Chapter 2
Number Systems and Data Formats

To understand how microcontrollers and microprocessors process data, we must adopt symbols and rules with proper models to mimic physical operations at a human friendly level. Digital systems operate with only two voltage levels to process information. As part of the modeling process, two symbols are associated to these levels, '0' and '1'. Any consecutive concatenation of these two symbols, in any order, is referred to as a word. Words are used as mathematical entities and abstract symbols to represent all sort of data. We refer to words as integers, thereby simplifying not only the description and manipulation of the strings, but also programming notation and hardware processes.

The meaning of a word is context dependent and may be completely arbitrary for user-defined data or assignments, set by the hardware configurations or else follow conventions and standards adopted in the scientific community. We limit our discussion here to numerical, text, and some particular cases of special interest.

This chapter contains more material than needed in a first introduction to the topic of embedded systems. A first time reader may skip Sects. 2.9 and 2.10. Arithmetic operations with integers, on the other hand, are needed to go beyond transfer operations.

2.1 Bits, Bytes, and Words

A *bit* is a variable which can only assume two values, 0 or 1. By extension, the name bit is also used for the constants 0 or 1. An ordered sequence of n bits is an *n-bit word*, or simply *word*. Normally, n is understood from the context; when it is not, it should be indicated. Particular cases of n-bit words with proper names are:

- *Nibble*: 4-bit word
- *Byte*: 8-bit word
- *Word*: 16-bit word
- *Double word*: 32-bit word.
- *Quad*: 64-bit word.

These lengths are associated to usual hardware. In the MSP430, for example, registers are 16 bits wide. Notice the specific use of the term “word” for 16 bits.

Individual bits in a word are named after their position, starting from the right with 0: bit 0 (b0), bit 1 (b1), and so on. Symbolically, an n-bit word is denoted as

$$b_{n-1}b_{n-2}\dots b_1b_0 \tag{2.1}$$

The rightmost bit, b_0, is the *least significant bit* (lsb), while the leftmost one, b_{n-1}, is the *most significant bit* (msb). Similarly, we talk of least or most significant nibble, byte, words and so on when making reference to larger blocks of bits.

Since each bit in (2.1) can assume either of two values, there are 2^n different n-bit words. For easy reference, Table 2.1 shows the powers of 2 from 2^1 to 2^{30}.

Certain powers of 2 have special names. Specifically:

- *Kilo* (K): $1\,\text{K} = 2^{10}$
- *Mega* (M): $1\,\text{M} = 2^{20}$
- *Giga* (G): $1\,\text{G} = 2^{30}$
- *Tera* (T): $1\,\text{T} = 2^{40}$

Hence, when “speaking digital”, 4 K means $4 \times 2^{10} = 2^{12} = 4,096$, 16 M means $16 \times 2^{20} = 2^{24} = 16,077,216$ and so on.[1]

Table 2.1 Powers of 2

N	2^N	N	2^N	N	2^N
1	2	11	2,048	21	2,097,152
2	4	12	4,096	22	4,194,304
3	8	13	8,192	23	8,388,608
4	16	14	16,384	24	16,777,216
5	32	15	32,768	25	33,554,432
6	64	16	65,536	26	67,108,864
7	128	17	131,072	27	134,217,728
8	256	18	262,144	28	268,435,456
9	512	19	524,288	29	536,870,912
10	1,024	20	1,048,576	30	1,073,741,824

2.2 Number Systems

Numbers can be represented in different ways using 0’s and 1’s. In this chapter, we mention the most common conventions, starting with the normal binary representation, which is a positional numerical system.

[1] The notation K, M, G and T was adopted because those powers are the closest to 10^3, 10^6, 10^9 and 10^{12}, respectively. However, manufacturers of hard drives use those symbols in the usual ten’s power meaning. Thus, a hard drive of 120 MB means that it stores 120×10^6 bytes.

2.2.1 Positional Systems

Our decimal system is *positional*, which means that any number is expressed by a permutation of digits and can be expanded as a weighted sum of powers of ten, the base of the system. Each digit contributes to the sum according to its position in the string. Thus, for example,

$$\begin{aligned} 32.23 &= 3 \times 10^1 + 2 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2}, \\ 578 &= 8 \times 10^0 + 7 \times 10^1 + 5 \times 10^2 \end{aligned}$$

This concept can be generalized to any base. A *fixed-radix*, or *fixed-point* positional system of *base* r has r ordered digits $0, 1, 2, \ldots$ "r−1". Number notations are composed of permutations of these r digits. When we run out of single-digit numbers we add another digit and form double-digit numbers, then when we run out of double-digit numbers we form three-digit numbers, and so on. There is no limit to the number of digits that can be used. Thus, we write

Base 2 : 0, 1, 10, 11, 100, 101, 110, . . . ;

Base 8 : 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, . . . , 17, 20, . . . , 77, 100, 101, . . . ;

Base 12 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, 10, 11, . . . , BB, 100, . . . , BBB, 1000, . . . ;

To avoid unnecessary complications, the same symbols are used for the digits $0, \ldots, 9$ in any base whenever are required, and letters $A, B, \ldots$ are digits with values of ten, eleven, etc. To distinguish between different bases, the radix is placed as a subscript to the string, as in 28_9, $1A_{16}$.

Numbers are written as a sequence of digits and a point, called *radix point*, which separates the *integer* from the *fractional part* of the number to the left and right side of the point, respectively. Consider the expression in equation (2.2):

$$a_{n-1}a_{n-2}\ldots a_1a_0.a_{-1}a_{-2}\ldots a_m \tag{2.2}$$

Here, each subscript stands for the exponent of the weight associated to the digit in the sum. The leftmost digit is referred to as the *most significant digit* (msd) and the rightmost one is the *least significant digit* (lsd). If there were no fractional part in (2.2), the radix point would be omitted and the number would be called simply an integer. If it has no integer part, it is customary to include a "0" as the integer part.

The number denoted by (2.2) represents a power series in r of the form

$$\begin{aligned} &\underbrace{a_{n-1}r^{n-1} + a_{n-2}r^{n-2} + \cdots + a_1r^1 + a_0r^0}_{\text{integer part}} \\ &\qquad + \underbrace{a_{-1}r^{-1} + \cdots + a_{-m}r^{-m}}_{\text{fractional part}} = \sum_{i=-m}^{n-1} a_i r^i \end{aligned} \tag{2.3}$$

An interesting and important fact follows from this expansion: *In a system of base r, r^n is written as 1 followed by n zeros.* Thus, $100_2 = 2^2$, $1000_5 = 5^3$, etc.

The systems of interest for us are the *binary* (base 2), *octal* (base 8), *decimal* (base 10), and *hexadecimal* (base 16) —*hex* for short,—systems. Table 2.2 shows equivalent representations to decimals 0–15 in these systems, using four binary digits, or bits, for binary numbers.

To simplify keyboard writing, especially when using text processors, the following conventions are adopted:

- For binary numbers use suffix 'B' or 'b'
- For octal numbers use suffix 'Q' or 'q'
- For hex numbers use suffix 'H' or 'h', or else prefix 0x. Numbers may not begin with a letter
- Base ten numbers have no suffix.

Hence, we write 1011B or 1011b instead of 1011_2, 25Q or 25q for 25_8, 0A5H or 0A5h or 0xA5 for $A5_{16}$.

Table 2.2 Basic numeric systems

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
0	0000	0	0	8	1000	10	8
1	0001	1	1	9	1001	11	9
2	0010	2	2	10	1010	12	A
3	0011	3	3	11	1011	13	B
4	0100	4	4	12	1100	14	C
5	0101	5	5	13	1101	15	D
6	0110	6	6	14	1110	16	E
7	0111	7	7	15	1111	17	F

2.3 Conversion Between Different Bases

The power expansion (2.3) of a number may be used to convert from one base to another, performing the right hand side operations in the target system. This requires, of course, that we know how to add and multiply in systems different from the common decimal one. This is why the decimal system is usually an intermediate step when a conversion is done by hand between non decimal systems. Yet, any algorithm is valid to work directly when knowledge or means are available.

2.3.1 Conversion from Base r to Decimal

The expansion (2.3) is the preferred method to convert from base r to decimal. Let us illustrate with some numbers. No subscript or suffix is used for the decimal result.

Example 2.1 *The following cases illustrate conversions to decimals:*

$$214.23_5 = 2 \times 5^2 + 1 \times 5^1 + 4 \times 5^0 + 2 \times 5^{-1} + 3 \times 5^{-2} = 59.52$$

$$\begin{aligned}10110.01B &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &\quad + 0 \times 2^{-1} + 1 \times 2^{-2} = 22.25\end{aligned}$$

$$B65FH = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = 46687$$

Notice that for hexadecimal conversion, all hex digits are interpreted in their decimal values for the sum.

Since binary numbers are so common in embedded systems with digits being only 0 and 1, it is handy to have a quick reference for weights using positive and negative powers of 2, as illustrated in Fig. 2.1 for some cases.

Example 2.2 *Let us convert* 1001011.1101B *to decimal using the powers as shown in Fig. 2.1, as shown in the following table, taking for the sum only those powers for bit 1:*

Number	1	0	0	1	0	1	1.	1	1	0	1
Exponents	6	5	4	3	2	1	0	−1	−2	−3	−4
Power	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625

Therefore, $1001011.1101B = 64 + 8 + 2 + 1 + 0.5 + 0.25 + 0.0625 = 75.8125$

2.3.2 Conversion from Decimal to Base r

Conversion into base r is easier if the integer and fractional parts are treated separately.

Integer Conversion

One popular procedure for converting decimal integers into base r is the *repeated division* method. This method is based on the division algorithm, and consists in successively dividing the number and quotients by the target radix r until the quotient

Weights:	128	64	32	16	8	4	2	1
Bits:	b7	b6	b5	b4	b3	b2	b1	b0

Weights:	0.5	0.25	0.125	0.0625	0.03125	0.015625
Bits:	b(-1)	b(-2)	b(-3)	b(-4)	b(-5)	b(-6)

Fig. 2.1 Weights for binary number $b_7b_6b_5b_4b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}b_{-5}b_{-6}$

is 0. The successive remainders of the divisions are the digits of the number in base r, starting from the least significant digit: divide the number by r and take the remainder as a_0; divide the quotient by r, and the remainder as a_1, and so on.[2] Let us illustrate with a pair of examples.

Example 2.3 *Convert decimal* 1993 *to expressions in bases 5 and 16.*

To convert to base 5, divide by 5. The remainders are the digits of the number we are looking for, going from lsd to msd:

Division	Quotient	Remainder
1993/5:	398	$a_0 = 3$
398/5:	79	$a_1 = 3$
79/5:	15	$a_2 = 4$
15/5:	3	$a_3 = 0$
3/5:	0	$a_4 = 3$

Hence, $1{,}993 = 30{,}433_5$

To convert to base 16, repeat the procedure using 16 as divisor. If the remainder is greater than or equal to 10, convert to hex equivalent digit.

Division	Quotient	Remainder
1993/16:	124	$9\ a_0 = 9$
124/16:	7	$12\ (a_1 = C)$
7/16:	0	$a_2 = 7$

The result is then $1993 = 7C9_{16} = 7C9h$.

Fractional Part

Conversion of decimal fractions can be done by the *repeated multiplication* method: Multiplying the number by r, the integer part of the first product becomes most significant digit a_{-1}. Discard then the integer part and multiply again the new

[2] Again, the algorithm is valid between any bases. Let M_k be the integer in a base k system, which we want to express in r base system. The power sum expansion means $M_k = (a_{h-1}a_{h-2}\ldots a_1a_0)_r = a_{h-1} \times r^{h-1} + \cdots a_1 \times r + a_0$, where the a_i's are the digits in the r base system. Dividing by r, the quotient is $a_h \times r^{h-1} + \cdots a_1$ and the remainder a_0. Repeating division, the quotient is $a_h \times r^{h-2} + \cdots a_2$ with a remainder a_1. Continuing this way, $a_2, a_3, \ldots a_h$ will be extracted, in that order.

fractional part by r to get the next digit a_{-2}. The process continues until one of the following conditions is met[3]:

- A zero decimal fraction is obtained, yielding a finite representation in radix r; or
- A previous fractional part is again obtained, having then found the periodic representation in radix r; or
- the expression in base r has the number of digits allocated for the process.

Since we are converting finite decimal fractions, the result must be either finite or periodic. The mathematical theory behind this assertion does not depend on the radix. Yet, periodicity might appear only after too many multiplications, so the third option above is just a matter of convenience. Let us look at some examples.

Example 2.4 *Convert the following decimal fractions to binary, limited to 8 digits if no periodicity appears before: (a)* 0.375, *(b)* 0.05, *(c)* 0.23.

(a) Let us begin with 0.375:

$$\begin{aligned} 2\times 0.375 &= 0.750 && \rightarrow\ a_{-1}=0 \\ 2\times 0.750 &= 1.50 && \rightarrow\ a_{-2}=1 \\ 2\times 0.5 &= 1.00 && \rightarrow\ a_{-3}=1 \quad \text{Zero fractional part. Stop} \end{aligned}$$

Therefore, since the decimal fractional part in the last product is 0.00*, the equivalent expression in binary is finite, specifically,* $0.375 = 0.011_2 = 0.011\text{B}$.

(b) Converting 0.05:

$$\begin{aligned} 2\times 0.05 &= 0.1 && \rightarrow\ a_{-1}=0 \\ 2\times 0.10 &= 0.2 && \rightarrow\ a_{-2}=0 \\ 2\times 0.2 &= 0.4 && \rightarrow\ a_{-3}=0 \\ 2\times 0.4 &= 0.8 && \rightarrow\ a_{-4}=0 \\ 2\times 0.8 &= 1.6 && \rightarrow\ a_{-5}=1 \\ 2\times 0.6 &= 1.2 && \rightarrow\ a_{-6}=1 \quad \text{Repeatingfractionalpart0.2.Stop} \end{aligned}$$

The decimal fraction 0.2 *in the last line has appeared before (third line), so the pattern* 0011 *will be periodic. Therefore,* $0.05 = 0.00\overline{0011}_2 = 0.0000110011\ldots\text{B}$.

(c) Converting 0.23:

$$\begin{aligned} 2\times 0.23 &= 0.46 \rightarrow a_{-1}=0 \\ 2\times 0.46 &= 0.92 \rightarrow a_{-2}=0 \\ 2\times 0.92 &= 1.84 \rightarrow a_{-3}=1 \end{aligned}$$

[3] Let $M_{10} = (a_{-1}a_{-2}\cdots a_{-m})_r = a_-1\times r^{-1} + a_2\times r^{-2}\cdots + a_{-m}\times r^{-m}$. Multiplying by r, the product is $a_{-1} + a_2\times r^{-1}\cdots + a_{-m}\times r^{-m+1)}$. The integer part here is a_{-1} and the fractional part follows the same pattern but starting with a_{-2}. Continuing with multiplication to the fractional part only, $a_{-2}, a_{-3},\ldots$ will be extracted, in that order.

$$2 \times 0.84 = 1.68 \rightarrow a_{-4} = 1$$
$$2 \times 0.68 = 1.36 \rightarrow a_{-5} = 1$$
$$2 \times 0.36 = 0.72 \rightarrow a_{-6} = 0$$
$$2 \times 0.72 = 1.44 \rightarrow a_{-7} = 1$$
$$2 \times 0.44 = 0.88 \rightarrow a_{-8} = 0 \quad \text{Stop because of predefined limit.}$$

We have reached 8 digits without finding the decimal fraction that will repeat.[4] *Within this approximation,* $0.23 \approx 0.00111010\text{B}$

Mixed Numbers with Integer and Fractional Parts
In this case, the conversion is realized separately for each part. Let us consider an example.

Example 2.5 *Convert* 376.9375_{10} *to base 8.*
First convert the integer part by successive divisions by 8:

Division	Quotient	Remainder
376/8:	47	$a_0 = 0$
47/8:	5	$a_1 = 7$
5/8:	0	$a_2 = 5$

which yields 376 = 570Q.
We now convert the fractional part by successive multiplications:

Multiplication	product	Int. Part.
$0.9375 \times 8 =$	7.5	$a_{-1} = 7$
$0.5 \times 8 =$	4.0	$a_{-2} = 4$

which yields 3.9375 = 0.74Q. *Therefore the final result is* 376.9375 = 570.74Q.

2.3.3 Binary, Octal and Hexadecimal Systems

The lower the base, the more digits required to represent a number. The binary numerical system, the 'natural' one for digital systems, is quite inconvenient for people. A simple three digit decimal number such as 389, becomes 110000101B in binary, a nine digit number. The octal and hexadecimal systems provide simpler alternatives for representing binary words. Since $8 = 2^3$ and $16 = 2^4$, conversion between binary and octal or binary and hex systems is straightforward:

Octal and Binary Associate each octal digit to three binary digits, from right to left in the integer part and left to right in the fractional part.

[4] The reader can continue and verify that the periodic pattern will eventually appear.

Hex and Binary Associate each hex digit to four binary digits, from right to left in the integer part and from left to right in the fractional part.

Table 2.2 shows the equivalences between octal and hex digits with binary numbers, where zeros to the left can be conveniently added. Using this table, let us look at some examples.

Example 2.6 *Convert (a)* 0x4AD.16 *to binary and octal expressions; (b)* 37.25Q *to binary and hex expressions.*

(a) Using Table 2.2 as reference, 4AD.16H *is first converted into binary as follows:*

$$\overbrace{0100}^{4}\overbrace{1010}^{A}\overbrace{1101}^{D}.\overbrace{0001}^{1}\overbrace{0110}^{6} \Rightarrow 10010101101.0001011B$$

Notice that the left zeros in the integer part and the right zeros of the fractional part have been suppressed in the final result. Now, to convert to octal, take the binary part and partition in groups of three bits as shown next, adding extra zeros as needed:

$$\overbrace{010}^{2}\ \overbrace{010}^{2}\ \overbrace{101}^{5}\ \overbrace{101}^{5}\ .\ \overbrace{000}^{0}\ \overbrace{101}^{5}\ \overbrace{100}^{4} \ \Rightarrow\ 2255.054Q$$

(b) For the second part, proceed as before, splitting the octal expression into binary, and then converting into hex expression.

$$\overbrace{011}^{3}\ \overbrace{111}^{7}\ .\ \overbrace{010}^{2}\ \overbrace{101}^{5}\ \Rightarrow\ 11111.010101B$$

Now to hex:

$$\overbrace{0001}^{1}\overbrace{1111}^{F}.\overbrace{0101}^{5}\overbrace{0100}^{4} \Rightarrow\ 1F.54H$$

2.4 Hex Notation for *n*-bit Words

The previous example showed the convenience of hexadecimal system for humans: *there is a one to one correspondence between each hex digit and a group of four bits, a nibble*. For this reason, to simplify writing and reading, it is customary to represent an *n*-bit word by a hex equivalent just as it were a binary integer, *irrespectively of the actual meaning of the string*. Thus, we refer to 1010011 as 53h or 0x53, to 1101100110000010 as 0xD982, and so on.

This convention is universally adopted in embedded systems literature, and also in debuggers. Thus, memory addresses, register contents, and almost everything is expressed in terms of hexadecimal integers, without any implication of them being a number.

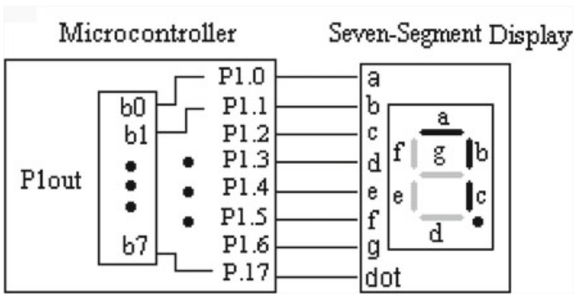

Fig. 2.2 Simplified functional connection diagram of a 7-segment display to a microcontroller port

Example 2.7 *The objective of this example is to illustrate how integers are used to represent hardware and data situations. Fig. 2.2 represents a functional connection, no hardware schematic shown, of a seven-segment display to Port 1 of a microcontroller. The port consists of eight pins, named* P1.0 *to* P1.7*, which are internally connected to a byte size register called* P1Out*. Let us call* b0, b1, ..., b7 *the outputs of the register, connected to* P1.0 *to* P1.7 *in the same order. The high and low voltage states of the register outputs are then applied to the seven-segment through the port pins.*

The seven-segment display consists of eight light-emitting diodes (LED), in the form of seven segments called a, b, c, d, e, f, *and* g *plus a* dot*. Each diode is connected to one port pin as shown in the figure. When the pin shows a high voltage state, the respective LED is turned on.*

In the diagram, the seven-segment display shows 7.*, i. e., it has LED's* a, b,c *and the* dot *on, while the others are off. This means that pins* P1.7, P1.6, P1.5, P1.4, P1.3, P1.2, P1.1, P1.0*,* **in that order**, *show the respective states* HIGH, LOW, LOW, LOW, LOW, HIGH, HIGH, HIGH. *Using the symbol* 1 *for the* HIGH *state and* 0 *for the* LOW *state, and suppressing commas, the register output state can be represented as* 10000111.

We use integers to represent this state, and say then that the contents of the register P1Out *is* 10000111B*, or* 87h*, or* 207q*, or* 135*. The binary number gives a one to one description of each bit inside the register. The hex and octal representations give the same information in a simplified form, using the property of the one to one relationship with the binary digits. The decimal representation is just the conversion of any of the previous expressions.*

If we want to display 6*, the register contents should be* 7Dh*. Verify it.*

The decimal and octal systems can also be used to this end (see Problem 2.27). The octal system is not so popular nowadays because of hardware, which has made nibbles, bytes, 16-bit words and double words the normal environment.

2.5 Unsigned Binary Arithmetic Operations

The familiar techniques applied to arithmetic operations of non negative decimal numbers are also valid for binary numbers. We review next the basic operations for this system.

2.5.1 Addition

The addition rules are

$$0 + 0 = 0 \tag{2.4a}$$

$$0 + 1 = 1 + 0 = 1 \tag{2.4b}$$

$$1 + 1 = 10 \tag{2.4c}$$

As in the decimal system, we start adding the two least significant digits of the operands and carry any excess digit into the sum of the next more significant pair of digits. Therefore, a sum of numbers with more than one digit should consider effectively the addition of three digits to include the carry of the previous column. This addition of three bits yields the so called complete adder case illustrated by Table 2.3.

Table 2.3 Rules for complete binary addition

a_i	+	b_i	+	$carry_i$	=	$carry_{i+1}$	sum_i
0		0		0		0	0
0		0		1		0	1
0		1		0		0	1
0		1		1		1	0
1		0		0		0	1
1		0		1		1	0
1		1		0		1	0
1		1		1		1	1

Example 2.8 *Add the binary equivalents of (a)* 27 *and* 18*; (b)* 152.75 *and* 236.375.

(a) The binary addition of 27 = 11011B *and* 18 = 10010B *is illustrated next.*

$$\begin{array}{rcc|cccccc|c}
\text{carries} & & \rightarrow & 1 & 0 & 0 & 1 & 0 & & \\
27 & + & \rightarrow & & 1 & 1 & 0 & 1 & 1 & + \\
18 & = & \rightarrow & & 1 & 0 & 0 & 1 & 0 & = \\
\hline
45 & & \rightarrow & 1 & 0 & 1 & 1 & 0 & 1 &
\end{array}$$

(b) *Since* 152.75 = 10011000.11B *and* 236.375 = 11101100.011B*, the addition is done as illustrated below. Here, the weight of each digit is shown for easy conversion.*

Weights:		256	128	64	32	16	8	4	2	1	0.5	0.25	0.125
Carries		1	1	1	1	1	0	0	0	1.	1	0	
152.75	+		1	0	0	1	1	0	0	0.	1	1	
236.375	=		1	1	1	0	1	1	0	0.	0	1	1
389.125		1	1	0	0	0	0	1	0	1.	0	0	1

2.5.2 Subtraction

Subtraction follows the usual rules too. When the minuend digit is less than the subtrahend digit, a borrow should be taken. For subtraction, the rules are

$$0 - 0 = 1 - 1 = 0 \tag{2.5a}$$

$$1 - 0 = 1 \tag{2.5b}$$

$$0 - 1 = 1 \text{ with a } \textbf{borrow} 1 \tag{2.5c}$$

When a borrow is needed, it is taken from the next more significant digit of the minuend, from which the borrow should be subtracted. Hence, actual subtraction can be considered to be carried out using three digits: the minuend, the subtrahend, and the borrowed digit. The result yields again a difference digit and a borrow that needs to be taken from the next column. This comment is better illustrated by the following Table 2.4 and example 2.9 below.

The table can be considered a formal expression of the usual procedure. Let us illustrate with an example.

Table 2.4 Complete subtraction

a_i	$-$	b_i	$-$	$borrow_i$	$=$	$borrow_{i+1}$	$difference_i$
0		0		0		0	0
0		0		1		1	1
0		1		0		1	1
0		1		1		1	0
1		0		0		0	1
1		0		1		0	0
1		1		0		0	0
1		1		1		1	1

Example 2.9 *Subtract* 137 *from* 216 *using binary subtraction.*

Since 137 = 10001001B *and* 216 = 11011000B*, we have the following operation, where the least significant borrow is 0 by default:*

$$\begin{array}{ccccccc}
\text{minuend} & 216 & - & \rightarrow & 1\,1\,0\,1\,1\,0\,0\,0 & - \\
\text{subtrahend} & 137 & = & \rightarrow & 1\,0\,0\,0\,1\,0\,0\,1 & \\
\text{Borrows} & \underline{110} & & & \underline{0\,0\,0\,1\,1\,1\,1\,0} & = \\
 & 79 & & \rightarrow & 0\,1\,0\,0\,1\,1\,1\,1 &
\end{array}$$

The binary system has been introduced up to now only for non negative numbers. Hence, to this point, the operation $137 - 216 = -79$ has no meaning. However, it is illustrative to look at the process when Table 2.4 is used.

Example 2.10 *Subtract* 216 *from* 137 *using binary subtraction according to Table 2.4 of the complete subtraction rule.*

We arrange again the operands and borrows, with the least significant borrow 0 by default:

$$\begin{array}{ccccccc}
\text{minuend} & 137 & - & \rightarrow & 1\,0\,0\,0\,1\,0\,0\,1 & - \\
\text{subtrahend} & 216 & = & \rightarrow & 1\,1\,0\,1\,1\,0\,0\,0 & \\
\text{Borrows} & & & & \underline{1\,1\,1\,0\,0\,0\,0\,0} & = \\
 & \overline{-79} & & \rightarrow ? & 1\,1\,0\,1\,1\,0\,0\,0\,1 &
\end{array}$$

Two interesting points to remark: First, the result contains one bit more than the operands; this "one" comes from the borrow, since the subtrahend is greater than the minuend. This feature is used to identify inequalities. Second, a binary interpretation of the result using the numeric transformation previously studied is not valid, since it would yield 481. Even discarding the most significant digit, we are still without a good result, 225.

Of course, we can proceed just as we have done it in our usual arithmetic practice: actually subtract $216 - 137$ and then append the negative sign in front of the result. However, we have no way of using a negative sign in embedded systems. Hence, we need a method to identify here -79. This method is discussed in Sect. 2.8, where the signed number representation is introduced. For the moment, let us introduce an associated concept, subtraction by complement's addition.

2.5.3 Subtraction by Complement's Addition

Let us consider two decimal numbers A and B with the same number of digits, say N. If necessary, left zeros can be appended to one of them. From elementary Algebra, it follows that

$$A - B = [A + (10^N - B)] - 10^N \tag{2.6}$$

The term $10^N - B$ in the addition is known as *ten's complement of* B. If $A > B$, then the addition $A + (10^N - B)$ results in $N + 1$ digits, i.e., with a carry of 1, which is eliminated by 10^N. If $A < B$, the addition will contain exactly N digits with no extra carry, and the result will be the negative of the tens complement of the addition. Let us illustrate with 127 and 31 (three digits required):

$$127 - 31 = (127 + 969) - 1000 = \underline{1}096 - 1000 = 96$$

and

$$31 - 127 = (31 + 873) - 1000 = 904 - 1000 = -(1000 - 904) = -96$$

Two's Complement Concept and Application

The method described above is applicable to any base system. This is so, because operations in (2.6) can be referred to any reference to base, and for any base r, $(r^N)_{10} = (10^N)_r$. When the base is 2, the complement is specifically called *two's complement*.

Notice that when considering two's complements, we need to specify the number of bits. For example, with four bits, the two's complement of 1010B is 10000B $-$ 1010B $=$ 0110B, but with 6 bits this becomes 1000000B $-$ 1010B $=$ 110110B. If you feel comfortable using decimal equivalents for powers of 2, these two examples can be thought as follows: 1010B is equivalent to decimal 10. For four bits, $2^4 = 16$ so the two's complement becomes $16 - 10 = 6 =$ 0110B; with 6 bits, $2^6 = 64$ and the two's complement is $64 - 10 = 54 =$ 110110B.

Example 2.11 *The operations (a)* 216 – 137 *and (b)* 137 – 216 *using binary numbers and the two's complement addition, and interpret accordingly, expressing your result in decimal numbers.*

The binary equivalents for the data are 216 = 11011000B *and* 137 = 10001001B, *respectively. For the two's complement, since both numbers have 8 bits, consider* $2^8 = 256$ *as the reference, so the two's complement of* 216 *is* $256 - 216 = 40 =$ 00101000B, *and that of* 137 *is* $256 - 137 = 119 =$ 01110111B. *With this information, we have then:*

(A) *For* 216 – 137*:*

$$\begin{array}{l} 11011000\,+ \\ \underline{01110111\,=} \\ 101001111 \end{array}$$

Since the answer has a carry (9 bits), the result is positive. Dropping this carry, we have the solution to this subtraction 01001111B = 79.

(B) *For* 137 – 216*:*

$$\begin{array}{l} 10001001\,+ \\ \underline{00101000\,=} \\ 10110001 \end{array}$$

Since the sum does not yield a carry (8 bits), the result is negative, with an absolute value equal to its two's complement. 10110001B = 177*, and the two's complement is* $256 - 177 = 79$. *Therefore, the solution is* -79*, as expected.*

Notice that with only eight bits, 10110001 *is the same result obtained in example 2.10 when considering only eight least significant bits.*

Important Remark When the subtraction of $A - B$ is realized by complement addition, the presence of a carry in the addition means that the difference is non-negative, $A \geq B$. On the other hand, if the carry is 0, there is a need for a borrow, $A < B$, and the result is the negative of the complement of the sum.

Calculating two's Complements
The operand's binary two's complement was calculated in the previous example through the interpretation of the algorithm itself. Two common methods to find the two's complement of a number expressed in binary form are mentioned next. The first one is practical for hardware realization and the second one is easy for fast hand conversion.

- *Invert-plus-sum*: Invert all bits (0–1 and viceversa) and add arithmetically 1.
  - To illustrate the proof, let us consider 4 bits, $b_3b_2b_1b_0$. The complement, in binary terms is

$$10000 - b_3b_2b_1b_0 = (1111 - b_3b_2b_1b_0) + 1$$

  According to the rules of subtraction (2.5a), the subtraction in parenthesis of the right hand side consists in inverting each bit.

- *Right-to-left-scan*: Invert only all the bits to the left of the rightmost '1'.
  - To illustrate with an example, assume that the six least significant bits of the original number are a 1 followed by five zeros, *****100000. After inverting the bits, all the bits to the right of the original rightmost 1 (now 0) will be 1's, and all those to the left will be inverted of the original bits, that is, after inversion we have xxxxxxx011111.... When we add 1, we will get xxxxxxx100000...., where the 'x' are the inverted bits of original number.

Both methods can be verified from the above examples, where N = 11011000B is the original number. For the two's complement take 00100111B + 1 = 00101000B.

2.5.4 Multiplication and Division

Multiplication and division also follow the normal procedures used in decimal system. We illustrate with two examples.

Example 2.12 *Multiply 19 by 26, and divide 57 by 13.*
For the multiplication, since 26 = 11010b *and* 19 = 10011b*,*

```
    11010  ×
    10011  =
    -----
    11010
   11010
11010
---------
111101110
```

The result is 111101110b = 1EEh = 494*, as it should be.*
Notice that the result here has 9 bits. In general, the number of bits in the product is at most equals to the sum of the number of bits in the operands.
For the division, where 57 = 111001B *and* 13 = 1101B*, the operation is shown next:*

```
                100
            -------
 1101       | 111001
             −1101
            ------
                101
```

This division yields a quotient 4 (100b) *and residue* 5 (101b)*, as it should be. The process followed the same rules as in decimal division.*

2.5.5 Operations in Hexadecimal System

It was pointed out the convenience of using the hexadecimal system as a method to simplify reading and writing binary words. This convenience extends to binary operations when dealing with numbers. While the rules can be set up for hexadecimal system, it is generally easier to “think” in decimal terms and write down in hexadecimal notation. Any result greater than 15 (F) yields a carry. Let us illustrate these remarks with examples 2.8 and 2.9 in hexadecimal notation.

Example 2.13 *Using the hexadecimal system, add* (a) 27 *and* 18*, and* (b) 152.75 *and* 236.375*; and* (c) *subtract* 137 *from* 216.
(a) *The binary addition of* 27 = 1Bh *and* 18 = 12h *is illustrated next.*

$$\begin{array}{rcccr} 27 & + & \rightarrow & \text{1Bh} & + \\ \underline{18} & = & \rightarrow & \underline{\text{12h}} & = \\ 45 & & \rightarrow & \text{2Dh} & \end{array}$$

For this addition, we can say 'B – *eleven* – *plus* 2 *is thirteen* – D –' *and* '1 + 1 = 2'.
(b) *Since* 152.75 = 98CH *and* 236.375 = EC.6H, *the addition is illustrated next.*

$$\begin{array}{lccr r} \text{Carries} & & & \text{1 1 1.0} & \\ 152.75 & + & \rightarrow & \text{98.Ch} & + \\ \underline{236.375} & = & \rightarrow & \underline{\text{EC.6h}} & = \\ 389.125 & & \rightarrow & \text{1 8 5.2h} & \end{array}$$

Reading: '12 (C) *plus* 6 *is* 18 (12h) *which makes* 2 *and carry* 1 *in hex.* 1 + 8 is 9, *plus* 12 (C), *is* 21 (15 h), *sum* 5 *and carry* 1, *and so on.*
(c) *Since* 137 = 89 h *and* 216 = D8h, *we have, with the least significant borrow* 0 *by default:*

$$\begin{array}{lrccrc} \text{minuend} & 216 & - & \rightarrow & \text{D 8h} & - \\ \text{subtrahend} & 137 & = & \rightarrow & \text{8 9h} & \\ \text{Borrows} & \underline{110} & & & \underline{\text{1 0h}} & = \\ & 79 & & \rightarrow & \text{4 Fh} & \end{array}$$

Again, let us read this subtraction: 8 *minus* 9 *needs a borrow,* 8 + *<u>sixteen from borrow</u>,* 24, *minus* 9 *makes* 15 (F). D *is* 13, *minus* 8 *minus the borrow* 1 *yields* 4.

Sixteen's Complements and Binary Two's Complements in Hex Notation

The concept of $10^N - X$ as complement of X when this one consists of N digits is independent of the base. The base is used as a reference to talk about the system. Thus, decimal if base ten, binary if base two, and so on. Hence, for hexadecimal systems we talk of *sixteen's complement* of A. Now, since there is a one to one correspondence to binary numbers, it can be shown that if A is transformed into binary expressions, than the complement will be the hex equivalent of the two's complement. Hence, one way to work fast the two's complements of bytes and words, is to work with their hex equivalents directly. A simple procedure to get the complement of a hex expression is as follows:
Sixteen's Complement: find the difference for each hex digit with respect to F and then add 1 to the result.

Example 2.14 *A set of data for bytes and words is given in hex notation as* 0×7A, 0×9F24, 91H *and* CFA20h *Find the two's complement of each one using hex notation and verify the response in binary.*

For 0×7A *we find* 100 h - 7Ah *as follows:* (FFh − 7Ah = 85 h) *and* (85h + 1) = 86 h. *Hence, the hex expression for the two's complement is* 86 h. *To verify:*

$$7Ah = 01111010B \Rightarrow \text{Two's Complement}: 10000110B = 86\,h$$

For 9F24h*, take the difference for each digit with respect to* F *and add* 1 : 60 DBh+ 1 = 60 DCh.
For 91h : 6Eh + 1 = 6 Fh.
For 0xCFA20 : 305DFh + 1 = 305 E0h.

The verification in binary form is left to the reader.

Important Remark: Hex notation not always represents four bits as is the case of the most significant hex digit. For example, 1EFh may be the hex representation for a binary expression of 12, 11, 10 or 9 bits and the two's complement will be different in each case. The hex subtraction to consider is different, depending on the case: 1000, 800, 400 and 200 h, respectively, since these are the hex expressions for 2^{12}, 2^{11}, 2^{10} and 2^9.

Example 2.15 *The hex expression* 26ACh *is being used to represent a binary number. Find the hex expression of the binary two's complement if* (a) *we are working with a* 16*-bit,* (b) 15*-bit, and* (c) 14*-bit binary number.*

(a) *In the 16-bit number, all hex digits are true digits, so we use the same method as in example 2.14 above:* 26ACh ⇒ D953h + 1 = D954h.
(b) *For the 15-bit number, the most significant hex digit represents actually three bits, so the complement is with respect to* 8000*h* = 7FFFh + 1. *Hence, for the digit* 2 *we take* 7 *as the initial minuend, getting,* 5953h + 1 ⇒ 5954h *as result.*
(c) *In the 14-bit number, the most significant hex digit represents actually two bits, so the complement is with respect to* 4000h = 3FFFh + 1. *Hence, for the digit* 2 *we take* 3 *as the initial minuend, getting,* 1953h + 1 ⇒ 1954h *as result.*

The reader can verify the results by transforming to binary and taking two's complement.

2.6 Representation of Numbers in Embedded Systems

Let us now see some methods to represent numbers in a digital environment. First for *integers* and then for *real* numbers; these are those containing a fractional component. Beginners may skip the representation of real numbers in the first experience, and come back to this chapter when the concepts may become necessary.

If only non-negative numbers are considered in the machine representation, we speak of *unsigned numbers*. If negative, zero and positive numbers are considered, then we talk of *signed numbers*. It is always important to define what type of representations we are considering, since this fact defines the interpretation of the results as well as the programming algorithms.

When trying to represent numbers with a digital system we have important constraints which can be summarized as follows:

1. Only 0's and 1's are allowed. No other symbols such as minus sign, point, radix, exponent, are available.
2. In a given environment or problem, all representations have a fixed length of N bits, so only a finite set of 2^N numbers can be represented.
3. Arithmetic operations may yield meaningless results irrespectively of their theoretical mathematical validity.

The first constraint follows from the very nature of digital systems, since we are using the "0" and "1" associated to voltage states. The second constraint tells us that the number of bits cannot be extended indefinitely and all numbers have the same number of bits, N, including left zeros. This value may be dictated either by hardware constraints or by a careful planning in our application. In any case, there is finite list of representable numbers, no matter how large or small the number is and we can always point out the "next" number following or preceding a given one. Thus, some properties of real numbers such as density[5] are lost.

The third constraint is exemplified by *overflow*, which occurs when the result of an operation is beyond the finite set of allowable numbers representable by finite-length words. An example in daily life of this fact can be appreciated in the car odometers, with a limited number of digits. An odometer with 4 digits cannot represent more than 9999 miles. Hence, adding 30 miles to 9990 does not yield 10020, but only 0020. We will have the opportunity to take a closer look to this phenomena.

2.7 Unsigned or Nonnegative Integer Representation

There are several ways in which unsigned integers can be represented in digital systems. The most common ones are the normal binary method and the 8421 Binary Coded Digit representation. Others such as Gray code, excess codes, etc., are suitable for certain applications.

2.7.1 Normal Binary Representation

In this convention, n-bit words are interpreted as normal nonnegative binary numbers in a positional system as discussed in Sect. 2.2.1. The interval of integers represented goes from 0 to $2^n - 1$. In particular,

- For bytes, $n = 8$: 0 (00h) to 255 (0xFF)
- For words, $n = 16$: 0 (0000h) to 65,535 (0xFFFF)
- For double words, $n = 32$: 0 (00000000h) to 4,294,967,295 (0xFFFFFFFF)

[5] Density of the real number system (or of rational numbers) means that between any two numbers there is an infinite quantity of real numbers.

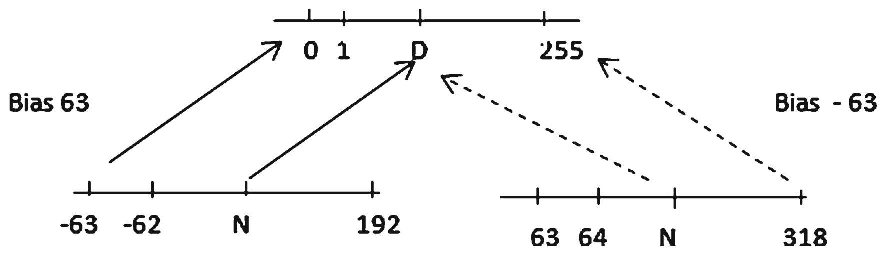

Fig. 2.3 Biased representation for number N : N + B = D

One advantage of this representation is that the hardware for the arithmetic addition is usually designed with a positional system in mind. Therefore, when using other representations it is often necessary to convert into normal binary to do the arithmetic, or else devise a special algorithm. When limited to n bits using unsigned representation, overflow in addition occurs when there is a carry.

2.7.2 Biased Binary Representation

To work "representable" number intervals that fall outside the normal binary interpretations of n-bit words, the convention of *biased* or *excess* notation is very useful. By "representable" intervals, we mean that the number of elements in the set is less than or equal to 2^n, so it is possible to assign an n-bit word to each element. The interval represented with bias B is $[-B, 2^n - B - 1]$. We illustrate the principle of bias representation with Fig. 2.3 for a byte.

If the number we want to represent is N, we add a bias B to this number and map it to D, which is then coded by normal binary convention. In other words:

$$N + B = D \tag{2.7}$$

Hence, we are mapping the interval of interest so that the lower bound corresponds to 0. Thus, the word 0 does not represent a decimal 0, but a nonzero integer $-B$, where B is the *bias*. The bias is also sometimes called *excess*, although the number can be negative.

Example 2.16 (a) *Represent* 35 *with bias* 63 *and bias* 127; (b) *What is the decimal equivalent of* 110011 *if it is biased by* 63*?*

(a) *For bias* 63*, we must find the normal binary form for* $35 + 63 = 98$*, which is* 1100010*. For bias* 127 *we should work* $35 + 127 = 162$*, for which the normal binary equivalent is* 10100010.

(b) *The decimal equivalent for a normal binary coding* 110011 *is* 51*. Since it is in excess* 63*, the actual equivalent is* $51 - 63 = -12$.

Example 2.17 *What is the interval that can be coded with bytes using a bias of (a)* 127 *and (b)* -63*?*

(a) The normal binary codification for bytes goes from 0 *to* 255*. Therefore, with bias* 127*, the range is from* -127 *to* 128.
(b) Proceeding as before, i.e., subtracting the bias from the bounds, the range is $+63$ *to* $+318$.

Using biased representations, we can include negative numbers, as seen in the above examples, or any other interval without the need of increasing the number of bits. Notice however that when arithmetic operations are involved, we need to compensate for the bias effect. This is usually done by software.

2.7.3 Binary Coded Decimal Representation -BCD-

In the *Binary Coded Decimal* representation (BCD), each digit of a decimal number is represented separately by a 4-bit binary string. It is common to use its equivalent 4-bit binary number, as shown in Table 2.2. This format is also referred to as *"8421 BCD"* to emphasize the normal weights for the individual bits: 8, 4, 2, and 1. This also differentiates it from other encoding by digits using other weights for the bits such as the 8 4 -2 -1, or codes by excess. Notice that 8421–BCD representation is similar to hex notation, except that nibbles 1010 to 1111 are excluded. This will be appreciated in the example below.

BCD encoding is very important, and it is the preferred one for many applications, especially those in which the exact representation of decimal numbers becomes an absolute need.[6] Many electronic systems are especially designed in hardware to work in this system, although most small embedded ones are based on binary hardware.

It is sometimes convenient to use bytes even when coding BCD. When using a byte, if the least significant nibble is used for the code and the most significant nibble has all 0's or all 1's, the representation is a *zoned BCD* or *unpacked BCD*. If the two nibbles of the byte are used for coding, it is a *packed BCD*. or *compressed BCD*.

Example 2.18 *Using bytes, represent the decimal* 236 *in (a) binary, (b) normal BCD and (c) packed BCD.*

(a) By conversion, since $236 < 256$*, one byte suffices and* $236 = 11101100\text{b}(0 \times \text{EC})$
(b) In normal BCD, three bytes will be required. Using the most significant nibble all 0's we get the following representation

$$\overbrace{00000010}^{2}\ \overbrace{00000011}^{3}\ \overbrace{00000110}^{6}$$

In hex notation: 02 h, 03 h*, and* 06 h

6 An introductory website for what is called *Decimal Arithmetic*, which makes all operations based on BCD encoding, is found at http://speleotrove.com/decimal/.

(c) *In packed BCD, two bytes will be necessary, with the most significant nibble taken as 0:*

$$\overbrace{0000}^{0}\overbrace{0010}^{2}\ \overbrace{0011}^{3}\overbrace{0110}^{6}$$

That is, 02 h 36 h

Sometimes, performing arithmetic directly from BCD without going through BCD-to-binary and binary-to-BCD conversions may be advantageous. However, even though the digits are individually coded, actual operations are done in binary arithmetic. There is hence the need to understand how to manipulate the binary operations in this case to yield the BCD representation of the actual result. This is explained next. For easiness, we use hexadecimal notation for the nibbles, recalling that for digits 0-9 both BCD and hex notations are identical.

BCD Arithmetic: Addition
When two decimal digits -or two digits plus a carry—are added, the difference in the result with respect to the hex notation appears as soon as the addition is greater than 9. That is, whenever the decimal addition falls between 10 and 19, generating a digit and a carry, which corresponds to 0Ah to 13h in the hexadecimal addition. To obtain the digits again, adding 6 is required, since 0Ah + 6h = 10h, and 13h + 6h = 19h. We summarize with the following
Rule for BCD addition: When adding hex digits (nibbles) in BCD representation

1. If the sum is between 0 and 9, leave it as it is;
2. Else, if it is between Ah to 13h, add 6 to the result of that sum.
3. These rules are applied separately to each column –units, tens, etc –.

Example 2.19 *Illustrate the addition rules for BCD representation with the hex digit representation for* 32 + 46, 56 + 35, *and* 85 + 54; 67 + 93; 99 + 88. *Since* 32 + 46 *yields no result greater than* 9 *in either column, the addition here is direct. On the other hand, the units column for* 56 + 35 *is greater than* 10*, so* 6 *should be added to that column. We omit carries for simplicity:*

Decimal	Hex	Decimal	Hex
32 +	32h +	56 +	56h +
46 =	46h =	35 =	35h =
78	78h	91	8Bh+
			06h =
			91h

Next, 85 + 54 *yields a result greater than* 9 *in the tens column, so we add* 60h *to the hex addition. On the other hand,* 67 + 93 *and* 99 + 88 *have both columns yielding results greater than* 9 *and we need then to add 66h to the hex addition in order to obtain the correct BCD representation as the final sum:*

```
 Decimal  Hex    ||  Decimal  Hex     ||  Decimal  Hex
  85 +    85h +  ||  67 +      67h +  ||  99 +      99h +
  54 =    54h =  ||  93 =      93h =  ||  88 =      88h =
 139      D9h    || 160        FAh +  || 187       121h+
          60h =  ||            66h =  ||            66h =
          139h   ||            160h   ||           187h
```

BCD Arithmetic: Subtraction
When taking the difference of decimal digits, the difference in the result with respect to the hex notation appears as soon as there is a need of a borrow. Since a borrow in hexadecimal system means 16 decimal units, while in decimal system means 10 units, we subtract 6 units more. Summarizing
Rule for BCD subtraction: When subtracting hex digits in BCD code,

1. If the minuend is greater or equal than the subtrahend, including a borrow, leave the difference that results;
2. Else, if a borrow is needed, subtract another 6h to the difference.
3. These rules are applied to each column –units, tens, etc –.

Example 2.20 *Illustrate the subtraction rules for BCD representation with the hex digit representation for* 59 – 27, 83 – 37, 129 – 65*, and* 141 – 48*. The subtractions are done below, without explicitly showing the borrows. The reader is invited to verify the operations*

```
Decimal  Hex    ||  Decimal  Hex
  59 –   59h –  ||  83 –     83h –
  27 =   27h =  ||  37 =     37h =
  32     32h    ||  46       4Ch –
                ||           06h =
                ||           46h
```

```
Decimal  Hex      ||  Decimal  Hex
 129 –   129h –   ||   141 –   141h –
  65 =     65h =  ||    48 =     48h =
  64       C4h    ||    93       F9h –
           60h =  ||             66h =
           64h    ||             93h
```

2.8 Two's Complement Signed Integers Representation:

To represent positive, negative, and zero numbers using strings of 0's and 1's, there are different methods, each one with its own advantages for particular applications. The most popular convention is the *two's complement* method, based on the two's complement operation introduced in Sect. 2.5.3. This method has the following important characteristics:

Number of bits: All strings must have the same number of bits and any interpretation of results must be limited to that length.
Range Of the total set of 2^n words, one half correspond to negative number representations. With n bits, the interval of integers is between -2^{n-1} and $2^{n-1} - 1$.
Backward compatibility and two's complement: To keep previous results with unsigned representations, we have the following constraints:

1. Representation of nonnegative integers are equivalent to their unsigned counterparts in normal binary system.
2. The binary representations for A and −A are two's complements of each other.
3. The representation of -2^{n-1}, has no two's complement within the set, so this number is represented by 1 followed by n−1 0's.

Sign bit: If the most significant bit is 0, the number is non-negative, if it is 1 the number is negative. The most significant bit is called *sign bit*.
Backward compatibility 2: Addition and subtraction follow the same rules as in the unsigned case.

Let us see how these principles generate the signed set of integers with four bits.

Example 2.21 *Derive the set of signed integers representable with 4 bits using the two's complement conditions. The range of integers is from* $-2^{4-1} = -8$ *to* $2^{4-1} - 1 = +7$. *From the backward compatibility property, integers* 0 *to* +7 *have the representation* 0000, 0001, ..., 0111. *Integers* $-1, -2, \dots -7$ *are the two's complements for those strings. These representations are summarized in the following table:*

Decimal	→	Nibble	Decimal	→	Nibble
+1	→	0001	−1	→	1111
+2	→	0010	−2	→	1110
+3	→	0011	−3	→	1101
+4	→	0100	−4	→	1100
+5	→	0101	−5	→	1011
+6	→	0110	−6	→	1010
+7	→	0111	−7	→	1001

On the other hand, 1000 *represents* $-2^3 = -8$.
Notice that 0000 *and* 1000 *are, respectively, two's complements of themselves. Thus, it is not possible to have representations for both* +8 *and* −8 *in this set.*

Because half of the integers are nonnegative, and must have the same representation as in the unsigned case, all of them are represented with a leading 0, covering exactly half of the set. Hence, all the rest, i.e., the words representing negative numbers, have the most significant bit equal to 1. This is why the most significant bit indicates whether the number is negative or nonnegative.

On the other hand, since the addition and subtraction rules are valid, we can decompose a string as follows:

$$b_{n-1}b_{n-2}\cdots b_1b_0 = \underbrace{b_{n-1}00\cdots 0}_{\text{msb followed by n}-1\text{ zeros}} \quad +0b_{n-2}\cdots b_1b_0$$

Now, since the second term in the right is a "normal" binary representation of a positive number, for which the power series (2.3) is applicable, and the first term is 0 if $b_{n-1} = 0$, and -2^{n-1} if $b_{n-1} = 1$, we arrive to the following conclusion:

When the word $A = b_{n-1}b_{n-1}\cdots b_1b_0$ represents a signed number in two's complement convention, it can be expanded in the following weighted sum, similar to the positional expansion:

$$A = -2^{n-1}\cdot b_{n-1}+2^{n-2}\cdot b_{n-2}+\cdots+2^1\cdot b_1+b_0 = -2^{n-1}\cdot b_{n-1} + \sum_{i=0}^{n-2} 2^i\cdot b_i \quad (2.8)$$

For example, 10110111 represents $-128 + (32 + 16 + 4 + 2 + 1) = -73$.

For bytes, words and double words, the range of signed integers becomes, respectively:

- Byte: $-2^7 = -128$ to $2^7 - 1 = +127$;
- Word: $-2^{15} = -32,768$ to $2^{15} - 1 = +32,767$
- Double Word: $-2^{31} = -2,147,403,608$ to $2^{31} - 1 = +2,147,403,607$

Sign Extension: Signed numbers using n-bit words can be extended to m-bit words, with $m > n$ bit strings, by appending to the left the necessary number of bits, all equal to the original sign bit. This procedure is called *sign extension*.

Applying, -5 with four bits has the code 1011 and with six bits it becomes 111011, while $+5$ becomes 000101. The proof of this property is left as an exercise.

The weighted sum (2.8), together with sign extension when applicable, can be applied for reading and generating representations of negative numbers. If the sign bit is 0, the number is non-negative and can be read as always. If the sign bit is 1, reduce the length scanning from left to right until you find the first 0. Read the "positive" component formed by the M remaining bits, including the 0, and subtract it from 2^M. For example, to read 1111101,consider only 01, which stands for 1, and $2^2 = 4$, so the result is $-(4 - 1) = -3$.

Let us illustrate again reading numbers and generating opposites with examples. This type of exercises become useful when working with programs and validating them. Remember that the programmer needs to verify the correctness by working examples with data that will be used in the process.

Example 2.22 *Find the signed decimal integer represented by the* 12*-bit word FD7h. Various methods are illustrated:*

(a) Since FD7h *represents* 111111010111 *and the sign bit is 1, the number is negative. By the sign extension principle, this number would be the same one*

represented with 7 bits, 1010111. *Here,* 010111B *is the binary representation for* 23*, so the expansion yields* $-2^6 + 23 = -41$. *Hence,* FD7h *is the representation for* -41.

(b) Logical complement plus one: The two's complement of 111111010111 *is*

$$000000101000 + 1 = 101001$$

which is the binary equivalent of 41. *Hence,* FD7h *stands for* -41.

(c) Working with hex complements: The hex complement for FD7h *with* 12 *bits is* 029h*, with decimal equivalent* $2 \times 16 + 9 = 41$. *Hence,* FD7h *stands for −41*

While the previous example was concerned with reading representations, now let us work the conversion process from signed decimal expressions. Positive ones have been previously considered, so let's just take the negative numbers now.

Example 2.23 *Find the two's complement representation for* -104 *with* 16 *bits and express it in hex notation.*

We solve by three methods:

(a) By the two's complement of 104*: With 16 bits, +104 is* 0000000001101000. *The two's complement becomes then* 1111 1111 1001 1000*, or* 0×FF98.

(b) Hex complement directly: $+104 = 0068\text{h}$ *and the hex complement becomes* 0×FF98h.

(c) By the weighted sum, starting with the minimum number of bits required and then sign extending: Since $-104 > -2^{8-1} = -128$*, eight bits are enough. Taking (2.8) as a reference,* $128 - 104 = 24 = 11000\text{B}$. *Therefore, inserting 0's as needed, with eight bits we have* $-128 + 24 = -104 \rightarrow 10011000$. *After sign extension with eight additional 1's to the left to complete the 16 bits we arrive at the hex notation* 0xFF98.

Example 2.24 *A byte* 83h *is used to represent numbers, but we need to extend the representation to* 16*-bit word size. What is the new representation and meaning, if (a) we are working with unsigned numbers, and (b) we are working with signed numbers.*

(a) Unsigned numbers are extended to any number of bits by simply adding leading zeros. Hence, the new representation would be 0×0083. *The decimal equivalent in both cases is* $8 \times 16 + 3 = 131$.

(b) The most significant bit is 1, so the number is negative, $83\text{h} \rightarrow -128 + 3 = -125$. *Since we are working with signed numbers, the sign extension rule should be applied, and the new representation becomes* 0×FF83.

2.8.1 Arithmetic Operations with Signed Numbers and Overflow

When adding and subtracting signed number representations, a carry or borrow may appear. This one is discarded if we are to keep the fixed length representation. *Overflow* will occur, however, if in this fixed length the result is nonsense. In particular;

- Addition of two numbers of the same sign should yield a result with the same sign;
- a positive number minus a negative number should yield a positive result and
- a negative number minus a positive number should yield a negative result.

When the results do not comply with any of these conditions, there is an overflow. More explicitly:

Overflow occurs if (a) addition of two numbers of the same sign yields a number of opposite sign or (b) subtraction involving different signed numbers yields a difference with the sign of the subtrahend.

Let us illustrate operations and overflow with 4-bit numbers, which were generated in Sect. 2.8.

Example 2.25 *(a) Check the validity of the following operations for signed numbers using two's complement convention with four bits:* $3+2, 4+(-4), (-6)+7, (-3)+(-5), 6-2, 4-4, (-2)-(-8), 3-(-4)$.*(b) Verify overflow in the following cases:* $3+5, (-5)+(-8), 4-(-6), (-6)-(+3)$.

(a) *All the following operations yield valid results discarding any carry or borrow when present. For example,* $3-(-4)$ *in binary yields* 10111*, but only* 0111 *is considered yielding* $+7$*, as expected. Notice that* $(+4)+(-4)$ *and* $4-4$ *both yield* 0 *when only four bits are taken, but the former yields a carry.*

$$\begin{array}{llllll}
3+ & \rightarrow & 0011+ & 4+ & \rightarrow & 0100+ \\
\underline{2=} & \rightarrow & \underline{0010}= & \underline{(-4)=} & \rightarrow & \underline{1100=} \\
5 & \rightarrow & 0101 & 0 & \rightarrow & 1\,0000 \\
 & & & & & \\
(-6)+ & \rightarrow & 1010+ & (-3)+ & \rightarrow & 1101+ \\
\underline{7=} & \rightarrow & \underline{0111=} & \underline{(-5)=} & \rightarrow & \underline{1011=} \\
1 & \rightarrow & 1\,0001 & (-8) & \rightarrow & 1\,1000 \\
 & & & & & \\
6- & \rightarrow & 0110- & 4- & \rightarrow & 0100- \\
\underline{2=} & \rightarrow & \underline{0010=} & \underline{4=} & \rightarrow & \underline{0100=} \\
4 & \rightarrow & 0100 & 0 & \rightarrow & 0000 \\
 & & & & & \\
(-2)- & \rightarrow & 1110- & 3- & \rightarrow & 0011- \\
\underline{(-8)=} & \rightarrow & \underline{1000=} & \underline{(-4)=} & \rightarrow & \underline{1100=} \\
6 & \rightarrow & 0110 & 7 & \rightarrow & 1\,0111
\end{array}$$

(b) *Overflow now occurs when the result of operation falls outside the range covered by the set of strings and is mainly shown by a result with a sign bit different from what was expected. Since* 4*-bit words cover from* $-2^3=-8$ *to* $2^3-1=7$*, the operations* $3+5=8, (-5)+(-8)=(-13)$ *and* $4-(-6)=10$ *do not make* *sense in this set. Let us look at the results in binary form interpreted from the standpoint of the two's complement convention:*

$$\begin{array}{lcrlcrlcr}
0011+ & \to & +3 & 1011+ & \to & -5 & 0100- & \to & +4 \\
\underline{0101=} & \to & +5 & \underline{1000=} & \to & -8 & \underline{1100=} & \to & -6 \\
1000 & \to & -8 & 10011 & \to & +3 & 11010 & \to & -6
\end{array}$$

We see in the first case an addition of two numbers with leading bit 0 (non negative) yielding a number with a sign bit 1. In the second case, two negative numbers add up to a positive result. In the third case, we subtract a negative number from a positive one, but the result is negative instead of positive. All these cases are deduced after analyzing the sign bits of the operands and results.

2.8.2 Subtraction as Addition of Two's Complement

From elementary Algebra it is known that A − B = A + (−B). On the other hand, for the two's complement signed integer representations in digital systems, $-B = \overline{B}+1$. Therefore, the subtraction operation can be reduced to an addition in the form

$$A - B = A + \overline{B} + 1 \tag{2.9}$$

In other words, both in unsigned and signed conventions, using two's complement addition for a subtraction is valid. The main difference is that this principle is limited to the n bits used in the signed representation and the result is directly interpreted for both positive and negative cases.

This principle allows us to use the same hardware for both representations. When performed in software, it is the programmer's responsibility to interpret results. One important fact to not forget is that in two's complement addition for a subtraction, carry and borrow are opposite. Look at cases 4 − 4 and 4 + (−4) in the example above. This difference should be taken into account when programming.

Remark: In the MSP430 microcontrollers, as in most microcontrollers, subtraction is actually realized through the addition of the two's complement of the subtrahend. Hence, a carry = 0 corresponds to the presence of a borrow.

2.9 Real Numbers

Previous sections emphasized the representation of integers using N-bit words. Now we consider real numbers, that is, numbers containing a fractional part. Two methods are the most common: the fixed-point and the floating-point representations.

Fig. 2.4 Format $Fm_1 \cdot m_2$ for an n-bit word, where $m_1 + m_2 = n$

Integer part, m_1 bits	Fractional part, m_2 bits
$b_{N-1} \quad b_{N-2} \cdots b_{m_2}$	$b_{m_2-1} \quad \cdots \quad b_0$

Fixed-point notation is preferred in situations where huge quantities of numbers are handled, and the values are neither very large nor very small. An advantage is that operations are the same as with integers, without changes in hardware, since the only difference is the presence of an implicit point. On the other hand, very large and very small numbers require a large number of bits and floating-point is preferred. The arithmetic with floating point number representation is more complicated than that of integer numbers and will be left out of the scope of this text.

2.9.1 Fixed-Point Representation

In a *fixed-point notation*, or a *fixed-point representation system*, numbers are represented with n-bit words assuming the radix point to always be in the same position, established by convention. The notation can be used to represent either unsigned or signed real numbers.

Let A be the n-bit word $b_{n-1}b_{n-2}\dots b_1b_0$ and $n = m_1+m_2$. The format $Fm_1 \cdot m_2$ for A means that the m_2 least significant bits constitute the fractional part and the m_1 most significant bits the integer part, as illustrated in Fig. 2.4.

From this interpretation and the power expansion (2.3) as well as the expansion (2.8) for signed numbers, we have

$$\begin{aligned} A &= \left(b_{n-1}\, b_{n-2} \cdots b_{m_2} b_{m_2-1} \cdots b_1 b_0\right)_{Fm_1 \cdot m_2} \\ &= \pm b_{n-1}2^{m_1-1} + b_{n-2}2^{m_1-2} + \cdots \\ &\quad + b_{m_2} + b_{m_2-1}2^{-1} + b_{m_2-2}2^{-2} + \cdots b_0 2^{-m_2} \end{aligned} \tag{2.10}$$

Factoring 2^{-m_2}, this can also be written as

$$\begin{aligned} A &= (b_{n-1}b_{Nn-2} \cdots b_1 b_0)_{Fm_1 \cdot m_2} \\ &= \frac{1}{2^{m_2}} \times \left(\pm b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0\right) \end{aligned} \tag{2.11}$$

In these equations, the plus (+) sign in the highest term applies to the unsigned case and the minus (−) to the signed one.

Two points are worth mentioning here. First, numbers increase in *steps* of $1/2^{m_2}$. So density in the approximation depends largely on the number of bits in the fractional part. Second, the number being represented is equal to the integer equivalent of the word divided by 2^{m_2}, both for signed and unsigned numbers. Based on this last property, the interval represented by n-bit word $b_{n-1}b_{n-2}\cdots b_1b_0$ in format $Fm_1 \cdot m_2$ is

- 0 to $(2^n - 1)/2^{m_2}$ for unsigned numbers, and
- $-2^{-m_2} \cdot 2^{n-1}$ to $2^{-m_2} \cdot (2^{n-1} - 1)$ for signed numbers

Let us now work some examples.

Example 2.26 *(a) Describe the unsigned numbers represented by* 4*-bit words in format* F2.2 *and format* F0.4*. (b) Repeat for signed numbers.*

(a) **Unsigned numbers***: Format* F2.2 *covers from* 0 *up to* $2^{-2} \times \left(2^4 - 1\right) = 3.75$ *in step size of* $2^{-2} = 0.25$*. On the other hand, format* F0.4 *covers from* 0 *to* $2^{-4} \times \left(2^4 - 1\right) = 0.9375$ *with a step size of* $2^{-4} = 0.0625$*.*

(b) **Signed numbers***: The F2.2 format range is from* $-2^{-2} \times 2^3 = -2$ *to* $2^{-2} \times (2^3 - 1) = 1.75$ *in steps of* $2^{-2} = 0.25$*. On the other hand, the F0.4 format range is from* $-2^{-4} \times 2^3 = -0.5$ *to* $2^{-4} \times (2^3 - 1) = 0.4375$ *in steps of* $2^{-4} = 0.0625$*. With this information, we can generate the representations for each case as described in the following tables:*

Unsigned numbers				Signed numbers			
Format	F2.2	Format	F0.4	Format	F2.2	Format	F0.4
0000	0.00	0000	0.0000	1000	−2.00	1000	−0.5000
0001	0.25	0001	0.0625	1001	−1.75	1001	−0.4375
0010	0.50	0010	0.1250	1010	−1.50	1010	−0.3750
0011	0.75	0011	0.1875	1011	−1.25	1011	−0.3125
0100	1.00	0100	0.2500	1100	−1.00	1100	−0.2500
0101	1.25	0101	0.3125	1101	−0.75	1101	−0.1875
0110	1.50	0110	0.3750	1110	−0.50	1110	−0.1250
0111	1.75	0111	0.4375	1111	−0.25	1111	−0.0625
1000	2.00	1000	0.5000	0000	0.00	0000	0.0000
1001	2.25	1001	0.5625	0001	0.25	0001	0.0625
1010	2.50	1010	0.6250	0010	0.50	0010	0.1250
1011	2.75	1011	0.6875	0011	0.75	0011	0.1875
1100	3.00	1100	0.7500	0100	1.00	0100	0.2500
1101	3.25	1101	0.8125	0101	1.25	0101	0.3125
1110	3.50	1110	0.8750	0110	1.50	0110	0.3750
1111	3.75	1111	0.9375	0111	1.75	0111	0.4375

Example 2.27 *What is the range of unsigned and signed numbers covered with a* 16*-bit word in formats* F8.8 *and* F0.16*? What is the step in each case? The steps for the* F8.8 *and* F0.16 *formats are, respectively,* $2^{-8} = 0.00390625 \approx 3.91 \times 10^{-3}$ *and* $2^{-16} = 0.0000152587290625 \approx 1.53 \times 10^{-5}$*. The intervals are shown in the following table.*

Number type	Format	Lower limit	Upper limit
Unsigned	F8.8	0	$\frac{2^{16}-1}{2^8} = 255.9960938$
Unsigned	F0.16	0	$\frac{2^{16}-1}{2^{16}} = 0.9999847412109375$
Signed	F8.8	$-\frac{2^{15}}{2^8} = -128$	$\frac{2^{15}-1}{2^8} = 127.99609375$
Signed	F0.16	$-\frac{2^{15}}{2^{16}} = -0.5$	$\frac{2^{15}-1}{2^{16}} = 0.4999847412109375$

These examples show how the number of bits in the fractional part defines the precision between steps. This precision gets better at the expense of the total range. We can also appreciate the effect of the step. In the example for unsigned numbers in format F0.4, for example, 0110 can be used to represent any number between 0.3750 up to, but not including, 0.4375. Hence, the step indicates the maximum error which can be introduced with this representation. Therefore, *to consistently keep an error lower than ϵ, the number of bits m_2 in the fractional part must satisfy* $2^{m_2} \geq \epsilon$.

Very small numbers: For very small numbers without integer part *extra scaling* may be used. This consists in implicit additional zeros to the left after the point, including the step. Thus, with an extra scaling of three decimals, the range for signed numbers in the format F0.4 (see example 2.26) becomes [–0.0005, 0.0004375].

Two's complement in fixed point representation

When working (unsigned) integers with n bits, it was pointed out that the two's complement of A is $2^n - A$. Now, for format Fm1.m2, following (2.11) it can be deduced that the difference is now to be considered with respect to 2^{m1}. This can be seen in Example 2.26. In format F2.2, numbers adding $2^2 = 4$ are represented by two's complement strings. The same can be said for F0.4 and $2^0 = 1$.

Example 2.28 *What is the signed decimal equivalent for* 10110010 *if the string is in format F3.5?*

(a) By power expansion,

$$10110010 \rightarrow -2^2 + 1 + 0.5 + 0.0625 = -2.4375$$

(b) By two's complement

$$01001110 \rightarrow -(2 + 0.25 + 0.125 + 0.0625) = -2.4375$$

(c) By two's complement in decimal representation:

$$10110010 = 5.5625 \rightarrow -(8 - 5.5625) = -2.4375$$

(d) By integer division:

$$10110010 = -78 \rightarrow \frac{-78}{32} = -2.4375$$

2.9.2 Floating-Point Representation

Floating-point is preferred for very large or very small numbers, or to cover large intervals and better precision when two large numbers are divided. There are several conventions for floating point representation, but IEEE 754 Floating Point Standard is the most common format. We discuss this one here. Texas Instruments uses both this one and another one adjusted to its products.

Floating-point representation is similar to the scientific notation, where a number is written in the form $b.a_{-1}a_{-2}\ldots \times 10^m$, with m an integer and $0 \leq |b| \leq 9$. For example, 193.4562 may be expressed as 1.934562×10^2, and $0.00132 = 1.32 \times 10^{-3}$. When $b \neq 0$, we say that the decimal point is in *normal position* and the notation is *normalized*. The number b is the *base* and the fractional part $a_{-1}a_{-2}\ldots$ the *mantissa*.

A similar principle is used for binary numbers. For easiness in reading, we use mixed notation, with the base in binary and power of 2 in decimal, as in $110.101 = 1.10101 \times 2^2$. As an extension, the mantissa is *normalized* if the decimal point is normalized, and *unnormalized* or *denormalized* when the integer part is 0. Using denormalized mantissa, $111.101 = 0.110101 \times 2^3$.

IEEE standard convention provides two representations: *single-precision floats* using 32 bits , and *double-precision floats* with 64 bits. Only the sign, the exponent, and the mantissa need to be included, with the following rules:

Sign: The most significant bit is the *sign bit*: 0 for positive, 1 for negative numbers
Exponent: The exponent is biased:

- Single precision: 8 bits [bits 30-23], with bias 127
- Double precision: 11 bits [bits 62-52], with bias 1023

Mantissa: The mantissa is normalized, except for exponent 0

- Single precision: 23 bits [bits 22-0]
- Double precision: 52 bit [bits 51-0]

The distribution is illustrated in Fig. 2.5.

The exponent is biased to avoid the need of representing negative exponents. Thus, an exponent of 00h in single precision floats means −127, and 000h is −1023 in double-precision numbers. All one's exponents (FFh in single-precision and 7FFh in double-precision floats) have special meaning. Moreover, all 0's exponent assume denormalized mantissa. The conventions for all 0's and all 1's exponents are shown in Table 2.5.

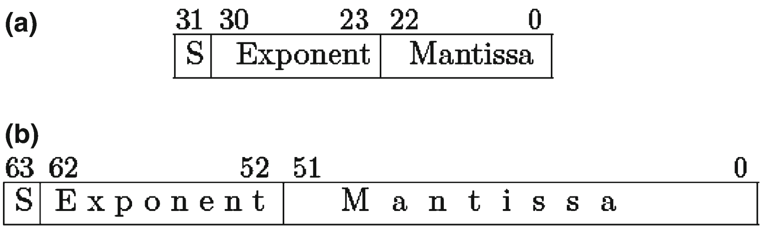

Fig. 2.5 Component distribution in real numbers

Table 2.5 Special convention for IEEE standard floating point

Sign	Exponent	Mantissa	Meaning
0	0	0	0
1	0	0	-0
0	All 1's	0	Infinity
1	All 1's	0	$-$Infinity
X	All 1's	$\neq 0$	NaN (Not a number)
X	00h	$\neq 0$	Mantissa denormalized

Let us work some examples of floating point numbers to get the feeling for this convention.[7]

Example 2.29 *Convert the following single-precision floating numbers to their decimal equivalents: (a)* 4A23100CH, *(b)* 002A2000H.

(a) 4A23100CH *stands for* 0100 1010 0010 0011 0001 0000 0000 1100. *Let us now separate the bits into the three components:*

Sign	Exponent	Mantissa
0	10010100	010 0011 0001 0000 0000 1100

The sign is 0, *which means a positive number.*
The decimal equivalent to the binary in the exponent, $10010100b = 148$ *is biased by 127, so the actual exponent is* $148-127 = 21$.
Including the leading 1, *and using the mixed notation mentioned before, we have* $1.01000110001000000001100 \times 2^{21} = 1010001100010000000011.00B = 2,671,619$.

(b) *The hex notation* 80300000H *means* 1000 0000 0011 0000 0000 0000 0000 0000.
Proceeding as before, the sign bit is 1, *so the number is negative. The exponent part is 0, and the mantissa is* 011, *disregarding the rightmost zeros. Since the*

[7] The reader may refer to the Prof. Vickery's website http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html in Queen's College of CUNY for automatic conversions. The program was devised by the student Quan Fei Wen.

exponent part is 0, the mantissa is denormalized and the actual exponent is –127. *Therefore the decimal equivalent number is computed as* $-\left(0.011\times 2^{-127}\right) = -\left(2^{-129}+2^{-130}\right) \approx -2.20405\times 10^{-39}$.

Example 2.30 *Obtain the single-precision floating point representation for* 532.379 *and for* -2.186×10^{-5}.

To obtain floating point representation in single-precision, we follow four steps:

1. *Express the binary equivalent of* 532.379 *with* 24 *bits:*
   - 532 = 1000010100B *yields ten bits;*
   - 0.379 *with* 14 *bits:* 0.379 = .01100001000001B
   - *Hence,* 532.379 = 1000010100.01100001000001b
2. *Express the* 24*-bit equivalent in binary 'scientific' form:*
   - $532.79 = 1.00001010001100001000001\times 2^{9}$
3. *Identify each component of the floating-point representation, biasing exponent by 127:*
   - *Sign: 0 for positive number*
   - *Exponent:* 9 + 127 = 136 = 10001000b
   - *Mantissa or significant:* 00001010001100001000001
4. *Combine the elements:*
   - *Result:* 0100 0100 0000 0101 0001 1000 0100 0001
   - *In hex-notation:* 44051841h

Now for -2.186×10^{-5}*:*

1. *Express the binary equivalent of* 0.00002186 *with* 24 *significant bits (that is, zeros before the first 1 do not count):*
   - 0.0000000000000001011011101011111111111110
2. *Express the* 24*-bit equivalent in binary 'scientific' form:*
   - $2.186\times 10^{-5} = 1.01101110101111111111110\times 2^{-16}$
3. *Identify each component of the floating-point representation, biasing exponent by 127:*
   - *Sign:* 1 *for negative number*
   - *Exponent:* −16 + 127 = 111 = 01101111b
   - *Mantissa or significant:* 01101110101111111111110
4. *Combine the elements:*
   - *Result:* 1011 0111 1011 0111 0101 1111 1111 1110
   - *In hex-notation:* B7B75FFEh

Example 2.31 *What are the smallest and largest positive decimal numbers representable in floating-point with (a) single-precision mode, and (b) double-precision mode?*

(a) *To find the smallest decimal number, the exponent in the floating-point should be* 0*, with the smallest non-zero mantissa. This number is therefore* 00000001H. *Interpreting this expression as before, the smallest number becomes*

$$0.00000000000000000000001 \times 2^{-127} = 2^{-23} \times 2^{-127} = 2^{-150} \approx 7.006 \times 10^{-46}$$

Any attempt to represent a smaller number results in underflow.
For the largest number, the exponent and mantissa should be as large as possible. Since the exponent cannot be all 1's, the largest possible exponent is 1111 1110*, equivalent to* $254 - 127 = +127$ *after unbiasing. Hence, the number we are looking for is* 7F7FFFFFH*, which corresponds to* $1.11111111111111111111111 \times 2^{127} = \left(2^{24} - 1\right) \times 2^{104} \approx 3.4 \times 10^{38}$.

(b) *For double-precision mode, similar considerations follow. For the smallest number,* 0000000000000001H*, using for convenience mixed hex and decimal notations,*

$$0.0000000000000001h \times 2^{-1023} \approx 5 \times 10^{-324}$$

For the largest number the exponent is 11111111110B*, equivalent to* 1023 *after unbiasing. Hence, the largest number is* 0x7FEFFFFFFFFFFFFF*, which is calculated as*

$$\left(2^{53} - 1\right) \times 2^{1023} \approx 1.8 \times 10^{308}$$

2.10 Continuous Interval Representations

Embedded systems are also applied in the measurement or monitoring of analog signals. This application requires encoding an analog signal with an n bit-word. Electronically, this is done with an Analog to Digital Converter (ADC), as discussed in Chap. 10. We introduce here the elementary theoretical basis.

Quantization of an interval of real numbers $[a, b]$ is the process of representing it by a finite set of values $Q_0, Q_1, \ldots Q_m$ called Quantization Levels. The interval $[a, b]$ is partitioned in m subintervals, one per quantization level. For digital encoding using n bit words, $m = 2^n$ subintervals. Each quantization level is representative of any value x in the associated sub interval.

The difference $x - Q_k$ between the actual value and its associated quantization level is the *Quantization Error*. The same n-bit word is used to encode both x and Q_k.

The most common way to define the quantization levels is with a uniform distribution in the interval of interest. With this method, the difference between two consecutive quantization levels is

$$\Delta = \frac{b-a}{2^n} \tag{2.12}$$

Δ is called *resolution* or *precision* of the quantization. Most often, the resolution is referred to by the number of bits, thus talking of an n-bit resolution. Figure 2.6 illustrates this concept with 3 bits. Using this figure as reference, let us illustrate several general properties.

1. The upper limit of the quantized interval, b, is not a quantization level. This value will be "represented" by level Q_{2^n-1}, with an error of 1 Δ.
2. Since any value x will be interpreted by one level Q_k, any quantization error in the even distributed levels satisfies

$$Quantization\, error\; = x - Q_k \leq \Delta \tag{2.13}$$

3. If we start at a with Q_0 and enumerate successively the levels, then the quantization levels are defined by

$$Q_k = a + k\Delta \quad k = 0, 1, \ldots, 2^n - 1 \tag{2.14}$$

That is,

$$a, \quad a + \Delta, \quad a + 2\,\Delta, \ldots, \quad a + \left(2^n - 1\right)\,\Delta$$

The level Q_k in (2.14) may be encoded with the unsigned binary equivalent n-bit word for k. In this case the encoding is said to be *straight binary*. Since we can go from one encoding to the next one by adding 1 to the LSB, the resolution Δ defined by (2.12) is also called 1*LSB resolution* . .

If a straight binary encoding is assigned with $Q_0 = a \neq 0$, as illustrated in Fig. 2.6, it is called *offset quantization*. If $a = 0$, then it is a *normal* quantization. It can be demonstrated that the midpoint of interval $[a, b]$ is always a quantization level $Q2^{n-1}$. In straight binary codes is encoded with 1000…00.

Example 2.32 *An interval between −10 and +10 is quantized with using equally separated quantization levels. (a) Find the LSB resolution, and the quantization levels for a 4-bit resolution. (b) For an 8-bit resolution, find the LSB resolution and enumerate the lowest four and the highest four quantization levels.*

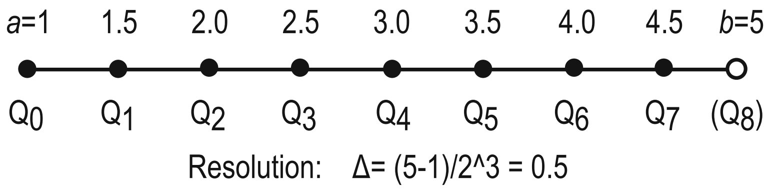

Fig. 2.6 Uniform distribution of quantization levels for 3 bits

Solution: *With 4 bits, the LSB resolution is found using (2.12):*

$$\Delta_4 = \frac{10-(-10)}{2^4} = 1.25$$

The sixteen quantization levels are then:

$Q_0 = -10.00$	$Q_4 = -5.00$	$Q_8 = 0.00$	$Q_{12} = +5.00$
$Q_1 = -8.75$	$Q_5 = -3.75$	$Q_9 = +1.25$	$Q_{13} = +6.25$
$Q_2 = -7.50$	$Q_6 = -2.50$	$Q_{10} = +2.50$	$Q_{14} = +7.50$
$Q_3 = -6.25$	$Q_7 = -1.25$	$Q_{11} = +3.75$	$Q_{15} = +8.75$

Notice that the greatest quantization value is 1 Δ less than $+10$, as expected.
(b) With 8 bits,

$$\Delta_8 = \frac{10-(-10)}{2^8} = 0.078125$$

The first four quantization levels are $Q_k = -10 + k\Delta$ for $k = 0, 1, 2,$ and 3:

$$Q_0 = -10,\ Q_1 = -9.921875\ Q_2 = -9.84375,\ \textit{and}\ Q_3 = -9.765625$$

The upper four apply the same formula with $k = 252, 253, 254, 255$:

$$Q_{252} = 9.6875\ Q_{253} = 9.765625\ Q_{254} = 9.84375\ Q_{255} = 9.921875$$

Encoding with regular intervals

It was pointed out that a subinterval is associated uniquely with a quantization level Q_k and any value x in the subinterval is encoded as Q_k. Now, encoding will depend in how we assign the subintervals.

As Fig. 2.6 illustrates, the quantization levels defined by Eq. (2.14) automatically determine a set of 2^n equal subintervals. Each of this has a quantization level as a lower bound. Let us associate then the subinterval to this bound. With this assignment, for an analog x we can find the encoding as the decimal equivalent k of the associated n-bit word by

$$k = \begin{cases} \text{int}\left[2^n \frac{x-a}{b-a}\right] & \text{if } x < b \\ 2^n - 1 & \text{otherwise} \end{cases} \tag{2.15}$$

where "int[•]" denotes the integer less than or equal to the argument value.

Let us illustrate the full process with an example.

Example 2.33 *Continuing with example 2.32, find the encoding, the quantization level, and the quantization errors for -2.20 and 7.4 for the 4 and 8-bit resolutions, when regular sub intervals are used in the quantization.*

Solution: *Since we have already the list of quantization levels from the previous example, we could look at it and find the associated quantization level for* -2.20 *as* $Q_6 = -2.50$*, The encoding is therefore* 0110.

However, for the sake of illustrating the process, let us use Eq. (2.15) *as*

$$N_Q = int\left[2^4 \frac{-2.20-(-20)}{20}\right] = int[6.24] = 6$$

with the quantization level, $-10 + 6(1.25) = -2.50$*. The quantization error is* $-2.20 - (-2.5) = 0.3$.

The table below summarizes the results for all requested cases. For later comparison, we have included the percentage of the error with respect to the 1 LSB resolution.

	4-bits		8-bits	
Parameter	−2.20	7.40	−2.20	7.40
k	6	13	99	222
Q_k	−2.5	6.25	−2.265625	7.34375
Error	0.30	1.15	0.065625	0.05625
Error/Δ	24 %	92 %	84 %	72 %

This previous example illustrates two facts. One is that the quantization error decreases as we increase the number of bits. This process has practical limitations such as hardware size, noise, and others. The second is that the maximum error is 1 LSB, which sometimes cannot be tolerated. Let us look at the following alternative which is most common.

2.10.1 Encoding with $\frac{1}{2}$LSB Offset Subintervals

The bound for a quantization error is minimum if the level is at the center of the subinterval. Since the distance between two levels is the 1 LSB resolution, then we can reduce the errors by shifting the subintervals half LSB. This is illustrated in Fig. 2.7.

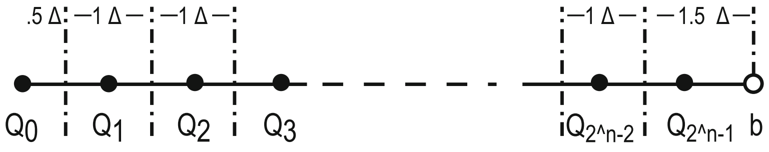

Fig. 2.7 Half LSB partition of interval $[a, b]$

In this case, with the exception of values in the upper subinterval, for any value x the quantization error is within $\pm\frac{1}{2}$LSB, which is the best we can get. Only those values falling in the range $[Q_{2^n-1} + 0.5\Delta, b]$ may yield a greater error, but always less than 1 LSB.

The encoding for x follows a similar computation, except that now we round to the nearest integer, and not to the lower bound integer. That is,

$$k = \begin{cases} \text{ROUND}\left(2^n \frac{x-a}{b-a}\right) & \text{if } x < \text{a and } k < 2^n \\ 2^n - 1 & \text{Otherwise} \end{cases} \tag{2.16}$$

where rounding is to the nearest integer.

Example 2.34 *Repeat example 2.33 using half LSB partition.*

Solution: *We now use equation* (2.16) *instead of* (2.15). *The reader can verify the following table.*

	4-bits		8-bits	
Parameter	−2.20	7.40	−2.20	7.40
k	6	14	100	223
Q_k	−2.5	7.5	−2.1875	7.421875
Error	0.30	−0.1	−0.0125	−0.021875
Error/Δ	24 %	−8 %	−16 %	−28 %

Notice the better approximation results. All errors are less than 50 % Δ.

2.10.2 Two's Complement Encoding of $[-a, a]$

In many applications, the interval to be encoded is symmetrical around the origin. In this case, it may be more convenient to encode using signed two's complement numbers, as illustrated in Fig. 2.8 for three bits. Notice that the quantization values remain the same.

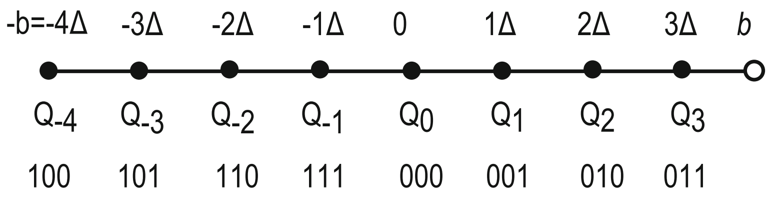

Fig. 2.8 Half LSB partition of an analog interval

In the encoding process, we apply formulas similar to the previous ones, but applying a 0 offset. Namely, we now have the following formulas where k may be a positive, negative, or zero integer, and its encoding is done with two's complement convention.

For quantization levels

$$Q_k = k\Delta \quad k = -2^{n-1}, \ldots -1, 0, 1, \ldots, 2^{n-1} - 1 \tag{2.17}$$

For regular subintervals, we have

$$k = \begin{cases} \text{int}\left[2^n \frac{x}{2b}\right] & \text{if } x < b \text{ and } k < 2^{n-1} \\ 2^{n-1} - 1 & \text{Otherwise} \end{cases} \tag{2.18}$$

And for half LSB partition

$$k = \begin{cases} \text{ROUND}\left(2^n \frac{x}{2b}\right) & \text{if } x < b \text{ and } k < 2^{n-1} \\ 2^{n-1} - 1 & \text{Otherwise} \end{cases} \tag{2.19}$$

2.11 Non Numerical Data

Besides numerical data, digital systems work non numerical information as well, represented with only 0's and 1's. In this section some of these codes are introduced.

2.11.1 ASCII Codes

There are many character codes, or *alphanumeric* codes, as they are also known. The most popular one for embedded systems is the *ASCII* code. ASCII is the acronym of *American Standard Code for Information Interchange*. Using seven bits, representations are given for the 26 upper and 26 lower case letters of the English alphabet, the ten numerals (0–9), miscellaneous symbols for punctuation such as, ":", ",", and other items.

ASCII was initially introduced for information exchange, it includes codes for control characters and operations used for rounding data and arranging printed text into a specific format. As such this set includes codes for layout control such as "backspace", "horizontal tabulation", and so on. Other control characters are used to separate data into divisions like paragraphs, pages, files; examples are "record separator", "file separator". Others are used for communications, such as "start of text", "end of text", etc.

Table 2.6 ASCII code chart

		Most Significant Digit							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	0	NUL	DLE	SP	0	@	P	‘	p
	1	SOH	DC	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
Least	**6**	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	’	7	G	W	g	w
Significant	**8**	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
Digit	**A**	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	−	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

Table 2.6 shows the ASCII code chart in hex notation, and Table 2.7 explains the acronyms for the different control functions. The seven bit code for a particular character or control function is found reading up the column for the most significant hex digit (three bits) and horizontally for the least significant one, or least significant four bits.

Example 2.35 *(a) Find the ASCII code for characters* 5 *and* S*; (b) Assuming an additional leading 0 to form bytes in an ASCII coding, decode* 01001101 01010011 01010000 00110100 00110011 00110000.

(a) From table 2.6 "5" is in the intersection of column 3 *and row* 5*. Hence, its ASCII code is* 35h*. Proceeding similarly, the ASCII code for* S *is* 53h*.*

01001101 01010011 01010000 00110100 00110011 00110000 can be expressed in hex notation as 4Dh 53h 50h 34h 33h 30h. From Table 2.6 we can verify that this code stands for MSP430.

Programmers usually don't have to memorize ASCII codes, since compilers take care of them with the *character constants* or *strings*, almost always enclosed by apostrophes. Thus, instead of writing 0x4D or 4Dh, the programmer writes `'M'`, both in C or asembly language. Similarly, it is easier to write `'The ball'` than `0x54 0x68 0x65 0x20 0x62 0x61 0x6C 0x6C`. Several of the ASCII control codes have made it into C, for example, some of the character escape codes, as illustrated in Table 2.8; others are simply normal ASCII's found on the keyboard, but included as escape characters because of the C syntax.

Parity bit. Notice that the ASCII code uses seven bits, so bit 7 in a byte becomes a don't care as far as the interpretation concerns. Thus, both 35h and B5h can be

Table 2.7 Meaning of control functions in ASCII code

NUL:	Null	SOH:	Start of heading
STX:	Start of text	ETX:	End of text
EOT:	End of transmission	ENQ:	Enquiry
ACK:	Acknowledge	BEL:	Bell
BS:	Backspace	HT:	Horizontal tab
LF:	Line feed	VT:	Vertical tab
FF:	Form feed	CR:	Carriage return
SO:	Shift out	SI:	Shift in
SP:	Space	DLE:	Data link escape
DC1:	Device control 1	DC2:	Device control 2
DC3:	Device control	DC4:	Device control 4
NAK:	Negative acknowledge	SYN:	Synchronous idle
ETB:	End of transmission block	CAN:	Cancel
EM:	End of medium	SUB:	Substitute
ESC:	Escape	FS:	File separator
GS:	Group separator	RS:	Record separator
US:	Unit separator	DEL:	Delete

Table 2.8 ASCII values of character escapes in C

Constant	Meaning	Hex	Constant	Meaning	Hex
‘\a’	BEL (Bell, “alert”)	07	‘\r’	CR (carriage return)	0D
‘\b’	BS (backspace)	08	‘\”’	double quote	22
‘\t’	HT (horizontal tab)	09	‘\”	single quote	27
‘\n’	LF (line Feed, “new line”)	0A	‘\?’	question mark	3F
‘\v’	VT (vertical tab)	0B	‘\\’	backslash	5C
‘\f’	FF (form feed)	0C			

used for character 5. ASCII was devised initially as a transmission code. Since the information can be corrupted, there is the need to devise ways to verify that the correct signal has been sent. One of these is the use of a *parity bit*, which is added to the ASCII code to form a byte. A word has *even parity* if the number of 1’s is even, and *odd parity* otherwise. Thus 41H$\rightarrow$ 0010 0001 is an even parity code for letter A, while A1H$\rightarrow$ 1010 0001 is an odd parity code. for the same letter.

2.11.2 Unicode

While English is the most important technical language nowadays, and undoubtely enough for many embedded applications, it is not the only one used in communications. Hence, the need for more codes. *Unicode*[8] is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The development of this standard is coordinated by the Unicode Consortium, whose official website is http://www.unicode.org. The origins of Unicode date back to 1987 and the development has continued since then without interruption. Nowadays, the Unicode Standard is the universal character encoding standard for written characters and text. It defines a consistent way of encoding multilingual text that enables the exchange of text data internationally and creates the foundation for global software.

Unicode characters are represented in one of three encoding forms: a 32-bit form (UTF- 32), a 16-bit form (UTF-16), and an 8-bit form (UTF-8). The 8-bit, byte-oriented form, UTF-8, has been designed for ease of use with existing ASCII-based systems.

The Unicode Standard contains 1,114,112 code points, most of which are available for encoding of characters. The majority of the common characters used in the major languages of the world are encoded in the first 65,536 code points, also known as the Basic Multilingual Plane (BMP). The overall capacity for more than 1 million characters is more than sufficient for all known character encoding requirements, including full coverage of all minority and historic scripts of the world.

The latest version, in 2011, is Unicode 6.0, and contains more than 109,384 characters and symbols, with ASCII as a subset code. These characters are more than sufficient not only for modern communication for the world languages, but also to represent the classical forms of many languages. The standard includes the European alphabetic scripts, Middle Eastern right-to-left scripts, and scripts of Asia and Africa. Many archaic and historic scripts are encoded. The Han script includes 74,616 ideographic characters defined by national, international, and industry standards of China, Japan, Korea, Taiwan, Vietnam, and Singapore. In addition, the Unicode Standard contains many important symbol sets, including currency symbols, punctuation marks, mathematical symbols, technical symbols, geometric shapes, dingbats, and emoji.

The reader is highly encouraged to visit the Unicode website to learn more about this standard.

2.11.3 Arbitrary Custom Codes

Users can create their own coding for specific applications. Also, reconfigurable or programmable hardware imposes special encodings for applications based on bit interpretations. In most situations, bits are interpreted either individually or by groups.

We illustrate this last comment using the Supply Voltage Supervisor (SVS) control register found in the MSP430 microcontrollers with the exception of the first generation family '3xx. The bit distribution is shown in Fig. 2.9.

[8] This subsection is based on http://www.unicode.org/versions/Unicode6.0.0/ch01.pdf

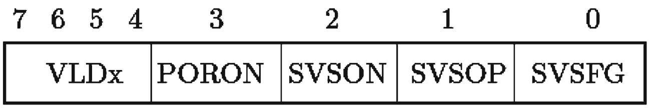

Fig. 2.9 SVS control register of the MSP430x1xx family (Courtesy of Texas Instruments Inc.)

The interpretation of bits is done working with the most significant nibble as a group, and the other bits individually as follows:

Bits 7-4 VLDx, Voltage level detect: They turn on the SVS and select the nominal SVS threshold voltage level:

0000 :	$SVSisoff$	1000 :	2.8 V
0001 :	1.9 V	1001 :	2.9 V
0010 :	2.1 V	1010 :	3.05 V
0011 :	2.2 V	1011 :	3.2 V
0100 :	2.3 V	1100 :	3.35 V
0101 :	2.4 V	1101 :	3.5 V
0110 :	2.5 V	1110 :	3.7 V
0111 :	2.65 V	1111 :	$ComparesexternalinputvoltageSVSINto1.2\,V.$

Bit 3 PORON: This bit enables the SVSFG flag to cause a POR device reset: 0 SVSFG does not cause a POR; 1 SVSFG causes a POR
Bit 2 SVSON, SVS on: This bit reflects the status of SVS operation, it DOES NOT turn on the SVS. (The SVS is turned on by setting VLDx > 0) This bit is 0 when the SVS is Off and 1 when the SVS is On
Bit 1 SVSOP, SVS output: This bit reflects the output value of the SVS comparator. Its value is 0 when SVS comparator output is low, and 1 when it is high.
Bit 0 SVSFG, SVS flag: This bit indicates a low voltage condition. SVSFG remains set (1) after a low voltage condition occurs until reset (0) by software or a brownout reset.

The above interpretation is defined from hardware design, and used to configure the peripheral. Hence, to select a 3.5 V threshold voltage and enable a power-on-reset we must force the word 11011xxx in the register. Notice that the three least significant bits are mostly read-only.

2.12 Summary

- A bit is a 0 or a 1. A sequence of bits is a word. A nibble is a 4-bit word, a byte is an 8-bit word, a double word is a 32-bit word, and a quad a 64-bit word. The term word is also used to denote a 16–bit word.

- Unsigned integers are most commonly represented using the normal positional binary system.
- The BCD system is a representation where each integer is represented separately by a four digit normal binary equivalent.
- With n bits, the range of unsigned integers that can be covered is between 0 and $2^n - 1$.
- The hexadecimal system has the characteristic of a one to one relationship between each hex digit and four bits. This property is used to introduce the hex notation.
- Signed integers are normally represented with the 2's complement notation. With n bits the covered range is from -2^{n-1} to $2^{n-1} - 1$.
- Overflow occurs when the result of an operation falls outside the range covered by the representation.
- Real numbers are coded by either the fixed point format or the floating point format. The Fm.n format for fixed point format codifies the real number with the most significant m bits for the integer part, and the n least significant bits for the fractional part, in steps of 2^{-n}.
- Floating point formats are single–precision, with 32 bits, and double–precision, with 64 bits. Floating point is preferred for very large and very small data.
- The ASCII code is the most popular for alphanumeric encoding. It does not include encoding for non-English letters. Unicode and other codes are used for most general cases.
- Continuous intervals are discretized by assigning words to intervals. The number of bits in the words determine the resolution of the codification.
- Non conventional codes are needed for specific applications or determined by the hardware of the system.

2.13 Problems

2.1 Express the following numbers as powers of 2:

a. 256
b. 64K
c. 32M
d. 512
e. 4G

2.2 Express the following powers of 2 in terms of K, M G:

a. 2^{14}
b. 2^{16}
c. 2^{24}
d. 2^{32}
e. 2^{20}

2.3 Convert the following binary (base 2) integers to decimal (base 10) equivalents.

a. 010000B
b. 11111B
c. 011101B
d. 11111111B
e. 10111111B
f. 10000000B

2.4 Convert the following decimal (base 10) numbers into their equivalent binary (base 2) and hexadecimal (base 16) numbers.

a. 10
b. 32
c. 40
d. 64
e. 156
f. 244

2.5 Convert the following binary (base 2) numbers to hexadecimal (base 16) numbers.

a. 10B
b. 101B
c. 1011B
d. 1010111B
e. 11101010B
f. 10000000B
g. 1011010110101110B

2.6 Convert the following unsigned hexadecimal (base 16) numbers into their equivalent decimal (base 10) and binary (base 2) numbers.

a. 1Ch
b. 7ABCDh
c. 1234h
d. 1FD53h
e. 9D23Ah
f. 0A1B2Ch

2.7 Construct a table for the equivalence of powers of 2, from 2^1 to 2^{24} in hexadecimal numbers.
2.8 Starting with the fact that r^n is represented in base r as a 1 followed by n zeros, demonstrate that the decimal equivalent for the binary number consisting of n 1's is $2^n - 1$.
2.9 Make the following conversions:

a. 10011.101B to decimal, hexadecimal, and octal
b. 2A1F.Bh to binary, octal, and decimal
c. 275.32Q to binary, hexadecimal, and decimal
d. 1327.76 to binary (up to 6 binary fractional digits).

2.10 The following words using bits have no special meaning. Express the words in hexadecimal and octal notations.

a. 101011011010
b. 1011011001010
c. 01101011011

2.11 Construct a table for example 2.7 on page 47 showing the different contents of register P1Out to display numbers 0 to 9, with and without dot.

2.12 Perform the following decimal additions and subtractions using the binary system.

a. $381 + 214$
b. $169 + 245$
c. $279 + 37$
d. $130 - 78$
e. $897 - 358$

2.13 Repeat the previous exercise with the hexadecimal system.

2.14 Express the 2's complement of 1 in each of the following formats

a. 4-bit binary (base 2) and hexadecimal (base 16) number
b. 8-bit binary (base 2) and hexadecimal (base 16) number
c. 16-bit binary (base 2) and hexadecimal (base 16) number

2.15 Find the two's complements of the following numbers with the specified number of bits.

a. 79 with 8 and 10 bits
b. 196 with 12 bits
c. 658 with 16 bits
d. 102 with 8, 10 and 16 bits.

2.16 Repeat the previous example using hex notation from the beginning. Notice that not all cases correspond to true hex representations.

2.17 Perform the following subtractions, both directly and with addition of complements, using binary and hex notation. For the case of negative results, take the complement of the result to interpret the subtraction. Notice in each case the difference between the borrow in the direct subtraction and the carry in the algorithm using two's complement addition.

a. $198 - 87$, with eight bits.
b. $56 - 115$, with eight bits.
c. $38496 - 6175$, with sixteen bits.
d. $1904 - 876$, with sixteen bits.
e. $2659 - 14318$, with sixteen bits.

2.18 Encode the following decimal numbers in (a) BCD and in (b) 127 biased binary code.

a. 10
b. 28
c. 40
d. 64
e. 156
f. 244

2.19 What is the minimum number of bits required to represent all integers in the closed interval [180, 307]? What is the bias needed to represent those integers in the interval using the minimum number of bits?

2.20 Encode the following negative numbers using 2's complement representation in the binary and hexadecimal number systems using 8 and 16 bits.

a. -12
b. -68
c. -128

2.21 Find the corresponding decimal numbers for each of the following words encoded as 2's complement signed numbers.

a. 1000b
b. 1111b
c. 11000110b
d. 10111110b
e. 01010101b
f. A2h
g. 7Ch
h. 43h
i. BEh
j. 62AFh
k. CCCCh
l. 3333h

2.22 Demonstrate the sign extension principle. Namely, show that if an n-bit word is converted into an $(n + h)$-bit word by appending the h bits to the left equal to the original sign bit, the signed decimal equivalent remains the same. (*Hint: use expansion* (2.8) *with the extended word, and reduce it to the equivalent of the n-bit one*)

2.23 Code the following decimal signed numbers in the best approximation using fixed point formats F4.5 and F4.7. Find the absolute and relative errors in each case when the result is interpreted again as decimal equivalent.

a. -5.125
b. 7.312
c. -0.772

d. 6.219
e. −3.856
f. 2.687

2.24 One disadvantage of floating point representations is that the steps are not uniform, but depend on the range in which the number is. To see this, find the decimal equivalent of the two consecutive numbers in single precision, hex notation, 41A34000 and 41A34001. Do the same thing for 4DA34000 and 4DA34001. How do the steps in consecutive numbers compare?

2.25 Using the ASCII code find the binary streams represented by following character strings.

a. abcd
b. ABCD
c. 123aB
d. ASCII
e. Microprocesors and Microcontrollers
f. YoUr nAMe

2.26 Convert the following binary streams into their equivalent character strings using the ASCII code.

a. "How Dy"
b. "university"
c. "012abCD"
d. "blank space"

2.27 In order for a computer or host to be able to connect to the Internet it must have a "logical address" known as "IP address". IP addresses are expressed using the dotted decimal notation where the decimal equivalent of each byte is separated from the following using a dot (.). Thus, when using the IPv4 version of the Internet Protocol (IP) a 32-bit IP address is expressed as four (4) decimal numbers separated by dots. For example, IP address 127.0.0.1, known as the loopback or local host IP address, would be expressed in binary as 01111111.00000000.00000000.00000001. Express each of the following IPv4 IP addresses as four (4) binary numbers separated by dots and viceversa.

a. 240.0.0.1
b. 10.240.68.11
c. 192.168.40.163
d. 11111111.11111111.11110000.00000000
e. 00001010.00000000.00000000.00000001
f. 10101100.10110010.00001111.11111111

2.28 Convert the following single-precision numbers to decimal.

a. AB1C3204
b. 01798642

2.29 Obtain the single-precision floating point representation for the following numbers:

a. 641.13
b. −1253.462

2.30 Find the dynamic range (smallest and largest positive and negative numbers) that can be expressed using 32-bits using with a floating point notation and with an integer notation.

2.31 A voltage takes values between 0 and 3 V. Conversion is done with 4-bit resolution. Specify the intervals and assignments.