1

Representation of Numbers

CONCEPTS COVERED IN THIS CHAPTER.–

This first part, devoted to the digital representation of information, primarily addresses numbers. Starting with numbers is appropriate because computers were originally designed for calculation purposes.

Since computers operate using only 0s and 1s, it is essential to understand the binary system and the conversions between the decimal and the binary systems, as well as vice versa. In addition, two other systems commonly used in computing, the octal system and the hexadecimal system, are discussed.

The approach to the machine representation of numbers involves working within a fixed range, as computers cannot recognize infinity. This necessitates methods for representing negative numbers and rational numbers. The common solutions used for these cases are described.

References: [STA 07, VEL 19, LIP 83].

Using a computer assumes that the information transmitted to it is in a format or language understandable by its components. Transforming data from a human-readable format into a format understandable by the computing machine is called "information coding".

Modern computers work exclusively with the binary system, which uses a reduced alphabet of $\{0, 1\}$. This raises the question of how to represent of common types of information (such as numbers, texts, images, sounds and videos), in binary. The ellipsis suggests that in the future, it might also be possible to represent other sensory experiences, such as smells and tastes, in binary form.

Mathematics for Digital Science 2,
by Gérard-Michel COCHARD and Mhand HIFI.

The study will begin with the coding of numbers, as computers were originally designed for computation. The following chapter will then cover the coding of other types of information, including texts, images, sounds and videos.

1.1. Representation of numbers

The decimal system is commonly used for manipulating numbers today, but historically, some civilizations used sexagesimal numbering systems. This approach is still evident in the measurement of time and angles: 1 hour = 60 minutes; 1 minute = 60 seconds, and angles are measured such that 1 flat angle = 180° = 3×60°.

Roman numerals were not very suitable for calculation, serving primarily for counting. The decimal system, which is quite ancient (originating in Egypt in the 3rd millennium BCE and later refined with the invention of 0), is based on the 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and uses positional numbering. As a result, the number 1234 is distinct from the number 3124, despite using the same digits, because the position of each digit is important for determining the number's value.

1.1.1. Position numbering

EXAMPLE 1.1.– Consider the number 1234. This number can be understood as follows: 1×1,000 + 2×100 + 3×10 + 4, or 1234 = 1×103 + 2×102 + 3×101 + 4×100.

In the decimal system, which as a base of 10 (with 10 digits), the position of each digit in a number corresponds to a power of 10.

In the binary system, which uses only the digits 0 and 1 and has a base of 2, a decimal number like 1234 is written as: 10011010010. This binary representation is understood as follows:

$10011010010 = 1\times2^{10} + 0\times2^{9} + 0\times2^{8} + 1\times2^{7} + 1\times2^{6} + 0\times2^{5} + 1\times2^{4} + 0\times2^{3} + 0\times2^{2} + 1\times2^{1} + 0\times2^{0}$.

Just like in the decimal system, the position numbering is used in the binary system. In the binary system, the base is 2, and the digits are 0 and 1, which are referred to as bits in computing. The binary can be verified to correspond to the number 1234 (from Example 1.1) by using powers of 2 (see Figure 1.1).

It should be noted that $2^{10} = 1{,}024$ is referred to as a kilobyte (KB) in computing, commonly abbreviated as "K". In this context, 1,024 bytes are equivalent to 1 KB.

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Figure 1.1. *Powers of 2*

Unlike human thinking, computers cannot process infinite numbers and are limited in the range of numbers they can manipulate. To represent an *n*-bit number, the largest integer that can be represented is $2^n - 1$. To represent the largest integer with m decimal digits in the binary system, which is $10^m - 1$, how many n bits are needed to represent this number in a computer?

To represent the largest integer with m decimal digits, the $10^m - 1 = 2^n - 1$ must be satisfied (assuming this is possible). Taking logarithms gives:

$$m \times \log_{10}(10) = n \times \log_{10}(2).$$

Since $\log_{10}(10) = 1$, this simplifies to

$$m = n \times \log_{10}(2)$$

Solving for n:

$$\mathrm{n} = \frac{m}{\log_{10}(2)} = \mathrm{m} \times \frac{\ln(10)}{\ln(2)} \approx 3.32 \times \mathrm{m}$$

It therefore takes approximately $(3.32 \times m)$ bits to represent an integer of m decimal digits in the binary system.

1.1.2. *The binary system*

As noted previously, the binary system is based on two symbols (bits): 0 and 1. Unlike the decimal system, operations in binary are extremely simple. Some may recall learning decimal multiplication tables in elementary school!

This is because, in the binary system, basic operations such as addition and multiplication are much more direct and less complex compared to the decimal system. Binary addition is performed bit by bit (see column 1 of Figure 1.2), where

$0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1$ (without carry) and $1 + 1 = 10$ (with carry). Binary multiplication follows simple rules (see column 2 of Figure 1.2), where $0 \times 0 = 0$, $0 \times 1 = 0$, $1 \times 0 = 0$ and $1 \times 1 = 1$.

addition	multiplication
0 + 0 = 0	0 x 0 = 0
0 + 1 = 1	0 x 1 = 0
1 + 0 = 1	1 x 0 = 0
1 + 1 = 10	1 x 1 = 1

Figure 1.2. *Addition and multiplication in binary*

This simplicity in binary operations is one of the reasons why computers use this system for internal calculations. This allows for efficient data manipulation and rapid calculations at the electronic circuit level.

The primary problem to solve is converting between decimal and binary systems. To convert a decimal number into a binary number, a straightforward method based on powers of 2 is used.

Consider the example of covering the decimal number 1234 to binary. Start by finding the largest power of 2 less than or equal to 1234, which is 1,024 = 2^{10}. Subtract 1,024 from 1234 to get 210. Next, find the largest power of 2 closest to 210, which is 128 = 2^7. Subtract 128 from 210 to get 82. Continue this process by finding the closest power of 2 to each resulting value and performing the corresponding subtractions. The sequence of operations is shown in Figure 1.3.

1234	
1024	2^{10}
210	
128	2^7
82	
64	2^6
18	
16	2^4
2	
2	2^1
0	

Figure 1.3. *Decimal-to-binary conversion*

The powers of 2 present in this decomposition are 2^1, 2^4, 2^6, 2^7 and 2^{10} and the missing powers of 2 are 2^0, 2^2, 2^3, 2^5, 2^8 and 2^9. To write the binary number, use the position numbering rule: place a 1 for each present power of 2 and a 0 for each absent power of 2. Thus, the binary representation is:

$$(1\times2^{10}) + (0\times2^9) + (0\times2^8) + (1\times2^7) + (1\times2^6) + (0\times2^5) + (1\times2^4) + (0\times2^3) + (0\times2^2) + (1\times2^1) + (0\times2^0) = (10011010010)_2$$

When necessary, the subscript 2 indicates a number in the binary system, and the subscript 10 indicates a number in the decimal system.

Now let us move on to the reverse problem: converting a binary number into a decimal number. It is actually simpler because you just write it down with the position numbering rule and do the math. Therefore, to convert this binary number to decimal, each bit of the binary number is multiplied by 2 raised to the power of its position, starting from zero at the far right.

$$(1001010010)_2 = (1 \times 2^{10}) + (0 \times 2^9) + (0 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 1024 + 128 + 64 + 16 + 2 = (1234)_{10}$$

Now, consider how to represent a rational number in base 2. Take the example of the decimal number $(14.80078125)_{10}$. To express this in base 2 follow this rule for a binary number of the form $(0.xyzt\ \ldots)_2$ where $x, y, z, t, \ldots$ are bits:

$$0 + x \times 2^{-1} + y \times 2^{-2} + z \times 2^{-3} + t \times 2^{-4} + \cdots$$

This method can be used to convert a decimal number with fractions into base 2. Applying this method will illustrate the conversion process for the example provided.

EXAMPLE 1.2.– Consider the decimal number 14.80078125 in base 10. First, convert the integer part $(14)_{10}$ into binary, which is $(1110)_2$. For the decimal part $(0.80078125)_{10}$, perform successive multiplications by 2,while recording the integer part of the result at each step:

$2 \times 0.80078125 = 1.6015625$ →1 (integer part: 1)
$2 \times 0.6015625 = 1.203125$ →1 (integer part: 1)
$2 \times 0.203125 = 0.40625$ →0 (integer part: 0)
$2 \times 0.40625 = 0.8125$ →0 (integer part: 0)
$2 \times 0.8125 = 1.625$ →1 (integer part: 1)

$2 \times 0.625 = 1.25$ $\rightarrow$1 (integer part: 1)
$2 \times 0.25 = 0.5$ $\rightarrow$0 (integer part: 0)
$2 \times 0.5 = 1$ $\rightarrow$1 (integer part: 1)

The decimal part can be expressed as a sum of fractions:

$$(0.80078125)_{10} = \frac{1}{2} + \frac{1}{4} + \frac{1}{32} + \frac{1}{64} + \frac{1}{256}$$

This corresponds to the binary representation $(0.11001101)_2$.

Combining this with the integer part $(14)_{10} = (1110)_2$, the full binary representation of $(14.80078125)_{10}$ is:

$$(14.80078125)_{10} = (1110.11001101)_2$$

The calculation performed in the previous example will be useful later. By converting the decimal number $(14.80078125)_{10}$ to binary, the complete binary representation is $(1110.11001101)_2$. This conversion illustrates the general method for converting decimal numbers to binary numbers using the position numbering rule. This method will be explored in more detail later.

1.1.3. *Octal system and hexadecimal system*

In the octal system , the base is 8, so there are eight digits available: 0, 1, 2, 3, 4, 5, 6 and 7. Converting a decimal number to octal follows the same principle as converting to binary. Using powers of 8, the number $(1234)_{10}$ can be represented in octal base. Given that

$$8^0 = 1, \quad 8^1 = 8, \quad 8^2 = 64, \quad 8^3 = 512, \quad 8^4 = 4096,$$

Break down $(1234)_{10}$ as follows:

$$1234 = 2 \times 512 + 3 \times 64 + 2 \times 8 + 2 \times 1$$

Converting to octal:

$$1234 = 2 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 2 \times 8^0 = (2322)_8$$

The conversion process involves starting with the binary representation of the number, grouping the into packets of 3 from the right, and then using the table in Figure 1.4 to convert each group to octal. For example:

$$(1234)_{10} = (10011010010)_2$$

Group the binary digits:

$$(10011010010)_2 = (010\ 011\ 010\ 010)_2$$

Convert each group to octal:

$$010_2 = 2_8, \quad 011_2 = 3_8, \quad 010_2 = 2_8, \quad 010_2 = 2_8$$

Thus,

$$(1234)_{10} = (2322)_8$$

binary	000	001	010	011	100	101	110	111
octal	0	1	2	3	4	5	6	7

Figure 1.4. *Binary-to-octal conversion*

This conversion method can also be applied for other bases, such as the hexadecimal system (base 16), using the corresponding powers.

In the hexadecimal system, the base is 16 and the 16 digits used are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The letters A to F represent the values 10 to 15, respectively, and are considered as digits.

To convert a decimal number to hexadecimal base, use powers of 16. By applying the corresponding powers of 16 $(16^0, 16^1, 16^2, 16^3, \ldots)$, the decimal number $(1234)_{10}$ can be decomposed into its hexadecimal representation:

– Find the largest power of 16 less than 1234:

$$16^2 = 256, \quad 16^1 = 16, \quad 16^0 = 1$$

– Decompose 1234 (and the successive values) using these powers:

$$1234 = 4 \times 256 + 210; \quad 210 = 13 \times 16 + 2$$

Here, 13 in hexadecimal is represented as D.

– Combine these results:

$$1234 = 4 \times 16^2 + 13 \times 16^1 + 2 \times 16^0$$

Thus,

$$(1234)_{10} = (4D2)_{16}$$

The conversion process can also be done using the table in Figure 1.5. First, convert the number to binary. By grouping the binary digits into groups of four from the right, the representation of $(1234)_{10}$ in binary is:

$$(1234)_{10} = (10011010010)_2$$

To ensure each group has four digits, add leading zeros if necessary:

$$(1234)_{10} = (10011010010)_2 = (\mathbf{0}100\ 1101\ 0010)_2 = (4D2)_{16}$$

binary	0000	0001	0010	0011	0100	0101	0110	0111
hexadecimal	0	1	2	3	4	5	6	7

binary	1000	1001	1010	1011	1100	1101	1110	1111
hexadecimal	8	9	A	B	C	D	E	F

Figure 1.5. *Binary-to-hexadecimal conversion*

1.2. Representation of numbers in a machine

So far, the discussion has focused mainly on positive integers in different number systems. However, it is also important to consider negative numbers and rational numbers (fractions) in this context.

1.2.1. *Machine representation of negative numbers*

Several approaches can be considered:

1) Sign-magnitude representation: a sign bit (0 or 1) is used to indicate whether a number is positive (0 in front) or negative (1 in front). For example,

– $(-1234)_{10}$ can be represented as $(\mathbf{1}10011010010)_2$ in sign-magnitude notation, where "**1**" denotes the negative sign and "10011010010" represents the magnitude.

– In this representation, a number of n bits has n-1 bits dedicated to the magnitude.

A drawback of this approach is the representation of "0", which can be ambiguous: $(0)_{10} \rightarrow$(**0** 0) and $(0)_{10} \rightarrow$(**1** 0). It is important to distinguish between the binary representation of numbers and their machine representation, which includes bit ($\rightarrow$).

2) One's complement representation: in one's *complement* notation, positive numbers are represented in their usual binary form (1 or 0) with a leading sign bit of "0". For negative numbers, the bitwise complement (inversion of all bits) of the positive number's binary representation is used, with a leading sign bit of "1". For example,

– $(+1234)_{10}$ in binary is $(+10011010010)_2$. In one's complement notation, this becomes $(\mathbf{0}10011010010)_2$ (with a sign bit).

– $(-1234)_{10}$ is represented by inverting the bits: $(101100101101)_2$

This method involves representing positive numbers in their normal binary form and converting negative numbers by flipping all bits of their positive counterparts.

The same difficulty with "0" arises, as it has two representations. Another issue is the imbalance between positive and negative numbers. For example, if a machine can represent $(1234)_{10}$, it is expected that it should also be able to represent $-(1234)_{10}$. In other words, the system should have the possibility to represent an equal number of positive and negative numbers.

In a machine, numbers are typically coded using n bits (often 32 or 64). For simplicity, consider a machine with 8-bit coding (7 bits for the absolute value). The largest positive number that can be represented is $2^7 - 1 = 127$. Its representation is therefore $(\mathbf{0}1111111)_2$.

In the one's complement method, -127 is represented by $(10000000)_2$ i.e. zero. This representation effectively indicates that there are not as many positive numbers as negative numbers in one's complement notation: positive numbers range from 0 to 127, while negative numbers range from -1 to -126. In Figure 1.6, negative numbers are obtained by taking the one's complement of positive numbers, which involves flipping all the bits. This results in a representation, where 1 denotes the sign bit for negative values.

One solution could be to use a representation where 0 is denoted by a number containing only zeros. However, this approach presents challenges for simple operations like addition. For example, consider $(127)_{10} - (127)_{10}$ in one's complement:

Representation: $(127)_{10}$ is represented as 01111111_2, and $-(127)_{10}$ is represented as 10000000_2 in one's complement.

Performing the addition:

$$01111111_2 \text{ (for } 127_{10}) + 10000000_2 \text{ (for } -127_{10}) = 11111111_2$$

In one's complement, 11111111_2 could be interpreted as -0. This result is problematic because it introduces an additional representation for zero and does not easily fit into a straightforward arithmetic interpretation.

decimal	representation 1
127	01111111
126	01111110
125	01111101
124	01111100
----	----
4	00000100
3	00000011
2	00000010
1	00000001
0	00000000
-1	11111110
-2	11111101
-3	11111100
-4	11111011
----	----
-124	10000011
-125	10000010
-126	10000001
-127	10000000

Figure 1.6. *Representation 1*

3) Two's *complement representation:* the two's *complement*, which involves adding 1 to the one's complement of a number. Let us illustrate this with the previous example and complete the table (see representation 2 in Figure 1.7).

For two's complement:

Representation: 127_{10} in binary is $\rightarrow 01111111_2$. To find -127_{10} in two's complement: first, get the one's complement of 127_{10}: 10000000_2; second, add 1 to this result: $10000000_2 + 1 = 10000001_2$.

Performing the subtraction:

$$127_{10} - 127_{10} = 01111111_2 - 10000001_2 = 00000000_2 = 0_{10}$$

The last carry on the left is ignored. With the two's complement method, it is possible to encode the number -128_{10} but not $+128_{10}$. In two's complement representation on n bits, the numbers can be encoded in the range from 0 to $2^{n-1} - 1$ and from –1 to -2^{n-1}

The two's complement representation allows additions to be performed "normally" by following the rules of binary addition.

Therefore, on 8 bits (where the left carry is always ignored), consider the following example.

decimal	representation 1	representation 2
127	01111111	01111111
126	01111110	01111110
125	01111101	01111101
124	01111100	01111100
----	----	----
4	00000100	00000100
3	00000011	00000011
2	00000010	00000010
1	00000001	00000001
0	00000000	00000000
-1	11111110	11111111
-2	11111101	11111110
-3	11111100	11111101
-4	11111011	11111100
----	----	----
-124	10000011	10000100
-125	10000010	10000011
-126	10000001	10000010
-127	10000000	10000001
-128		10000000

Figure 1.7. *Representation 2*

EXAMPLE 1.3.– Representation of the numbers $(60)_{10}$ and $(8)_{10}$:

$$(60)_{10} = (34)_{10} + (26)_{10} \rightarrow (00100010)_2 + (00011010)_2 = (00111100)_2 \rightarrow (00111100)_2 \rightarrow (60)_{10}$$

$$(8)_{10} = (34)_{10} - (26)_{10} \rightarrow (00100010)_2 + (11100110)_2 = (00001000)_2 \rightarrow (00001000)_2 \rightarrow (8)_{10}$$

Of course, the result of an operation on binary numbers must fit within the n bits used by the computer. When an arithmetic operation provides a result that exceeds the capacity of n bits, an overflow occurs.

To illustrate this, consider the following example:

EXAMPLE 1.4.– Let the number be $(280)_{10}$.

$$(280)_{10} = (120)_{10} + (160)_{10} \rightarrow (01111000)_2 + (10100000)_2 = (00011000)_2 \rightarrow (24)_{10}$$

The *result is incorrect* because $(280)_{10}$ cannot indeed be represented in 8 bits as $(00011000)_2$.

In general, if a number exceeds the representational range of a given number of bits, systems may produce an error or overflow message.

1.2.2. *Representation of rational numbers*

Rational numbers , also known as fractions, are numbers expressed as the ratio of two integers. They can be represented in several ways: as fractions, as decimal numbers with an integer part and a decimal part, or using scientific notation, which uses powers of 10. The focus here is on the latter representation, also known as *floating point* notation.

In scientific notation , a number is written as $s.m \times 10^p$, where s is the sign ($+$ or $-$), m is the mantissa (which is always positive), and p is the exponent (which can be positive or negative). For example, consider the rational number $x = \frac{22}{7}$, whose approximate decimal value is $3.1428\ldots$ (an approximation of π). The scientific notation for this number could be:

$$x = 3.1428 \approx +0.31428 \times 10^{1}$$

where $+$ is the sign, 0.31428 is the mantissa, and 1 is the exponent. Alternatively, this number can be represented as:

$$x = 3.1428 \approx +31{,}428 \times 10^{-6}$$

where + is the sign, $31{,}428$ is the mantissa, and -6 is the exponent.

As demonstrated, the mantissa can vary depending on the value of the exponent. To standardize the representation, it is common to impose the following condition for the mantissa: $0.1 \leq m < 1$. This means that the mantissa must be between 0.1 (inclusive) and 1 (exclusive). This rule ensures a standardized representation of numbers in scientific notation and facilitates calculations and comparisons between numbers. With this mantissa rule, the digit before the decimal point is the first significant digit in floating-point notation. This notation is useful for expressing very small numbers or very large numbers. For example, the mass of an electron is approximately 0.9×10^{-30} kg, and the mass of the Sun is approximately 0.2×10^{31} kg.

Using the mantissa rule , express the decimal part of the mantissa that is after the decimal point, which always starts with 0. For clarity, this part of mantissa is denoted as m', where $m' = 10m$. The range of values for m' is therefore $[1, 10[$.

Consider a representation where the mantissa is expressed using four digits and the exponent uses three digits. In this configuration, the range of positive numbers that can be represented is from 0.1000×10^{-999} to 0.9999×10^{999}. This range covers an extremely large number of values, yet each number can be represented with just four digits for the mantissa and three digits for the exponent, totaling seven digits including the exponent sign.

However, an exponent can be negative. In such cases, if the usual representation is maintained, the first digit will determine the sign of the exponent. If t bits are allocated to store the exponent, then $t - 1$ bits will be used for the absolute value of the exponent.

To handle negative exponent, a fixed value e, called the *excess* or bias , can be added to the actual exponent to ensure that the encoded exponent is always positive. For t bits reserved for the exponent*, the excess* e is given by 2^{t-1}. For instance, with $t = 8$ bits for the exponent, the excess $e = 2^7 - 1$. This allows the exponent to be represented in the form 0 to 254. Therefore, an exponent range of $[-127, +127]$ is

mapped to the encoded range $[0, 254]$. This method enables the representation of both negative and positive exponents using t bits.

To recap, a number in scientific notation (or floating-point representation) is written as:

$$s.m' \times 2^E$$

where s represents the sign ($+$ or $-$), m' is the mantissa satisfying $1 \leq m' < 10$ in $decimal$, e is the fixed excess or bias, and E is the encoded exponent, which is the true exponent plus the excess (E = real exponent + e).

This scientific notation can be translated into binary representation as follows (see Figure 1.8):

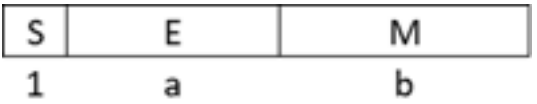

Figure 1.8. *Structure of a rational number*

– S on 1 bit represents the sign, with 0 indicating a positive number and 1 indicating a negative number (i.e. 0 for + and 1 for −).

– E on a bits represents the exponent. It is calculated using the rule $E = p + e$, where p is the real exponent of the number and e is a fixed excess (bias) chosen to ensure that E is always positive. The number of bits allocated to the exponent is a.

– M on b bits represents the mantissa, which contains the significant part of the number. The number of bits allocated to the mantissa is b.

By combining these elements, the complete binary representation of a rational number in a machine is obtained. The IEEE 754 standard specifies the following rules for the binary representation:

a) Formats: there are two formats, single precision and double precision.

b) Representation: a number is coded in the form $S.1, M' \times 2^c$. The bit 1 preceding the decimal point of the mantissa is not explicitly encoded (it is called the hidden bit); only the part $M'$of the mantissa is encoded. The exponent is encoded as $E = c + e$, where e is a fixed excess.

c) The single precision format: it has the following structure (see Figure 1.9). The exponent is encoded on 8 bits, with an excess of $e = (127)_{10} = (01111111)_2$.

The mantissa (excluding the hidden bit) M' is encoded on 23 bits. The total number of bits for encoding is 32 bits.

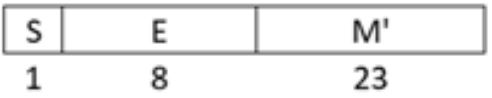

Figure 1.9. *IEEE 754 single precision*

d) The double precision format: it has the following structure (see Figure 1.10).

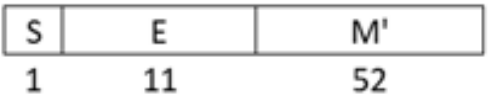

Figure 1.10. *IEEE 754 double precision*

The exponent is coded using 11 bits, with an excess of $e = (1023)_{10} = (0111111111)_2$; the mantissa (excluding the hidden bit) M' is encoded using 52 bits. The enter number is encoded using 64 bits.

Let us consider an example using the single precision format.

EXAMPLE 1.5.– To encode the number $(14.80078125)_{10}$ in single precision format, follow these steps:

– Determine the sign bit: since the number is positive, the sign bit S is 0.

– Convert the number to binary: the integer part $(14)_{10}$ converts to $(1110)_2$, and the decimal part $(0.80078125)_{10}$ converts to $(0.11001101)_2$ through binary fraction conversion.

– Combining these two parts, the binary representation is as follows:

$$(14.80078125)_{10} = (1110.11001101)_2$$

– Normalize the binary number: normalize (1110.11001101)2 to scientific notation in binary provides the following result:

$(1.11011001101)_2 \times 2^3$.

– Determine the exponent: to find the binary value of the exponent, first calculate the exponent E by adding the true exponent to the fixed excess e. For this example, the true exponent p is 3, the fixed excess e is 127. Hence, the encoded E is:

$$E = p + e = (00000011)_2 + (01111111)_2 = (10000010)_2 = 130.$$

– Determine the mantissa: the reduced mantissa M', omitting the hidden bit, is:

$M' = 11011001101000000000000_2$ (padded with zeros to fit 23 bits)

– Combine the components: the sign bit $S = 0$, the exponent $E = (10000010)_2$, and the mantissa $M' = 11011001101000000000000_2$. Therefore, the 32-bit single precision representation of $(14.80078125)_{10}$ is:

0 10000010 11011001101000000000000

This representation is the binary encoding of the number $(14.80078125)_{10}$ in single precision format according to the IEEE 754 standard.