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. The goal of this chapter is to provide an understanding of the basic principles of binary number systems.

Learning Outcomes—After completing this chapter, you will be able to:

2.1 Describe the formation and use of positional number systems.
2.2 Convert numbers between different bases.
2.3 Perform binary addition and subtraction by hand.
2.4 Use two's complement numbers to 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 represent 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 lesser) 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,... 19, 20, 21,...). This is repeated each time a position runs out of symbols (e.g., . . ., 99, 100, 101. . . 999, 1000, 1001,. . .).

First, 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*.

B.J. LaMeres, *Introduction to Logic Circuits & Logic Design with Verilog*,
DOI 10.1007/978-3-319-53883-9_2

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.

Radix $=$ Base $\equiv$ the number of numerals in the number system

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 (i.e., "."). A particular number system often renames this radix point to reflect its base. For example, in the base 10 number system (i.e., 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.

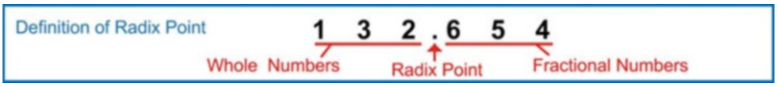


Fig. 2.1
Definition of radix point

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.

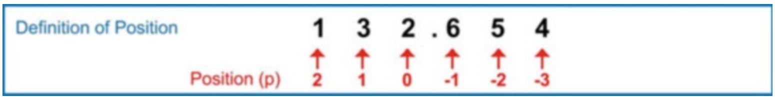


Fig. 2.2
Definition of position number (p) within the number

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.

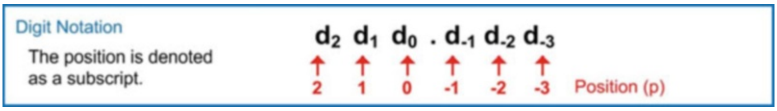

Digit Notation
The position is denoted as a subscript.
d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3}
2 1 0 -1 -2 -3
Position (p)

Fig. 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 affecting 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)**.

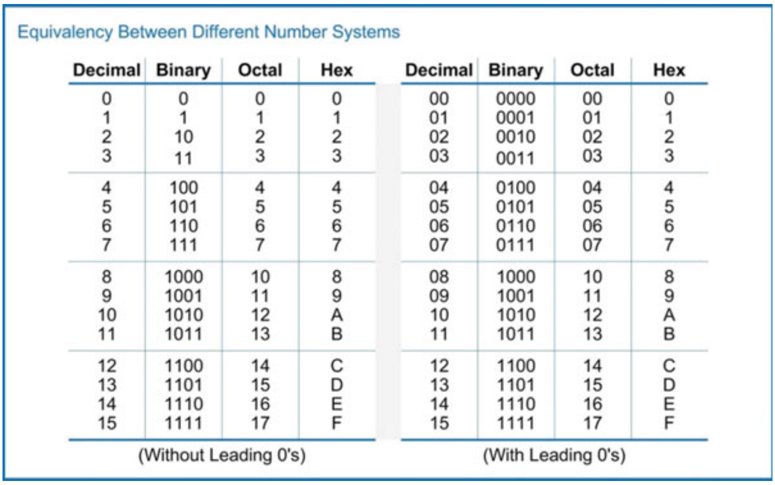
2.1.4 Octal Number System (Base 8)

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”. Table 2.1 lists the equivalent values in each of the 4 number systems just 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 zeros. The equivalencies of decimal, binary and hexadecimal in this table are typically committed to memory.



Equivalency Between Different Number Systems

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

Concept Check

CC2.1 The base of a number system is arbitrary and is commonly selected to match a particular aspect of the physical system in which it is used (e.g., base 10 corresponds to our 10 fingers, base 2 corresponds to the 2 states of a switch). If a physical system contained 3 unique modes and a base of 3 was chosen for the number system, what is the base 3 equivalent of the decimal number 3?

(A) $3_{10} = 11_3$ (B) $3_{10} = 3_3$ (C) $3_{10} = 10_3$ (D) $3_{10} = 21_3$

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:

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

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.

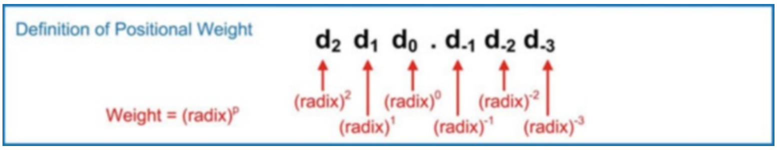


Fig. 2.4
Weight definition

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:

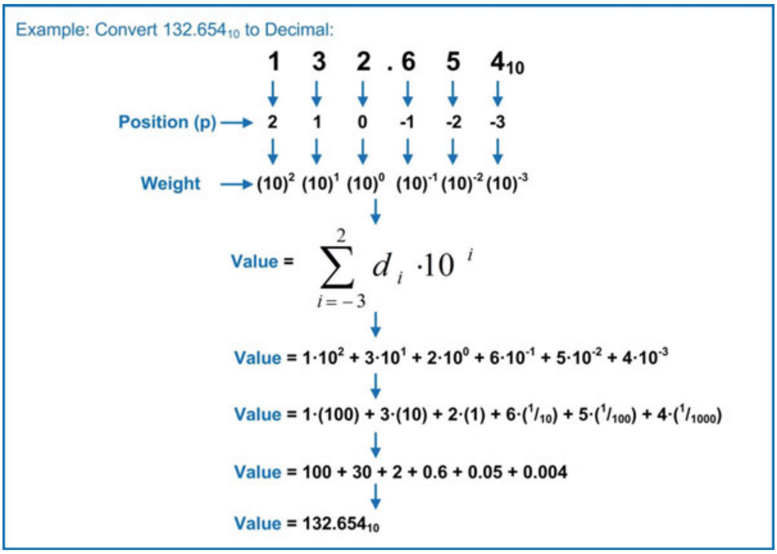
$$Total\ Decimal\ Value = \sum_{i=p_{min}}^{p_{max}} d_i \cdot (radix)^i$$

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 $(\text{radix})^p$ 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}. Example 2.1 shows this process step-by-step.



$$\text{Value} = \sum_{i=-3}^{2} d_i \cdot 10^i$$

$$\text{Value} = 1\cdot10^2 + 3\cdot10^1 + 2\cdot10^0 + 6\cdot10^{-1} + 5\cdot10^{-2} + 4\cdot10^{-3}$$

$$\text{Value} = 1\cdot(100) + 3\cdot(10) + 2\cdot(1) + 6\cdot({}^1/_{10}) + 5\cdot({}^1/_{100}) + 4\cdot({}^1/_{1000})$$

$$\text{Value} = 100 + 30 + 2 + 0.6 + 0.05 + 0.004$$

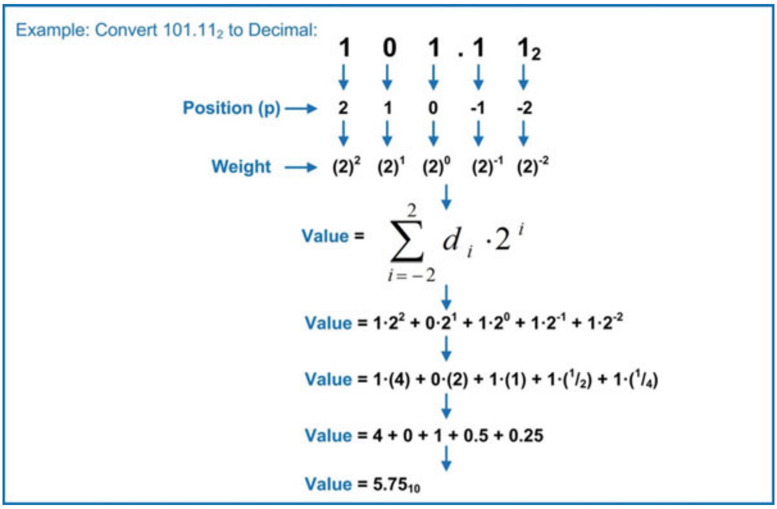
$$\text{Value} = 132.654_{10}$$

Example 2.1
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. Example 2.2 shows the step-by-step process converting a binary number to decimal.



Example: Convert 101.11_2 to Decimal:

$1 \quad 0 \quad 1 \;.\; 1 \quad 1_2$

Position (p) → $2 \quad 1 \quad 0 \quad -1 \quad -2$

Weight → $(2)^2 \quad (2)^1 \quad (2)^0 \quad (2)^{-1} \quad (2)^{-2}$

$$\text{Value} = \sum_{i=-2}^{2} d_i \cdot 2^i$$

$$\text{Value} = 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 + 1\cdot2^{-1} + 1\cdot2^{-2}$$

$$\text{Value} = 1\cdot(4) + 0\cdot(2) + 1\cdot(1) + 1\cdot({}^1/_2) + 1\cdot({}^1/_4)$$

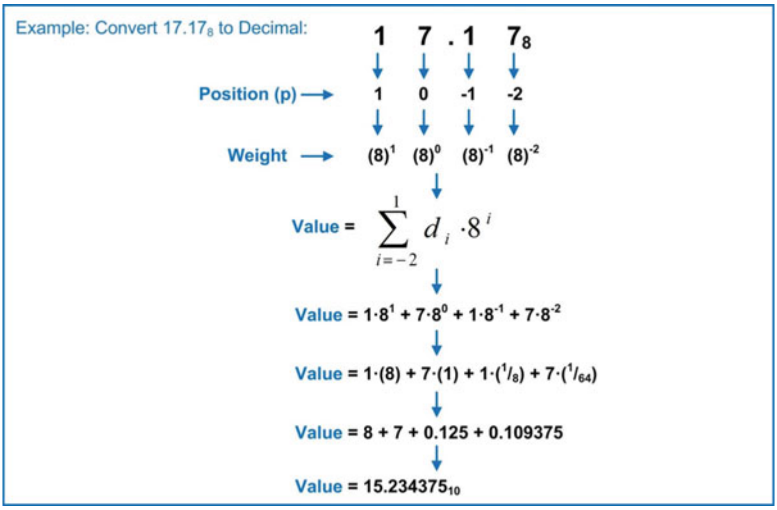
$$\text{Value} = 4 + 0 + 1 + 0.5 + 0.25$$

$$\text{Value} = 5.75_{10}$$

Example 2.2
Converting binary to decimal

2.2.1.2 Octal to Decimal

When converting from octal to decimal, the same process is followed with the exception that the base in the weight is changed to 8. Example 2.3 shows an example of converting an octal number to decimal.



Example: Convert 17.17_8 to Decimal:

$1 \quad 7 \;.\; 1 \quad 7_8$

Position (p) → $1 \quad 0 \quad -1 \quad -2$

Weight → $(8)^1 \quad (8)^0 \quad (8)^{-1} \quad (8)^{-2}$

$$\text{Value} = \sum_{i=-2}^{1} d_i \cdot 8^i$$

$$\text{Value} = 1\cdot8^1 + 7\cdot8^0 + 1\cdot8^{-1} + 7\cdot8^{-2}$$

$$\text{Value} = 1\cdot(8) + 7\cdot(1) + 1\cdot({}^1/_8) + 7\cdot({}^1/_{64})$$

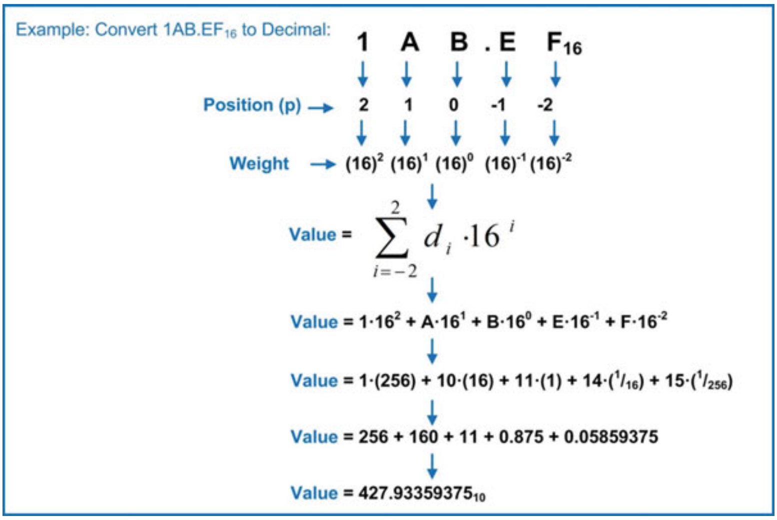
$$\text{Value} = 8 + 7 + 0.125 + 0.109375$$

$$\text{Value} = 15.234375_{10}$$

Example 2.3
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. Example 2.4 shows the step-by-step process converting a hexadecimal number to decimal.


Example: Convert $1AB.EF_{16}$ to Decimal:
1 A B . E F_{16}
Position (p)
2 1 0 -1 -2
Weight
$(16)^2$ $(16)^1$ $(16)^0$ $(16)^{-1}$ $(16)^{-2}$
Value = $\sum_{i=-2}^{2} d_i \cdot 16^i$
Value = $1\cdot16^2 + A\cdot16^1 + B\cdot16^0 + E\cdot16^{-1} + F\cdot16^{-2}$
Value = $1\cdot(256) + 10\cdot(16) + 11\cdot(1) + 14\cdot(^1/_{16}) + 15\cdot(^1/_{256})$
Value = $256 + 160 + 11 + 0.875 + 0.05859375$
Value = 427.93359375_{10}

Example 2.4
Converting hexadecimal to decimal

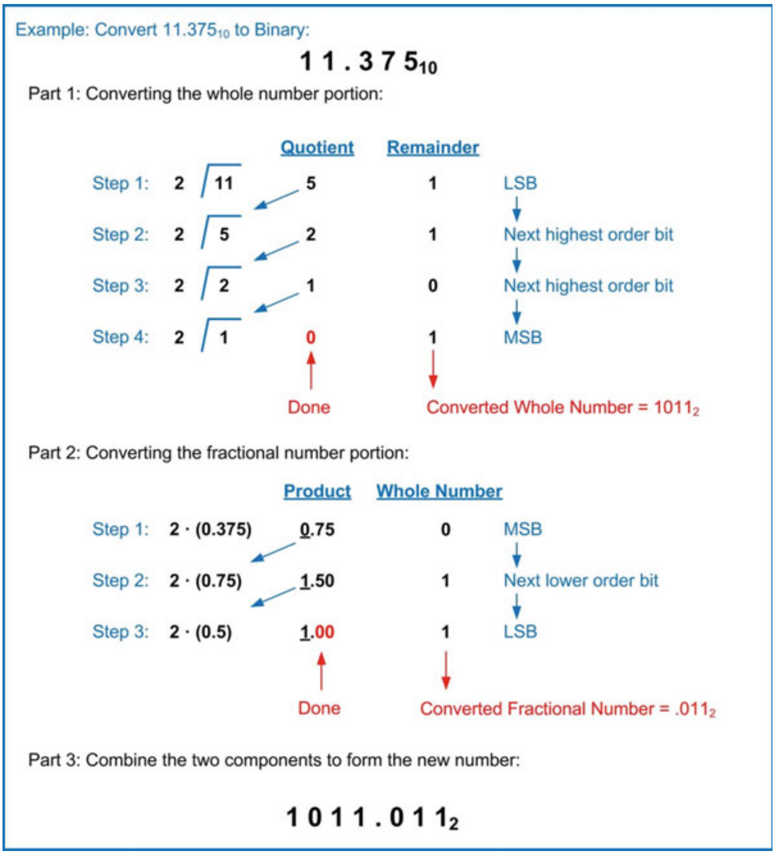
2.2.2 Converting From Decimal

The process of converting from decimal to another base consists of two separate algorithms. There is one algorithm for converting the whole number portion of the number and another algorithm for converting 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 fractional 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 4-bits of fraction had been achieved in the conversion.

2.2.2.1 Decimal to Binary

Let’s convert 11.375_{10} to binary. Example 2.5 shows the step-by-step process converting a decimal number to binary.



Example: Convert 11.375_{10} to Binary:

11.375_{10}

Part 1: Converting the whole number portion:

		Quotient	Remainder	
Step 1:	2 / 11	5	1	LSB
Step 2:	2 / 5	2	1	Next highest order bit
Step 3:	2 / 2	1	0	Next highest order bit
Step 4:	2 / 1	0	1	MSB

Done

Converted Whole Number = 1011_2

Part 2: Converting the fractional number portion:

		Product	Whole Number	
Step 1:	2 · (0.375)	0.75	0	MSB
Step 2:	2 · (0.75)	1.50	1	Next lower order bit
Step 3:	2 · (0.5)	1.00	1	LSB

Done

Converted Fractional Number = $.011_2$

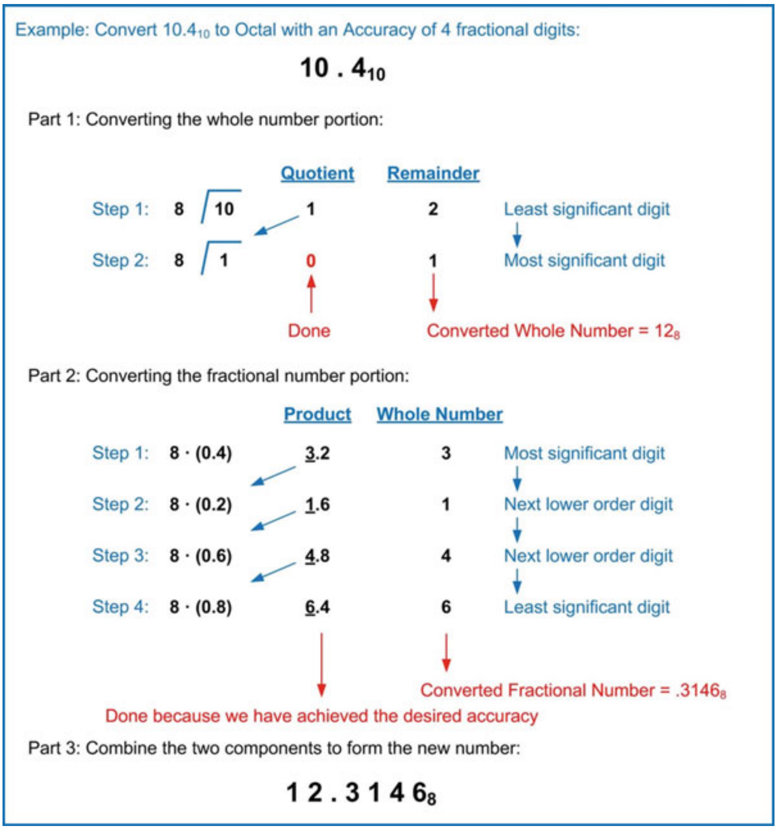
Part 3: Combine the two components to form the new number:

1011.011_2

Example 2.5
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. Example 2.6 shows the step-by-step process converting a decimal number to octal with a fractional accuracy of 4 digits.



Example: Convert 10.4_{10} to Octal with an Accuracy of 4 fractional digits:

$10\,.\,4_{10}$

Part 1: Converting the whole number portion:

		Quotient	Remainder	
Step 1:	8 ⟌ 10	1	2	Least significant digit
Step 2:	8 ⟌ 1	0	1	Most significant digit

Done

Converted Whole Number = 12_8

Part 2: Converting the fractional number portion:

		Product	Whole Number	
Step 1:	8 · (0.4)	3.2	3	Most significant digit
Step 2:	8 · (0.2)	1.6	1	Next lower order digit
Step 3:	8 · (0.6)	4.8	4	Next lower order digit
Step 4:	8 · (0.8)	6.4	6	Least significant digit

Converted Fractional Number = $.3146_8$

Done because we have achieved the desired accuracy

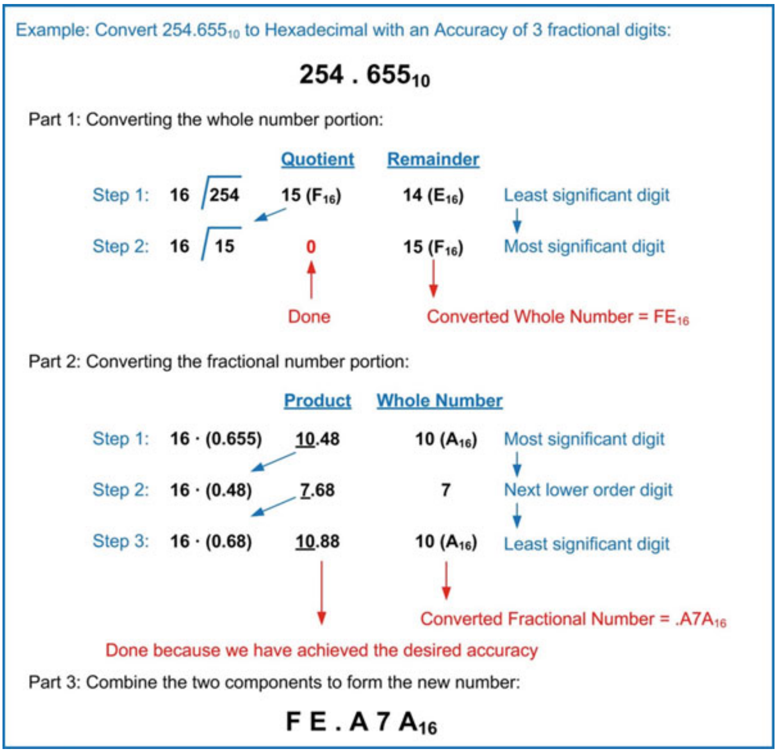
Part 3: Combine the two components to form the new number:

12.3146_8

Example 2.6
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 $15_{10,}$ then the decimal numbers are substituted with their hex symbol equivalent (i.e., A to F). Example 2.7 shows the step-by-step process of converting a decimal number to hex with a fractional accuracy of 3 digits.



Example: Convert 254.655_{10} to Hexadecimal with an Accuracy of 3 fractional digits:

$254\,.\,655_{10}$

Part 1: Converting the whole number portion:

		Quotient	Remainder	
Step 1:	16 / 254	15 (F_{16})	14 (E_{16})	Least significant digit
Step 2:	16 / 15	0	15 (F_{16})	Most significant digit

Done

Converted Whole Number = FE_{16}

Part 2: Converting the fractional number portion:

		Product	Whole Number	
Step 1:	16 · (0.655)	10.48	10 (A_{16})	Most significant digit
Step 2:	16 · (0.48)	7.68	7	Next lower order digit
Step 3:	16 · (0.68)	10.88	10 (A_{16})	Least significant digit

Converted Fractional Number = $.A7A_{16}$

Done because we have achieved the desired accuracy

Part 3: Combine the two components to form the new number:

$FE.A7A_{16}$

Example 2.7
Converting decimal to hexadecimal

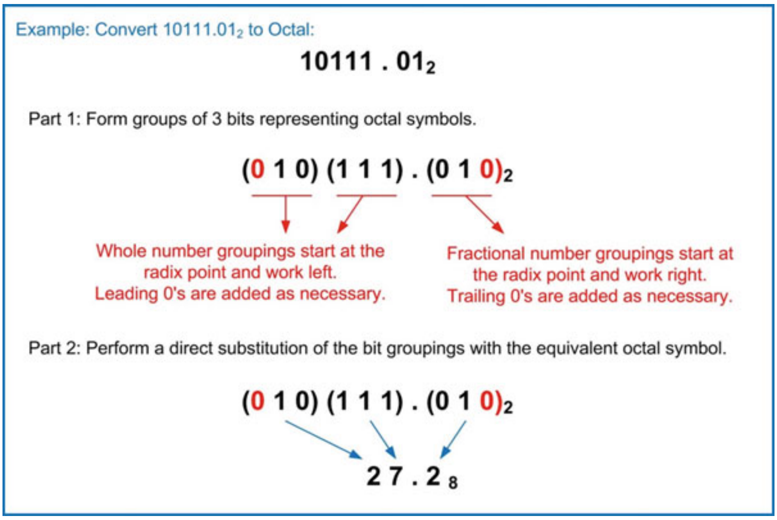
2.2.3 Converting Between 2^n Bases

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 (i.e., $0_8 = 000_2$, $7_8 = 111_2$). Base 16 numbers take exactly 4 binary bits to represent all 16 symbols (i.e., $0_{16} = 0000_2$, $F_{16} = 1111_2$).

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 is 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 is then directly substituted with their 2^n base symbol.

2.2.3.1 Binary to Octal

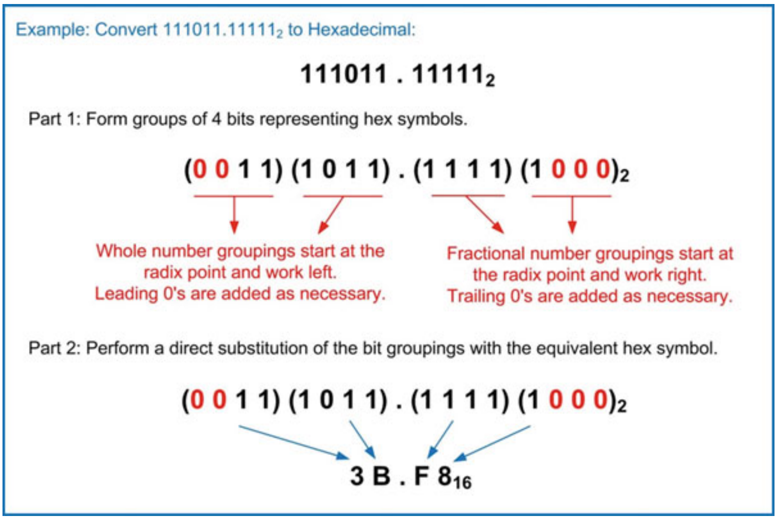
Example 2.8 shows the step-by-step process of converting a binary number to octal.


Example: Convert 10111.01₂ to Octal:
10111 . 01₂
Part 1: Form groups of 3 bits representing octal symbols.
(0 1 0) (1 1 1) . (0 1 0)₂
Whole number groupings start at the radix point and work left. Leading 0's are added as necessary.
Fractional number groupings start at the radix point and work right. Trailing 0's are added as necessary.
Part 2: Perform a direct substitution of the bit groupings with the equivalent octal symbol.
(0 1 0) (1 1 1) . (0 1 0)₂
2 7 . 2 ₈

Example 2.8
Converting binary to octal

2.2.3.2 *Binary to Hexadecimal*

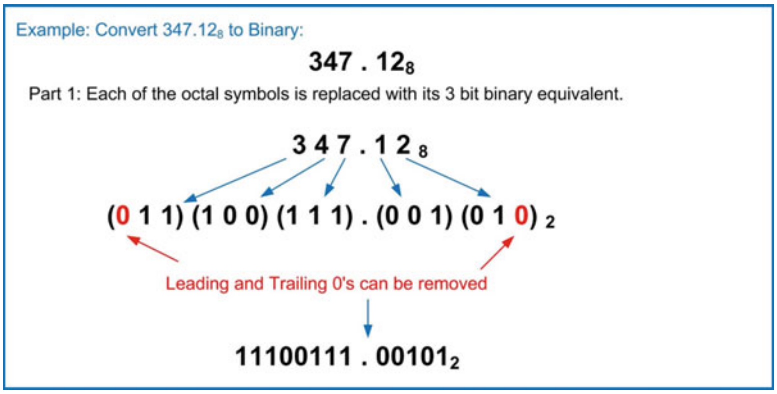
Example 2.9 shows the step-by-step process of converting a binary number to hexadecimal.


Example: Convert 111011.11111₂ to Hexadecimal:
111011 . 11111₂
Part 1: Form groups of 4 bits representing hex symbols.
(0 0 1 1) (1 0 1 1) . (1 1 1 1) (1 0 0 0)₂
Whole number groupings start at the radix point and work left. Leading 0's are added as necessary.
Fractional number groupings start at the radix point and work right. Trailing 0's are added as necessary.
Part 2: Perform a direct substitution of the bit groupings with the equivalent hex symbol.
(0 0 1 1) (1 0 1 1) . (1 1 1 1) (1 0 0 0)₂
3 B . F 8₁₆

Example 2.9
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. Example 2.10 shows the step-by-step process of converting an octal number to binary.



Example: Convert 347.12_8 to Binary:

$347\,.\,12_8$

Part 1: Each of the octal symbols is replaced with its 3 bit binary equivalent.

$3\,4\,7\,.\,1\,2_8$

$(0\;1\;1)\,(1\;0\;0)\,(1\;1\;1)\,.\,(0\;0\;1)\,(0\;1\;0)_2$

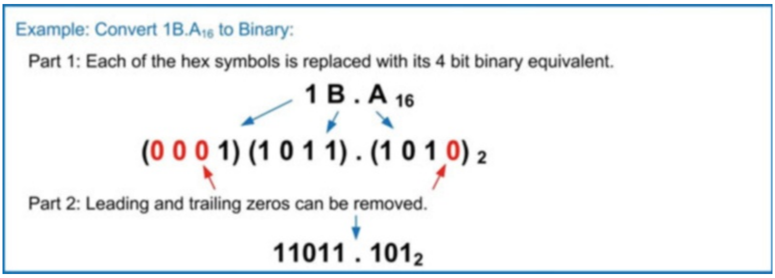
Leading and Trailing 0's can be removed

$11100111\,.\,00101_2$

Example 2.10
Converting octal to binary

2.2.3.4 Hexadecimal to Binary

Example 2.11 shows the step-by-step process of converting a hexadecimal number to binary.



Example: Convert $1B.A_{16}$ to Binary:

Part 1: Each of the hex symbols is replaced with its 4 bit binary equivalent.

$1\,B\,.\,A_{16}$

$(0\;0\;0\;1)\,(1\;0\;1\;1)\,.\,(1\;0\;1\;0)_2$

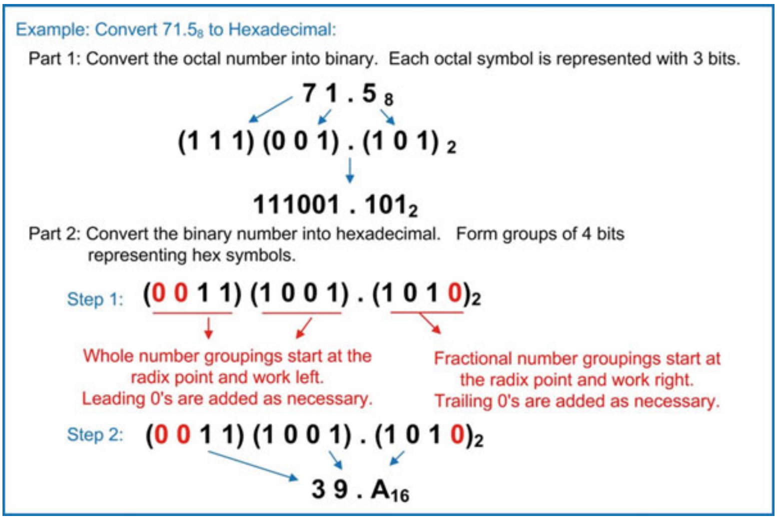
Part 2: Leading and trailing zeros can be removed.

$11011\,.\,101_2$

Example 2.11
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. Example 2.12 shows the step-by-step process of converting an octal number to hexadecimal.



Example: Convert 71.5_8 to Hexadecimal:

Part 1: Convert the octal number into binary. Each octal symbol is represented with 3 bits.

$7\ 1\ .\ 5_8$

$(1\ 1\ 1)\ (0\ 0\ 1)\ .\ (1\ 0\ 1)_2$

$111001\ .\ 101_2$

Part 2: Convert the binary number into hexadecimal. Form groups of 4 bits representing hex symbols.

Step 1: $(0\ 0\ 1\ 1)\ (1\ 0\ 0\ 1)\ .\ (1\ 0\ 1\ 0)_2$

Whole number groupings start at the radix point and work left.
Leading 0's are added as necessary.

Fractional number groupings start at the radix point and work right.
Trailing 0's are added as necessary.

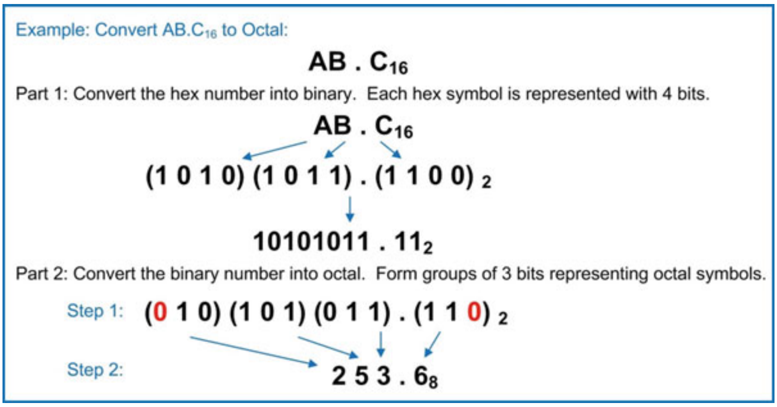
Step 2: $(0\ 0\ 1\ 1)\ (1\ 0\ 0\ 1)\ .\ (1\ 0\ 1\ 0)_2$

$3\ 9\ .\ A_{16}$

Example 2.12
Converting octal to hexadecimal

2.2.3.6 Hexadecimal to Octal

Example 2.13 shows the step-by-step process of converting a hexadecimal number to octal.



Example: Convert $AB.C_{16}$ to Octal:

$AB\ .\ C_{16}$

Part 1: Convert the hex number into binary. Each hex symbol is represented with 4 bits.

$AB\ .\ C_{16}$

$(1\ 0\ 1\ 0)\ (1\ 0\ 1\ 1)\ .\ (1\ 1\ 0\ 0)_2$

$10101011\ .\ 11_2$

Part 2: Convert the binary number into octal. Form groups of 3 bits representing octal symbols.

Step 1: $(0\ 1\ 0)\ (1\ 0\ 1)\ (0\ 1\ 1)\ .\ (1\ 1\ 0)_2$

Step 2: $2\ 5\ 3\ .\ 6_8$

Example 2.13
Converting hexadecimal to octal

Concept Check

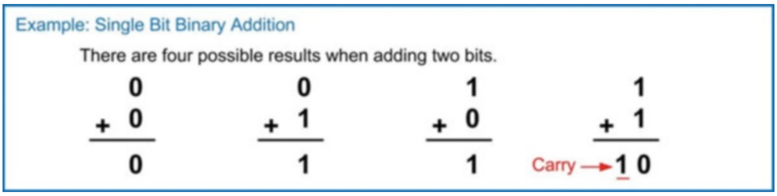
CC2.2 A “googol” is the term for the decimal number 1e100. When written out manually this number is a 1 with 100 zeros after it (e.g., 10,000,000,000,000,000,000,000,000,000,000,000,000, 000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000). This term is more commonly associated with the search engine company Google, which uses a different spelling but is pronounced the same. How many bits does it take to represent a googol in binary?

(A) 100 bits (B) 256 bits (C) 332 bits (D) 333 bits

2.3 Binary Arithmetic

2.3.1 Addition (Carries)

Binary addition is a straightforward 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. Example 2.14 shows how addition is performed on two individual bits.

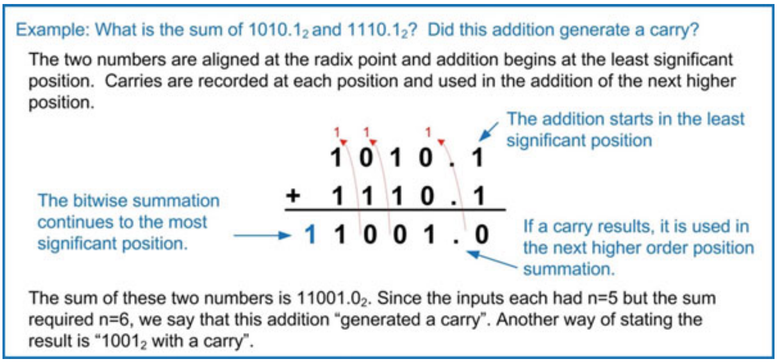


Example: Single Bit Binary Addition

There are four possible results when adding two bits.

Example 2.14
Single bit binary addition

When performing binary addition, the width of the inputs and output is fixed (i.e., n-bits). Carries 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. Example 2.15 shows this process when adding two 4-bit numbers.



Example: What is the sum of 1010.1_2 and 1110.1_2? Did this addition generate a carry?

The two numbers are aligned at the radix point and addition begins at the least significant position. Carries are recorded at each position and used in the addition of the next higher position.

The sum of these two numbers is 11001.0_2. Since the inputs each had n=5 but the sum required n=6, we say that this addition "generated a carry". Another way of stating the result is "1001_2 with a carry".

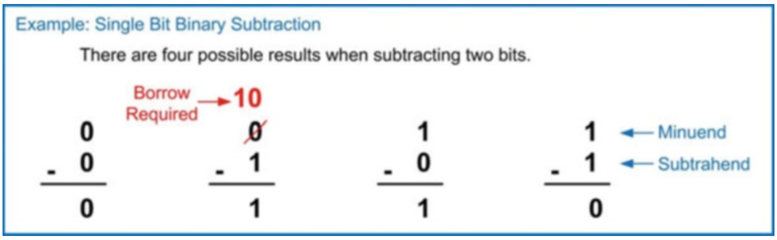
1 1 1
1 0 1 0 . 1
+ 1 1 1 0 . 1
1 1 0 0 1 . 0
The addition starts in the least significant position
The bitwise summation continues to the most significant position.
If a carry results, it is used in the next higher order position summation.

Example 2.15
Multiple bit 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} (i.e., $1111\ 1111_2$). 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 (i.e., 0), then it in turn needs to borrow from the next higher position, and so forth. Example 2.16 shows how subtraction is performed on two individual bits.

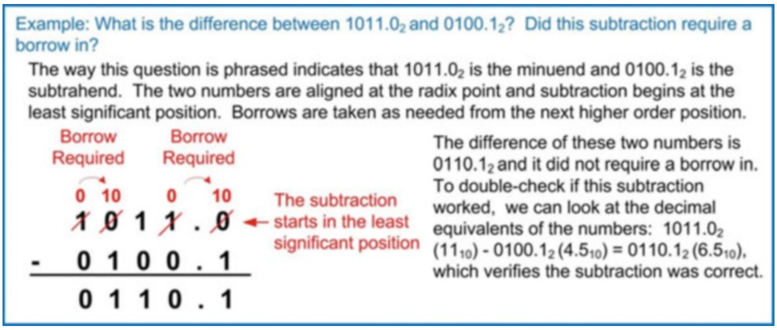


Example: Single Bit Binary Subtraction

There are four possible results when subtracting two bits.

Example 2.16
Single bit binary subtraction

As with binary addition, binary subtraction is accomplished on fixed widths of inputs and output (i.e., 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 move 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 2.17 shows this process when subtracting two 4-bit numbers.



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 phrased 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 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.

Example 2.17
Multiple bit 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.

Concept Check

CC2.3 If an 8-bit computer system can only perform unsigned addition on 8-bit inputs and produce an 8-bit sum, how is it possible for this computer to perform addition on numbers that are larger than what can be represented with 8-bits (e.g., $1{,}000_{10} + 1{,}000_{10} = 2{,}000_{10}$)?

(A) There are multiple 8-bit adders in a computer to handle large numbers.

(B) The result is simply rounded to the nearest 8-bit number.

(C) The computer returns an error and requires smaller numbers to be entered.

(D) The computer keeps track of the carry out and uses it in a subsequent 8-bit addition, which enables larger numbers to be handled.

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

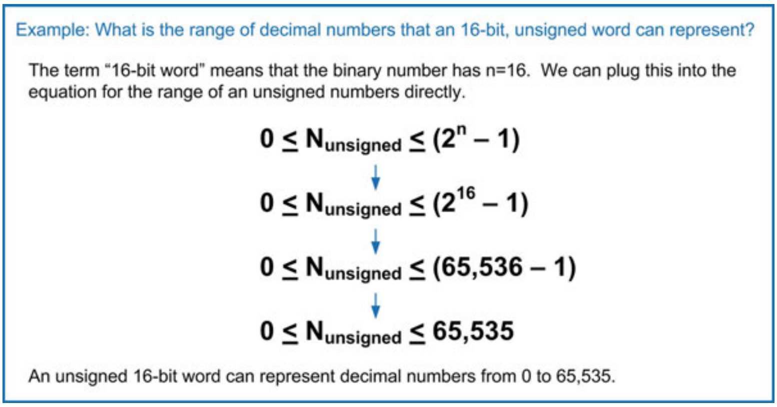
An unsigned number is one that does not allow negative numbers. When talking about this type of code, the number of bits is fixed and stated up front. We use the variable *n* to represent the number of bits in the number. For example, if we had an 8-bit number, we would say, “This is an 8-bit, unsigned 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: $0 < \mathrm{N_{unsigned}} < (2^n - 1)$

$$\textbf{Range of an UNSIGNED number} \Rightarrow 0 \leq \mathbf{N_{unsigned}} \leq (\mathbf{2^n}-\mathbf{1})$$

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 2.18 shows this process for a 16-bit number.



Example: What is the range of decimal numbers that an 16-bit, unsigned 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.

$$0 \leq \mathbf{N_{unsigned}} \leq (2^n - 1)$$
$$\downarrow$$
$$0 \leq \mathbf{N_{unsigned}} \leq (2^{16} - 1)$$
$$\downarrow$$
$$0 \leq \mathbf{N_{unsigned}} \leq (65{,}536 - 1)$$
$$\downarrow$$
$$0 \leq \mathbf{N_{unsigned}} \leq 65{,}535$$

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

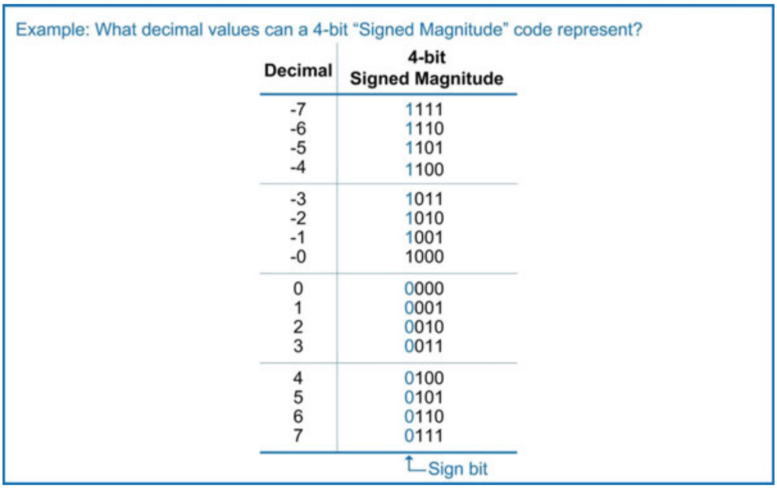
Example 2.18
Finding the range of an unsigned number

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 *two’s complement*. There are two other encoding techniques called *signed magnitude* and *one’s complement* that are rarely used but are studied to motivate the power of two’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, two’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 (i.e., 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. These are shown in Example 2.19.



Example: What decimal values can a 4-bit "Signed Magnitude" code represent?

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

Sign bit

Example 2.19
Decimal values that a 4-bit, signed magnitude code can represent

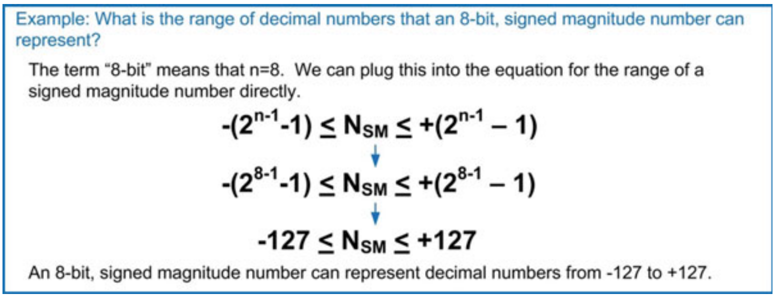
There are drawbacks of signed magnitude encoding that are apparent from this example. First, the value of 0_{10} has two signed magnitude codes (0000_2 and 1000_2). This is an inefficient use of the available codes and leads to complexity when building arithmetic circuitry since it must account for two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

$$\textbf{Range of a SIGNED MAGNITUDE number} \Rightarrow -\left(2^{n-1}-1\right) \leq N_{SM} \leq +\left(2^{n-1}-1\right)$$

Example 2.20 shows how to use this expression to find the range of decimal values that an 8-bit, signed magnitude code can represent.



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.

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

$$\downarrow$$

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

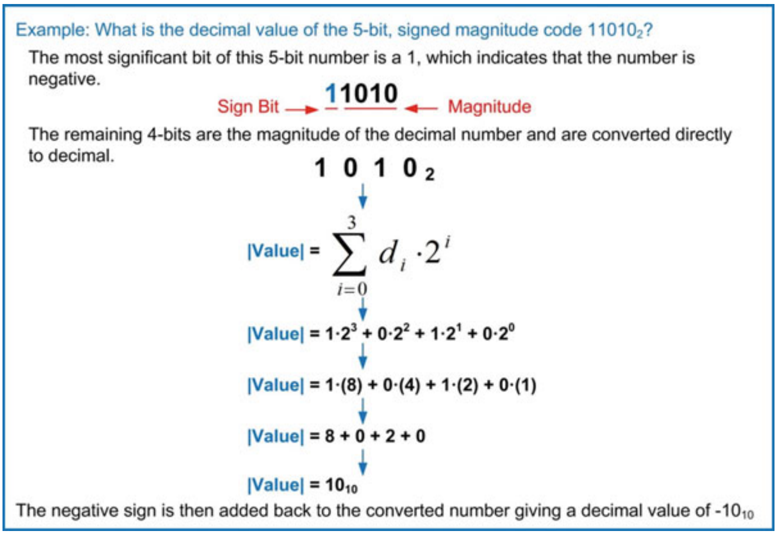
$$\downarrow$$

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

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

Example 2.20
Finding the range of a signed magnitude number

The process to determine the decimal value from a signed magnitude binary code involves treating the sign bit separately from the rest of the code. The sign bit provides the polarity of the decimal number (0 = Positive, 1 = Negative). The remaining bits in the code are treated as unsigned numbers and converted to decimal using the standard conversion procedure described in the prior sections. This conversion yields the magnitude of the decimal number. The final decimal value is found by applying the sign. Example 2.21 shows an example of this process.



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.

$$\text{Sign Bit} \rightarrow \underline{1}\,\underline{1010} \leftarrow \text{Magnitude}$$

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

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

$$\downarrow$$

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

$$\downarrow$$

$$|\text{Value}| = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

$$\downarrow$$

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

$$\downarrow$$

$$|\text{Value}| = 8 + 0 + 2 + 0$$

$$\downarrow$$

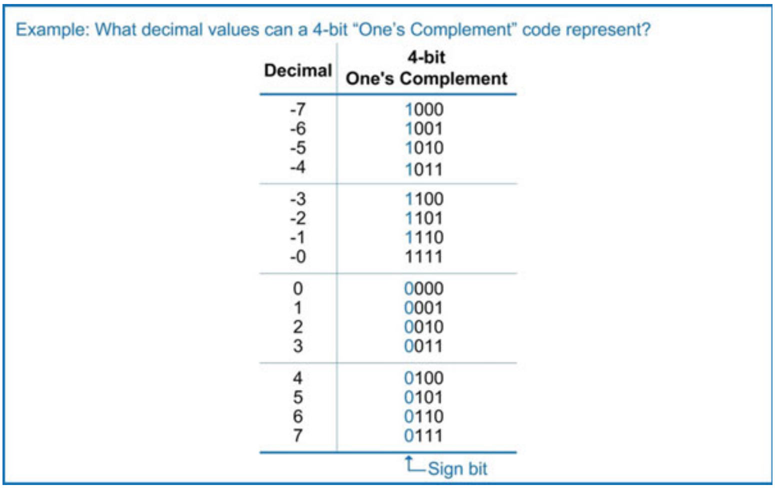
$$|\text{Value}| = 10_{10}$$

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

Example 2.21
Finding the 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. As an example of this approach, let's look at the decimal values that a 4-bit, one's complement number can take on. These are shown in Example 2.22.



Example: What decimal values can a 4-bit "One's Complement" code represent?

Decimal	4-bit One'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

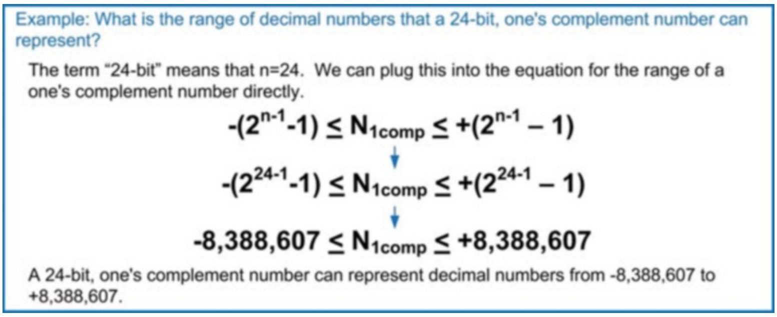
Example 2.22
Decimal values that a 4-bit, one's complement code can represent

Again, we notice that there are two different codes for 0_{10} (0000_2 and 1111_2). This is a drawback of one'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. There are advantages of one'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 one's complement the result yields -4, just as in a traditional decimal system. Notice in one'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 one'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.

If we use the notation N_{1comp} to represent any possible value that an n-bit, one's complement number can take on, the range is defined as:

$$\textbf{Range of a ONE'S COMPLEMENT number} \Rightarrow -\left(2^{n-1}-1\right) \leq N_{1\text{'s comp}} \leq +\left(2^{n-1}-1\right)$$

Example 2.23 shows how to use this expression to find the range of decimal values that a 24-bit, one's complement code can represent.



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

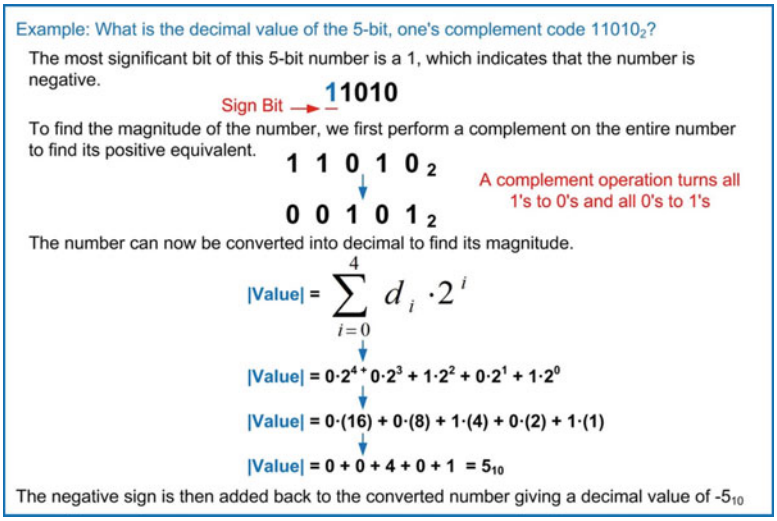
The term "24-bit" means that n=24. We can plug this into the equation for the range of a one's complement number directly.

$$-(2^{n-1}-1) \leq N_{1comp} \leq +(2^{n-1} - 1)$$
$$\downarrow$$
$$-(2^{24-1}-1) \leq N_{1comp} \leq +(2^{24-1} - 1)$$
$$\downarrow$$
$$-8{,}388{,}607 \leq N_{1comp} \leq +8{,}388{,}607$$

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

Example 2.23
Finding the range of a 1's complement number

The process of finding the decimal value of a one's complement number involves first identifying whether the number is positive or negative by looking at the sign bit. If the number is positive (i.e., the sign bit is 0), then the number is treated as an unsigned code and is converted to decimal using the standard conversion procedure described in prior sections. If the number is negative (i.e., the sign bit is 1), then the number sign is recorded separately and the code is complemented in order to convert it to its positive magnitude equivalent. This new positive number is then converted to decimal using the standard conversion procedure. As the final step, the sign is applied. Example 2.24 shows an example of this process.



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

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

Sign Bit
11010

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

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

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.

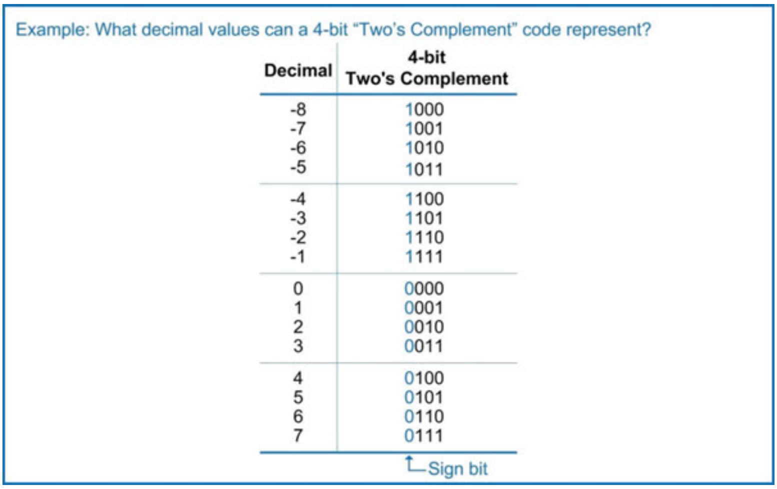
$$|\text{Value}| = \sum_{i=0}^{4} d_i \cdot 2^i$$
$$\downarrow$$
$$|\text{Value}| = 0{\cdot}2^4 + 0{\cdot}2^3 + 1{\cdot}2^2 + 0{\cdot}2^1 + 1{\cdot}2^0$$
$$\downarrow$$
$$|\text{Value}| = 0{\cdot}(16) + 0{\cdot}(8) + 1{\cdot}(4) + 0{\cdot}(2) + 1{\cdot}(1)$$
$$\downarrow$$
$$|\text{Value}| = 0 + 0 + 4 + 0 + 1 = 5_{10}$$

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

Example 2.24
Finding the 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 one. If a carry is generated, it is discarded. This procedure is called "*taking the two's complement of a number*". The procedure of complementing each bit and adding one is the most common technique to perform a two'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 two'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, two's complement number can take on. These are shown in Example 2.25.



Example: What decimal values can a 4-bit "Two's Complement" code represent?

Decimal	4-bit Two'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

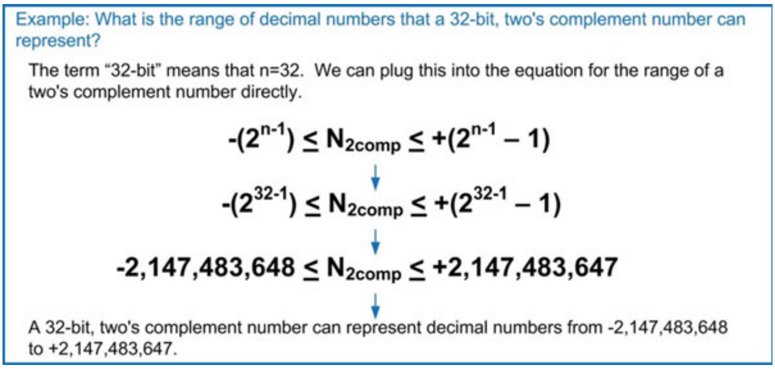
Example 2.25
Decimal values that a 4-bit, two's complement code can represent

There are many advantages of two'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. Another advantage of two'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 rollover characteristic is preserved from one's complement. Incrementing +7 by +1 will result in -8.

If we use the notation N_{2comp} to represent any possible value that an n-bit, two's complement number can take on, the range is defined as:

$$\textbf{Range of a TWO'S COMPLEMENT number} \Rightarrow -\left(2^{n-1}\right) \leq N_{2's\ comp} \leq +\left(2^{n-1}-1\right)$$

Example 2.26 shows how to use this expression to find the range of decimal values that a 32-bit, two's complement code can represent.



Example: What is the range of decimal numbers that a 32-bit, two'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 two's complement number directly.

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

$$\downarrow$$

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

$$\downarrow$$

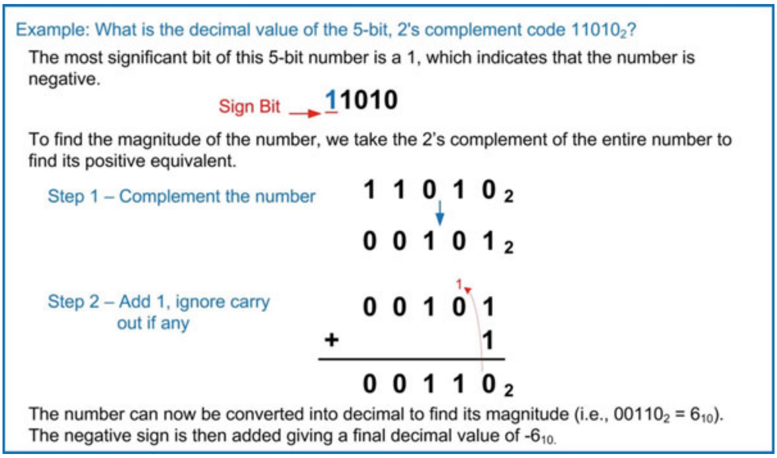
$$-2{,}147{,}483{,}648 \leq N_{2comp} \leq +2{,}147{,}483{,}647$$

$$\downarrow$$

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

Example 2.26
Finding the range of a two's complement number

The process of finding the decimal value of a two's complement number involves first identifying whether the number is positive or negative by looking at the sign bit. If the number is positive (i.e., the sign bit is 0), then the number is treated as an unsigned code and is converted to decimal using the standard conversion procedure described in prior sections. If the number is negative (i.e., the sign bit is 1), then the number sign is recorded separately and a two's complement is performed on the code in order to convert it to its positive magnitude equivalent. This new positive number is then converted to decimal using the standard conversion procedure. The final step is to apply the sign. Example 2.27 shows an example of this process.



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.

Sign Bit → 11010

To find the magnitude of the number, we take the 2's complement of the entire number to find its positive equivalent.

Step 1 – Complement the number

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

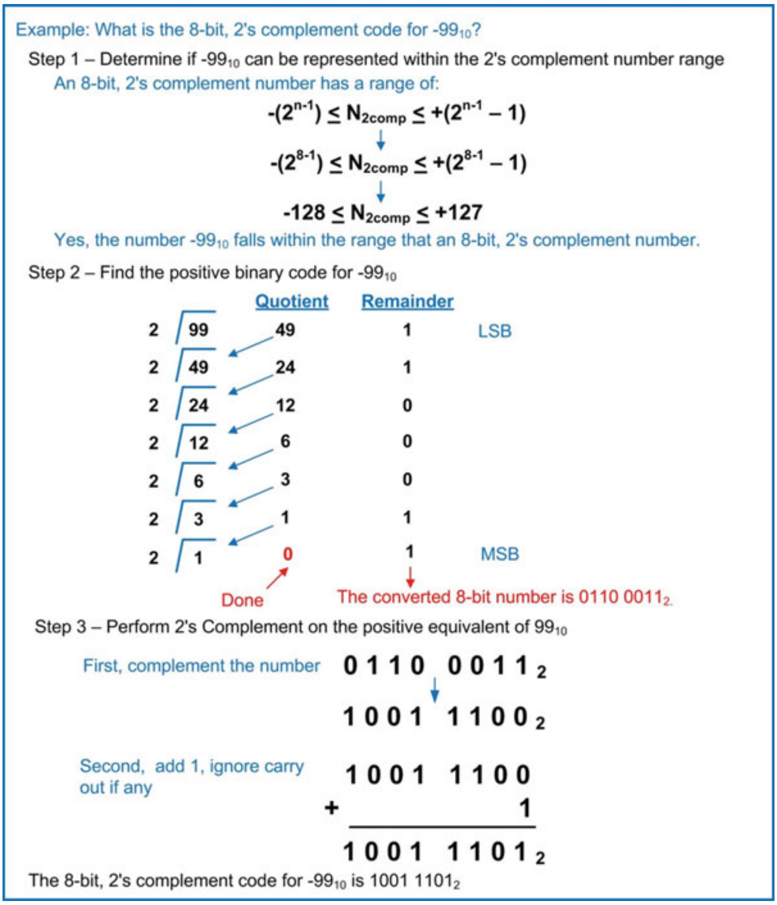
Step 2 – Add 1, ignore carry out if any

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

The number can now be converted into decimal to find its magnitude (i.e., $00110_2 = 6_{10}$). The negative sign is then added giving a final decimal value of -6_{10}.

Example 2.27
Finding the decimal value of a two's complement number

To convert a decimal number into its two's complement code, the range is first checked to determine whether the number can be represented with the allocated number of bits. The next step is to convert the decimal number into unsigned binary. The final step is to apply the sign bit. If the original decimal number was positive, then the conversion is complete. If the original decimal number was negative, then the two's complement is taken on the unsigned binary code to find its negative equivalent. Example 2.28 shows this procedure when converting -99_{10} to its 8-bit, two's complement code.



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:

$$-(2^{n-1}) \leq N_{2comp} \leq +(2^{n-1}-1)$$
$$\downarrow$$
$$-(2^{8-1}) \leq N_{2comp} \leq +(2^{8-1}-1)$$
$$\downarrow$$
$$-128 \leq N_{2comp} \leq +127$$

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

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

	Quotient	Remainder	
2 ⟌ 99	49	1	LSB
2 ⟌ 49	24	1	
2 ⟌ 24	12	0	
2 ⟌ 12	6	0	
2 ⟌ 6	3	0	
2 ⟌ 3	1	1	
2 ⟌ 1	0	1	MSB

Done

The converted 8-bit number is 0110 0011$_2$.

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

First, complement the number

$$0110\ 0011_2$$
$$\downarrow$$
$$1001\ 1100_2$$

Second, add 1, ignore carry out if any

$$\begin{array}{r} 1001\ 1100 \\ +\quad\quad\quad 1 \\ \hline 1001\ 1101_2 \end{array}$$

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

Example 2.28
Finding the two's complement code of a decimal number

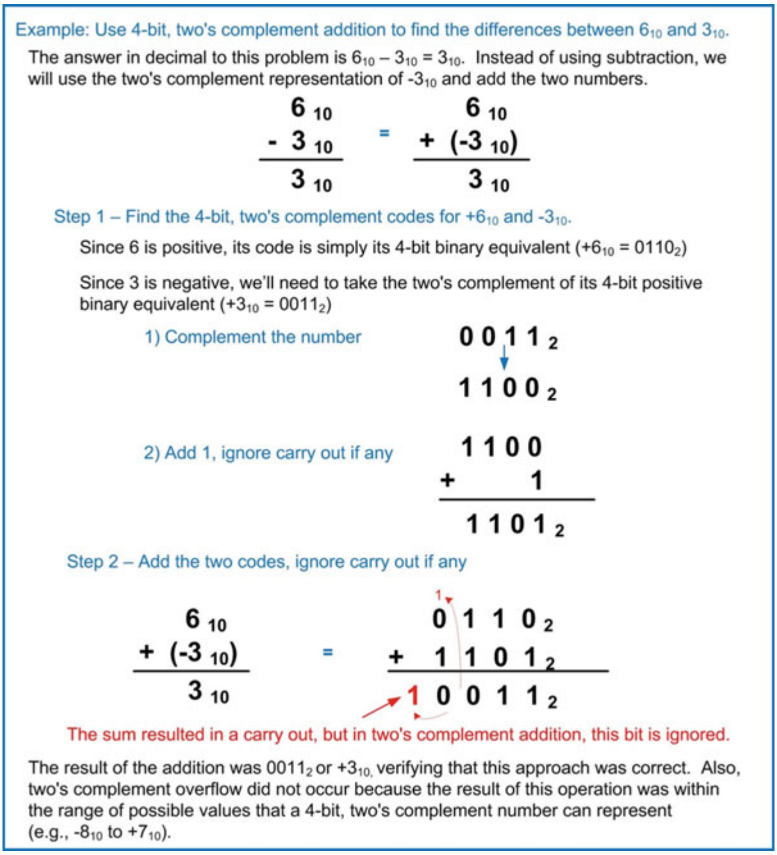
2.4.2.4 Arithmetic with Two's Complement

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 *two'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 two's complement of a number. Creating a circuit to perform two'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 two'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 **two'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, two'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 two'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 two'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 (i.e., Positive + Positive = Negative or Negative + Negative = Positive)
- The subtraction of a positive number from a negative number results in a positive number (i.e., Negative – Positive = Positive)
- The subtraction of a negative number from a positive number results in a negative number (i.e., Positive – Negative = Negative)

Computer systems that use two'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 straightforward since they simply monitor the sign bits of the input and output codes. Example 2.29 shows how to use two's complement in order to perform subtraction using an addition operation.



Example: Use 4-bit, two'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 two's complement representation of -3_{10} and add the two numbers.

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

Step 1 – Find the 4-bit, two'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 two's complement of its 4-bit positive binary equivalent ($+3_{10} = 0011_2$)

1) Complement the number

$$\begin{array}{c} 0011_2 \\ \downarrow \\ 1100_2 \end{array}$$

2) Add 1, ignore carry out if any

$$\begin{array}{r} 1100 \\ +\quad 1 \\ \hline 1101_2 \end{array}$$

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

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

The sum resulted in a carry out, but in two'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, two's complement overflow did not occur because the result of this operation was within the range of possible values that a 4-bit, two's complement number can represent (e.g., -8_{10} to $+7_{10}$).

Example 2.29
Two's complement addition

Concept Check

CC2.4 A 4-bit, two's complement number has 16 unique codes and can represent decimal numbers between -8_{10} to $+7_{10}$. If the number of unique codes is even, why is it that the range of integers it can represent is not symmetrical about zero?

(A) One of the positive codes is used to represent zero. This prevents the highest positive number from reaching $+8_{10}$ and being symmetrical.

(B) It is asymmetrical because the system allows the numbers to roll over.

(C) It isn't asymmetrical if zero is considered a positive integer. That way there are eight positive numbers and eight negatives numbers.

(D) It is asymmetrical because there are duplicate codes for 0.

Summary

- The base, or radix, of a number system refers to the number of unique symbols within its set. The definition of a number system includes both the symbols used and the relative values of each symbol within the set.
- The most common number systems are base 10 (decimal), base 2 (binary), and base 16 (hexadecimal). Base 10 is used because it is how the human brain has been trained to treat numbers. Base 2 is used because the two values are easily represented using electrical switches. Base 16 is a convenient way to describe large groups of bits.
- A positional number system allows larger (or smaller) numbers to be represented beyond the values within the original symbol set. This is accomplished by having each position within a number have a different *weight*.
- There are specific algorithms that are used to convert any base to or from decimal. There are also algorithms to convert between number systems that contain a power-of-two symbols (e.g., binary to hexadecimal and hexadecimal to binary).
- Binary arithmetic is performed on a fixed width of bits (n). When an n-bit addition results in a sum that cannot fit within n-bits, it generates a *carry out* bit. In an n-bit subtraction, if the minuend is smaller than the subtrahend, a *borrow in* can be used to complete the operation.
- Binary codes can represent both unsigned and signed numbers. For an arbitrary n-bit binary code, it is important to know the encoding technique and the range of values that can be represented.
- Signed numbers use the most significant position to represent whether the number is negative (0 = positive, 1 = negative). The width of a signed number is always fixed.
- Two's complement is the most common encoding technique for signed numbers. It has an advantage that there are no duplicate codes for zero and that the encoding approach provides a monotonic progression of codes from the most negative number that can be represented to the most positive. This allows addition and subtraction to work the same on two's complement numbers as it does on unsigned numbers.
- When performing arithmetic using two's complement codes, the carry bit is ignored.
- When performing arithmetic using two's complement codes, if the result lies outside of the range that can be represented it is called *two's complement overflow*. Two's complement overflow can be determined by looking at the sign bits of the input arguments and the sign bit of the result.

Exercise Problems

Section 2.1: Positional Number Systems

2.1.1 What is the radix of the binary number system?

2.1.2 What is the radix of the decimal number system?

2.1.3 What is the radix of the hexadecimal number system?

2.1.4 What is the radix of the octal number system?

2.1.5 For the number 261.367, what position (p) is the number 2 in?

2.1.6 For the number 261.367, what position (p) is the number 1 in?

2.1.7 For the number 261.367, what position (p) is the number 3 in?

2.1.8 For the number 261.367, what position (p) is the number 7 in?

2.1.9 What is the name of the number system containing 10_2?

2.1.10 What is the name of the number system containing 10_{10}?

2.1.11 What is the name of the number system containing 10_{16}?

2.1.12 What is the name of the number system containing 10_8?

2.1.13 Which of the four number systems covered in this chapter (i.e., binary, decimal, hexadecimal, and octal) could the number 22 be part of? Give all that are possible.

2.1.14 Which of the four number systems covered in this chapter (i.e., binary, decimal, hexadecimal, and octal) could the number 99 be part of? Give all that are possible.

2.1.15 Which of the four number systems covered in this chapter (i.e., binary, decimal, hexadecimal, and octal) could the number 1F be part of? Give all that are possible.

2.1.16 Which of the four number systems covered in this chapter (i.e., binary, decimal, hexadecimal, and octal) could the number 88 be part of? Give all that are possible.

Section 2.2: Base Conversions

2.2.1 If the number 101.111 has a radix of 2, what is the weight of the position containing the bit 0?

2.2.2 If the number 261.367 has a radix of 10, what is the weight of the position containing the numeral 2?

2.2.3 If the number 261.367 has a radix of 16, what is the weight of the position containing the numeral 1?

2.2.4 If the number 261.367 has a radix of 8, what is the weight of the position containing the numeral 3?

2.2.5 Convert $1100\ 1100_2$ to decimal. Treat all numbers as unsigned.

2.2.6 Convert 1001.1001_2 to decimal. Treat all numbers as unsigned.

2.2.7 Convert 72_8 to decimal. Treat all numbers as unsigned.

2.2.8 Convert 12.57_8 to decimal. Treat all numbers as unsigned.

2.2.9 Convert F3_{16} to decimal. Treat all numbers as unsigned.

2.2.10 Convert 15B.CEF_{16} to decimal. Treat all numbers as unsigned. Use an accuracy of 7 fractional digits.

2.2.11 Convert 67_{10} to binary. Treat all numbers as unsigned.

2.2.12 Convert 252.987_{10} to binary. Treat all numbers as unsigned. Use an accuracy of 4 fractional bits and don't round up.

2.2.13 Convert 67_{10} to octal. Treat all numbers as unsigned.

2.2.14 Convert 252.987_{10} to octal. Treat all numbers as unsigned. Use an accuracy of 4 fractional digits and don't round up.

2.2.15 Convert 67_{10} to hexadecimal. Treat all numbers as unsigned.

2.2.16 Convert 252.987_{10} to hexadecimal. Treat all numbers as unsigned. Use an accuracy of 4 fractional digits and don't round up.

2.2.17 Convert $1\ 0000\ 1111_2$ to octal. Treat all numbers as unsigned.

2.2.18 Convert $1\ 0000\ 1111.011_2$ to hexadecimal. Treat all numbers as unsigned.

2.2.19 Convert 77_8 to binary. Treat all numbers as unsigned.

2.2.20 Convert F.A_{16} to binary. Treat all numbers as unsigned.

2.2.21 Convert 66_8 to hexadecimal. Treat all numbers as unsigned.

2.2.22 Convert AB.D_{16} to octal. Treat all numbers as unsigned.

Section 2.3: Binary Arithmetic

2.3.1 Compute $1010_2 + 1011_2$ by hand. Treat all numbers as unsigned. Provide the 4-bit sum and indicate whether a *carry out* occurred.

2.3.2 Compute $1111\ 1111_2 + 0000\ 0001_2$ by hand. Treat all numbers as unsigned. Provide the 8-bit sum and indicate whether a *carry out* occurred.

2.3.3 Compute $1010.1010_2 + 1011.1011_2$ by hand. Treat all numbers as unsigned. Provide the 8-bit sum and indicate whether a *carry out* occurred.

2.3.4 Compute $1111\ 1111.1011_2 + 0000\ 0001.1100_2$ by hand. Treat all numbers as unsigned. Provide the 12-bit sum and indicate whether a *carry out* occurred.

2.3.5 Compute $1010_2 - 1011_2$ by hand. Treat all numbers as unsigned. Provide the 4-bit difference and indicate whether a *borrow in* occurred.

2.3.6 Compute $1111\ 1111_2 - 0000\ 0001_2$ by hand. Treat all numbers as unsigned. Provide the 8-bit difference and indicate whether a *borrow in* occurred.

2.3.7 Compute $1010.1010_2 - 1011.1011_2$ by hand. Treat all numbers as unsigned. Provide the 8-bit difference and indicate whether a *borrow in* occurred.

2.3.8 Compute $1111\ 1111.1011_2 - 0000\ 0001.1100_2$ by hand. Treat all numbers as unsigned. Provide the 12-bit difference and indicate whether a *borrow in* occurred.

Section 2.4: Unsigned and Signed Numbers

2.4.1 What range of decimal numbers can be represented by 8-bit, two's complement numbers?

2.4.2 What range of decimal numbers can be represented by 16-bit, two's complement numbers?

2.4.3 What range of decimal numbers can be represented by 32-bit, two's complement numbers?

2.4.4 What range of decimal numbers can be represented by 64-bit, two's complement numbers?

2.4.5 What is the 8-bit, two's complement code for $+88_{10}$?

2.4.6 What is the 8-bit, two's complement code for -88_{10}?

2.4.7 What is the 8-bit, two's complement code for -128_{10}?

2.4.8 What is the 8-bit, two's complement code for -1_{10}?

2.4.9 What is the decimal value of the 4-bit, two's complement code 0010_2?

2.4.10 What is the decimal value of the 4-bit, two's complement code 1010_2?

2.4.11 What is the decimal value of the 8-bit, two's complement code $0111\ 1110_2$?

2.4.12 What is the decimal value of the 8-bit, two's complement code $1111\ 1110_2$?

2.4.13 Compute $1110_2 + 1011_2$ by hand. Treat all numbers as 4-bit, two's complement codes. Provide the 4-bit sum and indicate whether *two's complement overflow* occurred.

2.4.14 Compute $1101\ 1111_2 + 0000\ 0001_2$ by hand. Treat all numbers as 8-bit, two's complement codes. Provide the 8-bit sum and indicate whether *two's complement overflow* occurred.

2.4.15 Compute $1010.1010_2 + 1000.1011_2$ by hand. Treat all numbers as 8-bit, two's complement codes. Provide the 8-bit sum and indicate whether *two's complement overflow* occurred.

2.4.16 Compute $1110\ 1011.1001_2 + 0010\ 0001.1101_2$ by hand. Treat all numbers as 12-bit, two's complement codes. Provide the 12-bit sum and indicate whether *two's complement overflow* occurred.

2.4.17 Compute $4_{10} - 5_{10}$ using 4-bit two's complement addition. You will need to first convert each number into its 4-bit two's complement code and then perform binary addition (i.e., $4_{10} + (-5_{10})$). Provide the 4-bit result and indicate whether two's complement overflow occurred. Check your work by converting the 4-bit result back to decimal.

2.4.18 Compute $7_{10} - 7_{10}$ using 4-bit two's complement addition. You will need to first convert each decimal number into its 4-bit two's complement code and then perform binary addition (i.e., $7_{10} + (-7_{10})$). Provide the 4-bit result and indicate whether two's complement overflow occurred. Check your work by converting the 4-bit result back to decimal.

2.4.19 Compute $7_{10} + 1_{10}$ using 4-bit two's complement addition. You will need to first convert each decimal number into its 4-bit two's complement code and then perform binary addition. Provide the 4-bit result and indicate whether two's complement overflow occurred. Check your work by converting the 4-bit result back to decimal.

2.4.20 Compute $64_{10} - 100_{10}$ using 8-bit two's complement addition. You will need to first convert each number into its 8-bit two's complement code and then perform binary addition (i.e., $64_{10} + (-100_{10})$). Provide the 8-bit result and indicate whether two's complement overflow occurred. Check your work by converting the 8-bit result back to decimal.

2.4.21 Compute $(-99)_{10} - 11_{10}$ using 8-bit two's complement addition. You will need to first convert each decimal number into its 8-bit two's complement code and then perform binary addition (i.e., $(-99_{10}) + (-11_{10})$). Provide the 8-bit result and indicate whether two's complement overflow occurred. Check your work by converting the 8-bit result back to decimal.

2.4.22 Compute $50_{10} + 100_{10}$ using 8-bit two's complement addition. You will need to first convert each decimal number into its 8-bit two's complement code and then perform binary addition. Provide the 8-bit result and indicate whether two's complement overflow occurred. Check your work by converting the 8-bit result back to decimal.