WIKIPEDIA

Fixed-point arithmetic

In computing, **fixed-point** refers to a method of representing fractional (non-integer) numbers by storing a fixed number of digits of their fractional part. Dollar amounts, for example, are often stored with exactly two fractional digits, representing the cents (1/100 of dollar). More generally, the term may refer to representing fractional values as integer multiples of some fixed small unit, e.g. a fractional amount of hours as an integer multiple of ten-minute intervals. Fixed-point number representation is often contrasted to the more complicated and computationally demanding floating-point representation.

In the fixed-point representation, the fraction is often expressed in the same number base as the integer part, but using negative powers of the base b. The most common variants are decimal (base 10) and binary (base 2). The latter is commonly known also as **binary scaling**. Thus, if n fraction digits are stored, the value will always be an integer multiple of b^{-n}. Fixed-point representation can also be used to omit the low-order digits of integer values, e.g. when representing large dollar values as multiples of $1000.

When decimal fixed-point numbers are displayed for human reading, the fraction digits are usually separated from those of the integer part by a radix character (usually '.' in English, but ',' or some other symbol in many other languages). Internally, however, there is no separation, and the distinction between the two groups of digits is defined only by the programs that handle such numbers.

Fixed-point representation was the norm in mechanical calculators. Since most modern processors have fast floating-point unit (FPU), fixed-point representations are now used only in special situations, such as in low-cost embedded microprocessors and microcontrollers; in applications that demand high speed and/or low power consumption and/or small chip area, like image, video, and digital signal processing; or when their use is more natural for the problem. Examples of the latter are accounting of dollar amounts, when fractions of cents must be rounded to whole cents in strictly prescribed ways; and the evaluation of functions by table lookup.

Contents

Representation

A fixed-point representation of a fractional number is essentially an integer that is to be implicitly multiplied by a fixed scaling factor. For example, the value 1.23 can be stored in a variable as the integer value 1230 with implicit scaling factor of 1/1000 (meaning that the last 3 decimal digits are implicitly assumed to be a decimal fraction), and the value 1 230 000 can be represented as 1230 with an implicit scaling factor of 1000 (with "minus 3" implied decimal fraction digits, that is, with 3 implicit zero digits at right). This representation allows standard integer arithmetic units to perform rational number calculations.

Fixed-point representation

Value represented	Internal representation
0.00	0
0.5	50
0.99	99
2	200
−14.1	−1410
314.160	31416

Negative values are usually represented in binary fixed-point format as a signed integer in two's complement representation with an implicit scaling factor as above. The sign of the value will always be indicated by the first stored bit (1 = negative, 0 = non-negative), even if the number of fraction bits is greater than or equal to the total number of bits. For example, the 8-bit signed binary integer $(11110101)_2 = -11$, taken with -3, +5, and +12 implied fraction bits, would represent the values $-11/2^{-3} = -88$, $-11/2^5 = -0.343\,75$, and $-11/2^{12} = -0.002\,685\,546\,875$, respectively.

Alternatively, negative values can be represented by an integer in the sign-magnitude format, in which case the sign is never included in the number of implied fraction bits. This variant is more commonly used in decimal fixed-point arithmetic. Thus the signed 5-digit decimal integer $(-00025)_{10}$, taken with -3, +5, and +12 implied decimal fraction digits, would represent the values $-25/10^{-3} = -25000$, $-25/10^5 = -0.00025$, and $-25/10^{12} = -0.000\,000\,000\,025$, respectively.

A program will usually assume that all fixed-point values that will be stored into a given variable, or will be produced by a given instruction, will have the same scaling factor. This parameter can usually be chosen by the programmer depending on the precision needed and range of values to be stored.

The scaling factor of a variable or formula may not appear explicitly in the program. Good programming practice then requires that it be provided in the documentation, at least as a comment in the source code.

Choice of scaling factors

For greater efficiency, scaling factors are often chosen to be powers (positive or negative) of the base b used to represent the integers internally. However, often the best scaling factor is dictated by the application. Thus one often uses scaling factors that are powers of 10 (e.g. 1/100 for dollar values), for human convenience, even when the integers are represented internally in binary. Decimal scaling factors also mesh well with the metric (SI) system, since the choice of the fixed-point scaling factor is often equivalent to the choice of a unit of measure (like centimetres or microns instead of metres).

However, other scaling factors may be used occasionally, e.g. a fractional amount of hours may be represented as an integer number of seconds; that is, as a fixed-point number with scale factor of 1/3600.

Even with the most careful rounding, fixed-point values represented with a scaling factor S may have an error of up to ±0.5 in the stored integer, that is, ±0.5 S in the value. Therefore, smaller scaling factors generally produce more accurate results.

On the other hand, a smaller scaling factor means a smaller range of the values that can be stored in a given program variable. The maximum fixed-point value that can be stored into a variable is the largest integer value that can be stored into it, multiplied by the scaling factor; and similarly for the minimum value. For example, the table below gives the implied scaling factor S, the minimum and maximum representable values $V_{\min}$ and $V_{\max}$, and the accuracy $\delta = S/2$ of values that could be represented in 16-bit signed binary fixed point format, depending on the number f of implied fraction bits.

Parameters of some 16-bit signed binary fixed-point formats

f	S	δ	$V_{\min}$	$V_{\max}$
−3	$1/2^{-3} = 8$	4	−262 144	+262 143
0	$1/2^{0} = 1$	0.5	−32 768	+32 767
5	$1/2^{5} = 1/32$	< 0.016	−1024.000 00	+1023.968 75
14	$1/2^{14} = 1/16\,384$	< 0.000 031	−2.000 000 000 000 00	+1.999 938 964 843 75
15	$1/2^{15} = 1/32\,768$	< 0.000 016	−1.000 000 000 000 000	+0.999 969 482 421 875
16	$1/2^{16} = 1/65\,536$	< 0.000 008	−0.500 000 000 000 000 0	+0.499 984 741 210 937 5
20	$1/2^{20} = 1/1\,048\,576$	< 0.000 000 5	−0.031 250 000 000 000 000 00	+0.031 249 046 325 683 593 75

Fixed-point formats with scaling factors of the form 2^n-1 (namely 1, 3, 7, 15, 31, etc.) have been said to be appropriate for image processing and other digital signal procssing tasks. They are supposed to provide more consistent conversions between fixed- and floating-point values than the usual 2^n scaling. The Julia programming language implements both versions.[1]

Exact values

Any binary fraction $a/2^m$, such as 1/16 or 17/32, can be exactly represented in fixed-point, with a power-of-two scaling factor $1/2^n$ with any $n \geq m$. However, most decimal fractions like 0.1 or 0.123 are infinite repeating fractions in base 2. and hence cannot be represented that way.

Similarly, any decimal fraction $a/10^m$, such as 1/100 or 37/1000, can be exactly represented in fixed point with a power-of-ten scaling factor $1/10^n$ with any $n \geq m$. This decimal format can also represent any binary fraction $a/2^m$, such as 1/8 (0.125) or 17/32 (0.53125).

More generally, a rational number a/b, with a and b relatively prime and b positive, can be exactly represented in binary fixed point only if b is a power of 2; and in decimal fixed point only if b has no prime factors other than 2 and/or 5.

Comparison with floating-point

Fixed-point computations can be faster and/or use less hardware than floating-point ones. If the range of the values to be represented is known in advance and is sufficiently limited, fixed point can make better use of the available bits. For example, if 32 bits are available to represent a number between 0 and 1, a fixed-point representation can have error less than 1.2×10^{-10}, whereas the standard floating-point representation may have error up to 596×10^{-10} — because 9 of the bits are wasted with the sign and exponent of the dynamic scaling factor. Specifically, comparing 32-bit fixed-point to floating-point audio, a recording requiring less than 40 dB of headroom has a higher signal-to-noise ratio using 32-bit fixed.

Programs using fixed-point computations are usually more portable than those using floating-point, since they don't depend on the availabilty of an FPU. This advantage was particularly strong before the IEEE Floating Point Standard was widely adopted, when floating-point computations with the same data would yield different results depending on the manufacturer, and often on the computer model.

Many embedded processors lack an FPU, because integer arithmetic units require substantially fewer logic gates and consume much smaller chip area than an FPU; and software emulation of floating-point on low-speed devices would be too slow for most applications. CPU chips for the earlier personal computers and game consoles, like the Intel 386 and 486SX, also lacked an FPU.

The *absolute* resolution (difference between successive values) of any fixed-point format is constant over the whole range, namely the scaling factor S. In contrast, the *relative* resolution of a floating-point format is approximately constant over their whole range, varying within a factor of the base b; whereas their *absolute* resolution varies by many orders of magnitude, like the values themselves.

In many cases, the rounding and truncation errors of fixed-point computations are easier to analyze than those of the equivalent floating-point computations. Applying linearization techniques to truncation, such as dithering and/or noise shaping is more straight-forward within fixed-point arithmetic. On the other hand, the use of fixed point requires greater care by the programmer. Avoidance of overflow requires much tighter estimates for the ranges of variables and all intermediate values in the computation, and often also extra code to adjust their scaling factors. Fixed-point programming normally requires the use of integer types of different widths. Fixed-point applications can make use of block floating point, which is a fixed-point environment having each array (block) of fixed-point data be scaled with a common exponent in a single word.

Applications

A common use of decimal fixed-point is for storing monetary values, for which the complicated rounding rules of floating-point numbers are often a liability. For example, the open source money management application GnuCash, written in C, switched from floating-point to fixed-point as of version 1.6, for this reason.

Binary fixed-point (binary scaling) was widely used from the late 1960s to the 1980s for real-time computing that was mathematically intensive, such as flight simulation and in nuclear power plant control algorithms. It is still used in many DSP applications and custom made microprocessors. Computations involving angles would use binary angular measurement (BAM).

Binary fixed point is used in the STM32G4 series CORDIC co-processors and in the discrete cosine transform (DCT) algorithms used to compress JPEG images.

Electronic instruments such as electricity meters and digital clocks often use polynomials to compensate for introduced errors, e.g. from temperature or power supply voltage. The coefficients are produced by polynomial regression. Binary fixed point polynomials can utilize more bits of precision than floating-point, and do so in fast code using inexpensive CPUs. Accuracy, crucial for instruments, compares well to equivalent-bit floating-point calculations, if the fixed-point polynomials are factored (e.g. y = d + x(c + x(b + xa))) to reduce the number of times that rounding occurs, and the fixed-point multiplications utilize rounding addends.

Operations

Addition and subtraction

To add or subtract two values with the same implicit scaling factor, it is sufficient to add or subtract the underlying integers; the result will have their common implicit scaling factor, can thus can be stored in the same program variables as the operands. These operations yield the exact mathematical result, as long as no overflow occurs—that is, as long as the resulting integer can be stored in the receiving program variable. If the values have different scaling factors, then they must be converted to a common scaling factor before the operation.

Multiplication

To multiply two fixed-point numbers, it suffices to multiply the two underlying integers, and assume that the scaling factor of the result is the product of their scaling factors. The result will be exact, with no rounding, provided that it does not overflow the receiving variable.

For example, multiplying the numbers 123 scaled by 1/1000 (0.123) and 25 scaled by 1/10 (2.5) yields the integer 123×25 = 3075 scaled by (1/1000)×(1/10) = 1/10000, that is 3075/10000 = 0.3075. As another example, multiplying the first number by 155 implicitly scaled by 1/32 (155/32 = 4.84375) yields the integer 123×155 = 19065 with implicit scaling factor (1/1000)×(1/32) = 1/32000, that is 19065/32000 = 0.59578125.

In binary, it is common to use a scaling factor that is a power of two. After the multiplication, the scaling factor can be divided away by shifting right. Mechanically, this process is simple and fast in most computers. Rounding is possible by adding a 'rounding addend' of half of the scaling factor before shifting; The proof: round(x/y) = floor(x/y + 0.5) = floor((x + y/2)/y) = shift-of-n(x + 2^(n-1)) A similar method is usable in any scaling.

Division

To divide two fixed-point numbers, one takes the integer quotient of their underlying integers, and assumes that the scaling factor is the quotient of their scaling factors. In general, the first division requires rounding and therefore the result is not exact.

For example, division of 3456 scaled by 1/100 (34.56) and 1234 scaled by 1/1000 (1.234) yields the integer

3456÷1234 = 3 (rounded) with scale factor (1/100)/(1/1000) = 10, that is, 30. As another example, the division of the first number by 155 implicitly scaled by 1/32 (155/32 = 4.84375) yields the integer 3456÷155 = 22 (rounded) with implicit scaling factor (1/100)/(1/32) = 32/100 = 8/25, that is 22×32/100 = 7.04.

If the result is not exact, the error introduced by the rounding can be reduced or even eliminated by converting the dividend to a smaller scaling factor. For example, if r = 1.23 is represented as 123 with scaling 1/100, and s = 6.25 is represented as 6250 with scaling 1/1000, then simple division of the integers yields 123÷6250 = 0 (rounded) with scaling factor (1/100)/(1/1000) = 10. If r is first converted to 1,230,000 with scaling factor 1/1000000, the result will be 1,230,000÷6250 = 197 (rounded) with scale factor 1/1000 (0.197). The exact value 1.23/6.25 is 0.1968.

Scaling conversion

In fixed-point computing it is often necessary to convert a value to a different scaling factor. This operation is necessary, for example:

- To store a value into a program variable that has a different implicit scaling factor;
- To convert two values to the same scaling factor, so that they can be added or subtracted;
- To restore the original scaling factor of a value after multiplying or dividing it by another;
- To improve the accuracy of the result of a division;
- To ensure that the scaling factor of a product or quotient is a simple power like 10^n or 2^n;
- To ensure that the result of an operation can be stored into a program variable without overflow;
- To reduce the cost of hardware that processes fixed-point data.

To convert a number from a fixed point type with scaling factor R to another type with scaling factor S, the underlying integer must be multiplied by the ratio R/S. Thus, for example, to convert the value 1.23 = 123/100 from scaling factor R=1/100 to one with scaling factor S=1/1000, the integer 123 must be multiplied by (1/100)/(1/1000) = 10, yielding the representation 1230/1000.

If the scaling factor is a power of the base used internally to represent the integer, changing the scaling factor requires only dropping low-order digits of the integer, or appending zero digits. However, this operation must preserve the sign of the number. In two's complement representation, that means extending the sign bit as in arithmetic shift operations.

If S does not divide R (in particular, if the new scaling factor S is greater than the original R), the new integer may have to be rounded.

In particular, if r and s are fixed-point variables with implicit scaling factors R and S, the operation $r \leftarrow r \times s$ require multiplying the respective integers and explicitly dividing the result by S. The result may have to be rounded, and overflow may occur.

For example, if the common scaling factor is 1/100, multiplying 1.23 by 0.25 entails multiplying 123 by 25 to yield 3075 with an intermediate scaling factor of 1/10000. In order to return to the original scaling factor 1/100, the integer 3075 then must be multiplied by 1/100, that is, divided by 100, to yield either 31 (0.31) or 30 (0.30), depending on the rounding policy used.

Similarly, the operation $r \longleftarrow r/s$ will require dividing the integers and explicitly multiplying the quotient by S. Rounding and/or overflow may occur here too.

Conversion to and from floating-point

To convert a number from floating point to fixed point, one may divide it by the scaling factor S, then round the result to the nearest integer. Care must be taken to ensure that the result fits in the destination variable or register. Depending on the scaling factor and storage size, and on the range input numbers, the conversion may not entail any rounding.

To convert a fixed-point number to floating-point, one may convert the integer to floating-point and then multiply it by the scaling factor S. This conversion may entail rounding if the integer's absolute value is greater than 2^{24} (for binary single-precision IEEE floating point) or of 2^{53} (for double-precision). Overflow or underflow may occur if $|S|$ is *very* large or *very* small, respectively.

Hardware support

Scaling and renormalization

Typical processors do not have specific support for fixed-point arithmetic. However, most computers with binary arithmetic have fast bit shift instructions that can multiply or divide an integer by any power of 2; in particular, an arithmetic shift instruction. These instructions can be used to quickly change scaling factors that are powers of 2, while preserving the sign of the number.

Early computers like the IBM 1620 and the Burroughs B3500 used a binary-coded decimal (BCD) representation for integers, namely base 10 where each decimal digit was independently encoded with 4 bits. Some processors, such as microcontrollers, may still use it. In such machines, conversion of decimal scaling factors can be performed by bit shifts and/or by memory address manipulation.

Some DSP architectures offer native support for specific fixed-point formats, for example signed n-bit numbers with $n-1$ fraction bits (whose values may range between -1 and almost $+1$). The support may include a multiply instruction that includes renormalization -- the scaling conversion of the product from $2n-2$ to $n-1$ fraction bits. If the CPU does not provide that feature, the programmer must save the product in a large enough register or temporary variable, and code the renormalization explicitly.

Overflow

Overflow happens when the result of an arithmetic operation is too large to be stored in the designated destination area. In addition and subtraction, the result may require one bit more than the operands. In multiplication of two unsigned integers with m and n bits, the result may have $m+n$ bits.

In case of overflow, the high-order bits are usually lost, as the un-scaled integer gets reduced modulo 2^n where n is the size of the storage area. The sign bit, in particular, is lost, which may radically change the sign and the magnitude of the value.

Some processors can set a hardware overflow flag and/or generate an exception on the occurrence of an overflow. Some processors may instead provide saturation arithmetic: if the result of an addition or subtraction were to overflow, they store instead the value with largest magnitude that can fit in the receiving area and has the correct sign.

However, these features are not very useful in practice; it is generally easier and safer to select scaling factors and word sizes so as to exclude the possibility of overflow, or to check the operands for excessive values before executing the operation.

Computer language support

Explicit support for fixed-point numbers is provided by a few computer languages, notably PL/I, COBOL, Ada, JOVIAL, and Coral 66. They provide fixed-point data types, with a binary or decimal scaling factor. The compiler automatically generates code to do the appropriate scaling conversions when doing operations on these data-types, when reading or writing variables, or when converting the values to other data types such as floating-point.

Most of those languages were designed between 1940 and 1990. More modern languages usually do not offer any fixed-point data types or support for scaling factor conversion. That is also the case for several older languages that are still very popular, like FORTRAN, C and C++. The wide availability of fast floating-point processors, with strictly standardized behavior, have greatly reduced the demand for binary fixed point support. Similarly, the support for decimal floating point in some programming languages, like C# and Python, has removed most of the need for decimal fixed-point support. In the few situations that call for fixed-point operations, they can be implemented by the programmer, with explicit scaling conversion, in any programming language.

On the other hand, all relational databases and the SQL notation support fixed-point decimal arithmetic and storage of numbers. PostgreSQL has a special `numeric` type for exact storage of numbers with up to 1000 digits.[2]

Moreover, in 2008 the International Standards Organization (ISO) issued a proposal to extend the C programming language with fixed-point data types, for the benefit of programs running on embedded processors.[3] Also, the GNU Compiler Collection (GCC) has back-end support for fixed-point.[4][5]

Detailed examples

Decimal fixed point multiplication

Suppose there is the following multiplication with 2 fixed point 3 decimal place numbers.

$$\begin{aligned}(10.500)(1.050) &= 1 \times 10.500 + 0.050 \times 10.500 \\ &= 10.500 + 0.525000 = 11.025000\end{aligned}$$

Note how since there are 3 decimal places we show the trailing zeros. To re-characterize this as an integer multiplication we must first multiply by $1000\ (= 10^3)$ moving all the decimal places in to integer places, then we will multiply by $1/1000\ (= 10^{-3})$ to put them back the equation now looks like

$$\begin{aligned}(10.500)(10^3)(1.050)(10^3)(10^{-3})(10^{-3}) &= (10500)(1050)(10^{-6}) \\ &= 11\,025\,000(10^{-6}) \\ &= 11.025000\end{aligned}$$

This works equivalently if we choose a different base, notably base 2 for computing, since a bit shift is the same as a multiplication or division by an order of 2. Three decimal digits is equivalent to about 10 binary digits, so we should round 0.05 to 10 bits after the binary point. The closest approximation is then 0.0000110011.

$$\begin{aligned}10 &= 8 + 2 = 2^3 + 2^1\\ 1 &= 2^0\\ 0.5 &= 2^{-1}\\ 0.05 &= 0.0000110011_2\end{aligned}$$

Thus our multiplication becomes

$$\begin{aligned}(1010.100)(2^3)(1.0000110011)(2^{10})(2^{-13}) &= (1010100)(10000110011)(2^{-13})\\ &= (10110000010111100)(2^{-13})\\ &= 1011.0000010111100\end{aligned}$$

This rounds to 11.023 with three digits after the decimal point.

Binary fixed-point multiplication

Consider the task of computing the product of 1.2 and 5.6 with binary fixed point using 16 fraction bits. To represent the two numbers, one multiplies them by 2^{16}, obtaining 78 643.2 and 367 001.6; and round these values the nearest integers, obtaining 78 643 and 367 002. These numbers will fit comfortably into a 32-bit word with two's complement signed format.

Multiplying these integers together gives the 35-bit integer 28 862 138 286 with 32 fraction bits, without any rounding. Note that storing this value directly into a 32-bit integer variable would result in overflow and loss of the most significant bits. In practice, it would probably be stored in a signed 64-bit integer variable or register.

If the result is to be stored in the same format as the data, with 16 fraction bits, that integer should be divided by 2^{16}, which gives approximately 440 401.28, and then rounded to the nearest integer. This effect can be achieved by adding 2^{15} and then shifting the result by 16 bits. The result is 440 401, which represents the value 6.719 985 961 914 062 5. Taking into account the precision of the format, that value is better expressed as 6.719 986 ± 0.000 008 (not counting the error that comes from the operand approximations). The correct result would be 1.2 × 5.6 = 6.72.

For a more complicated example, suppose that the two numbers 1.2 and 5.6 are represented in 32-bit fixed point format with 30 and 20 fraction bits, respectively. Scaling by 2^{30} and 2^{20} gives 1 288 490 188.8 and 5 872 025.6, that round to 1 288 490 189 and 5 872 026, respectively. Both numbers still fit in a 32-bit signed integer variable, and represent the fractions

1.200 000 000 186 264 514 923 095 703 125 and
5.600 000 381 469 726 562 50

Their product is (exactly) the 53-bit integer 7 566 047 890 552 914, which has 30+20 = 50 implied fraction bits and therefore represents the fraction

6.720 000 458 806 753 229 623 609 513 510

If we choose to represent this value in signed 16-bit fixed format with 8 fraction bits, we must divide the integer product by $2^{50-8} = 2^{42}$ and round the result; which can be achieved by adding 2^{41} and shifting by 42 bits. The result is 1720, representing the value $1720/2^8$ = 6.718 75, or approximately 6.719 ± 0.002.

Notations

Various notations have been used to concisely specify the parameters of a fixed-point format. In the following list, *f* represents the number of fractional bits, *m* the number of magnitude or integer bits, *s* the number of sign bits, and *b* the total number of bits.

- The COBOL programming language originally supports decimal fixed-precision with arbitrary size and decimal scaling, whose format was specified "graphically" with the `PIC` directive. For example, `PIC S9999V99` specified a sign-magnitude 6-digit decimal integer with two decimal fraction digits.[6]
- The construct `REAL FIXED BINARY (`*p*`,`*f*`)` was used in the PL/I programming language, to specify a fixed-point signed binary data type with *p* total bits (not including sign) with *f* bits in the fraction part; that is a *p*+1 bit signed integer with a scaling factor of $1/2^f$. The latter could be positive or negative. One could specify `COMPLEX` instead of `REAL`, and `DECIMAL` instead of `BINARY` for base 10.
- In the Ada programming language, a numeric data type could be specified by, for example,`type F is delta 0.01 range -100.0 .. 100.0`, meaning a fixed-point representation consisting of a signed binary integer in two's complement format with 7 implied fraction bits (providin a scaling factor 1/128) and at least 15 bits total (ensuring an actual range from -128.00 to almost +128.00).[7]
- The Q notation was defined by Texas Instruments.[8] One writes `Q`*f* to specify a signed binary fixed-point value with *f* fraction bits; for example, `Q15` specifies a signed integer in two's complement notation with a scaling factor $1/2^{15}$. The code `Q`*m*`.`*f* specifies additionally that the number has *m* bits in the integer part of the value, not counting the sign bit. Thus `Q1.30` would describe a binary fixed-point format with 1 integer bit and 30 fractional bits, which could be stored as a 32-bit 2's complement integer with scaling factor $1/2^{30}$.[8][9] A similar notation has been used by ARM, except that they count the sign bit in the value of *m*; so the same format above would be specified as `Q2.30`.[10][11]
- The notation B*m* has been used to mean a fixed binary format with *m* bits in the integer part; the rest of the word being fraction bits. For example, the maximum and minimum values that can be stored in a signed B16 number are ≈32767.9999847 and −32768.0,respectively.
- The VisSim company used `fx`*m*`.`*b* to denote a binary fixed-point value with *b* total bits and *m* bits in the integer part; that is, a *b*-bit integer with scaling factor $1/2^{b-m}$. Thus `fx1.16` would mean a 16-bit number with 1 bit in the integer part and 15 in the fraction.[12]
- The PS2 GS ("Graphics Synthesizer") User's Guide uses the notation *s*`:`*m*`:`*f*, where *s* specifies the presence (0 or 1) of sign bit.[13] For example, `0:5:3` represents an unsigned 8-bit integer with a scaling factor of $1/2^3$.
- The LabVIEW programming language uses the notation <*s*`,`*b*`,`*m*> to specify the parameters of an 'FXP' fixed point numbers. The *s* component can be either '+' or '±', signifying either an unsigned or 2's complement signed number, respectively. The *b* component is the total number of bits, and *m* is the number of bits in the integer part.

Software application examples

- The popular TrueType font format uses 32-bit signed binary fixed-point with 26 bits to the left of the decimal for some numeric values in its instructions.[14] This format was chosen to provide the minimal amount of precision required for hinting and for performance reasons.[15]

- All 3D graphics engines on Sony's original PlayStation, Sega's Saturn, Nintendo's Game Boy Advance (only 2D), Nintendo DS (2D and 3D), Nintendo Gamecube[16] and GP2X Wiz video game systems, which lacked an FPU, used fixed-point arithmetic. The PlayStation included hardware support for 16-bit fixed point with 12 fraction bits in its transformation coprocessor.
- The TeX typesetting software, widely used by scientists and mathematicians, uses 32-bit signed binary fixed point with 16 fraction bits for all position calculations. The values are interpreted as fractions of a typographer's point. TeX font metric files use 32-bit signed fixed-point numbers, with 12 fraction bits.
- Tremor, Toast and MAD are software libraries which decode the Ogg Vorbis, GSM Full Rate and MP3 audio formats respectively. These codecs use fixed-point arithmetic because many audio decoding hardware devices do not have an FPU.
- The Wavpack lossless audio compressor uses fixed point arithmetic. The choice was justified by, among other things, the worry that different floating-point rounding rules in different hardware could corrupt the lossless nature of the compression.[17]
- The Nest Labs Utilities library (https://github.com/nestlabs/nlutilities), provides a limited set of macros and functions for fixed point numbers, particularly when dealing with those numbers in the context of sensor sampling and sensor outputs.
- The OpenGL ES 1.x specification includes a fixed point profile, as it is an API aimed for embedded systems, which do not always have an FPU.
- The dc and bc programs are arbitrary precision calculators, but only keep track of a (user-specified) fixed number of fractional digits.
- Fractint represents numbers as Q2.29 fixed-point numbers,[18] to speed up drawing on old PCs with 386 or 486SX processors, which lacked an FPU.
- *Doom* was the last first-person shooter title by id Software to use a 16.16 fixed point representation for all of its non-integer computations, including map system, geometry, rendering, player movement etc. For compatibility reasons, this representation is still used in modern *Doom* source ports.
- fixed point numbers are sometimes used for storing and manipulating images and video frames. Processors with SIMD units aimed at image processing may include instructions suitable for handling packed fixed point data.
- The Q# programming language for the Azure quantum computers, that implement quantum logic gates, contains a standard numeric library for performing fixed-point arithmetic on registers of qubits.[19]

See also

- Q (number format)
- Libfixmath - a library written in C for fixed-point math
- Floating-point arithmetic
- Logarithmic number system
- Minifloat
- Block floating-point scaling
- Modulo operation
- µ-law algorithm
- A-law algorithm

References

1. Julia programming language documentation FixedPointNumbers package (https://github.com/JuliaMath/FixedPointNumbers.jl).
2. PostgreSQL manual, section 8.1.2. Arbitrary Precision Numbers (http://www.postgresql.org/docs/8.3/static/datatype-numeric.html#DATATYPE-NUMERIC-DECIMAL)
3. JTC1/SC22/WG14 (2008), status of TR 18037: Embedded C (http://www.open-std.org/JTC1/SC22/WG14/www/projects#18037)
4. GCC wiki, Fixed-Point Arithmetic Support (https://gcc.gnu.org/wiki/FixedPointArithmetic)
5. Using GCC, section 5.13 Fixed-Point Types (https://gcc.gnu.org/onlinedocs/gcc/Fixed-Point.html)
6. IBM Corporation, "Numeric items (https://www.ibm.com/docs/en/i/7.2?topic=rules-numeric-items#dcaprnu)". Online documentation site, accessedon 2021-07-05.
7. Ada 83 documentation: "Rationale, 5.3.2: Fixed Point Types (http://archive.adaic.com/standards/83rat/html/ratl-05-03.html#5.3.2)". Accessed on 2021-07-05.
8. Texas Instruments, TMS320C64x DSP Library Programmer's Reference (http://focus.ti.com/lit/ug/spru565b/spru565b.pdf), Appendix A.2
9. "MathWorks Fixed-Point Toolbox Documentation Glossary" (http://www.mathworks.com/help/toolbox/fixedpoint/ref/bp7g699.html#f6811). *mathworks.com*.
10. "ARM Developer Suite AXD and armsd Debuggers Guide" (http://infocenter.arm.com/help/topic/com.arm.doc.dui0066d/CHDFAAEI.html?resultof=%22%51%2d%66%6f%72%6d%61%74%22%20%22%71%2d%66%6f%72%6d%61%74%22%20). 1.2. ARM Limited. 2001 [1999]. Chapter 4.7.9. AXD > AXD Facilities > Data formatting > Q-format. ARM DUI 0066D. Archived (https://archive.today/20171104110547/http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0066d/CHDFAAEI.html) from the original on 2017-11-04.
11. "Chapter 4.7.9. AXD > AXD Facilities > Data formatting > Q-format". *RealView Development Suite AXD and armsd Debuggers Guide* (http://infocenter.arm.com/help/topic/com.arm.doc.dui0066g/DUI0066.pdf) (PDF). 3.0. ARM Limited. 2006 [1999]. pp. 4–24. ARM DUI 0066G. Archived (https://web.archive.org/web/20171104105632/http://infocenter.arm.com/help/topic/com.arm.doc.dui0066g/DUI0066.pdf) (PDF) from the original on 2017-11-04.
12. Inc., solidThinking. "VisSim is now solidThinking Embed" (http://www.vissim.com/products/addons/vissim/fixed-point.html). *www.vissim.com*.
13. PS2 GS User's Guide, Chapter 7.1 "Explanatory Notes"
14. "The TrueType Instruction Set: Data types" (https://docs.microsoft.com/en-us/typography/opentype/otspec140/tt_instructions#dt).
15. "[Freetype] Why 26.6 ?" (https://lists.nongnu.org/archive/html/freetype/2002-09/msg00076.html).
16. "Dolphin Emulator" (http://dolphin-emu.org/blog/2014/03/15/pixel-processing-problems/). *Dolphin Emulator*. 15 March 2014.
17. "WavPack Technical Description" (http://www.wavpack.com/technical.htm). *www.wavpack.com*. Retrieved 2015-07-13.
18. Fractint, A Little Code (http://spanky.triumf.ca/www/fractint/periodicity.html#integer_math_anchor)
19. "Introduction to the Quantum Numerics Library" (https://docs.microsoft.com/en-us/quantum/libraries/numerics/?view=qsharp-preview). Retrieved 2019-11-13.

Further reading

- Warren Jr., Henry S. (2013). *Hacker's Delight* (2 ed.). Addison Wesley - Pearson Education, Inc. ISBN 978-0-321-84268-8.

External links

- Simple Fixed-Point Math (https://spin.atomicobject.com/2012/03/15/simple-fixed-point-math/)
- Fixed-Point Arithmetic - An Introduction (https://web.archive.org/web/20150912013429/http://www.digitalsignallabs.com/fp.pdf)
- Fixed Point Representation and Fractional Math (http://www.superkits.net/whitepapers/Fixed%20Point%20Representation%20&%20Fractional%20Math.pdf)
- A Calculated Look at Fixed-Point Arithmetic (https://web.archive.org/web/20020611080806/http://www.embedded.com/98/9804fe2.htm), (PDF) (https://web.archive.org/web/20111005183025/http://www.eetindia.co.in/ARTICLES/1998APR/PDF/EEIOL_1998APR03_EMS_TA.pdf)