Fixed-point arithmetic

This article is about a form of limited-precision arithmetic in computing. For the invariant points of a mathematical function, see Fixed point (mathematics).

In computing, a **fixed-point number** representation is a real data type for a number that has a fixed number of digits after (and sometimes also before) the radix point (after the decimal point '.' in English decimal notation). Fixed-point number representation can be compared to the more complicated (and more computationally demanding) floating-point number representation.

Fixed-point numbers are useful for representing fractional values, usually in base 2 or base 10, when the executing processor has no floating point unit (FPU) or if fixed-point provides improved performance or accuracy for the application at hand. Most low-cost embedded microprocessors and microcontrollers do not have an FPU.

1 Representation

A value of a fixed-point data type is essentially an integer that is scaled by an implicit specific factor determined by the type. For example, the value 1.23 can be represented as 1230 in a fixed-point data type with scaling factor of 1/1000, and the value 1,230,000 can be represented as 1230 with a scaling factor of 1000. Unlike floating-point data types, the scaling factor is the same for all values of the same type, and does not change during the entire computation.

The scaling factor is usually a power of 10 (for human convenience) or a power of 2 (for computational efficiency). However, other scaling factors may be used occasionally, e.g. a time value in hours may be represented as a fixed-point type with a scale factor of 1/3600 to obtain values with one-second accuracy.

The maximum value of a fixed-point type is simply the largest value that can be represented in the underlying integer type multiplied by the scaling factor; and similarly for the minimum value. For example, consider a fixed-point type represented as a binary integer with b bits in two's complement format, with a scaling factor of $1/2^f$ (that is, the last f bits are fraction bits): the minimum representable value is $-2^{b-1}/2^f$ and the maximum value is $(2^{b-1}-1)/2^f$.

2 Operations

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 R and divided by S; that is, multiplied by the ratio R/S. Thus, for example, to convert the value 1.23 = 123/100 from a type with scaling factor R=1/100 to one with scaling factor S=1/1000, the underlying integer 123 must be multiplied by (1/100)/(1/1000) = 10, yielding the representation 1230/1000. If S does not divide R (in particular, if the new scaling factor S is greater than the original R), the new integer will have to be rounded. The rounding rules and methods are usually part of the language's specification.

To add or subtract two values of the same fixed-point type, it is sufficient to add or subtract the underlying integers, and keep their common scaling factor. The result can be exactly represented in the same type, as long as no overflow occurs (i.e. provided that the sum of the two integers fits in the underlying integer type). If the numbers have different fixed-point types, with different scaling factors, then one of them must be converted to the other before the sum.

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. This operation involves no rounding. 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. If the two operands belong to the same fixed-point type, and the result is also to be represented in that type, then the product of the two integers must be explicitly multiplied by the common scaling factor; in this case 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. This then must be multiplied by 1/100 to yield either 31 (0.31) or 30 (0.30), depending on the rounding method used, to result in a final scale factor of 1/100.

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. The first division involves rounding in general. 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. One can obtain a more accurate result by first converting the dividend to a more precise type: in the same example, converting 3456 scaled by 1/100 (34.56) to 3,456,000 scaled by 1/100000, before dividing by 1234 scaled by 1/1000 (1.234), would yield 3456000÷1234 = 2801 (rounded) with scaling factor (1/100000)/(1/1000) = 1/100, that is 28.01 (instead of 30). If both operands and the desired result are represented in the same fixed-point type, then the quotient of the two integers must be explicitly divided by the common scaling factor.

3 Binary vs. decimal

The two most common classes of fixed-point types are decimal and binary. Decimal fixed-point types have a scaling factor that is a power of ten; for binary fixed-point types it is a power of two.

Binary fixed-point types are most commonly used, because the rescaling operations can be implemented as fast bit shifts. Binary fixed-point numbers can represent fractional powers of two exactly, but, like binary floating-point numbers, cannot exactly represent fractional powers of ten. If exact fractional powers of ten are desired, then a decimal format should be used. For example, one-tenth (0.1) and one-hundredth (0.01) can be represented only approximately by binary fixed-point or binary floating-point representations, while they can be represented exactly in decimal fixed-point or decimal floating-point representations. These representations may be encoded in many ways, including BCD.

4 Notation

Main article: Q (number format)

There are various notations used to represent word length and radix point in a binary fixed-point number. 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.

- **Q**f: The "Q" prefix. For example, Q15 represents a number with 15 fractional bits. This notation is ambiguous since it does not specify the word length, however it is usually assumed that the word length is either 16 or 32 bits depending on the target processor in use.[1]

- **Q**m**.**f: The unambiguous form of the "Q" notation. Since the entire word is a 2's complement integer, a sign bit is implied. For example, Q1.30 describes a number with 1 integer bit and 30 fractional bits stored as a 32-bit 2's complement integer.[1][2]

- **fx**m**.**b: The "fx" prefix is similar to the above, but uses the word length as the second item in the dotted pair. For example, fx1.16 describes a number with 1 magnitude bit and 15 fractional bits in a 16 bit word.[3]

- s**:**m**:**f: Yet other notations include a sign bit, such as this one used in the PS2 GS User's Guide.[4] It also differs from conventional usage by using a colon instead of a period as the separator. For example, in this notation, 0:8:0 represents an unsigned 8-bit integer.

5 Precision loss and overflow

Because fixed point operations can produce results that have more bits than the operands, there is possibility for information loss. For instance, the result of fixed point multiplication could potentially have as many bits as the sum of the number of bits in the two operands. In order to fit the result into the same number of bits as the operands, the answer must be rounded or truncated. If this is the case, the choice of which bits to keep is very important. When multiplying two fixed point numbers with the same format, for instance with I integer bits, and Q fractional bits, the answer could have up to $2I$ integer bits, and $2Q$ fractional bits.

For simplicity, many fixed-point multiply procedures use the same result format as the operands. This has the effect of keeping the middle bits; the I-number of least significant integer bits, and the Q-number of most significant fractional bits. Fractional bits lost below this value represent a precision loss which is common in fractional multiplication. If any integer bits are lost, however, the value will be radically inaccurate. Some model-based fixed-point packages[5] allow you to specify a result format different from the input formats. This allows you to maximize precision and avoid overflow.

Some operations, like divide, often have built-in result limiting so that any positive overflow results in the largest possible number that can be represented by the current format. Likewise, negative overflow results in the largest negative number represented by the current format. This built in limiting is often referred to as *saturation*.

Some processors support a hardware overflow flag that can generate an exception on the occurrence of an overflow, but it is usually too late to salvage the proper result at this point.

Modern development cycles include a prototyping phase which examines the potential precision loss and overflow of designs using fixed point calculations before proceeding to physical prototyping.[6]

6 Implementations

Very few computer languages include built-in support for fixed point values, because for most applications, binary or decimal floating-point representations are usually simpler to use and accurate enough. Floating-point representations are easier to use than fixed-point representations, because they can handle a wider dynamic range and do not require programmers to specify the number of digits after the radix point. However, if they are needed, fixed-point numbers can be implemented even in programming languages like C and C++, which do not commonly include such support.

A common use of fixed-point BCD numbers is for storing monetary values, where the inexact values of binary floating-point numbers are often a liability. Historically, fixed-point representations were the norm for decimal data types; for example, in PL/I or COBOL. The Ada programming language includes built-in support for both fixed-point (binary and decimal) and floating-point. JOVIAL and Coral 66 also provide both floating- and fixed-point types.

ISO/IEC TR 18037[7] specifies fixed-point data types for the C programming language; vendors are expected to implement the language extensions for fixed point arithmetic in coming years. Fixed-point support is implemented in GCC.[8][9]

Almost all relational databases, and the SQL query language, 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.[10]

6.1 Other

- libfixmath is a recent open-source library for the manipulation of fixed point numbers,[11] it currently supports Q16.16 and Q0.32 formats and provides an interface similar to math.h.

- GnuCash is an application for tracking money which is written in C. It switched from a floating-point representation of money to a fixed-point implementation as of version 1.6. This change was made to trade the less predictable rounding errors of floating-point representations for more control over rounding (for example, to the nearest cent).

- 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 (partly to save money, but primarily to save power - integer units are much smaller in silicon area than an FPU) and audio decoding requires performance to the extent a software implementation of floating-point on low-speed devices would not produce output in real time.

- 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[12] and GP2X Wiz video game systems use fixed-point arithmetic for the same reason as Tremor and Toast: to gain throughput on an architecture without an FPU.

- 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.

- TeX font metric files use 32-bit signed fixed-point numbers, with 12 bits to the left of the decimal, extensively.

- The dc and bc programs are arbitrary precision calculators, but only keep track of a (user-specified) fixed number of fractional digits.

- VisSim A visually programmed block diagram language supporting a fixed-point block set to allow simulation and automatic code generation of fixed-point operations. Both word size and radix point can be specified on an operator basis.

- Fractint represents numbers as Q2.29 fixed-point numbers,[13] 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. This was done in order for the game to be playable on 386 and 486SX CPUs without an FPU. For compatibility reasons, this representation is still used in modern Doom source ports.

- SystemC provides signed and unsigned fixed point data types.

7 See also

- Binary scaling
- Q (number format)
- Libfixmath - a library written in C for fixed-point math
- Floating-point arithmetic

8 References

[1] Texas Instruments, TMS320C64x DSP Library Programmer's Reference, Appendix A.2

[2] MathWorks Fixed-Point Toolbox Documentation Glossary

[3] VisSim Fixed-Point Toolbox

[4] PS2 GS User's Guide, Chapter 7.1 "Explanatory Notes"

[5] VisSim Fixed-Point User Guidelhttp://www.vissim.com/downloads/doc/EmbeddedControlsDeveloper_UGv80.pdf

[6] Equalis Fixed Point Module documentation.

[7] JTC1/SC22/WG14, status of TR 18037: Embedded C

[8] GCC wiki, Fixed-Point Arithmetic Support

[9] Using GCC, section 5.13 Fixed-Point Types

[10] PostgreSQL manual, section 8.1.2. Arbitrary Precision Numbers

[11] libfixmath, The project page

[12] http://dolphin-emu.org/blog/2014/03/15/pixel-processing-problems/

[13] Fractint, A Little Code

9 External links

- Fixed-Point Arithmetic - An Introduction Representing and implementing fixed-point arithmetic in digital signal processing, by Randy Yates
- A Calculated Look at Fixed-Point Arithmetic at the Wayback Machine (archived June 11, 2002)
- A Calculated Look at Fixed-Point Arithmetic (PDF)
- Working with floating point parameters in an integer world How to represent floating point values in a fixed point processor, by Dinu Madau
- Fixed Point Representation And Fractional Math
- Cladlab: Fixed-Point Mathematics Using fixed-point numbers on an embedded system

10 Text and image sources, contributors, and licenses

10.1 Text

- **Fixed-point arithmetic** *Source:* http://en.wikipedia.org/wiki/Fixed-point%20arithmetic?oldid=639077078 *Contributors:* Damian Yerrick, SimonP, Tedernst, RTC, Michael Hardy, Nixdorf, Julesd, Ciphergoth, Charles Matthews, Furrykef, Grendelkhan, Wernher, EpiVictor, Fredrik, Seth Ilys, Mfc, Tobias Bergemann, ManuelGR, Ryanrs, DavidCary, Sunny256, Jorge Stolfi, Nayuki, Elektron, Unixplumber, Bender235, Saraedum, El C, R. S. Shaw, Bookandcoffee, Forderud, Bluemoose, Kbdank71, YurikBot, Raggha, CyberRax, Gaius Cornelius, Alex Bakharev, EAderhold, Attilios, SmackBot, Fireman biff, Unyoyega, Bmearns, Eskimbot, Jpvinall, Sundaryourfriend, Charles Esson, Michael.Pohoreski, Radagast83, Cybercobra, Doodle77, Petedarnell, Lambiam, Jared Grainger, Yates, Vocaro, CapitalR, CmdrObot, Velle, Novous, Bill.albing, Khatru2, Sageev, Al Lemos, AntiVandalBot, Swampjedi, Gert4gt, Darklilac, Andy.Cowley, Rck289, Mrieser, Whoop whoop, SwiftBot, Gwern, MartinBot, R'n'B, Bvacaliuc, Daniel5Ko, Signalhead, Lights, Philip Trueman, Vipinhari, Berserkerus, Shdwjk, JL-Bot, Ktr101, Northernhenge, Eeekster, DanielPharos, Ra2007, Paulnwt, Addbot, Tsunanet, Scientus, Luckas-bot, Yobot, Jordsan, AnomieBOT, 1exec1, Isheden, Crno srce, ShashClp, Pinethicket, Diannaa, Tzervo, EmausBot, WikitanvirBot, Jsung123, ZéroBot, Tolly4bolly, Sbmeirow, Atcold, Nyarcel, EdoBot, ClueBot NG, Wdchk, Davidc103, Bequal4, Ajaykrishnachoppa, Comp.arch, Gbmhunter, Gheekaru and Anonymous: 102

10.2 Images

- **File:Wikibooks-logo-en-noslogan.svg** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/df/Wikibooks-logo-en-noslogan.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.

10.3 Content license

- Creative Commons Attribution-Share Alike 3.0