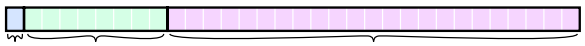
The *fluint8* Software Integer Library

Jim McCann*
TCHOW llc

Tom Murphy VII†
tom7.org foundation



$$= -1^S \cdot 2^{E-127-23} \cdot \left(2^{23} + M\right)$$

Figure 1: *Our library performs unsigned integer operations using only arithmetic operations on IEEE754 floating point numbers stored in binary32 format (pictured).*

Abstract

We present *fluint8*, a library for performing integer math, including basic arithmetic and bitwise logical operations, using only basic floating point operations.

CR Categories: $(1.10100100011)_2 \times 2^{11}$ [Software]: Software Engineering—Coding Tools and Techniques

1 Introduction

There are a surfeit of libraries that exist to perform floating point operations on processors that only support integer math. This is unsurprising, as many such processors exist – from ancient 286's to modern embedded microcontrollers. These libraries use many integer instructions to emulate the action of a floating point unit, providing correct and useful (if slow) results.

We present a small header-only C library to emulate integer operations – specifically 8-bit unsigned integer operations – using standard IEEE 754 single-precision (binary32) floating point math. Our presented operations have been designed to be succinct but also pleasantly puzzling.

As far as we are aware, no processor exists for which this library would be required. However, perhaps you should consider that a challenge.

2 Floating Point

An IEEE 754 single-precision floating point number (binary32 format) is stored as a sign bit, a 8-bit exponent, and a 23-bit mantissa (Figure 1). Except for special cases, the number represented by a floating point number with sign S, exponent E, and mantissa M is [‡]:

$$-1^S \cdot (1.M)_2 \cdot 2^{E-127}$$

Particularly, notice that the leading "1" in the fraction is implicit in the representation (it is implied by a non-zero exponent[§]).

This means that the range of integers that can be represented (without loss of precision) is

$$[-2^{24}, 2^{24}] = [-16777216, 16777216]$$

which, conveniently, is far more than the $[0, 255]$ range needed for storing 8-bit unsigned integers.

When floating point operations result in numbers that cannot be accurately represented, the results are rounded according to the current rounding mode. The default rounding mode assumed in this paper is *roundTiesToEven*. I would say that it does what you expect, but floating point numbers seldom manage that feat. Regardless, this rounding mode means that whenever a value is exactly halfway between two representable numbers, the number with a least-significant-bit of 0 is picked.

Rounding and precision loss leads to this fun fact:

```
16777216.0f + 1.0f - 1.0f == 16777215.0f
16777216.0f - 1.0f + 1.0f == 16777216.0f
```

(Hot take: floating-point operations are non-commutative.)

3 The *fluint8* Library

The fluint8 library provides all of the mathematical and logical operations one expects on 8-bit integers, using only floating point addition, subtraction, multiplication, and division – other than a loop with fixed bounds which could be unrolled by the compiler, no conditionals or function calls are required.

Full source code for the library (and this paper) are available at `https://github.com/ixchow/fluint8`.

In this section we go through the library operation by operation, explaining how each function works.

*e-mail: ix@tchow.com

†ə-mail: tom7@tom7.org

‡or at least this is what wikipedia says, so I'm going with that, and it seems to work out.

§An all-zero exponent is used for special numbers like zero, but we're not going to go into that. Wait, we just did.

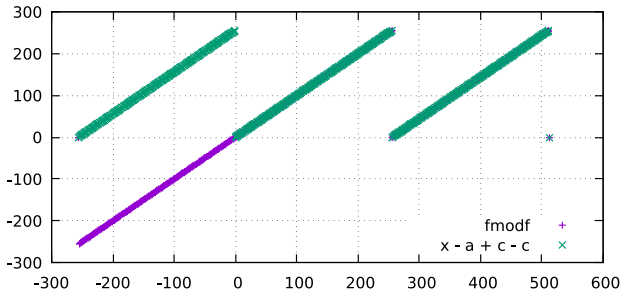


Figure 2: *Comparing* `fmodf(x, 256.0f)` *to the expression* `x - 127.5f + 3221225472.0f - 3221225472.0f` *over the range* `[-256.0f, 512.0f]`. *Notice that the expression is positive for negative numbers, making it more useful for simulating integer rollover. Plot created using gnuplot.*

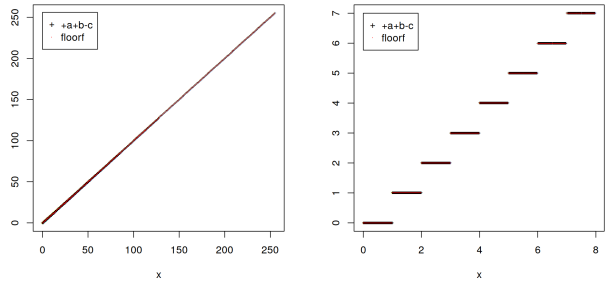


Figure 3: *Comparing* `floorf(x)` *to the expression* `x + 0.50390625f + 8388608.0f - 8388609.0f` *over the all quotients* `[0.0f, 255.0f] / [1.0f, 255.0f]`. *The functions match at all plotted points. Plot created using R.*

3.1 Storage Format

Our library represents unsigned 8-bit integers as their equivalent floating point values. In other words, the value `uint_t(127)` is represented as `127.0f`. This straightforward equivalence is convenient when writing basic mathematical functions.

In order to support, e.g., reading data from files, our library includes functions that convert between floating point numbers and bit-patterns of their equivalent 8-bit unsigned representation.

```
void fu8_to_bits(float a, void *out) {
    a += 8388608.0f;
    memcpy(out, &a, (size_t)(1.0f));
}
```

The function `fu8_to_bits` adds a large enough number to `a` that its mantissa's least-significant bit now represents 1. Essentially, the code is shoving the integer information stored in `a` to the least-significant-byte of the representation, and then copying[¶] it out to the destination.

The same trick works when setting a floating point number from an integer bit pattern:

```
float fu8_from_bits(void const *from) {
    float a = 8388608.0f;
    memcpy(&a, from, (size_t)(1.0f));
    return a - 8388608.0f;
}
```

[¶] The astute reader will notice that we've taken care to avoid using an integer constant as a parameter to `memcpy`. Presumably on processors without integer support `size_t` must be a floating-point type. And, yes, we promised above not to use function calls, but it's hard to copy a byte without integer types.

3.2 Arithmetic Functions

Our library implements +, −, ⋆, /, +, and − by treating floating point numbers as real numbers; an approach that often works, but requires some post-processing to deal with rollover:

```
float fu8_add(float a, float b) {
    float x = a + b;
    x += x - 127.5f + 3221225472.0f -
3221225472.0f;
    return x;
}
```

Here, the second line of the function computes `fmodf(x, 256.0f)` by rounding `x` to the next-greater multiple of `256.0f` (rounding is forced by adding `3221225472.0f` to make the least-significant-digit of the number have value 256), then subtracting this rounded value. Don't believe us? Examine the convincing graph in Figure 2.

Most of the remaining math functions follow this "operate then wrap" paradigm:

```
float fu8_sub(float a, float b) {
    float x = a - b;
    x -= x - 127.5f + 3221225472.0f -
3221225472.0f;
    return x;
}
float fu8_mul(float a, float b) {
    float x = a * b;
    x -= x - 127.5f + 3221225472.0f -
3221225472.0f;
    return x;
}
float fu8_pos(float a) {
    return a;
}
```

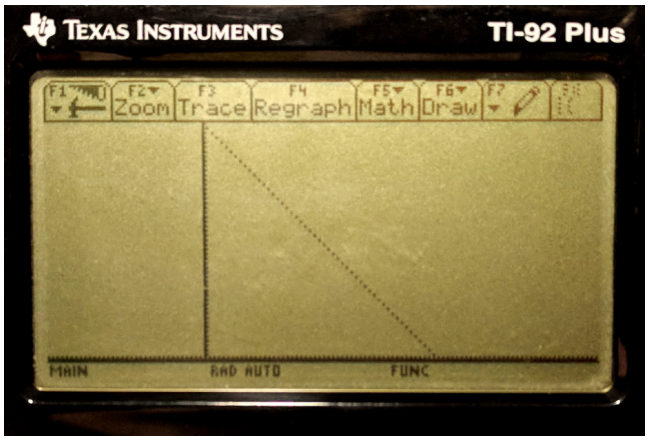


Figure 4: *graph of* `255.0f - x` *(the bitwise complement of* `x`*). Plot created using a TI-92 Plus graphing calculator.*

```
float fu8_neg(float a) {
   return (a + 127.5f + 3221225472.0f -
3221225472.0f) - a;
}
```

Our add-and-subtract modulus function always returns a positive number, which we take advantage of in the subtraction function. The negation function uses a similar trick to either subtract `a` from `256.0f` if positive or from `0.0f` otherwise.

```
float fu8_div(float a, float b) {
   float x = a / b;
   x = x + 0.50390625f + 8388608.0f -
8388609.0f;
   return x;
}
```

The division function computes the floor of a value by rounding that value to the next-largest and subtracting one (Figure 3). In this particular instance, the constant required is small enough that the subtraction of one can be rolled into the subtraction of the large constant (we choose this over rounding down mostly for `a e s t h e t i c` reasons).

For the division and modulus operations, fluint8 diverges from the normal behavior of integer instructions when the denominator is zero. A typical processor triggers a fault upon integer division by zero, but IEEE 754 instead returns one of the special values *Infinity*, *-Infinity* (or *NaN*) and continues calculating. After this point, fluint8 may produce nonsense results. However, this is strictly compliant with the C and C++ standard, for which integer division by zero is formally undefined behavior.[‖]

[‖] "If the second operand of / or % is zero the behavior is undefined." — C++03 5.6.4

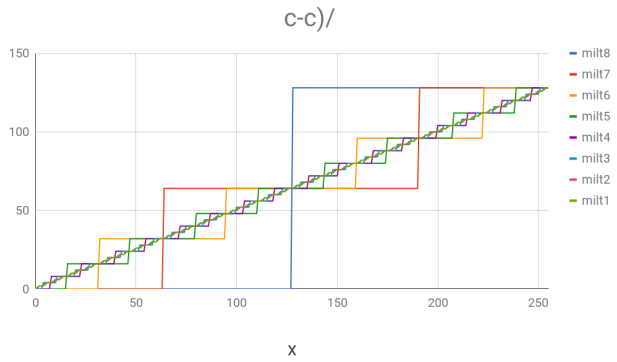


Figure 5: *The value of* `a + 1.0f + c-c)/ 2.0f` *for* `c` *ranging from* `2147483648.0f` *(milt8) to* `16777216.0f` *(milt1). Plot created in Google Sheets.*

3.3 Bitwise Operations

Things really get interesting when we begin to look at bitwise operations, which aren't standard operations on floating point numbers[**].

Let's begin with bitwise negation (`~`). This one is relatively easy to explain – an unsigned 8-bit integer plus its bitwise complement is always 255, which makes negation as easy as subtraction (Figure 4):

```
float fu8_not(float a) {
   return 255.0f - a;
}
```

Things get a bit more interesting when computing bitwise and (`&`):

```
float fu8_and(float a, float b) {
   float ax, bx, x = 0.0f;
   for (float c = 2147483648.0f;
          c != 8388608.0f; c *= 0.5f) {
      a -= ax = (a + 1.0f + c-c)/ 2.0f;
      b -= bx = (b + 1.0f + c-c)/ 2.0f;
      x = 0.5f * x + ax * bx;
   }
   return x;
}
```

Note that though this is presented as a loop, the loop has constant bounds and could be unrolled by a compiler into eight repetitions of the same code.

This code peels apart `a` and `b` bit-by-bit using a similar trick to the floating point modulus idea we explained earlier. In this case, however, we've formatted the code so it has a little waving guy in it, who we will call Milt:

```
c-c) /
```

[**] Though they seem well-defined; maybe a language-designer oversight?

Though it looks like Milt is just hanging out, minding its own business, and not changing the value of the expression, Milt is in fact doing something surprisingly nonlinear (Figure 5).

So when Milt's eyes are `2147483648.0f`, it is extracting twice the value of the MSB of `a`, which in turn is stored in `ax` and subtracted from `a`. In this way, the code peels off each successive most-significant bit from `a` and `b` and accumulates their product into the final result.

This leaves only the mystery of why `x` is being divided by two each loop iteration. But this isn't a mystery at all. Consider computing `171 & 226`. Notice that on the first iteration, the product `128.0f * 128.0f` would be added to `x`; the multiplications by a factor of `0.5f` on each subsequent iteration simply – in aggregate – bring it to the correct result of `128.0f`.

a	b	ax	bx	x	c
171	226				
43	98	128	128	16384	2147483648
43	34	0	64	8192	1073741824
11	2	32	32	5120	536870912
11	2	0	0	2560	268435456
3	2	8	0	1280	134217728
3	2	0	0	640	67108864
1	0	2	2	324	33554432
0	0	1	0	162	16777216

4 Optimization

The previous routine computes a bitwise function a single bit at a time. While clean and logically motivated, it seems possible to improve bandwidth by processing multiple bits at once. As a proof-of-concept, the following routine computes the exclusive-or function (`^`) for two `fluints` in the range 0-3.

```
float fu8_xor2bit(float a, float b) {
  return truncf(
    fmod(((a * -1.89269124e+30f) +
          (b * -1.09500709e+35f)) *
           -1.14474456e-18f,
          4.77664232f));
}
```

The routine works by computing a very noisy function that just happens to come close to the correct results for all 16 possible inputs. Don't believe us? Barbecue your eyes of Figure 6. This routine uses `fmod` and `truncf`, but the same loss-of-precision tricks from before can likely be used to avoid them. Four similar expressions can be composed to compute 8-bit exclusive-or, and/or it may be possible to find expressions that compute more bits at once.

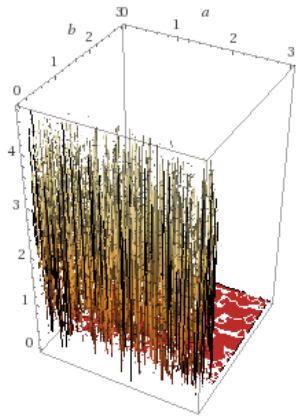


Figure 6: *3D graph of* $((a \times -1.89269124 \times 10^{30} + (b \times -1.09500709 \times 10^{35})) \times -1.14474456 \times 10^{-18}) \bmod 4.77664232$ *with* a *and* b *each ranging from* 0–3.0. *Plot created using Wolfram Alpha Computational Knowledge Engine.*

5 Not Optimization

The library should not be used with compiler options such as `-ffast-math` (which may assume properties that do not hold of IEEE754, like commutativity). This often optimizes away the entire fluint8 code, causing it to misbehave (whoa, not *that* fast, buddy).

6 Applications

While direct hardware applications of this technology are currently theoretical (Section 7), there is at least one compelling application for *fluint8* in everyday practice. Many pieces of software use primarily integer instructions, letting the floating point unit lie completely dormant for many nanoseconds at a time. As has been known for decades, while the integer registers and functional units are occupied, the otherwise idle floating point units can be used to perform useful tasks. Unfortunately, no useful tasks were known for floating point instructions. Now, we see that normal integer operations (such as cryptography) can be simulated with these instructions and registers. With compiler support, a separate thread of non-interfering floating point instructions could be emitted, and arbitrarily interleaved with the integer ones, scheduled only when the integer unit would likely stall for a data dependency. This technique is *ultrathreading*, since it is one step better than *hyperthreading*. For example, we hypothesize that during normal web browsing on a modern x86-64 processor, such code could mine as much as one Bitcoin per 107 trillion years, with no more than a 1% loss of efficiency.

7 Future Work

Design a processor for which this library is relevant.