Fast Rounding of Floating Point Numbers in C/C++ on Wintel Platform

Laurent de Soras

Updated on 2008.03.08

web: http://ldesoras.free.fr

ABSTRACT

This is a technical paper related to practical aspects of the C/C++ programming language. Number rounding is a rather common mathematical operation, which seems quite simple at a first glance. However both x86-compatible Floating Point Units and C/C++ standard constraints make this operation slower than expected for this level of complexity. For algorithms using extensively this operations, the slowdown could be critical. In this paper, we will review the rounding process and the causes of the slowdown. We will propose and analyse workarounds giving different compromises between accuracy and execution speed.

KEYWORDS

Rounding, trunc, round, floor, ceil, C/C++, x86

0. Document structure

0.1 Table of content

0.2 Revision history

Version	Date	Modifications
–	2003.10.19	Creation
1.0	2004.07.04	Published
1.1	2005.11.27	Typo in `truncate_int()` source code
1.2	2006.06.17	Misc. typos in source code
1.3	2008.03.08	

0.3 Glossary

CPU	Central Processing Unit
FPU	Floating Point Unit
IEEE	Institute of Electrical and Electronics Engineers
WINTEL	Combination of Microsoft Windows operating system and Intel x86 architecture.
x86	Family of the Intel processor, IA32 and IA64 architectures.

1. Why it takes ages with C/C++ instructions

In C language, when you cast a floating point number to an integer, rounding is done according to precise rules. The process is called truncation or *round towards 0*, and consists in removing everything after the decimal point. The compiler achieves this by setting the FPU rounding mode to the right state, truncating the number using `fld` and `fistp` assembly instructions and restoring the previous FPU state. `floor()` and `ceil()` instructions works very similarly, except that the number obtained is still in floating point representation and must be casted if true integer is wanted.

The following code is the `__ftol()` function, automatically included by the Microsoft Visual C++ 6.0 compiler to do the cast.

```
push     ebp                     ; Stack management
mov      ebp, esp                ;
add      esp, -12                ;
wait
fnstcw   word ptr [ebp-2]     ; Retrieve FPU Ctrl Word (rounding mode)
wait
mov      ax, word ptr [ebp-2]
or       ah, 12                  ; Modifies CW to set the rounding mode
mov      word ptr [ebp-4], ax
fldcw    word ptr [ebp-4]        ; Changes rounding mode
fistp    qword ptr [ebp-12]      ; Rounds the number
fldcw    word ptr [ebp-2]        ; Restores old rounding mode
mov      eax, dword ptr [ebp-12] ; Gets lowest 32 bits of the result
mov      edx, dword ptr [ebp-8]  ; Gets highest 32 bits
leave                            ; Stack management
ret
```

There are many flaws here:

- The function is not inlined and does useless stack management.
- It is a generic function giving 64-bit results. If we need only a signed 32-bit number, this is a waste of cycles.
- Whereas the `fistp` instruction executes very quickly, this is not true for the rounding mode change. Indeed, reading and writing the FPU Control Word flushes the instruction pipeline. On modern CPU, it is a cause of major slowdown if the rounding takes place in a tight loop.

2. The fastest way

We will describe in this section two solutions giving fast result and relatively accurate results. The fastest solution depends entirely of the application, context and wanted accuracy.

2.1 FILD/FISTP

The code reviewed in the previous section did absolutely no assumption on the *current* FPU state. That is why it needed to change explicitly the rounding mode. But if we expect the FPU being always set to a certain rounding mode (the one we want), we do not have to save, change and restore the FPU Control Word every time.

However to make this assumption, we need to have a complete control on the program source code, in order to be sure that the rounding mode is consistent in every place the

function is used. If it is changed somewhere, we have to take it into account to keep right the results. FPU Control Word is thread-wise, meaning that the rounding modes of two different threads are independent. This consideration is important in a *host/plug-in* architecture, where codes from multiple sources cooperate in the same process or thread.

So we can set once the rounding mode, do the rounding multiple times, and restore the rounding mode at the end of the operation. When a program starts in Windows, the default FPU rounding mode is always set to *round to nearest integer*. This is a good thing to know, because this rounding mode is quite common. And by playing with the sign and offsets, we could realize other rounding modes – we will review them later. The following function rounds a number in the current rounding mode, therefore *round to nearest integer*, given the above-mentioned conditions.

```
inline int conv_float_to_int (float x)
{
   int      a;
   __asm
   {
      fld     x
      fistp    a
   }

   return (a);
}
```

We can also overload this function for `double` type, with exactly the same code.

2.2 Using memory

The second solution is to use the memory storage format of the floating point numbers. We can achieve the rounding by manipulating the bits of the floating point representation.

2.2.1 Summary of floating point number structure

An IEEE floating point number is made of three bit fields: the sign, the exponent and the mantissa. Sign is always one bit, but its working is different of the classic 2-complement representation of integer numbers. It just means "the rest of the bits represents the absolute value of the number, which is actually negative".

Exponent is the power of two, scaling the mantissa. It is biased: a constant value has been added to make it an unsigned integer. Precision and bias depends on the global width of the number. For 32-bit `float`, exponent is 8 bits and bias B = 127. For 64-bit double-precision floating point numbers, exponent is 11 bits and B = 1023.

Mantissa takes the remaining P bits. It is an unsigned integer. To save bits, the first, most significant bit is never represented. Indeed, it is always set to 1. Therefore we get the following formula:

$$x = \pm(1 + \frac{M}{2^P}) \times 2^{E-B}$$

Where M is the mantissa value, P the mantissa precision, E the exponent value and B the bias. This formula is valid for normal numbers. There are specific cases, like denormal values or NaN (Not a Number) where the meaning of the bits is modified.

2.2.2 The method

The method consist in making appear the rounded number in the mantissa field, without needing to do extra shift or masking. Thus, we just have to store the floating point number in memory and read part of it as it was an integer. How can we do this magic?

The most difficult part is to scale right the mantissa. Indeed, in the natural representation, mantissa bits are aligned to left. The most significant bit set is always at position P, whatever the number magnitude. We want to align it to the right, in order to conform to a fixed point representation. The solution here is to add a huge constant number C, bigger than the input number. Thus, the most significant bit set is always the C'one. Our number can now be aligned to the right, all the bits fixed within the mantissa. But what contain these bits? If C is too big, significant bits of the input will be lost. If it is too small, we will get extra-bits in the lowest part we do not want. During the addition, the rounding rules of the FPU determine the lowest bits of the mantissa. So we have to choose C in order to put the bit 0 of the rounded number on the bit 0 of the mantissa. Therefore the integer number will be calculated according to the FPU rounding rules.

The simplest way to do it is to find the number to add to 1.0 in order to get a mantissa full of 0's excepted the lower bit (1 in integer representation). Absolute value of the mantissa is P bits. We know that the most significant bit is implicit. So we have to obtain the binary number 1000...001 within P + 1 bits. Thus, we choose:

$$C = 2^P$$

What is the range of the input? For the upper bound, it depends on the mantissa resolution: it has to be less than 2^P . The lower bound is 0. Indeed, this method will fail with negative numbers. Why ? Because by adding negative numbers to C, we obtain numbers in the form 0111...xxx. The most significant bit is now at the position P – 1 instead of P, shifting our number. There is a solution to this problem: choosing C higher, still containing zeros in the lower bits, and with the same most significant bit. The optimal solution is in the binary form 11000...000, because we get the widest range for both signs. So:

$$C_S = 3 \times 2^{P-1}$$

The valid range becomes $\left[-2^{P-1} ; 2^{P-1}-1\right]$

In the code below, we have to split C_S in two parts because it is not representable as an integer number ; C/C++ standard only guarantees that `long` type is at least 32-bit wide. Also, the bits we have to read are located at the beginning of the number, because the memory model on x86 is Litte Endian (lower significant bit first). The range checks are done in two times: the first test is rough because we do not assume anything on the used rounding mode. The second test verify that both original and rounded numbers are of the same sign. This should detect minor range errors, which would have passed the first tests.

```
inline int conv_float_to_int_mem (double x)
{
   const int      p = 52;
   const double   c_p1 = static_cast <double> (1L << (p / 2));
   const double   c_p2 = static_cast <double> (1L << (p - p / 2));
   const double   c_mul = c_p1 * c_p2;
   assert (x > -0.5 * c_mul - 1);
   assert (x < 0.5 * c_mul);
   assert (x > static_cast <double> (INT_MIN) - 1.0);
   assert (x < static_cast <double> (INT_MAX) + 1.0);

   const double   cs = 1.5 * c_mul;
   x += cs;

   const int      a = *(reinterpret_cast <const int *> (&x));
   assert (a * x >= 0);

   return (a);
}
```

2.3 Why it fails

The two described methods are quite fast. However they rely on the processor rounding mode, which is *round to nearest integer* by default. So we wrote round-to-nearest functions? Not quite. Actually the exact rounding mode is really called *round to closest even integer*. Practically, it is almost the same thing. Except for the numbers which are exactly half-way between two integers: $\frac{3}{2}$, $\frac{25}{2}$, $-\frac{7}{2}$, etc. In this case, the x86 rounding mode chooses the closest even integer, whereas the mathematical operation one generally expects chooses the immediate greater integer.

To point out the differences, here are a few examples of both rounding modes:

Number	To nearest integer	To closest even integer
–2.25	–2	–2
–1.75	–2	–2
–1.50	**–1**	**–2**
–1.25	–1	–1
–0.75	–1	–1
–0.50	0	0
–0.25	0	0
0.25	0	0
0.50	**1**	**0**
0.75	1	1
1.25	1	1
1.50	2	2
1.75	2	2
2.25	2	2

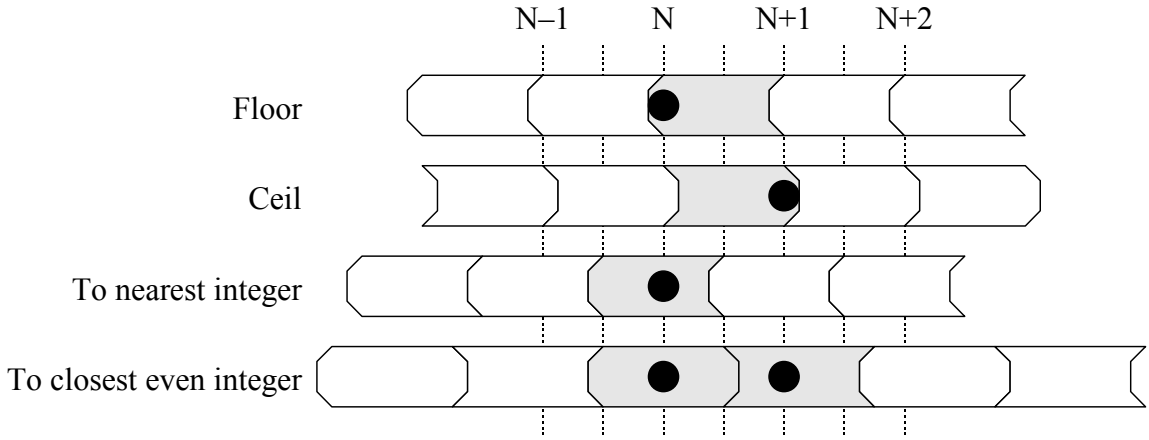
Depending on the application, this particularity may be critical or not. We will expose below a solution to obtain a real *to-nearest-integer* mode.

3. Accurate method

The first concept to understand is the model of rounding for each mode. This model applies to all modes except truncation. It is mainly characterised by a *periodicity* and a *scheme*. The scheme is like a pattern for rounding, which can be repeated with a certain period within the domain. To make things clear, we will describe the models for each pattern.

Rounding mode	Periodicity	Scheme
Towards minus infinity (`floor`)	1	$[N\ ;\ N+1[\to N$
Towards plus infinity (`ceil`)	1	$]N-1\ ;\ N] \to N$
To nearest integer	1	$[N-\frac{1}{2}\ ;\ N+\frac{1}{2}[\to N$

Rounding mode	Periodicity	Scheme
To closest even integer	2	$\begin{cases} [2N-\frac{1}{2}\ ;\ 2N+\frac{1}{2}] \rightarrow 2N \\ ]2N+\frac{1}{2}\ ;\ 2N+\frac{3}{2}[\rightarrow 2N+1 \end{cases}$

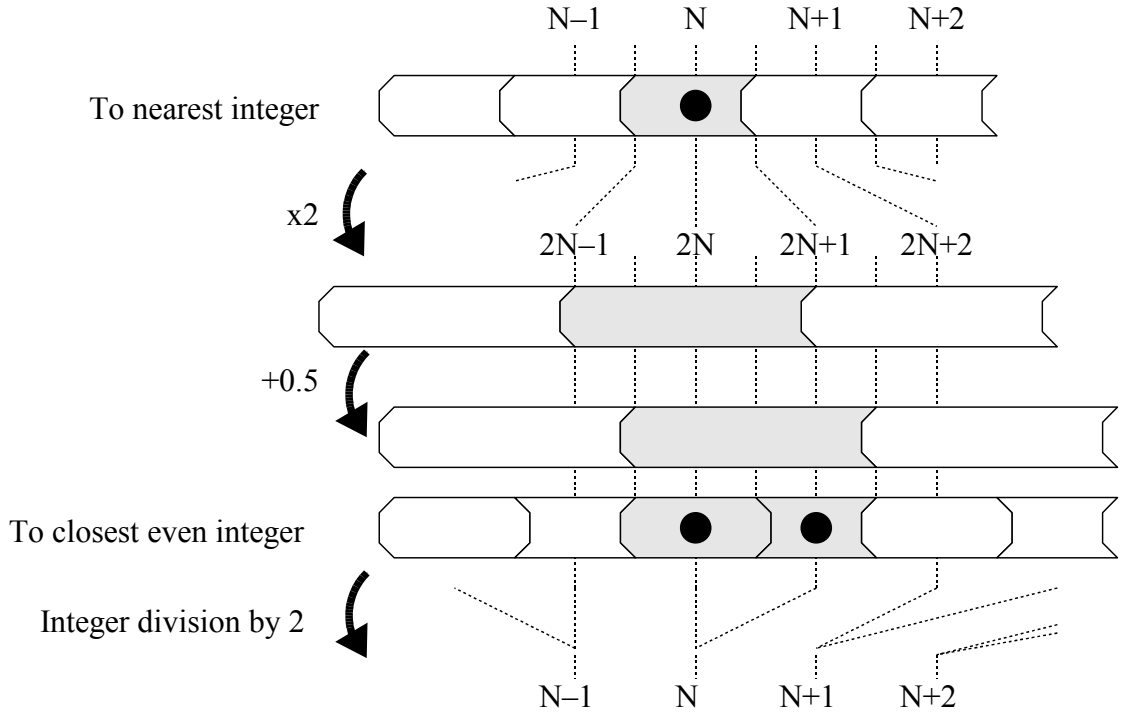


3.1 Concept

With the *to closest even integer* mode, we can achieve easily and accurately the other rounding modes. Indeed, we observe a similarity in the rounding schemes. They have globally the same shape. By shifting the ranges, scaling the input and playing with sign, it is possible to always operate in the *closest even integer* mode and get back to the original range to extract the result.

3.2 Round to nearest integer

We will detail only this mode, others are simple to reconstruct once the concept is understood. The drawing below shows the transforms we apply on the target mode scheme to reach the *closest even integer* mode.



The last step is an integer division, but we have to be careful. In order to make it work in the negative range, the division must always round *towards minus infinity*. C/C++ division uses the truncation method, so it has to be avoided. Instead, we use an arithmetical shift to the right, which has the right property. Second pitfall, do not use the >> C/C++ operator, because standard does not guarantee that sign of negative numbers will be preserved. The use of assembly language is unavoidable here if we want to keep execution as fast as possible.

```
int round_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_to_nearest = 0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fadd  round_to_nearest
      fistp i
      sar   i, 1
   }

   return (i);
}
```

The integer division reduces the range of the rounded numbers from -2^{30} to $2^{30}-1$. It is possible to obtain the full 32-bit range by temporarily using 64-bit arithmetic:

```
int round_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_to_nearest = 0.5f;
   int            i;
   __int64        tmp;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fadd  round_to_nearest
```

```
      fistp qword ptr tmp
      mov   eax, dword ptr tmp
      sar   dword ptr tmp + 4, 1
      rcr   eax, 1
      mov   i, eax
   }

   return (i);
}
```

The same trick can be used for the other rounding functions.

3.3 Round towards minus infinity (floor)

The only difference between floor and the *to nearest integer* mode is the offset.

```
int floor_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_towards_m_i = -0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fadd  round_towards_m_i
      fistp i
      sar   i, 1
   }

   return (i);
```

3.4 Round towards plus infinty (ceil)

This mode is a kind of mirror of the *towards plus infinity* mode.

```
int ceil_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_towards_p_i = -0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)
      fsubr round_towards_p_i
      fistp i
      sar   i, 1
   }

   return (-i);
}
```

3.5 Truncate

Truncate mode is the default rounding mode in C. This consists in suppressing the decimals, giving a *towards minus infinity* mode for positive numbers, and *towards plus infinity* mode for negative numbers.

```
int truncate_int (double x)
{
   assert (x > static_cast <double> (INT_MIN / 2) - 1.0);
   assert (x < static_cast <double> (INT_MAX / 2) + 1.0);

   const float    round_towards_m_i = -0.5f;
   int            i;
   __asm
   {
      fld   x
      fadd  st, st (0)

      fabs
      fadd  round_towards_m_i
      fistp i
      sar   i, 1
   }

   if (x < 0)
   {
      i = -i;
   }

   return (i);
}
```

CONCLUSION

We have reviewed several fast methods to achieve fast rounding, assuming we know the default processor rounding mode, and without the need of changing it. It is also possible to extend this idea to produce any [desired rounding mode / processor rounding mode] combinations.