# FLOATING POINT REPRESENTATIONS

It is assumed that the student is familiar with the discussion in Appendix B of the text by A. Tanenbaum, *Structured Computer Organization,* 4th Ed., Prentice-Hall, 1999. Here we are concerned in particular with the discussion of floating-point numbers and normalization in pp. 643–651.

In the notation used here floating point representations for real numbers have the form:

$$\pm M \times R^{\pm E} \ ,$$

where M is a signed *mantissa,* R is the *radix of normalization,* and E is a signed *exponent.* It is the signed values M and E that are typically packed into computer "words" of varying length to encode single, double, and extended precision (short, long, and extended) floating point numbers. The number of bits used in the representation of M determines the *precision* of the representation. The number of bits used to represent the signed value E determines the *range* of the representation. Only a finite number of values among those in the representable range can be encoded in the $n$ bits available, and this number is determined by the precision.

To facilitate fixed-point comparisons of floating-point data representations such as might occur in sorting applications three things are usually done. The first is to place the sign of the mantissa M denoted $S_M$ on the left in the same location as in the sign representation for fixed point numbers. This provides easy detection of the sign of the floating-point data in the same manner as for fixed-point data. The second thing is to place the signed exponent representation between $S_M$ and M just to the left of M as packed in the high order word, and the third thing is to bias the k-bit representation of the signed exponent so as to shift the represented range. In a k-bit field the range of signed exponents is:

$$-(2^{k-1}-1) \leq E \leq +(2^{k-1}-1).$$

One of the more common ways of biasing the exponent (but not the only way as we shall see in the examples) is to add to the signed exponent a bias of $2^{k-1}$ This bias is appropriately subtracted in unpacking for output conversion and other arithmetic operations. The biased exponent is called the *characteristic* ($E_b$) of the representation. Biasing the number in this way alleviates the need to examine the sign of the exponent explicitly in making fixed point comparisons such as in sorting. The range of the characteristic (or biased exponent) is then:

$$-2^{k-1}+2^{k-1}+1 \leq E+2^{k-1} \leq 2^{k-1}+2^{k-1}-1 \ \ ,$$

which is equivalent to

$$1 \leq E_b \leq 2^k - 1.$$

The term usually used is that the characteristic represents the exponent *excess* $2^{k-1}$. Some computer manufacturers do not use a symmetric range and permit one more negative exponent in the representation of the characteristic by allowing a zero characteristic to be a valid representation.

A floating point number packed into a single word would then have the following form:

| $S_M$ | $CHARACTERISTIC$ | $MANTISSA$ |
|---|---|---|
| 1 | $k$ | $j$ |

The purpose of normalization is to preserve as many significant digits in the representation as possible in the $j$ bits available to represent the normalized mantissa. The greatest number of significant bits are preserved ($j$ of them) when the radix of normalization is binary ($R = 2$). When $R = 16$, the most significant non-zero hexadecimal digit in the normalized mantissa may have as many as three leading zeros in the $j$-bit binary representation in a binary computer.

Some representations assume a normalized fraction, in which case the radix point is assumed to lie at the boundary between the characteristic and the mantissa. Other representations assume the mantissa is

normalized as an integer (e.g., the CDC 6600, CYBER 70 and successor series machines) so that the radix point is assumed to lie immediately to the right of the mantissa.

In the case of binary normalization some representations assume the number to be normalized only if the most significant binary digit is a one. If this digit is always a one, there is no uncertainty and therefore no need to devote a bit in the $j$ bits available to represent this digit; it can thus be represented implicitly as a *hidden-bit* so that the remaining lower order j binary digits (in which there is some uncertainty from number to number) can be represented. By using a hidden bit the representation permits a precision of $j+1$ bits for the mantissa, but at the expense of possibly giving up an explicit representation for zero. For example, hidden bits are used in the DEC PDP-11 series floating point number representations and in the IEEE Standard Floating Point representations. In other representations zero is usually handled as a special case by letting a word with all zeros represent the value zero (or equivalently, in one's complement machines letting a vector of all one's represent minus zero).

The portion of the real line represented by floating point formats appears as follows:

| *Negative Overflow Region* | *Expressible Negative Numbers* | *Negative Underflow Region* | *Positive Underflow Region* | *Expressible Positive Numbers* | *Positive Overflow Region* |
|---|---|---|---|---|---|

$-\infty \leftarrow$ *Zero* $\rightarrow +\infty$

Using an $n$-bit word to represent a single precision floating-point word, we note that there are exactly $2^n$ values in the sets of Positive and Negative Expressible Numbers that can be represented (including zero).

Positive and negative floating point numbers are usually represented in one of two ways depending on the computer's choice of arithmetic circuitry and representation for fixed point numbers. Machines that use one's complement representations for fixed point numbers and have one's complement adders in their arithmetic units typically pack the positive floating point number into one (or more) word(s) and then complement all bits in this (these) word(s) to represent the negative floating point number. Examples of these one's complement representations include the UNIVAC 1100 series machines and the CDC 6600, CYBER 70 series and successor machines.

Machines that represent fixed point numbers in two's complement form and have two's complement adders in their arithmetic units typically use a sign magnitude representation for positive and negative floating point numbers with $S_M = 0$ for positive numbers and $S_M = 1$ for negative numbers. The negative mantissas are typically converted to their two's complement representation(s) when the floating point number is unpacked for floating point arithmetic operations; the result's mantissa is then converted back to a sign-magnitude format when it is repacked into the floating point representation after an arithmetic operation or upon input conversion. Machines that use this sign magnitude representation for floating point numbers include the IBM 360/370 and compatible instruction set architectures, the DEC PDP-11 series and upward compatible machines, the Cray-1 and successors and machines that use the IEEE Floating Point Standard representation.

We now consider specific examples to illustrate the concepts. Floating point formats used on machines of various manufacturers are considered; namely, the UNIVAC 1100 series, the IBM 360/370 series, the DEC PDP-11/VAX-11 series, the CDC 6600/CYBER 70 series, and the IEEE Floating Point Standard series such as the Intel 8087 series numeric data processors.For each machine format considered we shall present floating-point representations of the numbers $+29.2_{10}$ and $-29.2_{10}$ , and of the numbers $+0.03125_{10} and -0.03125_{10}$ . Recall that

$$29.2_{10} = 35.\overline{1463}_8 = 1D.\overline{3}_{16} = 11101.\overline{0011}_2 \ .$$

Furthermore, recall that

$$0.03125_{10} = \left(\frac{1}{32}\right)_{10} = 0.00001_2 = 0.02_8 = 0.08_{16} \ .$$

**UNIVAC 1100**

The UNIVAC 1100 series machines use 36-bit words to represent instructions and fixed point data and a one's complement representation for negative numbers with one's complement arithmetic circuitry. A single precision floating point datum is packed into a single 36-bit word, and a double precision floating point datum is packed into two consecutive 36-bit words with the second 36-bit word representing a continuation of the low order bits in the mantissa. The mantissa is a binary normalized fraction of the form $0.1xxxxxx_2$ or is either all zeros or all ones. A single precision datum has an 8-bit characteristic that represents a signed exponent with bias (excess) $128_{10} = 200_8$, and a double precision datum has an 11-bit characteristic that represents the signed exponent excess $1024_{10} = 2000_8$. Negative floating-point numbers are represented by packing the positive representation into the single (or double) word(s) and then taking the one's complement of this (these) word(s) by logically complementing each of the 36 (72) bit positions (including the characteristic field). A word(s) with all zeros represents floating-point +0, and a word(s) with all ones represents floating-point −0.

These two formats appear as follows:

*Single Precision:*

| 35 | 34 27 | 26 0 |
|---|---|---|
| $S_M$ | $CHARACTERISTIC$ | $MANTISSA$ |
| 1 | 8 | 27 |

*Double Precision:*

| 35 | 34 *word*1 24 | 23 0 , 35 *word*2 0 |
|---|---|---|
| $S_M$ | $CHARACTERISTIC$ | $MANTISSA$ |
| 1 | 11 | 60 |

*UNIVAC 1100 examples:* After first performing binary normalization on the binary representation of the number, computing the biased exponent and packing the $S_M$, CHARACTERISTIC, and MANTISSA fields of the word(s) in the corresponding floating point format, we represent the resulting 36-bit (or 72-bit) floating-point word(s) in octal shorthand with 12 octal digits for each word. The representation for negative values is then formed by taking the seven's complement of the octal shorthand representation for the positive number. A positive representation can be detected in the octal shorthand by noting that the first (*i.e.,* leftmost) octal digit is 0, or 1, or 2, or 3; each of these when converted to binary has a leftmost binary digit of zero, which corresponds to the bit in the $S_M$ position. Octal shorthand representations of negative floating-point numbers, therefore, have first (*i.e.,* leftmost) octal digit of 7, or 6, or 5, or 4; each of these when converted to binary has a leftmost binary digit of one, which corresponds to the bit in the $S_M$ position. We consider the single-precision case in detail first, and then present the double-precision result.

We first consider representations for $\pm 29.2_{10}$ .

(1) Normalize:

$$+29.2_{10} = +11101.\overline{0011}_2 = +0.\underbrace{111\ 010\ 011\ 001\ 100\ 110\ 011\ 001\ 100}11\overline{0011}_2 \times 2^{+5} \ .$$

(2) Compute biased exponent:

Exponent $= +5_{10} = +5_8$, and Bias $= 2^{8-1} = 2^7 = 128_{10} = 200_8$. Therefore, the Characteristic $E_b = (128 + 5)_{10} = 133_{10} = (200 + 5)_8 = 205_8$ .

(3) Convert to octal shorthand and pack result into floating-point format:

$$+29.2_{10} \equiv 205|723146314 \ .$$

(4) For negative number take diminished radix complement of representation for positive number:

$$-29.2_{10} \equiv 572|054631463 \ .$$

The double precision representation follows the same steps, except that a second 36-bit word is generated and the exponent bias differs.

(1d) Normalize as in step 1 above but continue writing out fraction bits and grouping them in groups of three after normalization to obtain 60 fraction bits (or equivalently 20 octal digits) instead of the 27 fraction bits (or 9 octal digits) obtained for the single precision case.

(2d) Compute biased exponent:

Exponent $= +5_{10} = +5_8$, and Bias $= 2^{11-1} = 2^{10} = 1024_{10} = 2000_8$. Therefore, the Characteristic $E_b = (1024 + 5)_{10} = 1029_{10} = (2000 + 5)_8 = 2005_8$ .

(3d) Convert to octal shorthand and pack result into floating-point format comprising two consecutive 36-bit words each of whose bit patterns is represented by a 12-digit octal number:

$$+29.2_{10} \equiv 2005|72314631 \quad \text{and} \quad 463146314631 \quad .$$

(4d) For negative number take diminished radix complement of representation for positive number:

$$-29.2_{10} \equiv 5772|05463146 \quad \text{and} \quad 314631463146 \quad .$$

We next consider representations for $\pm 0.03125_{10}$ .

(1) Normalize:

$$+0.03125_{10} = +0.00001_2 = +0.\underbrace{100\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000}_{2} \times 2^{-4} \ .$$

(2) Compute biased exponent:

Exponent $= -4_{10} = -4_8$, and Bias $= 2^{8-1} = 2^7 = 128_{10} = 200_8$. Therefore, the Characteristic $E_b = (128 - 4)_{10} = 124_{10} = (200 - 4)_8 = 174_8$ .

(3) Convert to octal shorthand and pack result into floating-point format:

$$+0.03125_{10} \equiv 174|400000000 \quad .$$

(4) For negative number take diminished radix complement of representation for positive number:

$$-0.03125_{10} \equiv 603|377777777 \quad .$$

The double precision representation follows the same steps, except that a second 36-bit word is generated and the exponent bias differs.

(1d) Normalize as in step 1 above but continue writing out fraction bits and grouping them in groups of three after normalization to obtain 60 fraction bits (or equivalently 20 octal digits) instead of the 27 fraction bits (or 9 octal digits) obtained for the single precision case; trailing zeros can obviously handled by inspection.

(2d) Compute biased exponent:

Exponent $= -4_{10} = -4_8$, and Bias $= 2^{11-1} = 2^{10} = 1024_{10} = 2000_8$. Therefore, the Characteristic $E_b = (1024 - 4)_{10} = 1020_{10} = (2000 - 4)_8 = 1774_8$ .

(3d) Convert to octal shorthand and pack result into floating-point format comprising two consecutive 36-bit words each of whose bit patterns is represented by a 12-digit octal number:

$$+0.03125_{10} \equiv 1774|40000000 \quad \text{and} \quad 000000000000 \quad .$$

(4d) For negative number take diminished radix complement of representation for positive number:

$$-0.03125_{10} \equiv 6003|37777777 \quad \text{and} \quad 777777777777 \quad .$$

**CDC 6600/7600**

The CDC 6600 series machines use 60-bit words to represent instructions and fixed point data and a one's complement representation for negative numbers with one's complement arithmetic circuitry. A single precision floating point datum is packed into a single 60-bit word, and a double precision floating point datum is packed into two consecutive 60-bit words with the second 60-bit word having a format identical to that of the first with $S_M$, Characteristic, and Mantissa fields; the mantissa field in this second word contains a continuation of the low order bits in the mantissa field in the first word; and the characteristic field (for the positive number representation) contains a value that is $48_{10} = 60_8$ less than the value in the characteristic field in the first word. The mantissa in the first word is a 48-bit binary normalized integer (of the form $1xxx\cdots x._2$), and the remaining 48 bits of mantissa in the second word are unnormalized. Each word in the floating point format(s) has an 11-bit characteristic that represents the signed exponent excess $1024_{10} = 2000_8$ if the original exponent is non-negative and excess $1023_{10} = 1777_8$ if the original exponent is negative. The reason for this is the way in which exponents are packed into the characteristic field. The original signed exponent is represented as an 11-bit one's complement binary integer, and then the high order bit (the exponent's sign bit position) is complemented. For a non-negative exponent with a zero in this exponent sign bit position this corresponds to adding $2000_8$ to the exponent. For a negative exponent with a one in this exponent sign bit position, complementing only the sign bit corresponds to the representation of an 11-bit binary integer resulting from the addition of $1777_8$ to the original negative exponent value; for example, if the original exponent were $-4_{10} = -4_8$, then computing the characteristic by adding $1777_8$ to $-4_8$ results in $1774_8$ which corresponds to the bit pattern obtained by first representing $-4$ as an 11-bit one's complement integer (namely, 11111111011) and then complementing (flipping) only the high order bit (resulting in the bit pattern $01111111011 \equiv 1773_8$). This exponent sign dependent use of different biases to represent the characteristic differs from the single excess biased exponent representations of the UNIVAC, IBM, and DEC PDP-11 formats; by comparison these latter three can be viewed as representing the 2's complement of the binary signed exponent in the k-bit characteristic field and then complementing (*i.e.,* flipping) the high order bit (*i.e.,* the bit in the exponent sign bit position). Negative floating-point numbers in CDC machines are represented by packing the positive representation into the single (or double) word(s) and then taking the one's complement of this (these) word(s) by logically complementing each of the 60 (120) bit positions (including the characteristic field). A word(s) with all zeros represents floating-point +0, and a word(s) with all ones represents floating-point $-0$.

The format for a floating point word is as follows:

| 59 | 58 48 | 47 0 |
|---|---|---|
| $S_M$ | *CHARACTERISTIC* | *MANTISSA* |
| 1 | 11 | 48 |

*CDC 6600/7600/CYBER-series examples:* In order to illustrate the binary normalization process to 48 bit integers we will first consider a simple example in addition to the two example numbers used above. This simple example is $\pm 0.5_{10} = \pm 0.1_2 = \pm 100000000\cdots 000000.0_2 \times 2^{-48}$ . Here we have normalized $0.1_2$ to a 48-bit binary integer comprising a one followed by 47 zeros. Because the exponent is negative (*i.e.,* $-48_{10} = -60_8$), the characteristic is $(1777 - 60)_8 = 1717_8$. Thus,

$$+0.5 \equiv 1717|4000000000000000 \ ; \ \text{and}$$
$$-0.5 \equiv 6060|3777777777777777 \ .$$

We now consider the other two examples from above, but in reverse order. We present the representation of $\pm 0.03125_{10} = \pm 0.00001_2$ first. Normalizing to a 48-bit binary integer requires shifting the binary point $48 + 4 = 52$ positions to the right; thus,

$$0.00001_2 = 100000\cdots total\ 47\ zeros\cdots 000.0_2 \times 2^{-52} \ .$$

Because the exponent is negative (namely, $-52_{10} = -64_8$) we use a bias of $1777_8$. The characteristic for the magnitude $0.03125_{10}$ and its positive representation is thus $(1777 - 64)_8 = 1713_8$ . Thus,

$$+0.03125 \equiv 1713|4000000000000000 \ ; \ \text{and}$$
$$-0.03125 \equiv 6064|3777777777777777 \ .$$

Now consider both the single and double precision representations of $\pm 29.2_{10} = \pm 11101.\overline{0011}_2$ . If a binary fraction starting with $0.1_2$ must be shifted 48 positions to the left during normalization, and the binary integer $1.0_2$ must be shifted 47 positions to the left, then the integer $11101.0_2$ is binary normalized to a 48 bit integer by shifting it left 43 positions relative to the binary point. Thus,

$$\pm 29.2_{10} = \pm 11101001100110011001100110011001100110011001100110011001.1\overline{0011}_2 \times 2^{-43} .$$

The exponent is again negative (*i.e.,* $-43_{10} = -53_8$) so the characteristic becomes $(1777 - 53)_8 = 1724_8$ . The single precision representation is the same as the first (leftmost) word in the two consecutive words of the double precision representation; so below are the double-precision representations.

$$+29.2 \equiv 1724|7231463146314631 \ \ \& \ \ 1644|4631463146314631 \ \ ; \ \ \text{and}$$
$$-29.2 \equiv 6053|0546314631463146 \ \ \& \ \ 6133|3146314631463146 \ \ .$$

We now consider one more example, a very large decimal number that we shall not bother to write down except for its value as a power of two; namely, $\pm 2^{50} = \pm 1000 \cdots total\ of\ 50\ zeros \cdots 00.0_2$ . Binary normalization of this number to a 48-bit binary integer requires that we shift the binary point three positions to the left. Thus, $2^{50} = 100 \cdots total\ of\ 47\ zeros \cdots 000.0_2 \times 2^{+3}$ . Representing this exponent as a one's complement number in the characteristic's 11-bit field results in $(00000000011)_2$, and complementing the leftmost bit produces $(10000000011)_2$, which is the same as adding a bias of $2000_8$ to the positive exponent of $+3_{10} = +3_8$ to produce the characteristic $2003_8$. Finally, the single precision representations are:

$$+2^{50} \equiv 2003|4000000000000000 \ \ ; \ \ \text{and}$$
$$-2^{50} \equiv 5774|3777777777777777 \ \ .$$

**Digital Equipment Corp. (DEC) PDP-11**

The DEC PDP-11 series machines use 16-bit words to represent instructions and fixed point data and a two's complement representation for negative integers with two's complement arithmetic circuitry. A single precision floating point datum is packed into two consecutive 16-bit words and is viewed as a single 32-bit representation. A double precision floating point datum is packed into four consecutive 16-bit words with the second pair of 16-bit words (treated as a 32-bit quantity) representing a continuation of the low order bits in the mantissa. The mantissa is a binary normalized fraction of the form $0.1xxx_2$. A single precision datum has an 8-bit characteristic that represents a signed exponent with bias (excess) $128_{10}$, and a double precision datum has the same 8-bit characteristic, so there is no increase in range by using double-precision. The designers must have taken the term "double-precision" literally. Negative floating-point numbers are represented in a sign-magnitude format by packing the positive representation into the single (or double) word(s) and then setting the $S_M$ bit only to one. Of the 32-bits used to represent single precision numbers only 23 remain in which to represent the normalized fraction. Because every number is assumed normalized in the form $0.1xxxx_2 \times 2^{\pm E}$, there is no need to represent the leading one and so only the x's in the fraction are put into the 23 remaining bits in the word, resulting in 24 bits of precision. We imagine that the leading one lies under and is hidden by an opaque characteristic field; it is therefore, called a "hidden bit." When unpacking a floating-point number for output conversion or for arithmetic manipulation, the hidden bit must be inserted at its rightful place in order to become visible again. A word(s) with all zeros is taken to mean floating-point +0, although (because of the hidden bit assumption) it is really $(2^{-128})_{10}$.

These two formats appear as follows:

*Single Precision (two consecutive 16-bit words):*

| 31 | 30 23 | 22 0 |
|---|---|---|
| $S_M$ | *CHARACTERISTIC* | *MANTISSA* |
| 1 | 8 | 23 |

*Double Precision (four consecutive 16-bit words):*

| 31 | 30 *word pair*1 23 | 22 0 , 31 *word pair*2 0 |
|---|---|---|
| $S_M$ | *CHARACTERISTIC* | *MANTISSA* |
| 1 | 8 | 55 |

*DEC PDP-11 examples:* After first performing binary normalization on the binary representation of the number, computing the biased exponent and packing the $S_M$, CHARACTERISTIC, and (hidden-bit)-MANTISSA fields of the word(s) in the corresponding floating point format, we represent the resulting consecutive 16-bit word(s) in octal shorthand with 6 octal digits for each word (appending leading binary zeros as necessary to the left end of each word to make the number of bits divisible by three for octal shorthand). The representation for negative values is then formed by changing the leftmost bit in the first 16-bit word of the representation for the positive number from zero to $S_M = 1$; so only the leftmost octal digit in the first word's shorthand representation changes from $0_8$ to $1_8$. We consider the single-precision case in detail first, and then present the double-precision result.

We first consider representations for $\pm 29.2_{10}$ .

(1) Normalize:

$$+29.2_{10} = +11101.\overline{0011}_2 = +0.\underline{1}1101\overline{0011}_2 \times 2^{+5} \ .$$

(2) Compute biased exponent:

Exponent $= +5_{10} = +101_2$, and Bias $= 2^{8-1} = 2^7 = 128_{10} = 10000000_2$. Therefore, the Characteristic $E_b = (128 + 5)_{10} = 133_{10} = (10000000 + 101)_2 = 10000101_2$ .

(3) Pack result into two 16-bit word floating-point format in binary:

$$+29.2_{10} \equiv |0|10000101|1101001| \ |1001100110011001| \ .$$

Now treating each of the two words separately as right justified 16-bit binary integers, we start at the right of each word and work left grouping the bits three at a time to obtain the octal shorthand representation of the two consecutive 16-bit words. The result (as would be printed out by a dump of the corresponding memory locations) is:

$$word1: \ 041351 \ \text{and}$$
$$word2: \ 114631 \ .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus,

$$-29.2_{10} \equiv |1|10000101|1101001| \ |1001100110011001| \ ,$$

and the octal shorthand representation is

$$word1: \ 141351 \ \text{and}$$
$$word2: \ 114631 \ .$$

The double precision representation follows the same steps, except that a second 32-bit word comprising two more consecutive 16-bit words with the continuation of the mantissa fraction bits is appended. These continuation bits in this case have the same form as shown in binary in the second 16-bit word in the single precision format. Therefore, we can write the octal shorthand representation of the four consecutive words by inspection. $+29.2_{10}$ is represented by the four words:

$$word1: \ 041351$$
$$word2: \ 114631$$
$$word3: \ 114631$$
$$word4: \ 114631 \ , \text{and}$$

$-29.2_{10}$ is represented by the four words:

$$\begin{aligned} word1: &\ 141351 \\ word2: &\ 114631 \\ word3: &\ 114631 \\ word4: &\ 114631 \quad . \end{aligned}$$

We next consider representations for $\pm 0.03125_{10}$ .

(1) Normalize:

$$+0.03125_{10} = +0.00001_2 = +0.\underline{1}0000000000000000\cdots 00000_2 \times 2^{-4} \ .$$

(2) Compute biased exponent:

Exponent $= -4_{10} = -100_2$, and Bias $= 2^{8-1} = 2^7 = 128_{10} = 1000000_2$. Therefore, the Characteristic $E_b = (128-4)_{10} = 124_{10} = (10000000-100)_2 = 01111100_2$ .

(3) Pack result into two 16-bit word floating-point format in binary:

$$+0.03125_{10} \equiv |0|01111100|0000000| \ \ |0000000000000000| \ .$$

The octal shorthand representation of these two 16-bit words is:

$$\begin{aligned} word1: &\ 037000 \ \text{ and} \\ word2: &\ 000000 \ . \end{aligned}$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus,

$$-0.03125_{10} \equiv |1|01111100|0000000| \ \ |0000000000000000| \ ,$$

with octal shorthand representation:

$$\begin{aligned} word1: &\ 137000 \ \text{ and} \\ word2: &\ 000000 \ . \end{aligned}$$

The double precision representation follows the same steps, except that two more consecutive 16-bit words containing all zeros are appended in each case.

**IEEE Floating Point Standard 754**

The IEEE Floating Point Standard uses four (or eight) 8-bit bytes to represent floating point numbers in machines whose fixed-point data use a two's complement representation for negative integers and that have two's complement arithmetic circuitry. A single precision floating point datum is packed into four consecutive 8-bit bytes viewed as a single 32-bit representation. A double precision floating point datum is packed into eight consecutive 8-bit bytes viewed as a single 64-bit representation, where the bits in the second 32-bits (or second set of four bytes) is continuation of the low order bits in the mantissa (called the *significand*). The mantissa (or significand) is a binary normalized mixed number (with integer and fraction parts) of the form $1.xxxx_2$. A single precision datum has an 8-bit characteristic that represents a signed exponent with bias (excess) $(128-1)_{10} = 127_{10}$, and a double precision datum has an 11-bit characteristic that represents the signed exponent excess $(1024-1)_{10} = 1023_{10}$. Negative floating-point numbers are represented in a sign-magnitude format by packing the positive representation into the single (or double) precision format and then setting the $S_M$ bit only to one. Of the 32-bits used to represent single precision numbers only 23 remain in which to represent the binary normalized mixed number. Because every number is assumed normalized in the form $1.xxxx_2 \times 2^{\pm E}$, there is no need to represent the leading one and so only the x's in the fraction are put into the 23 remaining bits in the word, resulting in 24 bits of precision. We imagine that the leading one lies under and is hidden by an opaque characteristic field; it is therefore, called a "hidden bit."

When unpacking a floating-point number for output conversion or for arithmetic manipulation, the hidden bit must be inserted at its rightful place in order to become visible again. A 32-bit (or 64-bit) floating-point word comprising four (or eight) bytes with all zeros is the representation for floating-point +0 by definition. In addition to defining formats for normalized floating point representations and zero, the standard also specifies valid formats for denormalized numbers, infinity, and quantities that are not a number (*e.g.,* see A. Tanenbaum, *Structured Computer Organization*, 3rd Ed., Prentice-Hall, 1990, pp.565–572).

The two formats for single and double precision appear as follows:

*Single Precision (four consecutive 8-bit bytes):*

| 31 | 30 23 | 22 0 |
|---|---|---|
| $S_M$ | *CHARACTERISTIC* | *MANTISSA* |
| 1 | 8 | 23 |

*Double Precision (eight consecutive 8-bit bytes):*

| 31 | 30 *high order 4 bytes* 20 | 19 0 , 31 *low order 4 bytes* 0 |
|---|---|---|
| $S_M$ | *CHARACTERISTIC* | *MANTISSA* |
| 1 | 11 | 52 |

*IEEE Floating Point Standard 754 examples:* After first performing binary normalization on the binary representation of the number, computing the biased exponent and packing the $S_M$, CHARACTERISTIC, and (hidden-bit)-MANTISSA fields of the word(s) in the corresponding floating point format, we represent the resulting consecutive 32-bit word(s) in hexadecimal shorthand with 8 hexadecimal digits for each 32-bit word The representation for negative values is then formed by changing the leftmost bit in the first 32-bit word of the representation for the positive number from zero to $S_M = 1$; so only the leftmost hexadecimal digit in the first word's shorthand representation changes. For non-negative numbers this leftmost hexadecimal digit is between $0_{16}$ and $7_{16}$, inclusive, because $S_M = 0$. For negative numbers this leftmost hexadecimal digit is between $8_{16}$ and $F_{16}$, inclusive, because in this case $S_M = 1$. We consider the single-precision case in detail first, and then present the double-precision result.

We first consider representations for $\pm 29.2_{10}$ .

(1) Normalize:

$$+29.2_{10} = +11101.\overline{0011}_2 = +\underline{1}.1101\overline{0011}_2 \times 2^{+4} \ .$$

(2) Compute biased exponent:

Exponent $= +4_{10} = +100_2$, and Bias $= 2^{8-1} - 1 = 2^7 - 1 = 127_{10} = 01111111_2$. Therefore, the Characteristic $E_b = (127 + 4)_{10} = (128 + 3)_{10} = 131_{10} = (01111111 + 100)_2 = 10000011_2$ .

(3) Pack result into the 32-bit word floating-point format in binary:

$$+29.2_{10} \equiv |0|10000011|11010011001100110011001| \ .$$

The hexadecimal shorthand representation of the bit pattern in these 32-bits is

$$+29.2_{10} \equiv |41E99999|_{16} \ .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus,

$$-29.2_{10} \equiv |1|10000011|11010011001100110011001| \ \equiv |C1E99999|_{16} \ .$$

The double precision representation follows the same steps, except that a second 32-bit word comprising four more consecutive 8-bit bytes with the continuation of the mantissa fraction bits is appended. Furthermore, the first 32-bit word in the first four bytes is modified to include 3 more characteristic bits and, therefore, 3

fewer mantissa bits, and an exponent bias of $1023_{10} = 01111111111_2$ is used. The characteristic for $\pm 29.2_{10}$ is then $(1023+4)_{10} = (1024+3)_{10} = (01111111111+100)_2 = (10000000000+011)_2 = 10000000011_2$ . Thus

$$+29.2_{10} \equiv$$
$$|0|10000000011|1101001100110011|00110011001100110011001100110011|$$
$$\equiv \; |403D3333|_{16}\; |33333333|_{16} \;\; , \;\; \text{and}$$

$$-29.2_{10} \equiv$$
$$|1|10000000011|1101001100110011|00110011001100110011001100110011|$$
$$\equiv \; |C03D3333|_{16}\; |33333333|_{16} \;\; .$$

We next consider representations for $\pm 0.03125_{10}$ .

(1) Normalize:
$$+0.03125_{10} = +0.00001_2 = +\underline{1}.0000000000000000\cdots 00000_2 \times 2^{-5} \;\; .$$

(2) Compute biased exponent:

Exponent $= -5_{10} = -101_2$, and Bias $= 2^7 - 1 = (128-1)_{10} = 127_{10} = 01111111_2$. Therefore, the Characteristic $E_b = (127-5)_{10} = (128-6)_{10} = 122_{10} = (01111111-101)_2 = (100000000-110)_2 = 01111010_2$ .

(3) Pack result into two 32-bit word floating-point format in binary:

$$+0.03125_{10} \equiv |0|01111010|00000000000000000000000| \;\; \equiv |3D000000|_{16} \;\; .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus,

$$-0.03125_{10} \equiv |1|01111010|00000000000000000000000| \;\; \equiv |BD000000|_{16} \;\; .$$

The double precision representation follows the same steps, except that the characteristic field in the first 32-bit word is expanded by 3 bits using a bias of $1023_{10}$ and one more consecutive 32-bit word containing all zeros is appended in each case. Thus, the characteristic is $(1023-5)_{10} = (01111111111-101)_2 = 01111111010_2$ , and

$$+0.03125_{10} \equiv$$
$$|0|01111111010|00000000000000000000000000000000000000000000000000000000|$$
$$\equiv \; |3FA00000|_{16}\; |00000000|_{16} \;\; , \;\; \text{and}$$

$$-0.03125_{10} \equiv$$
$$|1|01111111010|00000000000000000000000000000000000000000000000000000000|$$
$$\equiv \; |BFA00000|_{16}\; |00000000|_{16} \;\; .$$

**IBM 360/370 series**

The IBM 360/370 series machines use two, four, or six 8-bit bytes to represent instructions, 32-bit words comprising four bytes to represent fixed-point data, and a two's complement representation for negative integers with two's complement arithmetic circuitry. A single precision floating point datum is packed into four consecutive bytes and is viewed as a single 32-bit full word representation (called "floating-point short". A double precision floating point datum is packed into eight consecutive bytes viewed as a single 64-bit word (called "floating-point long") with the low order 32-bits representing a continuation of the low order bits in the mantissa. The mantissa is a hexadecimally normalized fraction of the form $0.yxxx_{16} \times 16^{\pm E}$, where $y$ is a non-zero hexadecimal digit. A single precision datum has a 7-bit characteristic that represents a signed exponent with bias (excess) $64_{10} = 40_{16}$, and a double precision datum has the same 7-bit characteristic, so there is no increase in the already large range by using double-precision. Negative floating-point numbers are represented in a sign-magnitude format by packing the positive representation into the single (or double)

word(s) and then setting the $S_M$ bit only to one. A 32-bit (or 64-bit) word with all zeros is the representation for floating-point zero.

These two formats appear as follows:

*Single Precision (four bytes):*

| 31 | 30 24 | 23 0 |
|---|---|---|
| $S_M$ | $CHARACTERISTIC$ | $MANTISSA$ |
| 1 | 7 | 24 |

*Double Precision (eight bytes):*

| 31 | 30 $word1$ 24 | 23 0 , 31 $word2$ 0 |
|---|---|---|
| $S_M$ | $CHARACTERISTIC$ | $MANTISSA$ |
| 1 | 7 | 56 |

*IBM 360/370 series examples:* After first performing hexadecimal normalization on the hexadecimal representation of the number, computing the biased exponent and packing the $S_M$, CHARACTERISTIC, and MANTISSA fields of the word(s) in the corresponding floating point format, we represent the resulting consecutive 32-bit word(s) in hexadecimal shorthand with 8 hexadecimal digits for each 32-bit word The representation for negative values is then formed by changing the leftmost bit in the first 32-bit word of the representation for the positive number from zero to $S_M = 1$; so only the leftmost hexadecimal digit in the first word's shorthand representation changes. For non-negative numbers this leftmost hexadecimal digit is between $0_{16}$ and $7_{16}$, inclusive, because $S_M = 0$. For negative numbers this leftmost hexadecimal digit is between $8_{16}$ and $F_{16}$, inclusive, because in this case $S_M = 1$. We consider the single-precision case in detail first, and then present the double-precision result.

We first consider representations for $\pm 29.2_{10}$ .

(1) Normalize:

$$29.2_{10} = 1D.\overline{3}_{16} = 0.1D3333\overline{3}_{16} \times 16^2 \ .$$

Note, that because the most significant digit of the hexadecimal fraction is $0.1_{16} = 0.0001_2$, the normalized hexadecimal fraction is not equivalent to a binary normalized fraction because of the three leading zeros in the binary representation. Thus, depending on the datum being represented, the use of three bytes (or 24-bits) to represent the IBM single-precision format hexadecimally normalized fraction provides only 21-bits of precision. The tradeoff, however, is that by using hexadecimal normalization one achieves a range using only 7-bits to represent the signed exponent of $16^{\pm 63} = (2^4)^{\pm 63} = 2^{\pm 252}$ that would require a 9-bit characteristic if binary normalization were used.

(2) Compute biased exponent:

Exponent $= +2_{10} = +2_{16}$, and Bias $= 2^{7-1} = 2^6 = 64_{10} = 40_{16}$. Therefore, the Characteristic $E_b = (64 + 2)_{10} = 66_{10} = (40 + 2)_{16} = 42_{16}$ .

(3) Pack result into floating-point format (in hexadecimal):

$$+29.2_{10} \equiv 42|1D3333 \ .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus, the bit pattern in the register or memory is (in hexadecimal shorthand):

$$-29.2_{10} \equiv C2|1D3333 \ .$$

The double precision representation follows the same steps, except that a second 32-bit word comprising four more consecutive 8-bit bytes with the continuation of the mantissa fraction bits is appended. We can write

the hexadecimal shorthand representation of these additional four consecutive bytes by inspection. $+29.2_{10}$ is represented by the following eight bytes, shown separated into two 32-bit words:

$$+29.2_{10} \equiv |42|1D3333|_{16} \quad |33333333|_{16} \quad \text{, and}$$
$$-29.2_{10} \equiv |C2|1D3333|_{16} \quad |33333333| \quad .$$

We next consider representations for $\pm 0.03125_{10}$ .

(1) Normalize:

$$+0.03125_{10} = 0.08_{16} = 0.8_{16} \times 16^{-1} \ .$$

(2) Compute biased exponent:

Exponent $= -1_{10} = -1_{16}$, and Bias $= 2^{7-1} = 2^6 = 64_{10} = 40_{16}$. Therefore, the Characteristic $E_b = (64-1)_{10} = 63_{10} = (40-1)_{16} = 3F_{16}$ .

(3) Pack result into floating-point format (in hexadecimal):

$$+0.03125_{10} \equiv 3F|800000 \quad .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus, the bit pattern in the register or memory is (in hexadecimal shorthand):

$$-0.03125_{10} \equiv BF|800000 \quad .$$

The double precision representation follows the same steps, except that a second 32-bit word of all zeros in each of these cases is appended. Thus

$$+0.03125_{10} \equiv |3F|800000|_{16} \quad |00000000|_{16} \quad \text{, and}$$
$$-0.03125_{10} \equiv |BF|800000|_{16} \quad |00000000| \quad .$$

IBM has also defined an extended precision floating-point format comprising 128 bits in 16 consecutive bytes that has a 112 bit hexadecimally normalized fraction, but we shall defer discussion of this case to another time.

## Addition of Floating Point Numbers

In order to reduce the number of bits to be processed in the examples we consider the following hypothetical 2's complement machine single precision 14-bit word floating point format:

*Hypothetical Machine:*

| 13 | 12 7 | 6 0 |
|---|---|---|
| $S_M$ | *CHARACTERISTIC* | *SIGNIFICAND* |
| 1 | 6 | 7 |

Here the SIGNIFICAND (or MANTISSAA) is a binary normalized fraction with no hidden bits, and negative numbers are represented in sign-magnitude format by simply setting the sign bit to 1. The CHARACTERISTIC is a biased exponent with exponent bias equal to $2^{6-1} = 2^5 = 32_{10}$.

We first consider representations for $\pm 10.75_{10}$ .

(1) Normalize:

$$+10.75_{10} = +1010.11_2 = +0.101011_2 \times 2^{+4} \ .$$

(2) Compute biased exponent:

Exponent $= +4_{10} = +100_2$, and Bias $= 2^{6-1} = 2^5 = 32_{10} = 100000_2$. Therefore, the Characteristic $E_b = (32 + 4)_{10} = 36_{10} = (100000 + 100)_2 = 100100_2$ .

(3) Pack result into the 14-bit word floating-point format in binary:

$$+10.75_{10} \equiv |0|100100|1010110| \ .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus,

$$-10.75_{10} \equiv |1|100100|1010110| \ .$$

We next consider representations for $\pm(11/32)_{10}$ .

(1) Normalize:

$$+(11/32)_{10} = +1011_2 \times 2^{-5} = +0.01011_2 = +0.1011_2 \times 2^{-1} \ .$$

(2) Compute biased exponent:

Exponent $= -1_{10} = -1_2$, and Bias $= 2^{6-1} = 2^5 = 32_{10} = 100000_2$. Therefore, the Characteristic $E_b = (32 + (-1))_{10} = 31_{10} = (100000 + (-1))_2 = 011111_2$ .

(3) Pack result into the 14-bit word floating-point format in binary:

$$+(11/32)_{10} \equiv |0|011111|1011000| \ .$$

(4) For negative numbers simply set $S_M = 1$ in the positive representation; thus,

$$-(11/32)_{10} \equiv |1|011111|1011000| \ .$$

**Floating Add Algorithm:**

1. Unpack each operand into two registers (one for the characteristic (C) and one for the signed significand (S)). If the significand is negative, convert it to its fixed-point 2's complement representation.
2. Compare the characteristics of the two operands and, if they are not equal, right shift the significand corresponding to the algebraically smaller characteristic while incrementing its characteristic for each right shift until the two characteristics are equal. This is called aligning the significands with respect to the radix point.

3. Prevent overflow of subsequent add by shifting both significands right one position and increment by one both characteristics. Because the two characteristics are now equal, assign one of them to be the characteristic of the result ($C_R$). Also, assume significand registers have one extra guard bit to prevent loss of precision in carrying out this step. (This step guarantees that renormalization of the result will require either no shifting or only left shifting; whereas, if magnitude overflow is allowed to occur during the add, one must also deal with right shifting and sign correction of the sum.)
4. Add the two significands.
5. If the sum is negative, set the sign bit to one in the final single precision format result and take the 2's complement of the sum significand so that the renormalization process works with the positive magnitude. (This is because there are some special case representations of 2's complement numbers that must otherwise be taken care of during renormalization.) If the sum is positive, set the sign bit to zero, and proceed to the next step.
6. Renormalize the binary fraction representing the sum by (if necessary) shifting it left until the high order fraction bit differs from its sign bit. (Recall we are now working with a positive signed magnitude; so the sign bit is a zero and the high order fraction bit should be a 1 for a non-zero sum.) For each left shift decrement by one the characteristic of the result. If after seven left shifts the high order fraction bit is still equal to the zero sign bit, then set the result to zero.
7. Repack the resulting characteristic and normalized significand into their fields in the final single precision floating-point format register. (Recall that the sign bit field was set already in step 5.)

**Example:** Floating add -10.75 to +(11/32).

$$-10.75_{10} \equiv |1|100100|1010110|$$
$$+(11/32)_{10} \equiv |0|011111|1011000|$$

1. Unpack

$$\text{C1} = |100100| \qquad \text{S1} = |1|0101010|x| \quad \leftarrow\ 2'\text{s complement;}\ \ x = \text{guard bit}$$
$$\text{C2} = |011111| \qquad \text{S2} = |0|1011000|x|$$

2. Compare characteristics: Recall C1 = (E1 + bias) and C2 = (E2 + bias), where E1 and E2 are signed integers. Thus, C1 - C2 = E1 + bias - E2 - bias = E1 - E2 = signed integer result. Therefore, take 2's complement of C2 and add it to C1. The leftmost bit of the result is the sign of the result. If this sign bit = 0, the result is positive and C1 $\geq$ C2; if this sign bit = 1, the result is negative and C1 $<$ C2 but it is sitting in its 2's complement form. In this case write down the minus sign and take the 2's complement to find the magnitude of the difference and thus the number of places to shift the contents of S1 to the right.

$$\begin{aligned} \text{C1} - \text{C2} &= 100100 \\ &\underline{+\ 100001} \\ &\quad 000101 \equiv +5 \end{aligned}$$

(Note that in using 2's complement arithmetic to compute the difference of the contents of these two 6-bit fixed length registers, a $7^{th}$ carry bit of 1 came out the left end and was discarded according to the rules of 2's complement arithmetic.) Now we see that C1 > C2; so we algebraically shift the content of S2 right 5 positions and increment C2 by 5, making its content equal to the content of C1 and thus align the fractions relative to the radix point. (Recall that an algebraic (or arithmetic) right shift preserves the sign of complement number representations by bringing in zeros on the left if the sign bit is a zero and bringing in ones if the sign bit is a one.) Thus, after shifting S2 we have:

$$\text{C2} = |100100| \qquad \text{S2} = |0|0000010|1|$$

Note that the 1 now in the guard bit position has already fallen off the end of our finite length 7 bit significand (plus 1 sign bit) register. The guard bit position is there to prevent loss of precision in this case for the zero to its left in the next step.

3. Overflow prevention: Algebraically shift right the contents of both S1 and S2 and increment by 1 both C1 and C2 (either of which is CR now).

$$\begin{aligned} \text{C1} &= |100101| \qquad & \text{S1} &= |1|1010101|0| \\ \text{C2} &= |100101| \qquad & \text{S2} &= |0|0000001|0| \end{aligned}$$

4. 2's complement fixed-point add the contents of the two 9-bit registers S1 and S2:

$$\begin{aligned} \text{S1} + \text{S2} &= |1|1010101|0| \\ &\underline{+\ |0|0000001|0|} \\ &\quad |1|1010110|0| \ = \text{SR} \quad \leftarrow \text{ Result is negative} \end{aligned}$$

5. Result is negative so first repack sign bit into final sum register:

$$\text{SUM} = |1|xxxxxx|yyyyyyy|$$

and in this case take the 2's complement of the content of SR.

$$\text{CR} = |100101| \qquad \text{SR} = |0|0101010|0|$$

6. Note that the content of SR is not normalized. Normalize it by shifting it left one position and decrementing by 1 the content of CR. (Note that for 2's complement numbers an algebraic (or arithmetic) left shift is the same as a logical left shift in which zeros are brought in from the right.) The renormalized result is then:

$$\text{CR} = |100100| \qquad \text{SR} = |0|\underbrace{1010100}|0|$$

7. Repack: The content of the CR register replaces the six x's in the CHARACTERISTIC field in the final SUM and the seven bits indicated by an underbrace replace the seven y's in the SIGNIFICAND field of the final SUM. Thus,

$$\text{SUM} = |1|100100|1010100|$$

is the final packed single precision result in its sign-magnitude form. Note that round-off (i.e., truncation) error has occurred during the computation of the result. It happened in the step that aligned the significands with respect to the radix point during which some bits were shifted out of the fixed length register and were lost. With infinite precision the result should be $-10\frac{13}{32}$; however, the finite precision result in the SUM register is $-10.5$, off by a difference of $\frac{3}{32}$.