2

FOUNDATIONS OF BINARY ARITHMETIC

What to Expect

The language of digital logic is the binary number system and this chapter introduces that system. Included are these topics:

- The various bases used in digital logic: binary (base 2), octal (base 8), decimal (base 10), and hexadecimal (base 16)
- Converting numbers between the bases
- Representing floating point numbers in binary

2.1 INTRODUCTION TO NUMBER SYSTEMS

2.1.1 *Background*

The expression of numerical quantities is often taken for granted, which is both a good and a bad thing in the study of electronics. It is good since the use and manipulation of numbers is familiar for many calculations used in analyzing electronic circuits. On the other hand, the particular system of notation that has been taught from primary school onward is not the system used internally in modern electronic computing devices and learning any different system of notation requires some re-examination of assumptions.

It is important to distinguish the difference between numbers and the symbols used to represent numbers. A number is a mathematical quantity, usually correlated in electronics to a physical quantity such as voltage, current, or resistance. There are many different types of numbers, for example:

- Whole Numbers: $1, 2, 3, 4, 5, 6, 7, 8, 9...$
- Integers: $-4, -3, -2, -1, 0, 1, 2, 3, 4...$
- Rational Numbers: $-5.3, 0, \frac{1}{3}, 6.7$
- Irrational Numbers: π (approx. 3.1416), e (approx. 2.7183), and the square root of any prime number
- Real Numbers: (combination of all rational and irrational numbers)

- Complex Numbers: $3 - j4$

Different types of numbers are used for different applications in electronics. As examples:

- Whole numbers work well for counting discrete objects, such as the number of resistors in a circuit.
- Integers are needed to express a negative voltage or current.
- Irrational numbers are used to describe the charge/discharge cycle of electronic objects like capacitors.
- Real numbers, in either fractional or decimal form, are used to express the non-integer quantities of voltage, current, and resistance in circuits.
- Complex numbers, in either rectangular or polar form, must be used rather than real numbers to capture the dual essence of the magnitude and phase angle of the current and voltage in alternating current circuits.

There is a difference between the concept of a "number" as a measure of some quantity and "number" as a means used to express that quantity in spoken or written communication. A way to symbolically denote numbers had to be developed in order to use them to describe processes in the physical world, make scientific predictions, or balance a checkbook. The written symbol that represents some number, like how many apples there are in a bin, is called a *cipher* and in western , the commonly-used ciphers are $0, 1, 2, 3, 4, 5, 6, 7, 8$, and 9.

2.1.2 *Binary Mathematics*

Binary mathematics is a specialized branch of mathematics that concerns itself with a number system that contains only two ciphers: zero and one. It would seem to be very limiting to use only two ciphers; however, it is much easier to create electronic devices that can differentiate between two voltage levels rather than the ten that would be needed for a decimal system.

2.1.3 *Systems Without Place Value*

HASH MARKS. One of the earliest cipher systems was to simply use a hash mark to represent each quantity. For example, three apples could be represented like this: |||. Often, five hash marks were "bundled" to aid in the counting of large quantities, so eight apples would be represented like this: 𝍸 |||.

ROMAN NUMERALS. The Romans devised a system that was a substantial improvement over hash marks, because it used a variety of ciphers to represent increasingly large quantities. The notation for one is the capital letter *I*. The notation for 5 is the capital letter *V*. Other ciphers, as listed in Table 2.1, possess increasing values:

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Table 2.1: Roman Numerals

If a cipher is accompanied by a second cipher of equal or lesser value to its immediate right, with no ciphers greater than that second cipher to its right, the second cipher's value is added to the total quantity. Thus, *VIII* symbolizes the number 8, and *CLVII* symbolizes the number 157. On the other hand, if a cipher is accompanied by another cipher of lesser value to its immediate left, that other cipher's value is subtracted from the first. In that way, *IV* symbolizes the number 4 (*V* minus *I*), and *CM* symbolizes the number 900 (*M* minus *C*). The ending credit sequences for most motion pictures contain the date of production, often in Roman numerals. For the year 1987, it would read: *MCMLXXXVII*. To break this numeral down into its constituent parts, from left to right:

$$(\mathrm{M} = 1000) + (\mathrm{CM} = 900) + (\mathrm{LXXX} = 80) + (\mathrm{VII} = 7)$$

Large numbers are very difficult to denote with Roman numerals; and the left vs. right (or subtraction vs. addition) of values can be very confusing. Adding and subtracting two Roman numerals is also very challenging, to say the least. Finally, one other major problem with this system is that there is no provision for representing the number zero or negative numbers, and both are very important concepts in mathematics. Roman culture, however, was more pragmatic with respect to mathematics than most, choosing only to develop their numeration system as far as it was necessary for use in daily life.

2.1.4 *Systems With Place Value*

DECIMAL NUMERATION. The Babylonians developed one of the most important ideas in numeration: cipher position, or place value,

to represent larger numbers. Instead of inventing new ciphers to represent larger numbers, as the Romans had done, they re-used the same ciphers, placing them in different positions from right to left to represent increasing values. This system also required a cipher that represents zero value, and the inclusion of zero in a numeric system was one of the most important inventions in all of mathematics (many would argue zero was the single most important human invention, period). The decimal numeration system uses the concept of place value, with only ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used in "weighted" positions to symbolize numbers.

Each cipher represents an integer quantity, and each place from right to left in the notation is a multiplying constant, or weight, for the integer quantity. For example, the decimal notation "1206" may be broken down into its constituent weight-products as such:

$$1206 = (1X1000) + (2X100) + (0X10) + (6X1)$$

Each cipher is called a "digit" in the decimal numeration system, and each weight, or place value, is ten times that of the place to the immediate right. So, working from right to left is a "ones" place, a "tens" place, a "hundreds" place, a "thousands" place, and so on.

While the decimal numeration system uses ten ciphers, and place-weights that are multiples of ten, it is possible to make a different numeration system using the same strategy, except with fewer or more ciphers.

BINARY NUMERATION. The binary numeration system uses only two ciphers and the weight for each place in a binary number is two times as much as the place to its right. Contrast this to the decimal numeration system that has ten different ciphers and the weight for each place is ten times the place to its right. The two ciphers for the binary system are zero and one, and these ciphers are arranged right-to-left in a binary number, each place doubling the weight of the previous place. The rightmost place is the "ones" place; and, moving to the left, is the "twos" place, the "fours" place, the "eights" place, the "sixteens" place, and so forth. For example, the binary number 11010 can be expressed as a sum of each cipher value times its respective weight:

$$11010 = (1X16) + (1X8) + (0X4) + (1X2) + (0X1)$$

The primary reason that the binary system is popular in modern electronics is because it is easy to represent the two cipher states (zero and one) electronically; if no current is flowing in the circuit it represents a binary zero while flowing current represents a binary one. Binary numeration also lends itself to the storage and retrieval of numerical information: as examples, magnetic tapes have spots of iron oxide that are magnetized for a binary one or demagnetized

for a binary zero and optical disks have a laser-burned pit in the aluminum substrate representing a binary one and an unburned spot representing a binary zero.

Digital numbers require so many bits to represent relatively small numbers that programming or analyzing electronic circuitry can be a tedious task. However, anyone working with digital devices soon learns to quickly count in binary to at least 11111 (that is decimal 31). Any time spent practicing counting both up and down between zero and 11111 will be rewarded while studying binary mathematics, codes, and other digital logic topics. Table 2.2 will help in memorizing binary numbers:

Bin	Dec	Bin	Dec	Bin	Dec	Bin	Dec
0	0	1000	8	10000	16	11000	24
1	1	1001	9	10001	17	11001	25
10	2	1010	10	10010	18	11010	26
11	3	1011	11	10011	19	11011	27
100	4	1100	12	10100	20	11100	28
101	5	1101	13	10101	21	11101	29
110	6	1110	14	10110	22	11110	30
111	7	1111	15	10111	23	11111	31

Table 2.2: Binary-Decimal Conversion

OCTAL NUMERATION. The octal numeration system is place-weighted with a base of eight. Valid ciphers include the symbols $0, 1, 2, 3, 4, 5, 6,$ and 7. These ciphers are arranged right-to-left in an octal number, each place being eight times the weight of the previous place. For example, the octal number 4270 can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$4270 = (4X512) + (2X64) + (7X8) + (0X1)$$

HEXADECIMAL NUMERATION. The hexadecimal numeration system is place-weighted with a base of sixteen. There needs to be ciphers for numbers greater than nine so English letters are used for those values. Table 2.3 lists hexadecimal numbers up to decimal 15:

The word "hexadecimal" is a combination of "hex" for six and "decimal" for ten

Hex	Dec	Hex	Dec
0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

Table 2.3: Hexadecimal Numbers

Hexadecimal ciphers are arranged right-to-left, each place being 16 times the weight of the previous place. For example, the hexadecimal number 13A2 can be expressed, just like a decimal number, as a sum of each cipher value times its respective weight:

$$13A2 = (1X4096) + (3X256) + (AX16) + (2X1)$$

Maximum Number Size

It is important to know the largest number that can be represented with a given number of cipher positions. For example, if only four cipher positions are available then what is the largest number that can be represented in each of the numeration systems? With the crude hash-mark system, the number of places IS the largest number that can be represented, since one hash mark "place" is required for every integer step. For place-weighted systems, however, the answer is found by taking the number base of the numeration system (10 for decimal, 2 for binary) and raising that number to the power of the number of desired places. For example, in the decimal system, a five-place number can represent 10^5, or $100,000$, with values from zero to $99,999$. Eight places in a binary numeration system, or 2^8, can represent 256 different values, $0 - 255$.

2.1.5 *Summary of Numeration Systems*

Table 2.4 counts from zero to twenty using several different numeration systems:

Text	Hash Marks	Roman	Dec	Bin	Oct	Hex
Zero	n/a	n/a	0	0	0	0
One	|	I	1	1	1	1
Two	||	II	2	10	2	2
Three	|||	III	3	11	3	3
Four	||||	IV	4	100	4	4
Five	𝍸	V	5	101	5	5
Six	𝍸 |	VI	6	110	6	6
Seven	𝍸 ||	VII	7	111	7	7
Eight	𝍸 |||	VIII	8	1000	10	8
Nine	𝍸 ||||	IX	9	1001	11	9
Ten	𝍸 𝍸	X	10	1010	12	A
Eleven	𝍸 𝍸 |	XI	11	1011	13	B
Twelve	𝍸 𝍸 ||	XII	12	1100	14	C
Thirteen	𝍸 𝍸 |||	XIII	13	1101	15	D
Fourteen	𝍸 𝍸 ||||	XIV	14	1110	16	E
Fifteen	𝍸 𝍸 𝍸	XV	15	1111	17	F
Sixteen	𝍸 𝍸 𝍸 |	XVI	16	10000	20	10
Seventeen	𝍸 𝍸 𝍸 ||	XVII	17	10001	21	11
Eighteen	𝍸 𝍸 𝍸 |||	XVIII	18	10010	22	12
Nineteen	𝍸 𝍸 𝍸 ||||	XIX	19	10011	23	13
Twenty	𝍸 𝍸 𝍸 𝍸	XX	20	10100	24	14

Table 2.4: Counting To Twenty

Numbers for Computer Systems

An interesting footnote for this topic concerns one of the first electronic digital computers: ENIAC. The designers of the ENIAC chose to work with decimal numbers rather than binary in order to emulate a mechanical adding machine; unfortunately, this approach turned out to be counter-productive and required more circuitry (and maintenance nightmares) than if they had they used binary numbers. "ENIAC contained 17,468 vacuum tubes, 7,200 crystal diodes, 1,500 relays, 70,000 resistors, 10,000 capacitors and around 5 million hand-soldered joints"[a]. Today, all digital devices use binary numbers for internal calculation and storage and then convert those numbers to/from decimal only when necessary to interface with human operators.

a http://en.wikipedia.org/wiki/Eniac

2.1.6 *Conventions*

Using different numeration systems can get confusing since many ciphers, like "1," are used in several different numeration systems. Therefore, the numeration system being used is typically indicated with a subscript following a number, like 11010_2 for a binary number or 26_{10} for a decimal number. The subscripts are not mathematical operation symbols like superscripts, which are exponents; all they do is indicate the system of numeration being used. By convention, if no subscript is shown then the number is assumed to be decimal.

In this book, subscripts are normally used to make it clear whether the number is binary or some other system.

Another method used to represent hexadecimal numbers is the prefix 0x. This has been used for many years by programmers who work with any of the languages descended from C, like C++, C#, Java, JavaScript, and certain shell scripts. Thus, 0x1A would be the hexadecimal number 1A.

One other commonly used convention for hexadecimal numbers is to add an *h* (for *hexadecimal*) after the number. This is used because that is easier to enter with a keyboard than to use a subscript and is more intuitive than using a 0x prefix. Thus, $1A_{16}$ would be written 1Ah. In this case, the *h* only indicates that the number 1A is hexadecimal; it is not some sort of mathematical operator.

Occasionally binary numbers are written with a 0b prefix; thus 0b1010 would be 1010_2, but this is a programmer's convention not often found elsewhere.

2.2 CONVERTING BETWEEN RADICES

2.2.1 *Introduction*

The radix of a system is also commonly called its "base."

The number of ciphers used by a number system (and therefore, the place-value multiplier for that system) is called the *radix* for the system. The binary system, with two ciphers (zero and one), is radix two numeration, and each position in a binary number is a *b*inary dig*it* (or *bit*). The decimal system, with ten ciphers, is radix-ten numeration, and each position in a decimal number is a *digit*. When working with various digital logic processes it is desirable to be able to convert between binary/octal/decimal/hexadecimal radices.

2.2.2 *Expanded Positional Notation*

Expanded Positional Notation is a method of representing a number in such a way that each position is identified with both its cipher symbol and its place-value multiplier. For example, consider the number 347_{10}:

$$347_{10} = (3X10^2) + (4X10^1) + (7X10^0) \tag{2.1}$$

The steps to use to expand a decimal number like 347 are found in Table 2.5.

Step	Result
Count the number of digits in the number.	Three Digits
Create a series of $(\_X\_)$ connected by plus signs such that there is one set for each of the digits in the original number.	$(\_X\_) + (\_X\_) + (\_X\_)$
Fill in the digits of the original number on the left side of each set of parenthesis.	$(3X\_) + (4X\_) + (7X\_)$
Fill in the radix (or base number) on the right side of each parenthesis.	$(3X10) + (4X10) + (7X10)$
Starting on the far right side of the expression, add an exponent (power) for each of the base numbers. The powers start at zero and increase to the left.	$(3X10^2) + (4X10^1) + (7X10^0)$

Table 2.5: Expanding a Decimal Number

Additional examples of expanded positional notation are:

$$2413_{10} = (2x10^3) + (4X10^2) + (1X10^1) + (3X10^0) \tag{2.2}$$

$$1052_8 = (1X8^3) + (0X8^2) + (5X8^1) + (2X8^0) \tag{2.3}$$

$$139_{16} = (1X16^2) + (3X16^1) + (9X16^0) \tag{2.4}$$

The above examples are for positive decimal integers; but a number with any radix can also have a fractional part. In that case, the number's integer component is to the left of the radix point (called the "decimal point" in the decimal system), while the fractional part is to the right of the radix point. For example, in the number 139.25_{10}, 139 is the integer component while 25 is the fractional component. If a number includes a fractional component, then the expanded positional notation uses increasingly negative powers of the radix for numbers to the right of the radix point. Consider this binary example: 101.011_2. The expanded positional notation for this number is:

$$101.011_2 = (1X2^2) + (0X2^1) + (1X2^0) + (0X2^{-1}) + (1X2^{-2}) + (1X2^{-3}) \tag{2.5}$$

Other examples are:

$$526.14_{10} = (5X10^2) + (2X10^1) + (6X10^0) + (1X10^{-1}) + (4X10^{-2}) \tag{2.6}$$

$$65.147_8 = (6X8^1) + (5X8^0) + (1X8^{-1}) + (4X8^{-2}) + (7X8^{-3}) \tag{2.7}$$

$$D5.3A_{16} = (13X16^1) + (5X16^0) + (3X16^{-1}) + (10X16^{-2}) \tag{2.8}$$

When a number in expanded positional notation includes one or more negative radix powers, the radix point is assumed to be to the immediate right of the "zero" exponent term, but it is not actually written into the notation. Expanded positional notation is useful in converting a number from one base to another.

2.2.3 *Binary to Decimal*

To convert a number in binary form to decimal, start by writing the binary number in expanded positional notation, calculate the values for each of the sets of parenthesis in decimal, and then add all of the values. For example, convert 1101_2 to decimal:

$$\begin{aligned} 1101_2 &= (1X2^3) + (1X2^2) + (0X2^1) + (1X2^0) \\ &= (8) + (4) + (0) + (1) \\ &= 13_{10} \end{aligned} \tag{2.9}$$

Binary numbers with a fractional component are converted to decimal in exactly the same way, but the fractional parts use negative powers of two. Convert binary 10.11_2 to decimal:

$$\begin{aligned} 10.11_2 &= (1X2^1) + (0X2^0) + (1X2^{-1}) + (1X2^{-2}) \\ &= (2) + (0) + (\frac{1}{2}) + (\frac{1}{4}) \\ &= 2 + .5 + .25 \\ &= 2.75_{10} \end{aligned} \tag{2.10}$$

Most technicians who work with digital circuits learn to quickly convert simple binary integers to decimal in their heads. However, for longer numbers, it may be useful to write down the various place weights and add them up; in other words, a shortcut way of writing expanded positional notation. For example, convert the binary number 11001101_2 to decimal:

```
Binary Number:  1 1 0 0 1 1 0 1
                - - - - - - - -
(Read Down)     1 6 3 1 8 4 2 1
                  2 4 2 6
                  8
```

A bit value of one in the original number means that the respective place weight gets added to the total value, while a bit value of zero means that the respective place weight does not get added to the total value. Thus, using the example above this paragraph, the binary number 11001101_2 is converted to: $128 + 64 + 8 + 4 + 1$, or 205_{10}.

Naming Conventions

The bit on the right end of any binary number is the Least Significant Bit (LSB) because it has the least weight (the ones place) while the bit on the left end is the Most Significant Bit (MSB) because it has the greatest weight. Also, groups of bits are normally referred to as *words*, so engineers would speak of 16-bit or 32-bit words. As exceptions, an eight-bit group is commonly called a *byte* and a four-bit group is called a *nibble* (occasionally spelled *nybble*).

2.2.4 *Binary to Octal*

"Five Seven Octal" is not pronounced "Fifty Seven" since "fifty" is a decimal number.

The octal numeration system serves as a "shorthand" method of denoting a large binary number. Technicians find it easier to discuss a number like 57_8 rather than 101111_2.

Because octal is a base eight system, and eight is 2^3, binary numbers can be converted to octal by creating groups of three and then simplifying each group. As an example, convert 101111_2 to octal:

```
101 111
 5   7
```

Thus, 101111_2 is equal to 57_8.

If a binary integer cannot be grouped into an even grouping of three, it is padded on the left with zeros. For example, to convert 11011101_2 to octal, the most significant bit must be padded with a zero:

```
011 011 101
 3   3   5
```

Thus, 11011101_2 is equal to 335_8.

A binary fraction may need to be padded on the right with zeros in order to create even groups of three before it is converted into octal. For example, convert 0.1101101_2 to octal:

```
0 . 110 110 100
0 .  6   6   4
```

Thus, 0.1101101_2 is equal to $.664_8$.

A binary mixed number may need to be padded on both the left and right with zeros in order to create even groups of three before it can be converted into octal. For example, convert 10101.00101_2 to octal:

```
010 101 . 001 010
 2   5  .  1   2
```

Thus, 10101.00101_2 is equal to 25.12_8.

Table 2.6 lists additional examples of binary/octal conversion:

Binary	Octal
100 101.011	45.3
1 100 010.1101	142.64
100 101 011.110 100 1	453.644
1 110 010 011 101.000 110 10	16 235.064
110 011 010 100 111.011 101	63 247.35

Table 2.6: Binary-Octal Conversion Examples

While it is easy to convert between binary and octal, the octal system is not frequently used in electronics since computers store and transmit binary numbers in words of 16, 32, or 64 bits, which are multiples of four rather than three.

2.2.5 *Binary to Hexadecimal*

The hexadecimal numeration system serves as a "shorthand" method of denoting a large binary number. Technicians find it easier to discuss a number like $2F_{16}$ rather than 101111_2. Because hexadecimal is a base 16 system, and 16 is 2^4; binary numbers can be converted to hexadecimal by creating groups of four and then simplifying each group. As an example, convert 10010111_2 to hexadecimal:

```
1001 0111
 9    7
```

Thus, 10010111_2 is equal to 97_{16}.

"Nine Seven Hexadecimal," or, commonly, "Nine Seven Hex," is not pronounced "Ninety Seven" since "ninety" is a decimal number.

A binary integer may need to be padded on the left with zeros in order to create even groups of four before it can be converted into hexadecimal. For example, convert 1001010110_2 to hexadecimal:

```
0010 0101 0110
 2    5    6
```

Thus, 1001010110_2 is equal to 256_{16}.

A binary fraction may need to be padded on the right with zeros in order to create even groups of four before it can be converted into hexadecimal. For example, convert 0.1001010110_2 to hexadecimal:

```
0 . 1001 0101 1000
  . 9    5    8
```

Thus, 0.1001010110_2 is equal to 0.958_{16}.

A binary mixed number may need to be padded on both the left and right with zeros in order to create even groups of four before it can be converted into hexadecimal. For example, convert 11101.10101_2 to hexadecimal:

```
0001 1101 . 1010 1000
 1    D   .  A    8
```

Thus, 11101.10101_2 is equal to $1D.A8_{16}$.

Table 2.7 lists additional examples of binary/hexadecimal conversion:

Binary	Hexadecimal
100 101.011	25.6
1 100 010.1101	62.D
100 101 011.110 100 1	12B.D2
1 110 010 011 101.000 110 10	1C9D.1A
110 011 010 100 111.011 101	66A7.74

Table 2.7: Binary-Hexadecimal Conversion Examples

2.2.6 *Octal to Decimal*

The simplest way to convert an octal number to decimal is to write the octal number in expanded positional notation, calculate the values for each of the sets of parenthesis, and then add all of the values. For example, to convert 245_8 to decimal:

$$\begin{aligned} 245_8 &= (2X8^2) + (4X8^1) + (5X8^0) \\ &= (2X64) + (4X8) + (5X1) \\ &= (128) + (32) + (5) \\ &= 165_{10} \end{aligned} \tag{2.11}$$

If the octal number has a fractional component, then that part would be converted using negative powers of eight. As an example, convert 25.71_8 to decimal:

$$\begin{aligned} 25.71_8 &= (2X8^1) + (5X8^0) + (7X8^{-1}) + (1X8^{-2}) \\ &= (2X8) + (5X1) + (7X0.125) + (1X0.015625) \\ &= (16) + (5) + (0.875) + (0.015625) \\ &= 21.890625_{10} \end{aligned} \tag{2.12}$$

Other examples are:

$$\begin{aligned} 42.6_8 &= (4X8^1) + (2X8^0) + (6X8^{-1}) \\ &= (4X8) + (2X1) + (6X0.125) \\ &= (32) + (2) + (0.75) \\ &= 34.75_{10} \end{aligned} \tag{2.13}$$

$$\begin{aligned} 32.54_8 &= (3X8^1) + (2X8^0) + (5X8^{-1}) + (4X8^{-2}) \\ &= (3X8) + (2X1) + (5X0.125) + (4X0.015625) \\ &= (24) + (2) + (0.625) + (0.0625) \\ &= 26.6875_{10} \end{aligned} \tag{2.14}$$

$$\begin{aligned} 436.27_8 &= (4X8^2) + (3X8^1) + (6X8^0) + (2X8^{-1}) + (7X8^{-2}) \\ &= (4X64) + (3X8) + (6X1) + (2X0.125) + (7X0.015625) \\ &= (256) + (24) + (6) + (0.25) + (0.109375) \\ &= 286.359375_{10} \end{aligned} \tag{2.15}$$

2.2.7 *Hexadecimal to Decimal*

The simplest way to convert a hexadecimal number to decimal is to write the hexadecimal number in expanded positional notation, calculate the values for each of the sets of parenthesis, and then add all of the values. For example, to convert $2A6_{16}$ to decimal:

$$\begin{aligned} 2A6_{16} &= (2X16^2) + (AX16^1) + (6X16^0) \\ &= (2X256) + (10X16) + (6X1) \\ &= (512) + (160) + (6) \\ &= 678_{10} \end{aligned} \tag{2.16}$$

If the hexadecimal number has a fractional component, then that part would be converted using negative powers of 16. As an example, convert $1B.36_{16}$ to decimal:

$$\begin{aligned} 1B.36_{16} &= (1X16^1) + (11X16^0) + (3X16^{-1}) + (6X16^{-2}) \\ &= (16) + (11) + (3X\frac{1}{16}) + (6X\frac{1}{256}) \\ &= 16 + 11 + 0.1875 + 0.0234375 \\ &= 27.2109375_{10} \end{aligned} \tag{2.17}$$

Other examples are:

$$\begin{aligned} A32.1C_{16} &= (AX16^2) + (3X16^1) + (2X16^0) + (1X16^{-1}) + (CX16^{-2}) \\ &= (10X256) + (3X16) + (2X1) + (1X\frac{1}{16}) + (12X\frac{1}{256}) \\ &= 2560 + 48 + 2 + 0.0625 + 0.046875 \\ &= 6300.109375_{10} \end{aligned} \tag{2.18}$$

$$\begin{aligned} 439.A_{16} &= (4X16^2) + (3X16^1) + (9X16^0) + (AX16^{-1}) \\ &= (4X256) + (3X16) + (9X1) + (10X\frac{1}{16}) \\ &= 1024 + 48 + 9 + 0.625 \\ &= 1081.625_{10} \end{aligned} \tag{2.19}$$

2.2.8 *Decimal to Binary*

2.2.8.1 *Integers*

Note: Converting decimal fractions is a bit different and is covered on page 27.

Converting decimal integers to binary (indeed, any other radix) involves repeated cycles of division. In the first cycle of division, the original decimal integer is divided by the base of the target numeration system (binary=2, octal=8, hex=16), and then the whole-number portion of the quotient is divided by the base value again. This process continues until the quotient is less than one. Finally, the binary, octal, or hexadecimal digits are determined by the "remainders" left over at each division step.

After a decimal number is converted to binary it can be easily converted to either octal or hexadecimal.

Table 2.8 shows how to convert 87_{10} to binary by repeatedly dividing 87 by 2 (the radix for binary) until reaching zero. The number in column one is divided by two and that quotient is placed on the next row in column one with the remainder in column two. For example, when 87 is divided by 2, the quotient is 43 with a remainder of one. This division process is continued until the quotient is less than one. When the division process is completed, the binary number is found by using the remainders, *reading from the bottom to top*. Thus 87_{10} is 1010111_2.

Integer	Remainder
87	
43	1
21	1
10	1
5	0
2	1
1	0
0	1

Table 2.8: Decimal to Binary

This repeat-division technique will also work for numeration systems other than binary. To convert a decimal integer to octal, for example, divide each line by 8; but follow the process as described above. As an example, Table 2.9 shows how to convert 87_{10} to 127_8.

Integer	Remainder
87	
10	7
1	2
0	1

Table 2.9: Decimal to Octal

The same process can be used to convert a decimal integer to hexadecimal; except, of course, the divisor would be 16. Also, some of the remainders could be greater than 10, so these are written as letters. For example, to convert 678_{10} to $2A6_{16}$ use the process illustrated in Table 2.10.

Integer	Remainder
678	
42	6
2	A
0	2

Table 2.10: Decimal to Hexadecimal

2.2.8.2 *Fractions*

Converting decimal fractions to binary is a repeating operation similar to converting decimal integers, but each step repeats multiplication rather than division. To convert 0.8215_{10} to binary, repeatedly multiply the fractional part of the number by two until the fractional part is zero (or whatever degree of precision is desired). As an example, in Table 2.11, the number in column two, 8215, is multiplied by two and the integer part of the product is placed in column one on the next row while the fractional part in column two. Keep in mind that the "Remainder" is a decimal fraction with an assumed leading decimal point. That process continues until the fractional part reaches zero.

Integer	Remainder
	8125
1	625
1	25
0	5
1	0

Table 2.11: Decimal to Binary Fraction

When the multiplication process is completed, the binary number is found by using the integer parts and *reading from the top to the bottom*. Thus 0.8125_{10} is 0.1101_2.

As another example, Table 2.12 converts 0.78125_{10} to 0.11001_2. The solution was carried out to full precision (that is, the last multiplication yielded a fractional part of zero).

Integer	Remainder
	78125
1	5625
1	125
0	25
0	5
1	0

Table 2.12: Decimal to Binary Fraction Example

Often, a decimal fraction will create a huge binary fraction. In that case, continue the multiplication until the desired number of binary places are achieved. As an example, in Table 2.13, the fraction 0.1437_{10} was converted to binary, but the process stopped after 10 bits.

Integer	Remainder
	1437
0	2874
0	5748
1	1496
0	2992
0	5984
1	1968
0	3936
0	7872
1	5744
1	1488

Table 2.13: Decimal to Long Binary Fraction

Thus, $0.1437_{10} = 0.0010010011_2$ (with 10 bits of precision).

To calculate this to full precision requires 6617 bits; thus, it is normally wise to specify the desired precision.

Converting decimal fractions to any other base would involve the same process, but the base is used as a multiplier. Thus, to convert a decimal fraction to hexadecimal multiply each line by 16 rather than 2.

2.2.8.3 *Mixed Numbers*

To convert a mixed decimal number (one that contains both an integer and fraction part) to binary, treat each component as a separate problem and then combine the result. As an example, Table 2.14 and Table 2.15 show how to convert 375.125_{10} to 101110111.001_2.

Integer	Remainder
375	
187	1
93	1
46	1
23	0
11	1
5	1
2	1
1	0
0	1

Table 2.14: Decimal to Binary Mixed Integer

Integer	Remainder
	125
0	25
0	5
1	0

Table 2.15: Decimal to Binary Mixed Fraction

A similar process could be used to convert decimal numbers into octal or hexadecimal, but those radix numbers would be used instead of two.

2.2.9 *Calculators*

For the most part, converting numbers between the various "computer" bases (binary, octal, or hexadecimal) is done with a calculator. Using a calculator is quick and error-free. However, for the sake of applying digital logic to a mathematical problem, it is essential to understand the theory behind converting bases. It will not be possible to construct a digital circuit where one step is "calculate the next answer on a hand-held calculator." Conversion circuits (like all circuits) need to be designed with simple gate logic, and an understanding of the theory behind the conversion process is important for that type of problem.

Online Conversion Tool

Excel will convert between decimal/binary/octal/hexadecimal integers (including negative integers), but cannot handle fractions; however, the following website has a conversion tool that can convert between common bases, both integer and fraction: http://baseconvert.com/. An added benefit for this site is conversion with twos complement, which is how negative binary numbers are represented and is covered on page 40.

2.2.10 *Practice Problems*

Table 2.16 lists several numbers in decimal, binary, octal, and hexadecimal form. To practice converting between numbers, select a number on any row and then covert it to the other bases.

Decimal	Binary	Octal	Hexadecimal
13.0	1101.0	15.0	D.0
1872.0	11 101 010 000.0	3520.0	750.0
0.0625	0.0001	0.04	0.1
0.457 031 25	0.011 101 01	0.352	0.75
43.125	101 011.001	53.1	2B.2
108.718 75	1 101 100.101 11	154.56	6C.B8

Table 2.16: Practice Problems

2.3 FLOATING POINT NUMBERS

Numbers can take two forms: Fixed Point and Floating Point. A fixed point number generally has no fractional component and is used for integer operations (though it is possible to design a system with a fixed fractional width). On the other hand, a floating point number has a fractional component with a variable number of places.

Before considering how floating point numbers are stored in memory and manipulated, it is important to recall that any number can be represented using *scientific notation*. Thus, 123.45_{10} can be represented as $1.2345X10^2$ and 0.0012345_{10} can be represented as $1.2345X10^{-3}$. Numbers in scientific notation with one place to the left of the radix point, as illustrated in the previous sentence, are considered *normalized*. While most people are familiar with normalized decimal numbers, the same process can be used for any other base, including binary. Thus, 1101.101_2 can be written as $1.101101X2^3$. Notice that for normalized binary numbers the radix is two rather than ten since binary is a radix two system, also there is only one bit to the left of the radix point.

IEEE Standard 754 defines Floating Point Numbers

By definition, floating point numbers are stored and manipulated in a computer using a 32-bit word (64 bits for "double precision" numbers). For this discussion, imagine that the number 10010011.0010_2 is to be stored as a floating point number. That number would first be normalized to $1.00100110010X2^7$ and then placed into the floating point format:

```
x    xxxxxxxx xxxxxxxxxxxxxxxxxxxxxxx
sign exponent mantissa
```

- SIGN: The sign field is a single bit that is zero for a positive number and one for a negative number. Since the example number is positive the sign bit is zero.
- EXPONENT: This is an eight-bit field containing the radix's exponent, or 7 in the example. However, the field must be able to contain both positive and negative exponents, so it is offset by

127. The exponent of the example, 7, is stored as $7 + 127$, or 134; therefore, the exponent field contains 10000110.

- MANTISSA (sometimes called *significand*): This 23-bit field contains the number that is being stored, or 100100110010 in the example. While it is tempting to just place that entire number in the mantissa field, it is possible to squeeze one more bit of precision from this number with a simple adjustment. A normalized binary number will always have a one to the left of the radix point since in scientific notation a significant bit must appear in that position and one is the only possible significant bit in a binary number. Since the first bit of the stored number is assumed to be one it is dropped. Thus, the mantissa for the example number is 00100110010000000000000.

Here is the example floating point number (the spaces have been added for clarity):

$$10010011.0010 = 0\ 10000110\ 00100110010000000000000$$

A few floating point special cases have been defined:

- ZERO: The exponent and mantissa are both zero and it does not matter whether the sign bit is one or zero.
- INFINITY: The exponent is all ones and the mantissa is all zeros. The sign bit is used to represent either positive or negative infinity.
- NOT A NUMBER (NAN): The exponent is all ones and the mantissa has at least one one (it does not matter how many or where). NaN is returned as the result of an illegal operation, like an attempted division by zero.

Two specific problems may show up with floating point calculations:

- OVERFLOW. If the result of a floating point operation creates a positive number that is greater than $(2 - 2^{-23})X2^{127}$ it is a positive overflow or a negative number less than $-(2 - 2^{-23})X2^{127}$ it is a negative overflow. These types of numbers cannot be contained in a 32-bit floating point number; however, the designer could opt to increase the circuit to 64-bit numbers (called "double precision" floating point) in order to work with these large numbers.
- UNDERFLOW. If the result of a floating point operation creates a positive number that is less than 2^{-149} it is a positive underflow or a negative number greater than -2^{-149} it is a negative underflow. These numbers are vanishingly small and are sometimes

simply rounded to zero. However, in certain applications, such as multiplication problems, even a tiny fraction is important and there are ways to use "denormalized numbers" that will sacrifice precision in order to permit smaller numbers. Of course, the circuit designer can always opt to use 64-bit ("double precision") floating point numbers which would permit negative exponents about twice as large as 32-bit numbers.

Table 2.17 contains a few example floating point numbers.

Decimal	Binary	Normalized	Floating Point
34.5	100010.1	$1.000101X2^5$	0 10000100 00010100000000000000000
324.75	101000100.11	$1.0100010011X2^8$	0 10000111 01000100010000111101100
-147.25	10010011.01	$1.001001101X2^7$	1 10000110 00100110100000000000000
0.0625	0.0001	$1.0X2^{-4}$	0 01111011 00000000000000000000000

Table 2.17: Floating Point Examples