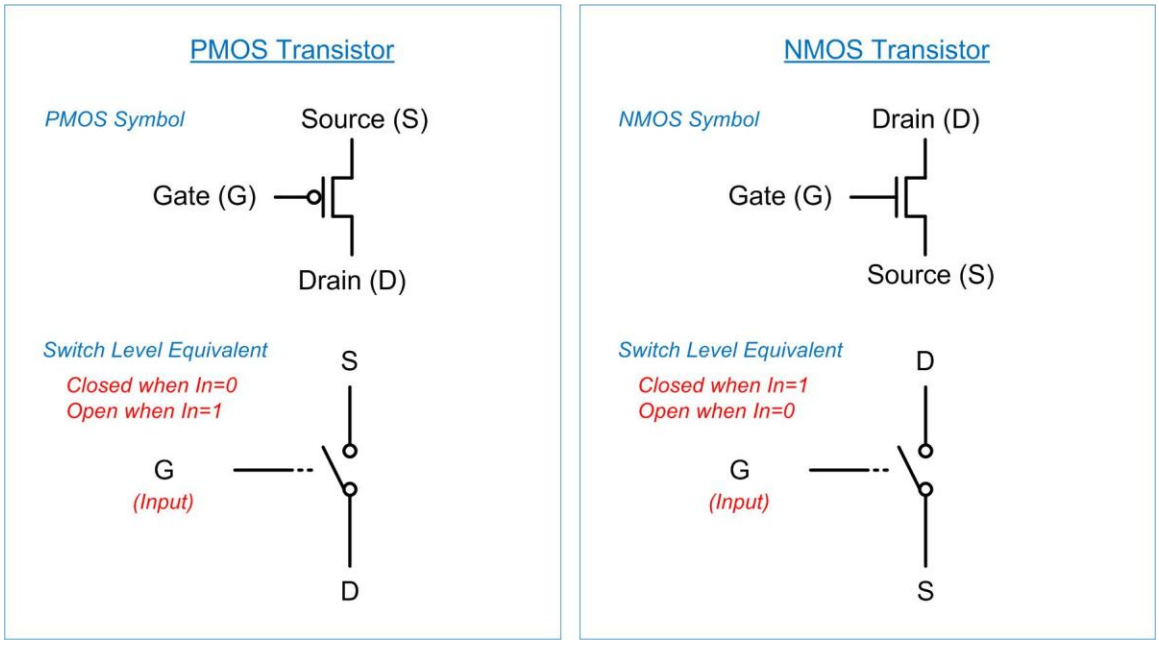


Figure 3.23
PMOS and NMOS Transistors

The basic operation of CMOS is that when driving a logic HIGH, the switches are used to connect the output to the power supply (V_{CC}) and when driving a logic LOW the switches are used to connect the output to GND. In CMOS, V_{CC} is considered an ideal logic HIGH and GND is considered an ideal logic LOW. The design of the circuit must never connect the output to V_{CC} and GND at the same time or else the device itself will be damaged due to the current flowing directly from V_{CC} to GND through the transistors. Due to the device physics of the MOSFETS, PMOS transistors are used to form the network that will connect the output to V_{CC} (a.k.a., the pull-up network) and NMOS transistors are used to form the network that will connect the output to GND (a.k.a., the pull-down network). Since PMOS transistors are closed when the input is a 0 (thus providing a logic HIGH on the output) and NMOS transistors are closed when the input is a 1 (thus providing a logic LOW on the output), CMOS implements negative logic gates. This means CMOS can implement inverters, NAND and NOR gates but not buffers, AND and OR gates directly. In order to create a CMOS AND gate, the circuit would implement a NAND gate followed by an inverter and similarly for an OR gate and buffer.

3.3.1.2 CMOS Inverter

Let's now look at how we can use these transistors to create a CMOS inverter. Consider the transistor arrangement shown in **Figure** 3.24. The inputs of both the PMOS and NMOS are connected together. The PMOS is used to connect the output to V_{CC} and the NMOS is used to connect the output to GND. Since the inputs are connected together and the switches operate in a complementary manner, this circuit ensures that both transistors will never be on at the same time. When In=0, the PMOS switch is closed and the NMOS switch is open. This connects the output directly to V_{CC}, thus providing a logic HIGH on the output. When In=1, the PMOS switch is open and the NMOS switch is closed. This connects the output directly to GND, thus providing a logic LOW. This configuration yields an inverter. This operation is shown graphically in **Figure** 3.25.

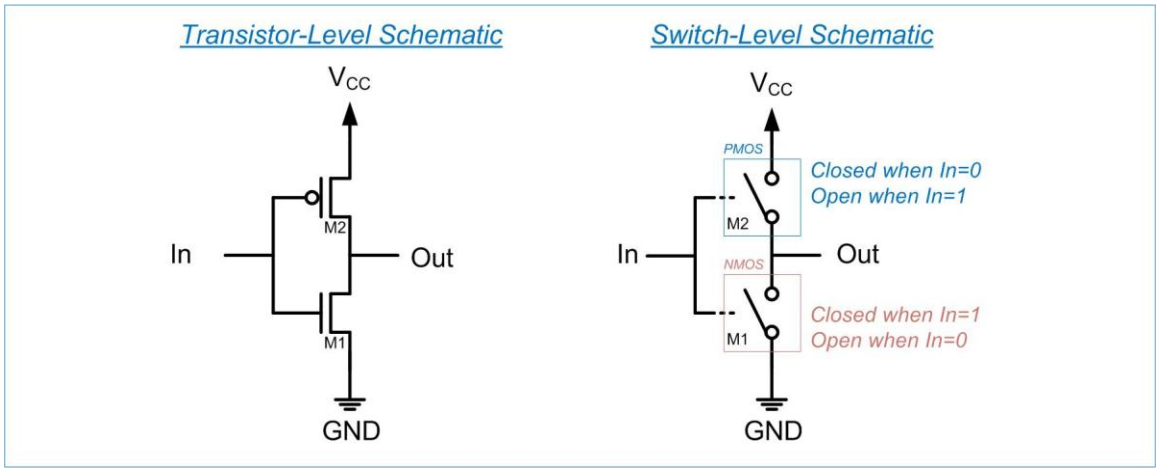


Figure 3.24
CMOS Inverter Schematic

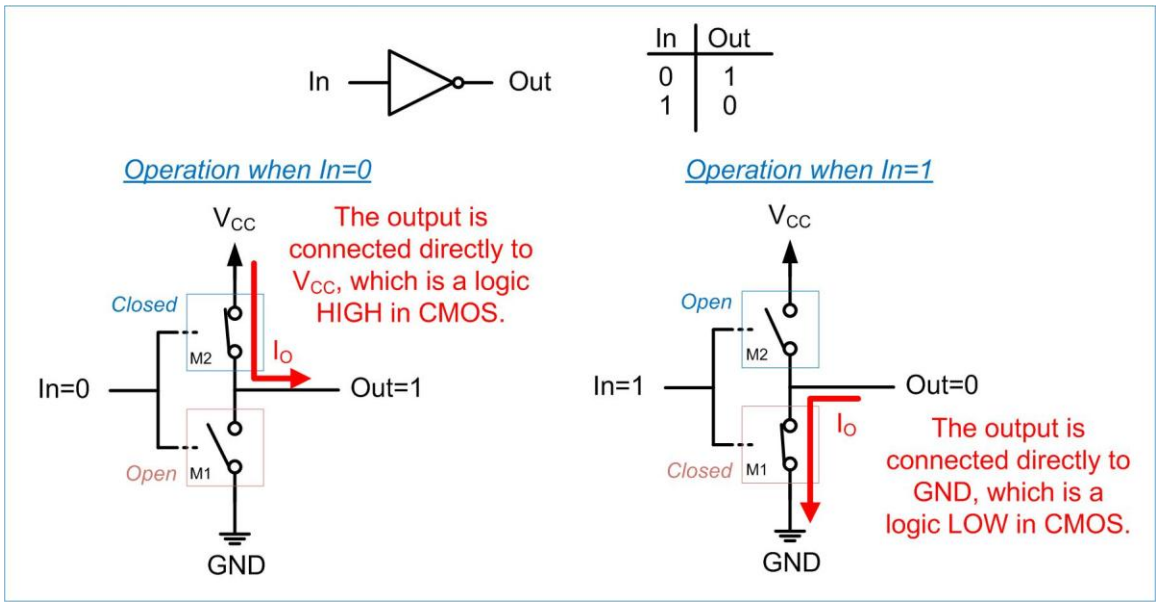


Figure 3.25
CMOS Inverter Operation

3.3.1.3 CMOS NAND Gate

Let's now look at how we use a similar arrangement of transistors to implement a 2-input NAND gate. Consider the arrangement show in **Figure** 3.26. The pull-down network consists of two NMOS transistors in series (M1 and M2) and the pull-up network consists of two PMOS transistors in parallel (M3 and M4). Let's go through each of the input conditions and examine which transistors are on and which are off and how they impact the output. The first input condition is when A=0 and B=0. This condition turns on both M3 and M4 creating two parallel paths between the output and V_{CC}. At the same time, it turns off both M1 and M2 preventing a path between the output and GND. This input condition results in an output that is connected to V_{CC} resulting in a logic HIGH. The second input condition is when A=0 and B=1. This condition turns on M4 in the pull-up network and M2 in the pull-down network. This condition also turns off M3 in the pull-up network and M1 in the pull-down network. Since the pull-up network is a parallel combination of PMOS transistors, there is still a path between the output and V_{CC} through M4. Since the pull-down network is a series combination of NMOS transistors, both M1 and M2 must be on in order to connect the output to GND. This input condition results in an output that is connected to V_{CC} resulting in a logic HIGH. The third input condition is when A=1 and B=0. This condition again provides a path between the output and V_{CC} through M3 and prevents a path between the output and ground by having M2 open. This input condition results in an output that is connected to V_{CC} resulting in a logic HIGH. The final input condition is when A=1 and B=1. In this input condition, both of the PMOS transistors in the pull-up network (M3 and M4) are off preventing the output from being connected to V_{CC}. At the same time, this input turns on both M1 and M2 in the pull-down network connecting the output to GND. This input condition results in an output that is connected to GND resulting in a logic LOW. Based on the resulting output values corresponding to the four input codes, this circuit yields the logic operation of a 2-Input NAND gate. This operation is shown graphically in Figure 3.27**.**

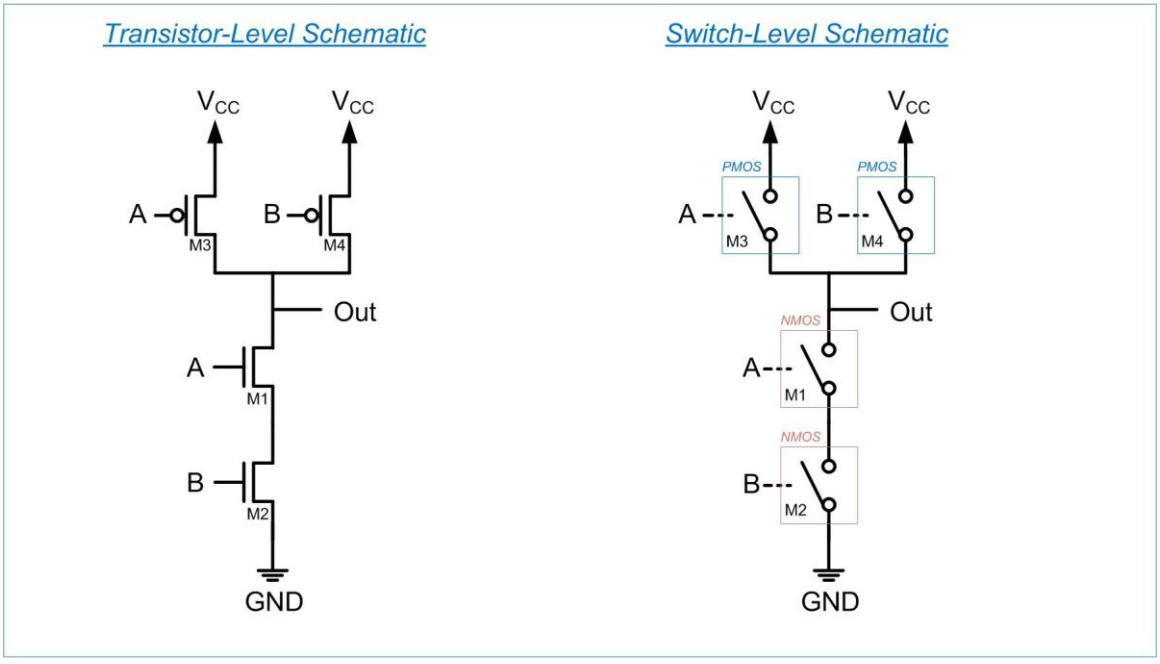


Figure 3.26
CMOS 2-Input NAND Gate Schematic

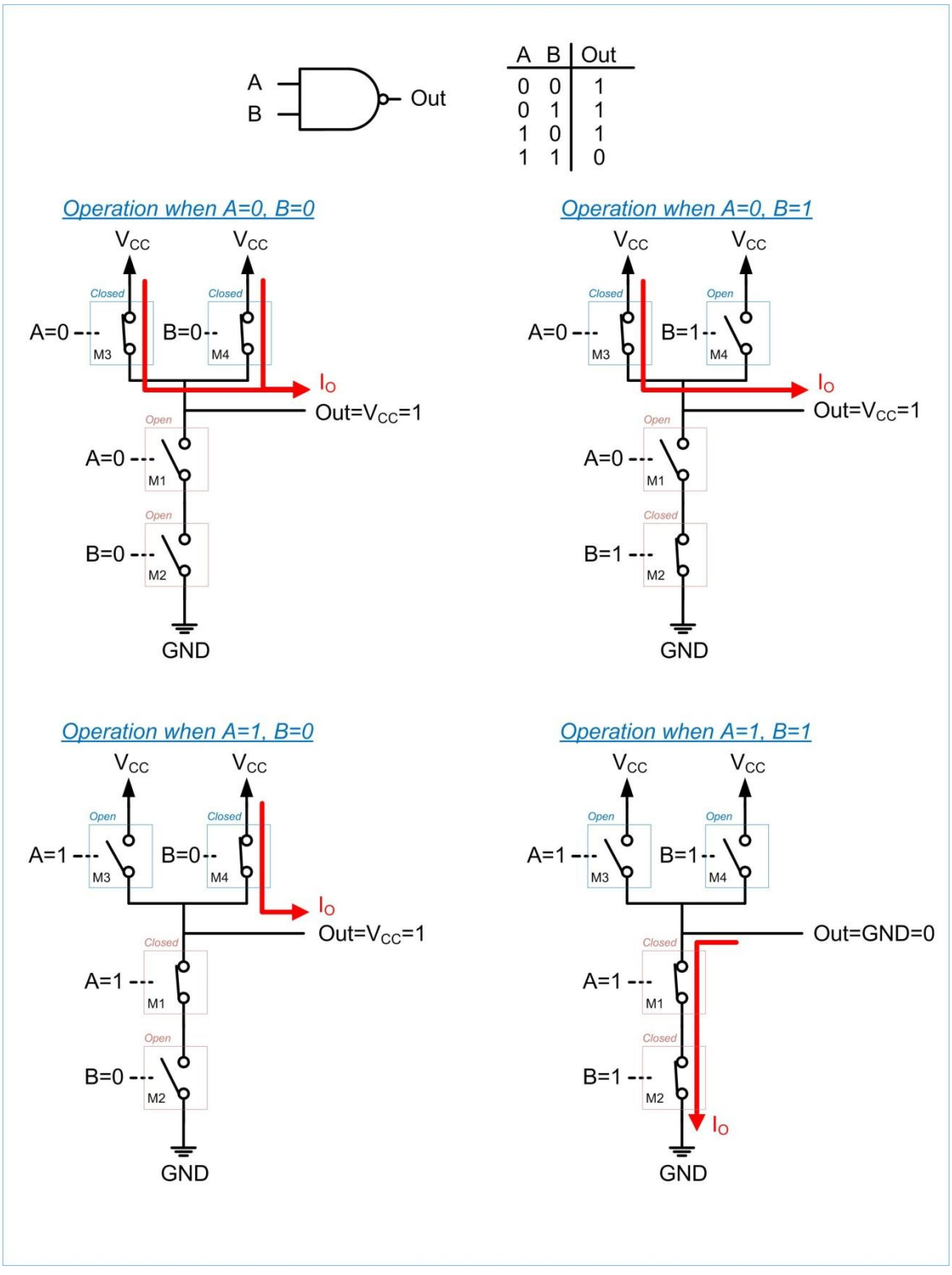


Figure 3.27
CMOS 2-Input NAND Gate Operation

Creating a CMOS NAND gate with more than 2 inputs is accomplished by adding additional PMOS transistors to the pull-up network in parallel and additional NMOS transistors to the pull-down network in series. **Figure** 3.28 shows the schematic for a 3-Input NAND gate. This procedure is followed for creating NAND gates with larger number of inputs.

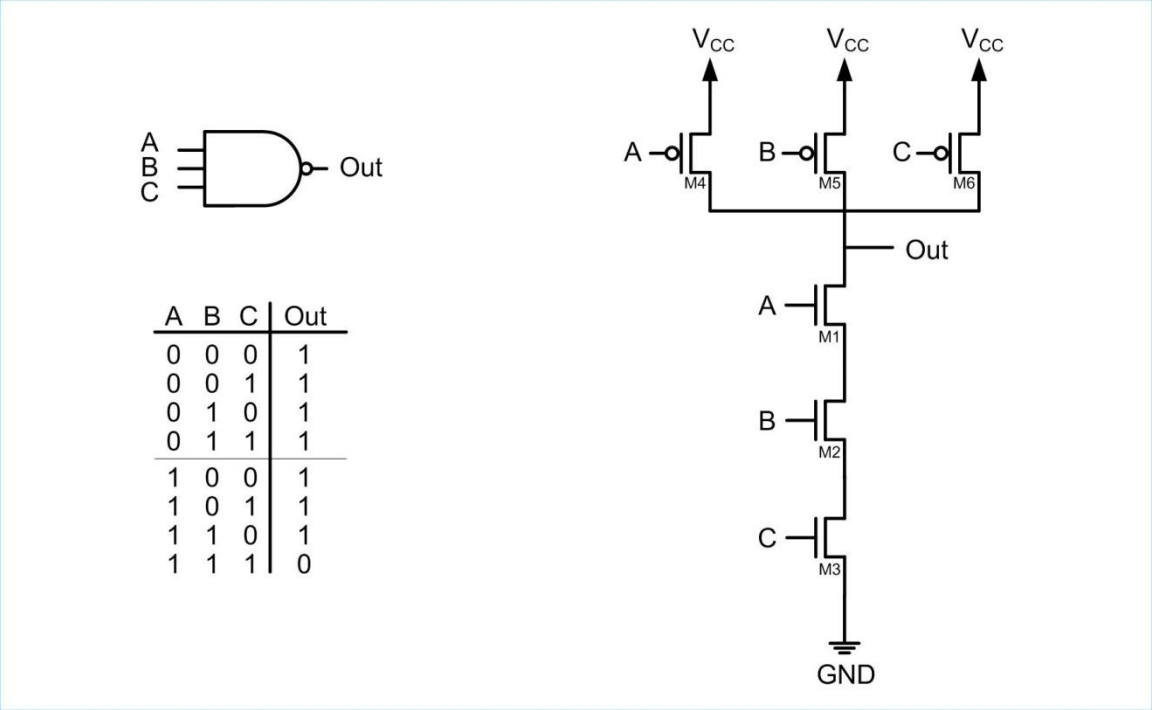


Figure 3.28
CMOS 3-Input NAND Gate Schematic

If the CMOS transistors were ideal switches, the approach of increasing the number of inputs could be continued indefinitely. In reality, the transistors are not ideal switches and there is a limit on how many transistors can be added in series and continue to operate. The limitation has to do with ensuring that each transistor has enough voltage to properly turn on or off. This is a factor in the series network because the drain terminals of the NMOS transistors are not all connected to GND. If a voltage develops across one of the lower transistors (e.g., M3), then it takes more voltage on the input to turn on the next transistor up (e.g., M2). If too many transistors are added in series, then the uppermost transistor in the series may not be able to be turned on or off by the input signals. The number of inputs that a logic gate can have within a particular logic family is called its **fan-in** specification. When a logic circuit requires a number of inputs that exceeds the fan-in specification for a particular logic family, then additional logic gates must be used. For example, if a circuit requires a 5-input NAND gate, but the logic family has a fan-in specifications of 4, this means that the largest NAND gate available only has 4-inputs. The 5-input NAND operation must be accomplished using additional circuit design techniques that use gates with 4 or less inputs. These design techniques will be covered in Chapter 4.

3.3.1.4 CMOS NOR Gate

A CMOS NOR gate is created using a similar topology as a NAND gate with the exception that the pull-up network consists of PMOS transistors in series and the pull-down network that consists of NMOS transistors in parallel. Consider the transistor configuration shown in **Figure** 3.29. The series configuration of the pull-up network will only connect the output to V_{CC} when both inputs are 0.

Conversely, the pull-down network prevents connecting the output to GND when both inputs are 0. When either or both of the inputs are true, the pull-up network is off and the pull-down network is on. This yields the logic function for a NOR gate. This operation is shown graphically **Figure** 3.30. As with the NAND gate, the number of inputs can be increased by adding more PMOS transistors in series in the pull-up network and more NMOS transistors in parallel in the pull-down network. The schematic for a 3-input NOR gate is given in **Figure** 3.31. This approach can be used to increase the number of inputs up until the fan-in specification of the logic family is reached.

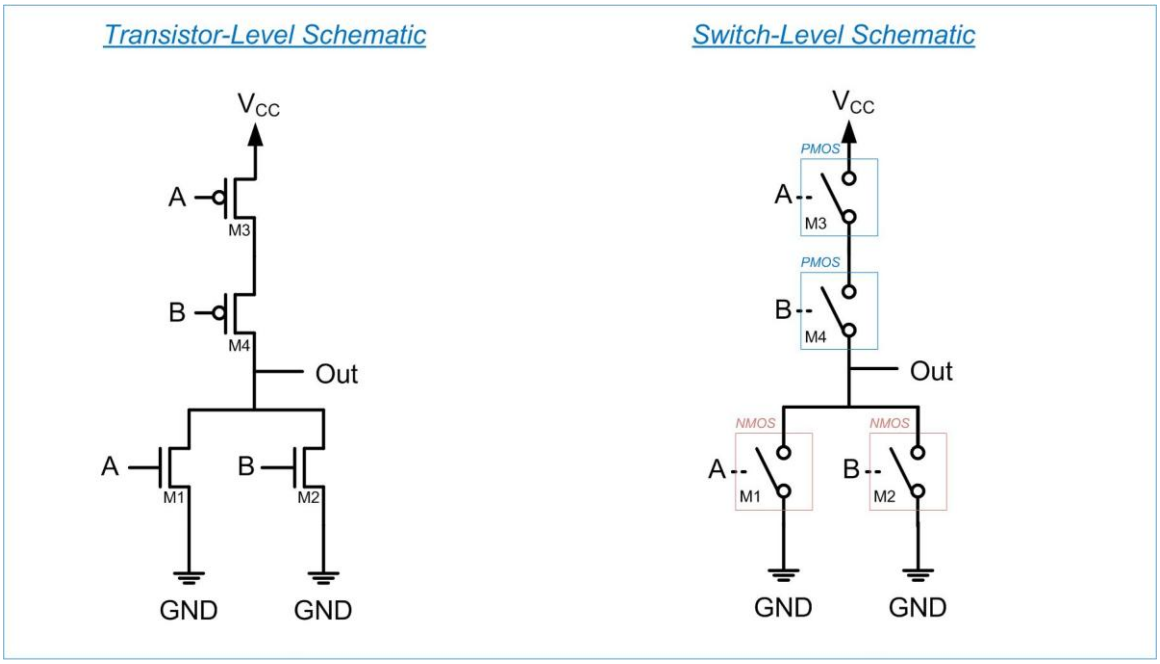


Figure 3.29
CMOS 2-Input NOR Gate Schematic

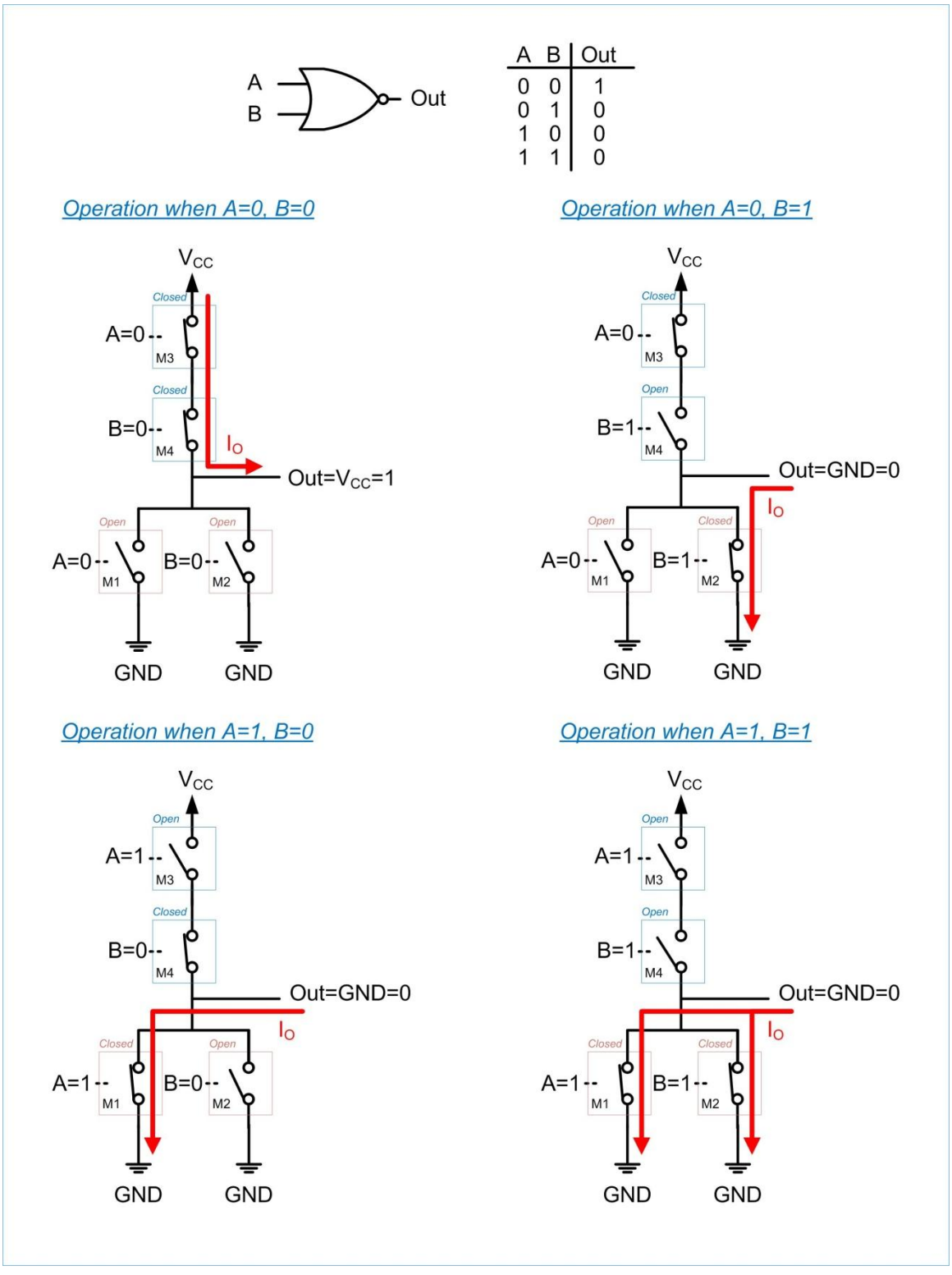


Figure 3.30
CMOS 2-Input NOR Gate Operation

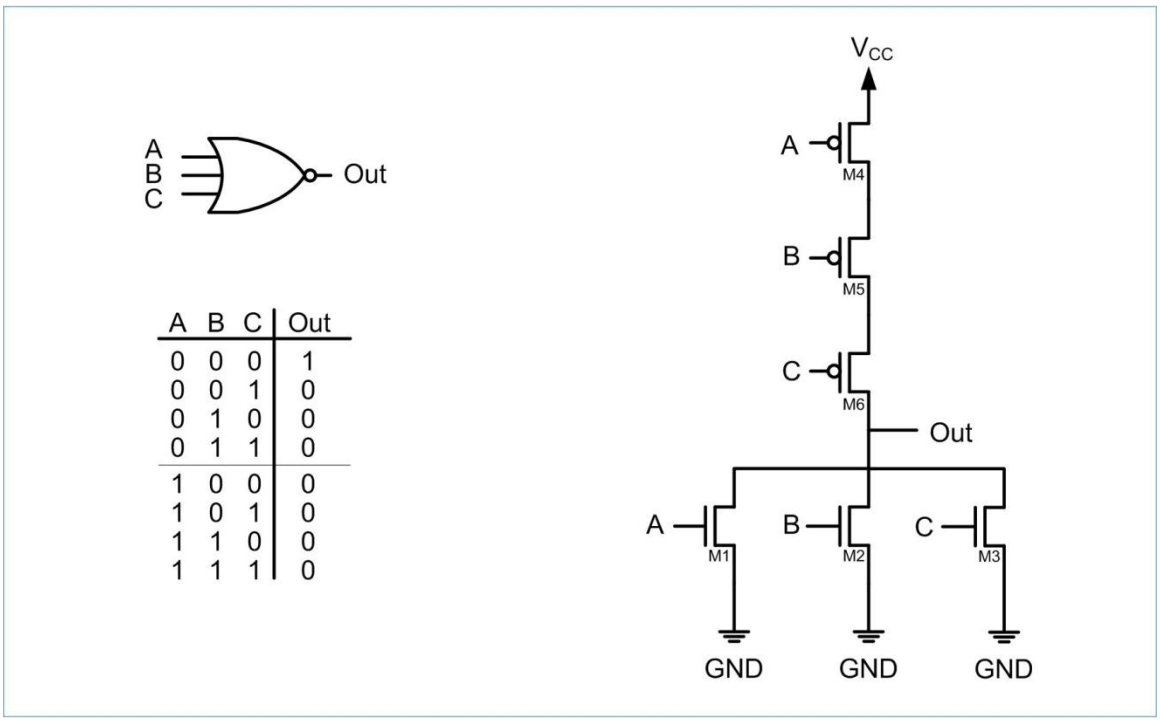


Figure 3.31
CMOS 3-Input NOR Gate Schematic

3.3.2 Transistor-Transistor Logic (TTL)

One of the first logic families that emerged after the invention of the integrated circuit was Transistor-Transistor Logic (TTL). This logic family uses bipolar junction transistor (BJT) as its fundamental switching item. This logic family defined a set of discrete parts that contained all of the basic gates in addition to more complex building blocks. TTL was used to build the first computer systems in the 1960's. TTL is not widely used today other than for specific applications because it consumes more power than CMOS and cannot achieve the density required for today's computer systems. TTL is discussed because it was the original logic family based on integrated circuits so it provides a historical perspective of digital logic. Furthermore, the discreet logic pinouts and part numbering schemes are still used today for discreet CMOS parts.

3.3.2.1 TTL Operation

TTL logic uses BJT transistors and resistors to accomplish the logic operations. The operation of a BJT transistor is more complicated than a MOSFET; however, it performs essentially the same switch operation when used in a digital logic circuit. An input is used to turn the transistor on, which in turn allows current to flow between two other terminals. **Figure** 3.32 shows the symbol for the two types of BJT transistors. The PNP transistor is analogous to a PMOS and the NPN is analogous to an NMOS. Current will flow between the Emitter and Collector terminals when there is a sufficient voltage on the Base terminal. The amount of current that flows between the Emitter and Collector is related to the current flowing into the Base. The primary difference in operation between BJTs and MOSFETs is that BJTs require proper voltage biasing in order to turn on and also draws current through the BASE in order to stay on. The detailed operation of BJTs is beyond the scope of this text so an overly simplified model of TTL logic gates is given.

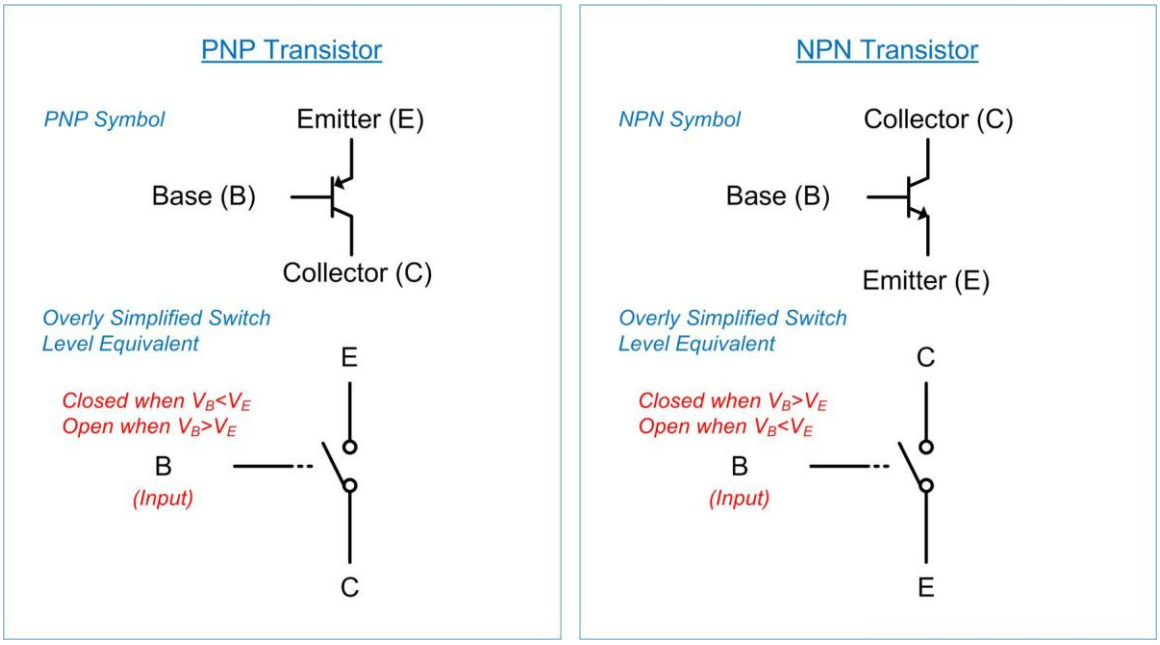


Figure 3.32
PNP and NPN Transistors

Figure 3.33 shows a simplified model of how TTL logic operates using BJTs and resistors. This simplified model does not show all of the transistors that are used in modern TTL circuits but instead is intended to provide a high level overview of the operation. This gate is an inverter that is created with an NPN transistor and a resistor. When the input is a logic HIGH, the NPN transistor turns on and conducts current between its collector and emitter terminals. This in effect closes the switch and connects the output to GND providing a logic LOW. During this state, current will also flow through the resistor to GND through Q1 thus consuming more power than the equivalent gate in CMOS. When the input is a logic LOW, the NPN transistor turns off and no current flows between its collector and emitter. This in effect is an open circuit leaving only the resistor connected to the output. The resistor pulls the output up to V_{CC} providing a logic HIGH on the output. One drawback of this state is that there will be a voltage drop across the resistor so the output is not pulled fully to V_{CC}.

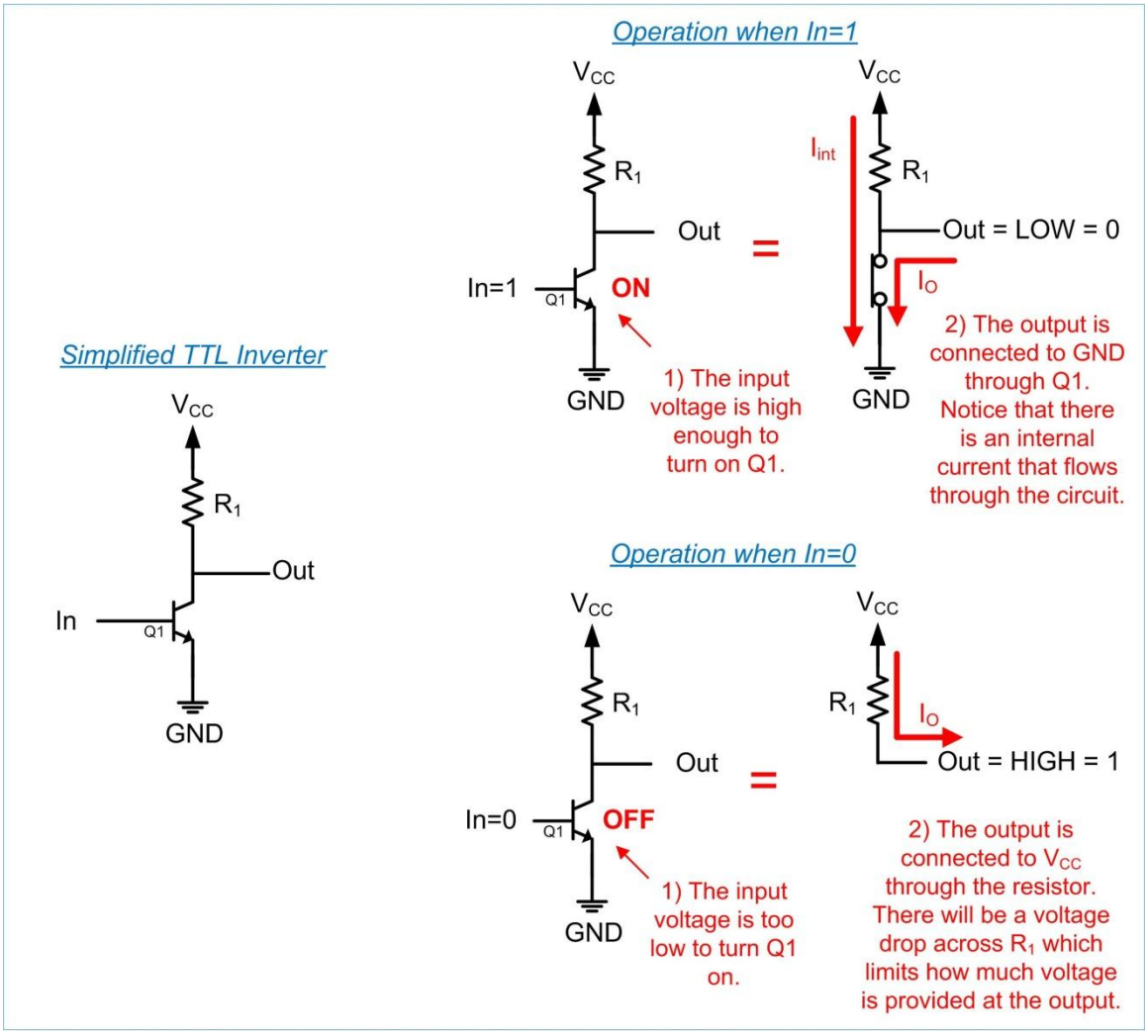


Figure 3.33
TTL Inverter

3.3.3 The 7400 Series Logic Families

The 7400 series of TTL circuits became popular in the 1960s and 1970s. This family was based on TTL and contained hundreds of different digital circuits. The original circuits came in either plastic or ceramic Dual-In-Line packages (e.g., DIP). The 7400 TTL logic family was powered off of a +5v supply. As mentioned before, this logic family set the pinouts and part numbering schemes for modern logic families. There were many derivatives of the original TTL logic family that made modifications to improve speed, reliability, decrease power and reduce power supplies. Today's CMOS logic families within the 7400 series still use the same pinouts and numbering schemes as the original TTL family.

3.3.3.1 Part Numbering Scheme

The part numbering scheme for the 7400 series and its derivatives contains 5 different fields: 1) manufacturer, 2) temperature range, 3) logic family, 4) logic function and 5) package type. The breakdown of these fields is shown in **Figure** 3.34.

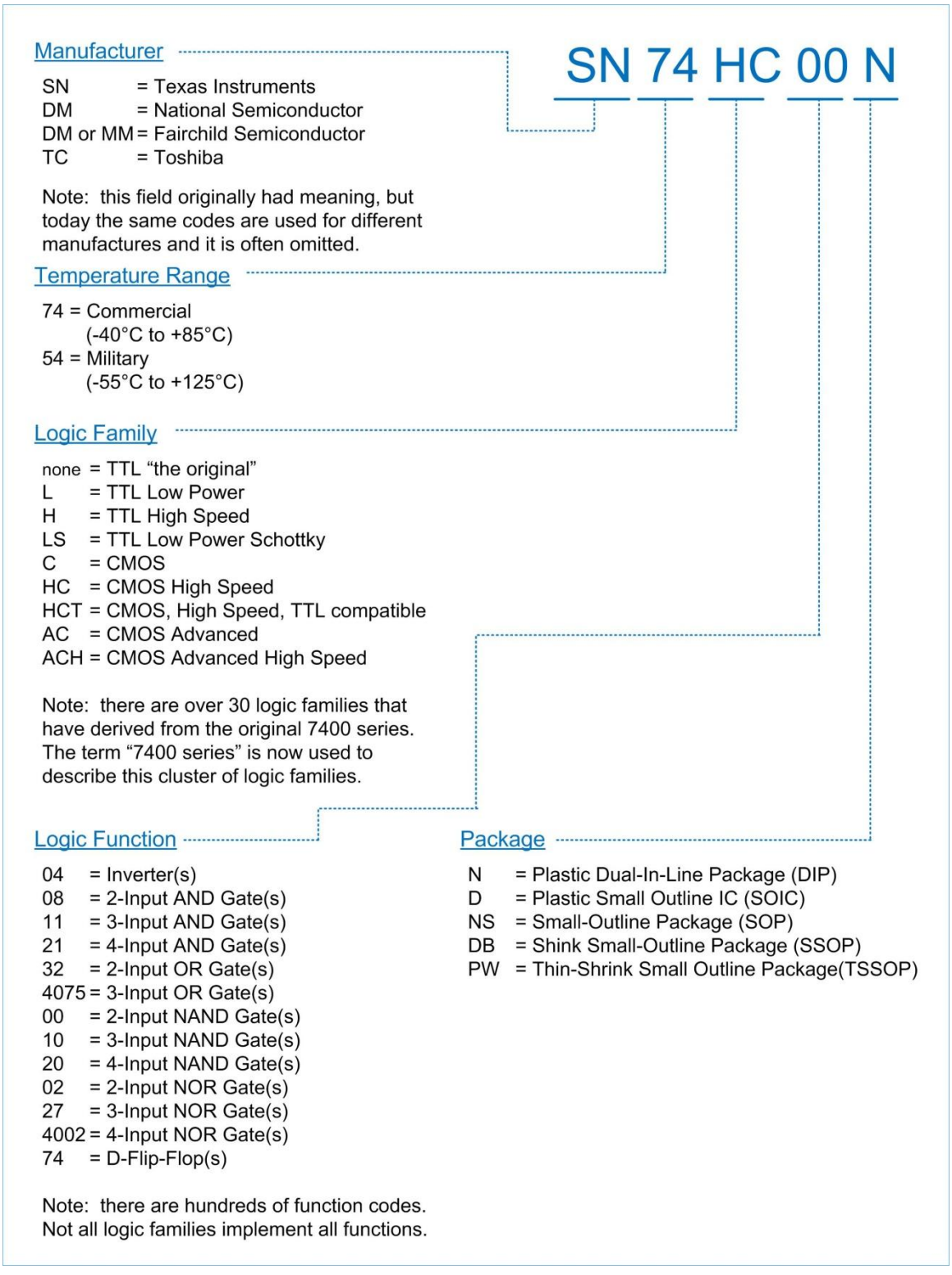


Figure 3.34
7400 Series Part Numbering Scheme

3.3.3.2 DC Operating Conditions

Table 3.2 gives the DC operating conditions for a few of the logic families within the 7400 series. Notice that the CMOS families consume much less power than the TTL families. Also notice that the TTL output currents are asymmetrical. The differences between the I_{OH} and I_{OL} within the TTL families has to do with the nature of the bipolar transistors and the resistors used to create the pull-up networks within the devices. CMOS has symmetrical drive currents due to using complementary transistors for the pull-up (PMOS) and pull-down networks (NMOS).

Logic Family	Year	DC Operating Condition											Speed (MHz)
		V_{CC}	V_{OHmax}	V_{OHmin}	V_{OLmax}	V_{OLmin}	V_{IHmax}	V_{IHmin}	V_{ILmax}	V_{ILmin}	I_{CC}	$I_{Omax\ (H/L)}$	
Original (TTL)	1964	+5	+5	+2.4	+0.4	GND	+5	+2	+0.8	GND	40m	-4/+16m	25
LS (TTL)	1976	+5	+5	+2.4	+0.4	GND	+5	+2	+0.8	GND	8.8m	-4/+8m	40
HC (CMOS)	1982	+2-6	V_{CC}	$0.8 \cdot V_{CC}$	0.33	GND	V_{CC}	$0.7 \cdot V_{CC}$	$0.3 \cdot V_{CC}$	GND	40u	+/-25m	50
AC (CMOS)	1985	+2-6	V_{CC}	$0.8 \cdot V_{CC}$	0.33	GND	V_{CC}	$0.7 \cdot V_{CC}$	$0.3 \cdot V_{CC}$	GND	80u	+/-50m	125

Note 1: All voltage specifications have units of volts. All current specifications have units of amps.

Note 2: The V_O and V_I specifications for the AC and HC logic families are worst case and vary depending on the V_{CC} selection and the output current.

Note 3: All specifications are given for the commercial temperature range (74 series).

Table 3.2
DC Operating Conditions for a Sample of 7400 Series Logic Families

3.3.3.3 Pin out Information for the DIP Packages

Figure 3.35 shows the pin out assignments for a subset of the basic gates from the 74HC logic family in the Dual-In-Line package form factor. Most of the basic gates within the 7400 series follow these assignments. Notice that each of these basic gates comes in a 14-pin DIP package, each with a single V_{CC} and single GND pin. It is up to the designer to ensure that the maximum current flowing through the V_{CC} and GND pins does not exceed the maximum specification. This is particularly important for parts that contain numerous gates. For example, the 74HC00 part contains 4, 2-Input NAND gates. If each of the NAND gates was driving a logic HIGH at its maximum allowable output current (e.g., 25mA from **Figure** 3.21), then a total of ($4 \cdot 25mA + I_q$) would be flowing through its V_{CC} pin. Since the V_{CC} pin can only tolerate a maximum of 50mA of current (from **Figure** 3.21), the part would be damaged since the output current of 100mA would also flow through the V_{CC} pin. The pinouts in **Figure** 3.35 are useful when first learning to design logic circuits because the DIP packages plug directly into a standard breadboard.

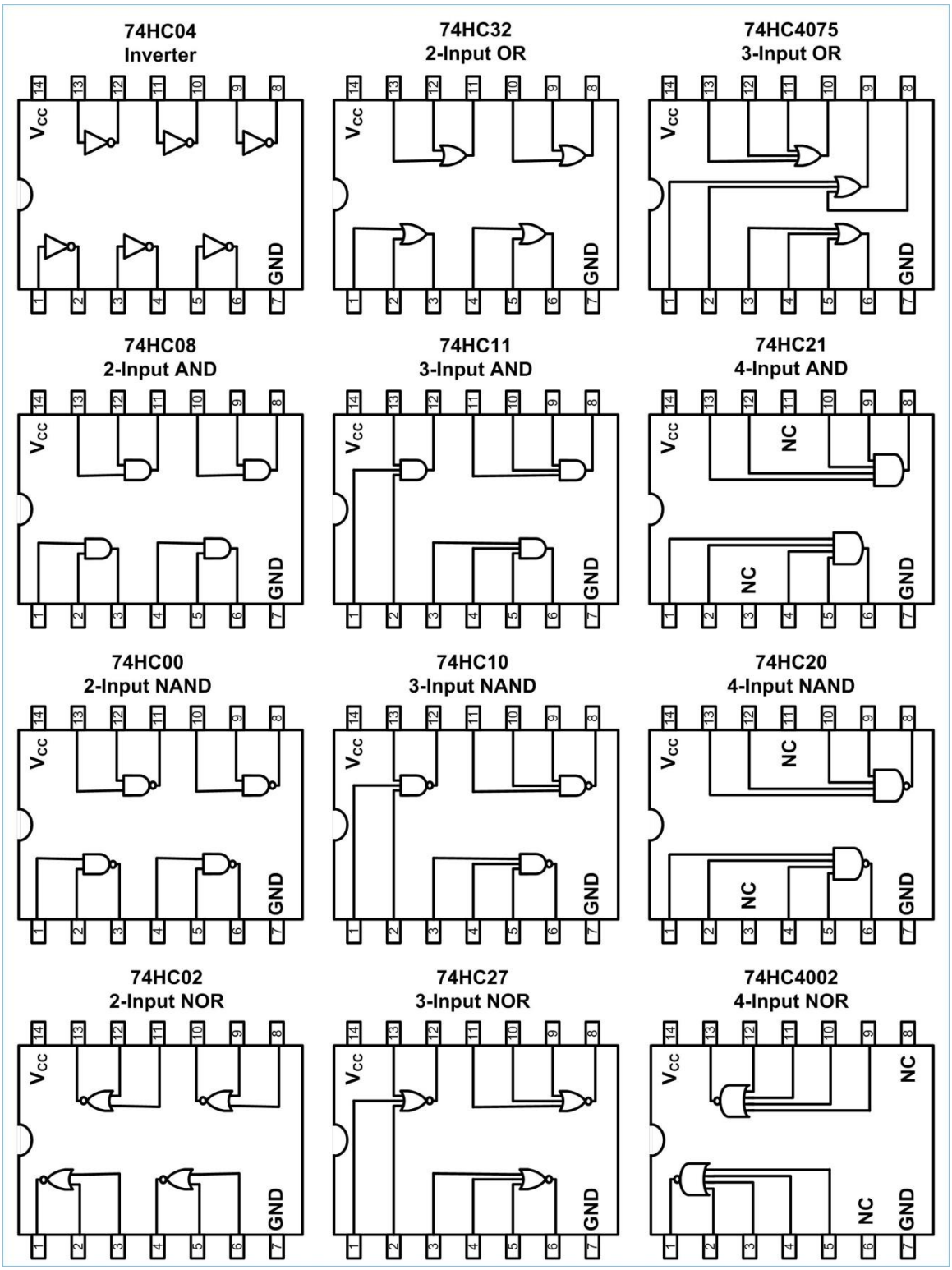


Figure 3.35
Pinouts for a subset of Basic Gates from the 74HC Logic Family in DIP Packaging

3.4 Driving Loads

At this point we've discussed in depth how proper care must be taken to ensure that not only do the output voltages of the driving gate meet the input specifications of the receiver in order to successfully transmit 1's and 0's, but that the output current of the driver does not exceed the maximum specifications so that the part is not damaged. The output voltage and current for a digital circuit depends greatly on the load that is being driven. The following sections discuss the impact of driving some of the most common digital loads.

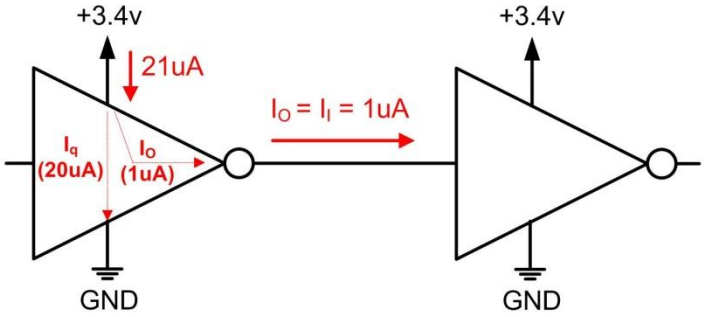
3.4.1 Driving Other Gates

Within a logic family, all digital circuits are designed to operate with one another. If there is minimal loss or noise in the interconnect system, then 1's and 0's will be successfully transmitted and no current specifications will be exceeded. Consider the example in **Figure** 3.36 for an inverter driving another inverter from the same logic family.

Example: Is either $I_{O\text{-}max}$ or $I_{CC\text{-}max}$ violated for the following circuit configuration?

Given: 74HC04 Specifications

$I_{I\text{-}max}$ = 1uA
I_{CC} = 20uA (e.g., I_q)
$I_{O\text{-}max}$ = 25mA
$I_{CC\text{-}max}$ = 50mA



Solution: The maximum input current of the load (e.g., the receiving inverter) is 1uA. The driver can easily provide this amount of current when sending 1's and 0's. This means that the I_O for the driver will be 1uA. This is far below the maximum output current of 25mA so the $I_{O\text{-}max}$ specification is not violated.

The driver will draw I_q through its V_{CC} pin to power its functional operation. In the datasheet, this current is often just called I_{CC}. This should not be confused with the specification for the maximum amount of current that can flow through the V_{CC} pin, which is often called $I_{CC\text{-}max}$. It is easy to tell the difference because I_{CC} (or I_q) is much smaller than $I_{CC\text{-}max}$ for CMOS parts. I_{CC} (or I_q) is specified in the uA to nA range while the maximum current that can flow through the V_{CC} pin is specified in the mA range. In addition to I_q, the driver will also pull a current equal to I_O through the V_{CC} pin while driving a logic HIGH. This means the maximum current pulled through the V_{CC} pin is $I_q + I_O$ = 20uA + 1uA = 21uA. Again, this is well below the specification for the maximum amount of current that can flow through the V_{CC} pin (50mA) so the I_{CC}-max specification is also not violated.

Figure 3.36
Example: Driving another Gate as the Load

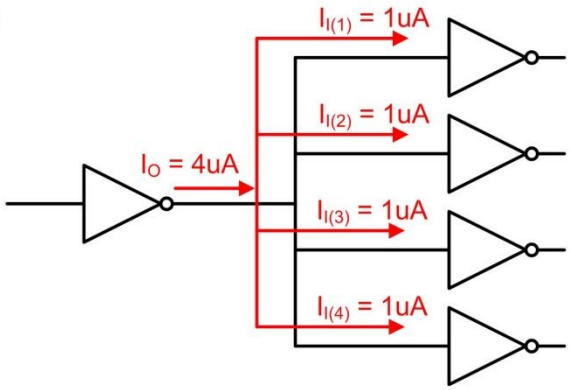
From this example, it is clear that there are no issues when a gate is driving another gate from the same family. This is as expected because that is the point of a logic family. In fact, gates are designed to drive multiple gates from within their own family. Based on solely the DC specifications for input and

output current, it could be assumed that the number of other gates that can be driven is simply ($I_{O\text{-}max}$ / $I_{I\text{-}max}$). For the example in **Figure** 3.36, this would result in a 74HC gate being able to drive 25,000 other gates (e.g., 25mA / 1uA = 25,000). In reality, the maximum number of gates that can be driven is dictated by the switching characteristics. This limit is called the **fan-out** specification. The fan-out specification states the maximum number of other gates from within the same family that can be driven. As discussed earlier, the output signal needs to transition quickly through the uncertainty region so that the receiver does not have time to react and go to an unknown state. As more and more gates are driven, this transition time is slowed down. The fan-out specification provides a limit to the maximum number of gates from the same family that can be driven while still ensuring that the output signal transitions between states fast enough to avoid the receivers from going to an unknown state.

Example: What is the driver's output current in the following circuit configuration where its driving the maximum number of gates allowed by its fan-out specification?

Given: 74HC04 Specifications

$I_{I\text{-}max}$ = 1uA
Fan-out = 4



Solution: The fan-out specification is 4, which means that the transmitting inverter can drive up to 4 other gates from its own logic family. Each of the receivers will draw their input current of I_I=1uA. This will be provided by the driver, so the total amount of output current is 4·1uA = 4uA.

Figure 3.37
Example: Driving Multiple Gates as a Load (Fan-Out)

3.4.2 Driving Resistive Loads

There are many situations where a resistor is the load in a digital circuit. A resistive load can be an actual resistor that is present for some other purpose such as a pull-up, pull-down, or for impedance matching. More complex loads such as buzzers, relays or other electronics can also be modeled as a resistor. When a resistor is the load in a digital circuit, care must be taken to avoid violating the output current specifications of the driver. The electrical circuit analysis technique that is used to evaluate how a resistive load impacts a digital circuit is **Ohm's Law**. Ohm's Law is a very simple relationship between the current and voltage in a resistor. **Figure** 3.38 gives a primer on Ohm's Law. For use in digital circuits, there are only a select few cases that this technique will be applied to, so no prior experience with Ohm's Law is required at this point.

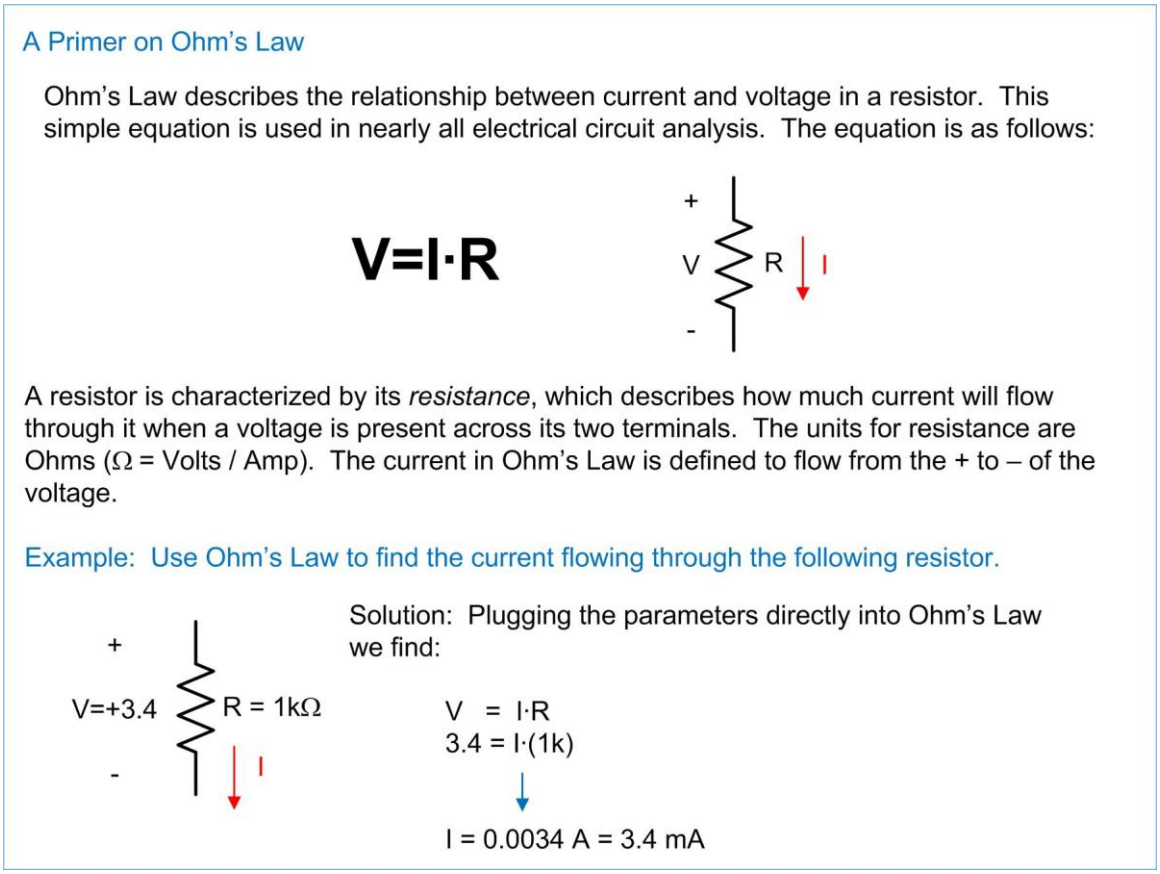


Figure 3.38
A Primer on Ohm's Law

Let's see how we can use Ohm's Law to analyze the impact of a resistive load in a digital circuit. Consider the circuit configuration in **Figure** 3.39. The load in this case is a resistor connected between the output of the driver and the power supply (+5v). When driving a logic HIGH, the output of the driver will be approximately the power supply (e.g., +5v). Since in this situation both terminals of the resistor are at +5v, there is no voltage difference present. That means when plugging into Ohm's Law, the voltage component is 0v, which gives 0 Amps of current. In the case where the driver is outputting a logic LOW, the output will be approximately GND. In this case, there is a voltage drop of +5v across the resistor (5v-0v). Plugging this into Ohm's Law yields a current of 50mA flowing through the resistor. This can become problematic because the current flows through the resistor and then into the output of the driver. For the 74HC logic family, this would exceed the I_O max specification of 25mA and damage the part. Additionally, as more current is drawn through the output, the output voltage becomes less and less ideal. In this example, the first order analysis used a V_O=GND; however, in reality as the output current increases the output voltage will move further away from its ideal value eventually reaching a value that may enter the uncertainty region.

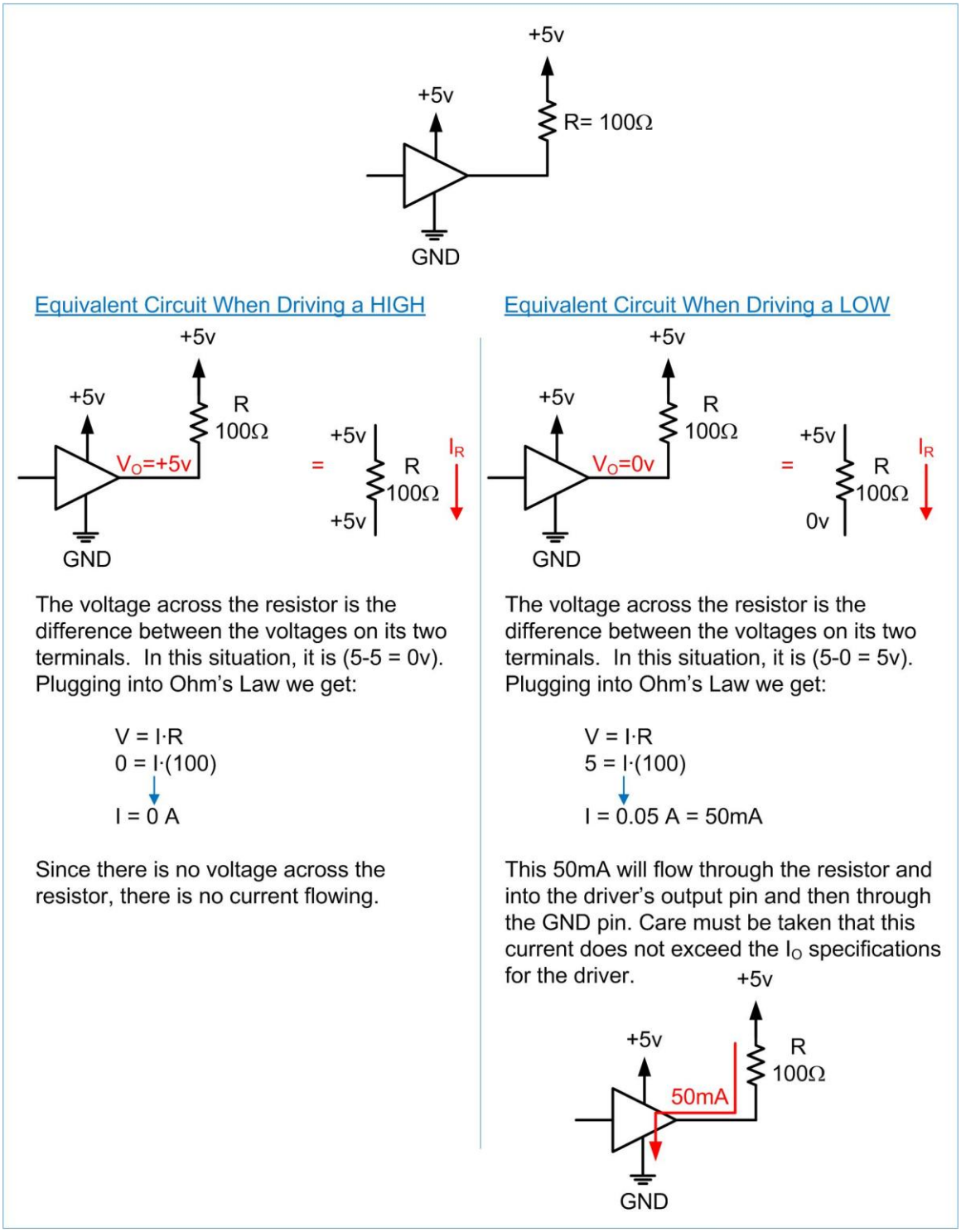


Figure 3.39
Example: Driving a Resistive Load (Pull-up)

A similar example is shown in **Figure** 3.40 for a driver connected to a resistive load between the output and GND.

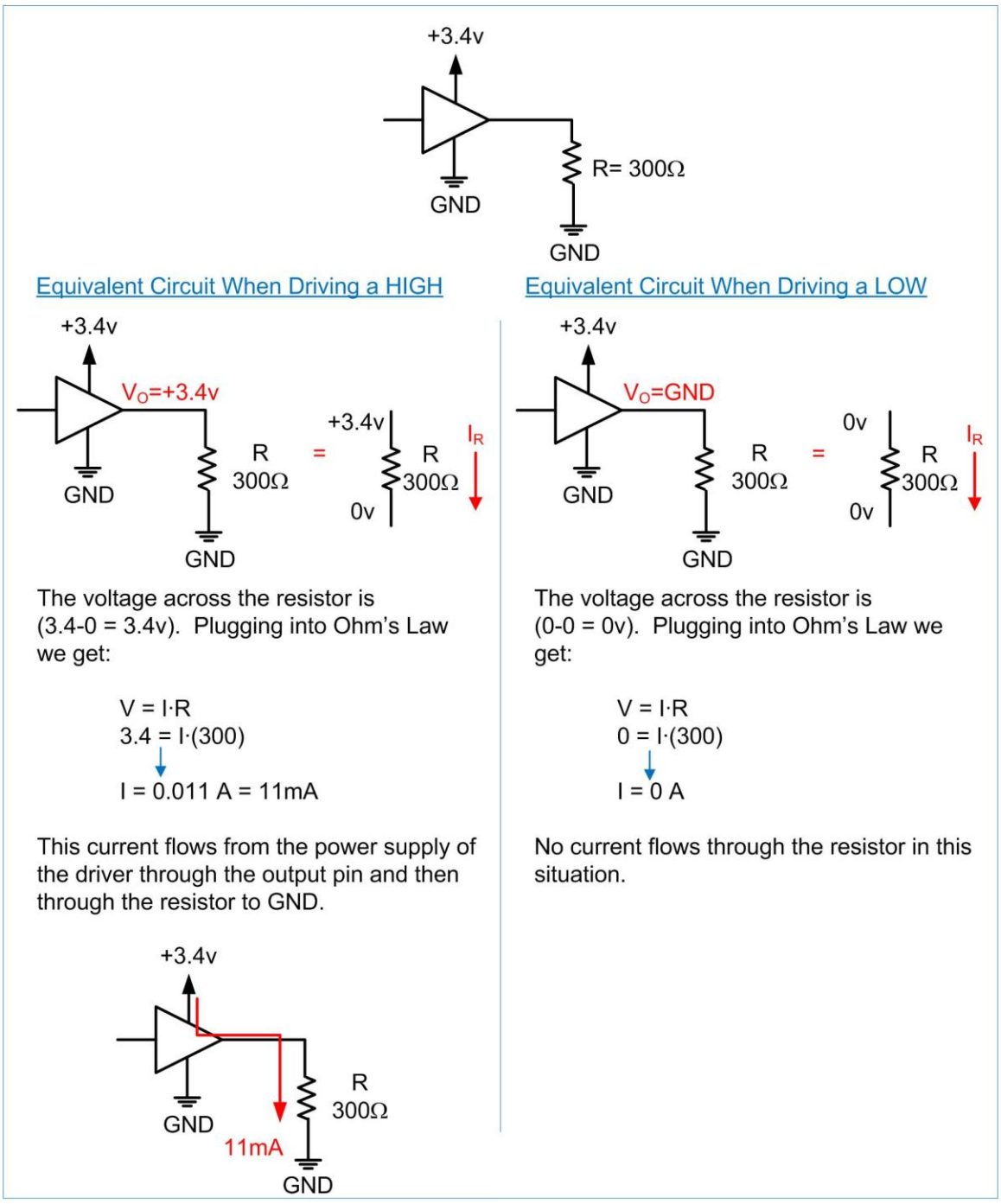


Figure 3.40
Example: Driving a Resistive Load (Pull-down)

3.4.3 Driving LEDs

A light emitting diode (LED) is a very common type of load that is driven using a digital circuit. The behavior of diodes is typically covered in an analog electronics class. Since it is assumed that the reader has not been exposed to the operation of diodes, the behavior of the LED will be described using a highly simplified model. A diode has two terminals, the anode and cathode. Current that flows from the anode to the cathode is called the *forward current*. A voltage that is developed across a diode from its anode to cathode is called the *forward voltage*. A diode has a unique characteristic that when a forward voltage is supplied across its terminal, it will only increase up to a certain point. The amount is specified as the LED's forward voltage (v_f) and is typically between 1.5v – 2v in modern LEDs. When a power supply circuit is connected to the LED, no current will flow until this forward voltage has been reached. Once it has been reached, current will begin to flow and the LED will prevent any further voltage from developing across it. Once current flows, the LED will begin emitting light. The more current that flows, the more light will be emitted up until the point that the maximum allowable current through the LED is reached and then the device will be damaged. When using an LED, there are two specifications of interest, the forward voltage and the recommended forward current. The symbols for a diode and an LED are given in **Figure** 3.41

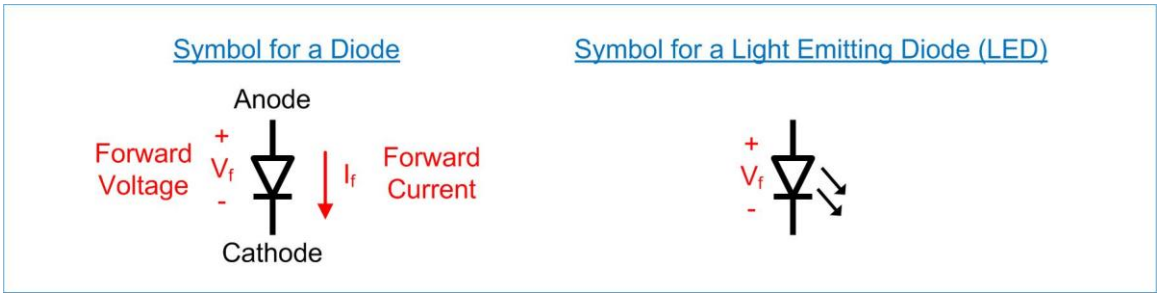
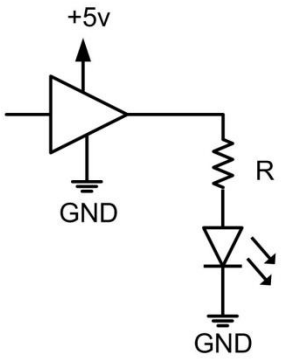


Figure 3.41
Symbols for a Diode and a Light Emitting Diode

When designing an LED driver circuit, a voltage must be supplied in order to develop the forward voltage across the LED so that current will flow. A resistor is included in series with the LED for two reasons. The first reason is to set the current going into the LED. The second reason is to provide an additional place for voltage to develop since the driver typically provides a higher voltage than the forward voltage of the diode. Consider the LED driver configuration shown in **Figure** 3.42. The LED has a specified forward voltage of +2v and a recommended forward current of 10mA. We need to select the value of the resistor so that when the driver outputs a logic HIGH, +2v will develop across the LED and 10mA will flow through it. When the driver outputs a logic HIGH, its V_O will go to +5v. While the output may not be exactly +5v, it is close enough to complete this design. The +5v will develop between the output and GND across the series combination of the resistor and LED. The LED's forward voltage will increase up to its forward voltage of +2v and then stop increasing. This means the remaining +3v will develop across the resistor. The value of the resistor will dictate how much current flows out of the driver, through the resistor and also through the LED. Since we know the voltage across the resistor and the desired current, we can use Ohm's Law to calculate the resistance.

Example: The following LED has a forward voltage of +2v and a recommended current of 10mA. Find the value of the resistor to set the LED current to 10mA.



Solution: When the driver outputs a logic LOW, it will provide V_O=0v. This means there will be no voltage that develops across the series combination of the resistor and LED. Since there is not enough voltage to meet the forward voltage requirements of the LED, no current will flow and the LED will be OFF.

When the driver outputs a logic HIGH, it will provide V_O=+5v. This voltage will develop across the series combination of the resistor and LED. The LED will increase up to its forward voltage of +2v and then remain there. The rest of the output voltage will develop across the resistor (e.g., +3v). We can choose the value of the resistor to set the current that will flow through the series combination using Ohm's Law since we know the voltage across the resistor and the desired current. In this case, the LED will be ON when the driver outputs a logic HIGH.

$V = I \cdot R$
$3 = (10mA) \cdot R$

$R = 300\ \Omega$

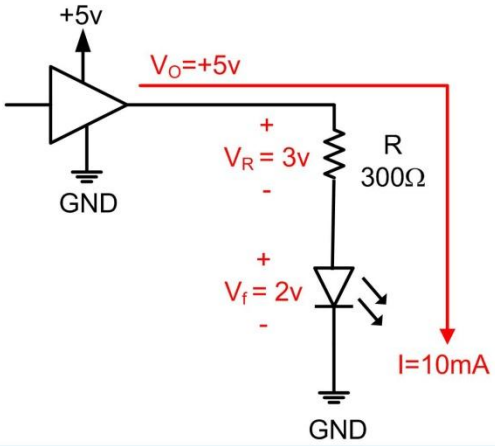
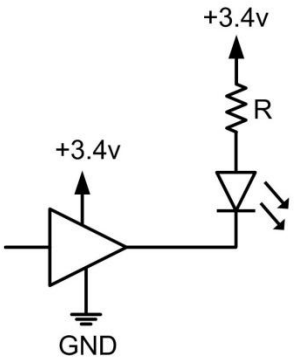


Figure 3.42
Example: Driving an LED with a Logic HIGH

Figure 3.43 shows another example of driving an LED, but this time using a different configuration where the LED will be on when the driver outputs a logic LOW.

Example: The following LED has a forward voltage of +1.8v and a recommended current of 4mA. Find the value of the resistor to set the LED current to 4mA.



Solution: When the driver outputs a logic HIGH, it will provide V_O=+3.4v. This means there will be no voltage that develops across the series combination of the resistor and LED since the other end of the combination is also at +3.4. This means when driving a logic HIGH, the LED will be OFF.

When the driver outputs a logic LOW, it will provide V_O=0v. Since the resistor is tied to +3.4v, this voltage will develop across the series combination of the resistor and LED. The LED will increase up to its forward voltage of +1.8v and then remain there. The rest of the output voltage will develop across the resistor (e.g., +1.6v). We can choose the value of the resistor to set the current that will flow through the series combination using Ohm's Law since we know the voltage across the resistor and the desired current. In this case, the LED will be ON when the driver outputs a logic LOW.

$V = I \cdot R$
$1.6 = (4mA) \cdot R$

$R = 400\ \Omega$

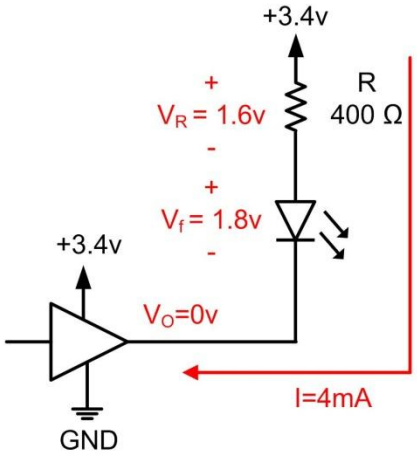


Figure 3.43
Example: Driving an LED with a Logic LOW

Exercise Problems

3.1 **Give the truth table for the following basic gates. Use the input variables A, B, C and call the output F.**

a) 3-Input AND Gate b) 3-Input OR Gate c) 3-Input XNOR Gate

3.2 **Give the logic expression for the following basic gates. Use the input variables A, B, C and call the output F.**

a) 3-Input AND Gate b) 3-Input OR Gate c) 3-Input XNOR Gate

3.3 **Give the logic waveform for the following basic gates. Use the input variables A, B, C and call the output F.**

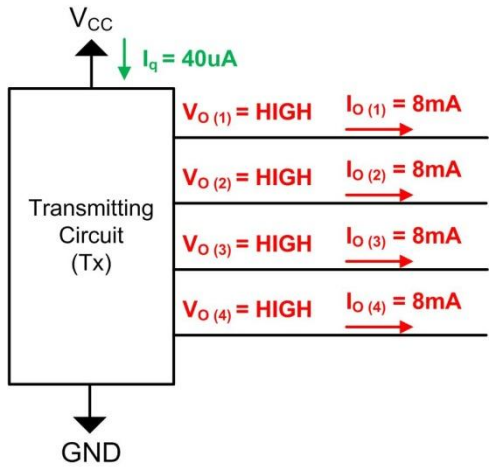
a) 3-Input AND Gate b) 3-Input OR Gate c) 3-Input XNOR Gate

3.4 **Using the DC Operating Conditions from Table 3.2 give the noise margins (NM_H and NM_L) for the following logic families:**

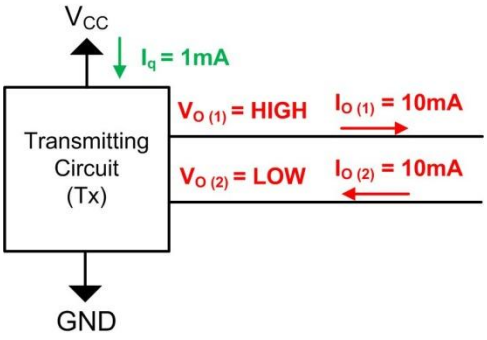
a) 74LS b) 74HC with V_{CC}=+5v c) 74HC with V_{CC}=+3.4v

3.5 **For the following driver configurations give the current flowing through the V_{CC} and GND pins of the driver:**

a)



b)



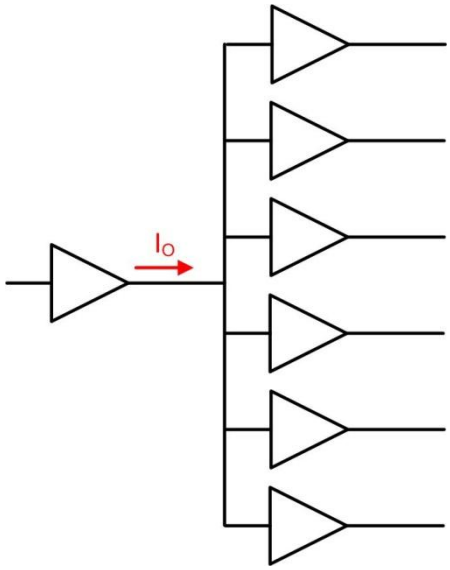
3.6 **Using the data sheet except from Figure 3.22, provide the following maximum switching characteristics for the 74HC04 inverter when powered with V_{CC}=+2v.**

a) t_{pd} b) t_{PLH} c) t_{PHL}
d) t_t e) t_r f) t_f

3.7 **Provide the transistor-level schematic for the following gates when implemented in CMOS. (Hint: The AND, OR and BUFFER will each require two gates)**

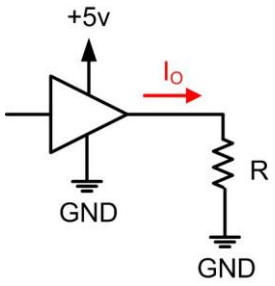
a) 4-Input NAND Gate b) 4-Input NOR Gate c) 2-Input AND Gate
d) 2-Input OR Gate e) Buffer

3.8 In the following driver configuration the buffer is driving its maximum fan-out specification of 6. The maximum input current for this logic family is I_I=1nA. What is the maximum output current that the driver will need to source?

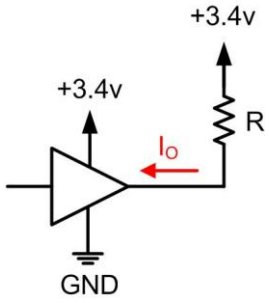


3.9 For the following driver configurations calculate the values of the resistors in order to ensure that the output current does not exceed 20mA:

a)

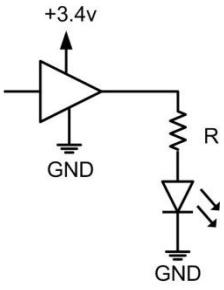


b)



3.10 For the following driver configurations calculate the values of the resistors in order to set the LED forward current to 5mA. The LEDs have a forward voltage of 1.9v.

a)



b)