

Coffee into Bugs: Floating Point from Scratch

*“A software engineer is a device for converting coffee into bugs.”*

Dave Allison

Show of hands, who understands floating point? Be truthful now. Like most software engineers, you probably use them all the time without caring how they work. I’m going to try to explain how floating point actually works[[1]](#footnote-1).

While working on a retro 6502 project[[2]](#footnote-2), I discovered that I needed to support floating point math with no hardware to help. Rather than trying to copy existing code, I thought I’d figure it out from scratch. This is the result,

The article describes:

1. The theory of fixed point numbers.
2. The theory of floating point numbers.
3. IEEE 754 single precision numbers.
4. C functions for math operations on IEEE 754 single precision floating point numbers.
   1. Conversion from integer.
   2. Conversion to integer.
   3. Multiplication, division, addition and subtraction.
   4. Comparisons.
5. Conversion from ASCII to floating point.
6. Conversion from floating point to ASCII.

Throughout the article I will provide functions, written in C[[3]](#footnote-3), to perform all the operations in the hope that grokking the operation of a C function is clearer than understanding an English[[4]](#footnote-4) description. That’s how I think, anyway.

All of this is my own work. I am not claiming to have invented anything and haven’t copied the code from anywhere else. The code probably isn’t optimal but should suffice to provide a decent description of how floating point works.

# Fixed point numbers

Before diving into code, first we need to cover some theory.

Integers are the most common representation that software uses, but what if we want to represent real numbers, with fractions of whole numbers in them? Well, an integer can be thought of as having a **binary point** just to the right of its least significant bit (20) with a bunch of un-stored zero bits to its right. The number 0x12345678 can be thought of as 0x12345678.0000… (going on for infinity number of bits).

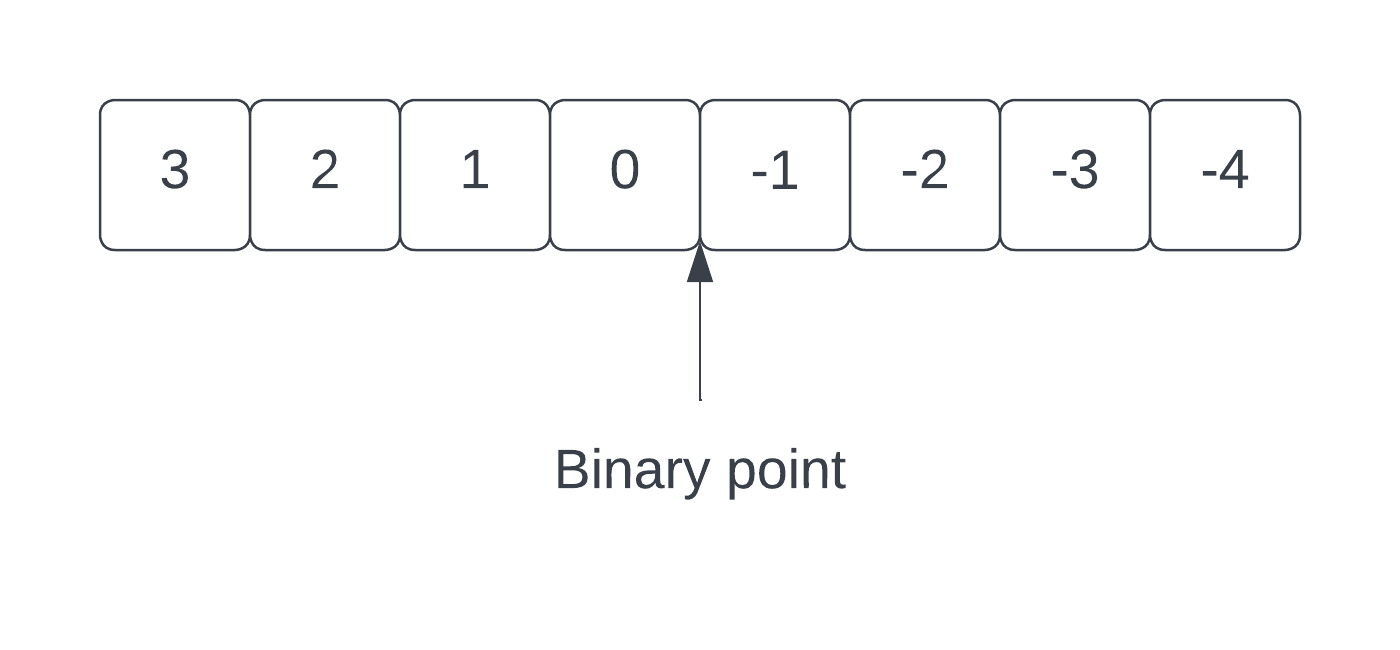
That’s not much use because it’s still an integer, but how about using the bottom bit of the number to represent, instead of the value 20 (1), the value 2-1 or 0.5? This has moved the binary point to the position between bits 0 and 1 and given us the possibility of representing a number between integers.

Let’s take the 8-bit binary number 00001111. As an integer, that is the value 15 (20 + 21 + 22 + 23 = 1 + 2 + 4 + 8 = 15). If we decide that we want to support one bit to the right of the binary point, the real number 15.0 would be represented by 00011110, or 0001111.0. Notice that we’ve shifted the number one bit to the left to make room for the fractional part of the number (1 bit). Whereas the original 8-bit number could hold the integer value from 0 to 255 in increments of 1, the new, real number can hold the numbers 0 to 127.5 in increments of 0.5. So, by using 1 bit for the fractional part, we’ve taken one bit away from the range of numbers we can represent in the same number of bits.

Using this representation, the binary number 00011111, or 0001111.1 is 15.5.

It would be more useful if we could have more than 1 bit for a fractional part. So, let’s do that, let’s use 4 bits. This will allow us to have a lot more fractions, but at the expense of only allowing 4 bits for the integer part of the number.

Just as each bit in an integer, going left, is an increasing power of 2, each bit in the fractional part (to the right of the binary point), going right, is a decreasing power of 2, starting at -1 at the leftmost bit and decreasing by 1 as we go right. Given an 8-bit number with 4 bits for the integer and 4 bits for the fraction, the bits represent the following powers of 2:



Now we can have numbers in the range 0000.0000 to 1111.1111, or 0.0 to 15.9375 (23 + 22 + 21 + 20 + 2-1 + 2-2 + 2-3 + 2-4 = 8 + 4 + 2 + 1 + 0.5 + 0.25 + 0.125 + 0.0625). The **precision** of the numbers we can represent is 0.0625.

This is known as **fixed point** representation, because the binary point is always in the same position.

Performing arithmetic on fixed point numbers is exactly the same as that used for integers. You just have to use the same representation for all the numbers. For example, consider adding 2.5 and 1.25. In our 8-bit fixed point binary (with the binary point shown), these are:

2.5 = 0010.1000

1.25 = 0001.0100

00101000 + 00010100 = 00111100 = 0011.1100 = 3.75 (21 +20 + 2-1 + 2-2)

Multiplication and division by *shifting* are also exactly the same. Consider 7.5 \* 2:

0111.1000 << 1 = 1111.000 = 15.0 (<< 1 means shift left by 1 bit, or multiply by 2).

Arbitrary multiplication and division are a little more complex[[5]](#footnote-5).

Multiplication produces a result with twice the number of bits in both the integral and fractional parts so will require twice the storage for the result and a right shift to move the binary point back to the correct place.

For example, consider 2.5 2.25. In binary:

0010.1000 0010.0100 = 0x28 0x24 = 0x5a0 = 00000101.10100000 = 5.625

Notice that the result is 16 bits long and the binary point is at bit 8 because we have doubled the number of bits in the fraction. To obtain the 8-bit result, shift it right by 4 bits, giving 0101.1010.

Division of fixed point numbers without losing precision requires that the dividend be first scaled up to into the the high bits of twice the amount of memory and divided by the unshifted divisor. Consider division without scaling:

5.625 2.5 = 0101.1010 0010.1000 = 0x5a 0x28 = 2

The actual result of 5.625 2.5 is 2.25, so we’ve lost precision. However, if we first scale the dividend up to a 16-bit number and divide we get:

0101101000000000 = 0x5a00 0x28 = 0x240 = 0000001001000000

The result of 2.25 in binary is 00100100 and you can see that this is contained in the result but shifted left by 4 bits. Therefore, like multiplication, our binary point is at bit 8 and shifting right by 4 bits will yield the correct 8-bit result without loss of precision.

To remove the fraction and leave the integer (an *abs* function), just shift the number right by 4 bits.

Obviously, a single byte with 4 bits for the integral part and 4 for the fraction is very limiting, so we would want to use more bits. Fixed point arithmetic is always a tradeoff between the range of numbers that can be represented (the number of bits to the left of the binary point), and the precision – the number of bits to the right of the point. I found, when working on a 3D graphical game on an Acorn Archimedes ARM3 computer back in the 1990’s, that using 14 bits for the fraction and 18 bits for the integer gave a reasonable range and accuracy for 3D vectors.

It’s probably obvious, but the representation chosen for how numbers are handled is purely a function of the program manipulating them. There is nothing special about any chosen binary point location or the use of fixed point arithmetic at all. There are no special machine instructions or arithmetic units needed to use them.

# Floating point numbers

So, why don’t we use fixed point arithmetic in modern computers? The answer is that they are too inflexible. Having a fixed number of bits for the integral and fractional parts will only work well for a limited set of problems. If a program needs a bigger range of numbers than the chosen fixed point representation will allow, there’s no easy way to enable that. Likewise, the precision of the fixed point number may be insufficient.

An alternative to fixed point numbers is the **floating point** number representation. Floating point means that the binary point can be in any position in the number – it floats. The representation is often compared with **scientific** or **exponential** representation seen in mathematics, and indeed the comparison is accurate.

In regular math, a scientific number is a real number multiplied by 10 to the power of something. This has two parts, the real number (sometimes called the **mantissa**) and the power-of-ten multiplier, called the **exponent**. For floating point representation in a computer, we use powers of 2 rather than 10.

The **value** of a floating point number is the *mantissa* shifted left or right by the number of bits specified by the *exponent* (multiplication and division by 2 is the same as shifting). If the exponent is positive, the mantissa is shifted left, and, for a negative exponent, to the right. If we place the binary point all the way left, just to the right of the most significant bit, we can represent any number from 0.0 to just less than 2.0 in the mantissa.

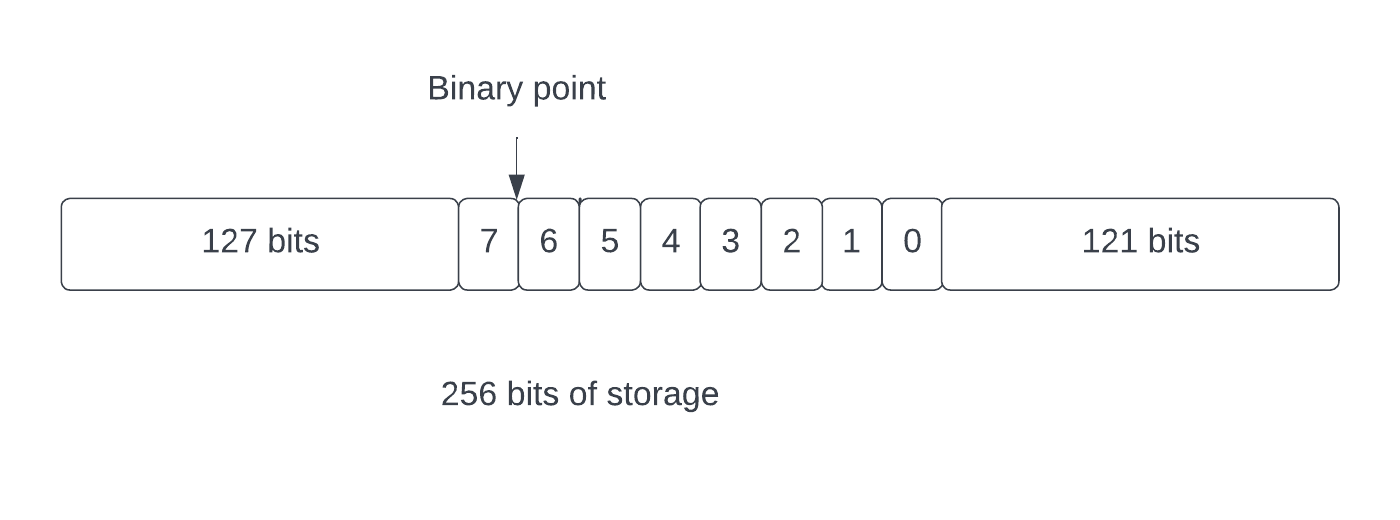
Consider an 8-bit mantissa and an 8-bit exponent (both unsigned for now). If we place the binary point to the right of bit 7 in the mantissa, we can represent the numbers 0.0000000 to 1.1111111, or 0.0 to 1.992188 in decimal. We have an 8-bit exponent, so this can be multiplied by any power of 2 from 20 to 2255.

Using this form, we can represent a 256-bit number (with the max value of 1.15 \* 1077) without actually using 256 bits – we are only using 16 bits. However, we obviously can’t represent all values in the full range as that would require actually using 256 bits!

Using an unsigned number for the exponent won’t accommodate numbers less than 1, so instead of unsigned, let’s use a *signed* 8-bit exponent. This will allow us to shift the mantissa to the left for positive exponent values, or to the right for negative values.

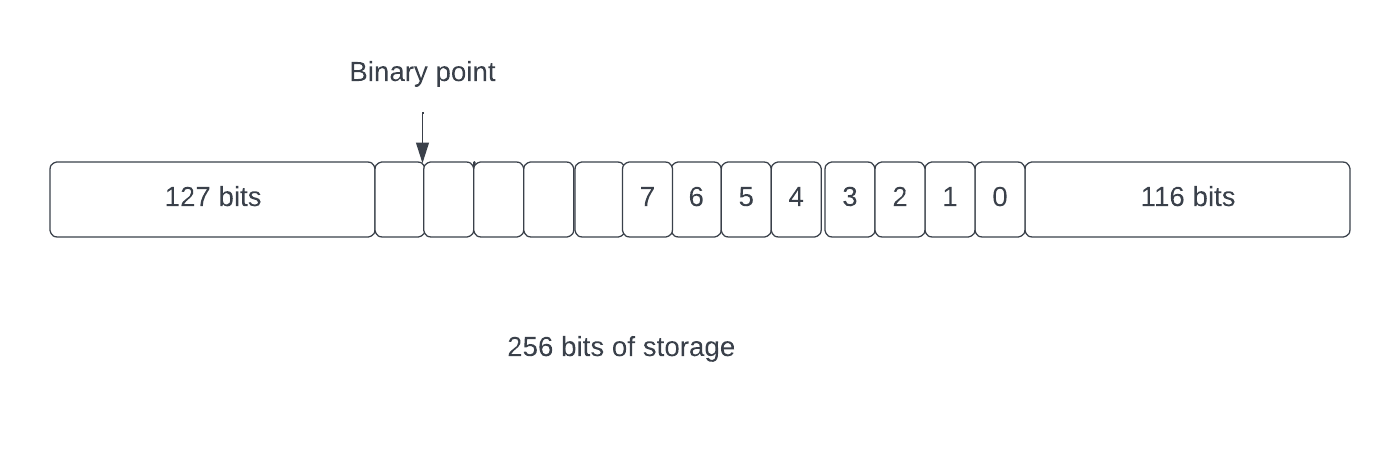
A signed 8-bit exponent allow us to use values of -128 to 127.

Let’s consider converting the 16-bit floating point number (8-bits for mantissa, 8 for exponent) into a 256-bit fixed point number. In order to allow us to use both positive and negative exponents, let’s place the binary point in the middle of 256 bits of storage – between bits 127 and 128. Since our mantissa has its binary point to the right of bit 7, we line up the binary points and place the high bit of the mantissa at bit 128 of the 256 bits.



Here we have the 8 bits of mantissa (numbered 7…0) in the middle of the 256 bits, with the binary point between bits 6 and 7. We have 127 zero bits to the left of bit 7 of the mantissa (bit 7 is at bit position 128) and 121 bits to the right of the 0 (128-7).

If our exponent is, say, -5, we shift the number 5 bits to the right. The binary point stays where it is, in the exact center of the 256 bits, but we multiply our number by 2-5 (divide by 25), leaving zeroes in all the vacated bits around the binary point[[6]](#footnote-6).



Notice that our example of putting the binary point to the right of the most significant bit allowed us to have a 1 or a 0 in bit 7. Using a value of 0 in bit 7 doesn’t buy us anything since any number between 0 and 1 can be represented as a number between 1 and 2 with a different exponent. For example, 0.5 can be represented as 1.0 2-1. So that’s basically wasting a bit in our number. This is what a practical floating point representation does – it moves the binary point to the left of the highest bit and adds an *implicit 1* to its left, that is not stored in the number.

What about negative mantissas? Whereas we can use a 2’s complement number for the exponent, the mantissa is really a sign-and-magnitude quantity. In decimal, the number -1.5 is just the number 1.5 with a symbol telling us it’s negative. In all mathematical operations we do on it, the sign is treated as independent of the value (we take it into account, of course, but it doesn’t affect the operations on the value itself). Instead of a 2’s complement mantissa, we use a single bit to specify whether the mantissa is positive or negative and keep the mantissa itself as a positive value. Let’s choose to use a value of 0 in the sign bit to mean positive, and 1 for negative.

To summarize, a floating point number comprises the following parts:

1. A signed 2’s complement exponent specifying a power of 2
2. A positive mantissa with its binary point to the left of its high bit
3. An implicit 1 to the left of high bit in the mantissa
4. A sign bit specifying if the mantissa is positive or negative.

Thus, a floating point number’s value is:

**±1.x 2e**

Where **x** is the value of the mantissa and **e** is the exponent. That is, **1.x** shifted left or right by **e** bits.

Let’s take a couple of examples, with our 8-bit mantissa and 8-bit exponent. The mantissa is in binary with its binary point to the extreme left and there’s an implicit (not stored) 1 to the left of the binary point. So, a mantissa value of 10000000 is, with its implicit 1 and binary point, 1.10000000, or 20 + 2-1, or 1.5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Sign** | **Fixed point binary** | **Mantissa** | **Exponent** |
| 1.5 | 0 | 1.1 | 1.10000000 | 0 |
| 0.5 | 0 | 0.1 | 1.00000000 | -1 |
| -1.25 | 1 | 1.01 | 1.01000000 | 0 |
| 4.75 | 0 | 100.11 | 1.00110000 | 2 |
| -4.75 | 1 | 100.11 | 1.00110000 | 2 |

Picking on 4.75, we see that the binary for this is 100.11, or 22 + 2-1 + 2-2, or 4 + 0.5 + 0.25, or 4.75. The mantissa must have a single 1 to the left of the binary point, called a **normalized** mantissa. To normalize this mantissa, we have to shift it to the right by 2 bits, thus giving the exponent the value of 2.

If the mantissa must be normalized, with a 1 to the left of its binary point, the steely-eyed reader might wonder how we can represent the value 0. The answer is that we use a special value of all zero bits in the mantissa and exponent to represent 0. This is obviously an exception to the normalization rules. A practical floating point system will also have other exceptions to represent **Not A Number (NaN)** and other special values.

Also, if you are really one the ball, you might recognize that if we have all bits 0 in the mantissa and exponent, we still have a sign bit, thus we can have both +0 and -0. Interesting.

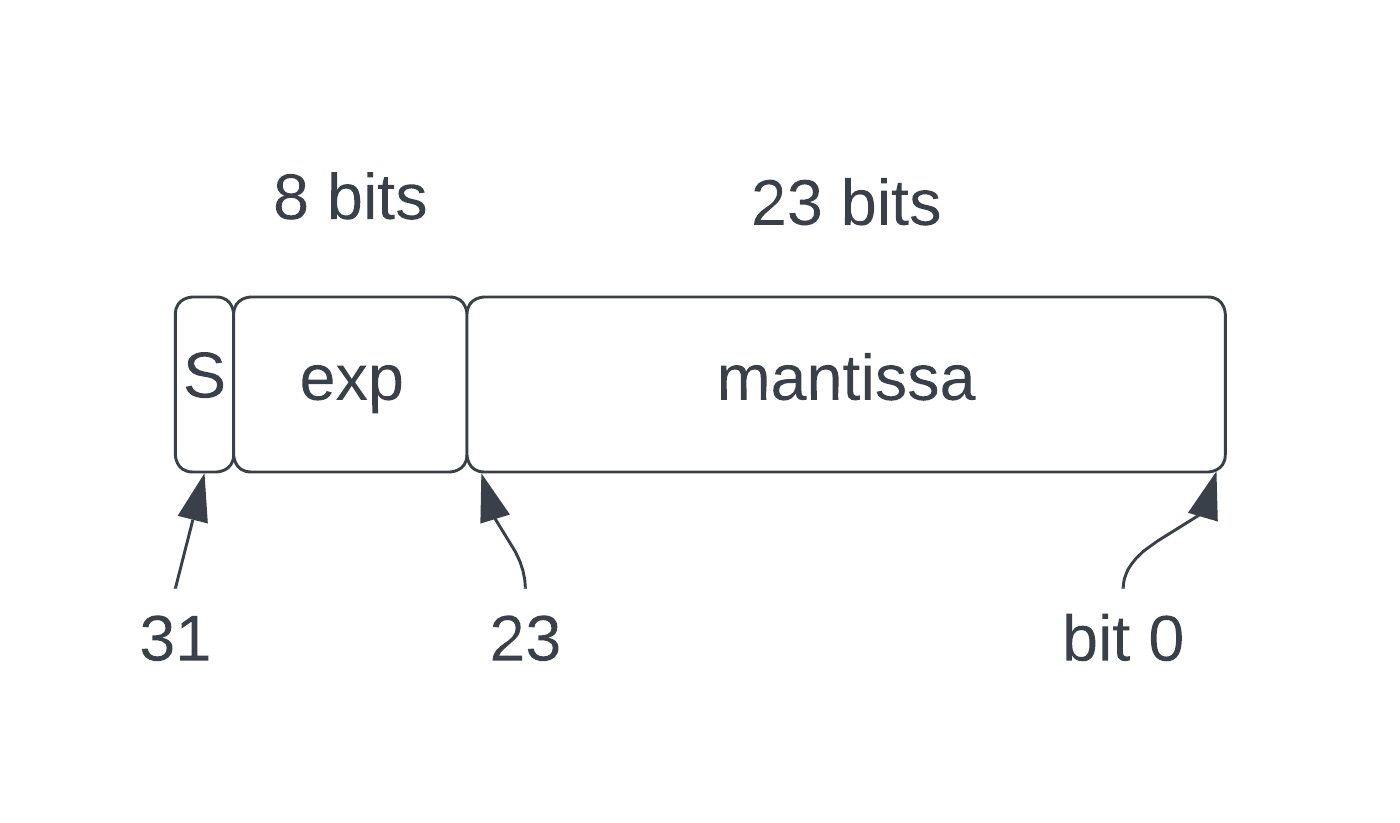
# IEEE 754 – The Practical Floating Point Standard

The modern standard for floating point number is the Institute of Electrical and Electronic Engineers (IEEE) technical standard 754 (<https://en.wikipedia.org/wiki/IEEE_754>). This specifies the bit layout and behavior for multiple precisions of floating point numbers. The most common precisions are *single* and *double* precision, where a single precision number fits into 32 bits and double precision is a 64 bit quantity.

Let’s only consider *single precision* floating point since other precisions are very similar and only differ in the number of bits assigned to each part. An IEEE 754 single precision number has the following parts:

1. A 23-bit mantissa with an implicit (not stored) 1 bit to the left of the binary point.
2. An 8-bit signed exponent with a **bias** of 127 added to it.
3. A single sign bit specifying if the mantissa is positive or negative
4. Special patterns for zero, infinity and Not-A-Number (NaN) values
5. Specified rounding rules for dealing with un-stored bits

This fits into a 32-bit register and the layout is as follows:



The bit called **S** is the sign bit (with 1 meaning the mantissa is a negative number). The exponent (marked as **exp**) is 8 bits and spans from bit 30 to bit 23. The remaining bits are for the **mantissa**. The mantissa is actually 24 bits long but only 23 bits are stored. The mantissa’s binary point is to the left of bit 23 and it must always be normalized to 1.x, so there will always be a 1 to the left of the binary point and there is no point in wasting a bit.

The exponent has a **bias** of 127 applied to it. This just means that instead of the exponent being stored as a signed 8-bit number, it has the number 127 added to it before storing. You can extract the proper exponent value by subtracting 127 from the number stored in the memory. The reason for this is so that the correct ordering of values can be maintained for negative and positive exponents, thus allowing integer comparisons to be used when comparing floating point numbers.

# IEEE 754 in C

Let’s do some coding. This section will present a bunch of C functions that perform all the necessary operations to support IEEE 7854 single precision floating point.

## Unpacking and packing IEEE 754 numbers

If we are to do any manipulation of the IEEE 754 encoded number, they need to be unpacked into the constituent parts. Let’s define a struct to hold the unpacked values:

// This is an unpacked IEEE754 single precision floating point number.

typedef struct {

uint8\_t sign; // Either 0x80 or 0.

uint8\_t exponent; // Exponent with bias applied.

uint32\_t mantissa; // Mantissa with implicit 1 in top bit.

} Unpacked;

The *exponent* is unpacked and stored with its *bias* still applied – the bias can easily be removed and reapplied during any calculations. The *mantissa* is stored in the top 24 bits of a 32-bit number and has the implicit 1 bit reintroduced at bit 31 of the value – the 23 bits of the IEEE 754 mantissa are then in bits 30...8. The bottom 8 bits will be 0 when the unpacking is done. The *sign* bit is held as either 0x80 or 0. There’s no need to shift it down to the bottom bit to make it into a boolean.

Let’s look at the function to unpack an IEEE 754 single precision number.

To obey the strict aliasing rules of modern C, we define a union to allow us to access the bit pattern of a float. This will allow us to access the float bits as either an unsigned integer or a signed integer.

typedef union Alias {

uint32\_t i; // Unsigned integer.

int32\_t si; // Signed integer.

float f; // Single precision float.

} Alias;

Aliasing (or in this case, type punning) in C is a technique to circumvent the type-system to allow a program to treat one type as if it was another. If done incorrectly, the program may or may not work depending on what the compiler decides to do when it’s optimizing. The use of a union is a valid way to do type punning that the compiler is aware of.

// Unpack a IEEE754 single precision number.

Unpacked Unpack(float f) {

Alias a = {.f = f};

Unpacked u;

u.sign = (a.i & 0x80000000) >> 24;

u.exponent = (a.i >> 23) & 0xff; // Still has bias.

u.mantissa = ((a.i & 0x007fffff) << 8) | 0x80000000;

return u;

}

This takes a **float** argument and unpacks the components into the *Unpacked* struct, which it returns. First, it gets the bit pattern from the argument by copying it to an *Alias* union and extracting the unsigned 32-bit number member. The *sign* member of the *Unpacked* struct is the top bit of the bits shifted down to the low byte and masked to a single byte (the value 0x80 means the sign bit was set). The exponent consists of the remaining 7 bits of the top byte and the top bit of the second byte – that is, the bits shifted right by 23 and masked to 8 bits (bias is not removed). The mantissa is the bottom 23 bits, shifted left by 8 bits and then the top bit is set (adding back the implicit 1).

The inverse of unpacking is to pack an *Unpacked* struct to an IEEE 754 single precision format.

// Pack into a IEEE754 single precision number.

float Pack(Unpacked u) {

Alias bits = {.i = 0};

bits.i |= (u.mantissa >> 8) & 0x007fffff; // Top 23 bits.

bits.i |= u.exponent << 23; // Already biased.

bits.i |= u.sign << 24;

return bits.f;

}

We start out with an empty 32-bit value inside an *Alias* local variable and fill in the bits by shifting an ORing them in. First, we take the top 23 bits of the mantissa from *Unpacked* and place that in the bottom 23 bits of the result. Then comes the exponent, which is 8-bits shifted left by 23 places and ORed in. Finally, we have the sign which is just shifted left by 24. Remember that the sign is either 0x80 or 0. ORing these with the top byte will either set bit 31 or not, with no effect on the other bits. The resultant bit pattern is then treated as a float and returned.

## Special values

IEEE 754 defines a number of special encodings for values that do not fit into the normal mathematical rules. These are:

|  |  |
| --- | --- |
| **Special Value** | **Encoding** |
| Zero | 0x00000000 |
| Infinity | 0x7f800000 |
| Not a Number (NaN) | 0x7fffffff |

We can also have a negative value for zero, Nan and Infinity, with the sign bit set.

The standard also defines the concepts of a denormalized number, which is identified by an exponent value of 0. We aren’t going to handle those.

## Normalizing and rounding

The IEEE 754 standard requires that the mantissa be normalized so that it has a 1 to the left of its binary point (unless it’s a special denormalized number). This is pretty easy to accomplish simply by shifting the mantissa to the left until there is a 1 in bit 31. However, each shift is a multiplication by 2, so we need to cancel that out by decrementing the exponent each time we shift. Here’s a function to normalize an *Unpacked* struct.

// Normalize an unpacked number by shifting until we get a 1 in the MSB. A

// normalized number always has a 1 to the left of the binary point.

Unpacked Normalize(Unpacked u) {

if (u.mantissa == 0) {

return u;

}

// Shift left until bit 31 of the mantissa is set and decrement exponent

// with each shift.

while ((u.mantissa & 0x80000000) == 0) {

u.mantissa <<= 1;

u.exponent--;

}

return u;

}

The function first checks that we don’t have a mantissa that’s all zeroes as that would cause an infinite loop. Assuming we have at least 1 bit that is non-zero, we continually shift the mantissa left until it has bit 31 set, decrementing the exponent each time we shift. We then return the, possibly modified, *Unpacked* struct[[7]](#footnote-7).

In order to improve accuracy and consistency, IEEE 754 specifies how to round the result to fit into the bits we have in the result. There are only 24 bits available for the mantissa (23 bits stored in memory with an implicit 1), so if the result of a calculation contains more bits than can be stored, we have a choice of how to handle those bits.

There are five rounding modes available:

1. Round to nearest, with even numbers taking priority
2. Round to nearest, with odd numbers taking priority
3. Round to positive infinity
4. Round to negative infinity
5. Round to zero

In this treatise we will only consider the first rounding mode, with the others being left as an exercise. That’s the default on most operating systems anyway.

To round using this mode, we look at the highest bit of the part of the number that will not be stored (bit 7 of the 32-bit mantissa) and if it is 1, we round the mantissa up by adding 1 to it. Remember that, moving to the right, each subsequent bit in the mantissa represents half of the previous value. So, if the highest bit of the bits to be discarded is 1, it says that the discarded value is greater or equal to half of the value of the bits to its left.

If it is exactly half there is a wrinkle, because the rounding mode says that even numbers should be considered as higher priority than odd numbers. This means that if the only bit set in the discarded bits is the high order bit (all the others being zero), it represents exactly half, and we should add 1 to the mantissa, but only if it will produce an even number.

Here’s a function to perform the rounding of an *Unpacked* struct.

Unpacked Round(Unpacked v) {

// The encoded mantissa is 24 bits long. This means we discard the bottom

// byte of the 32-bit mantissa in v. We round to nearest, so if the top

// bit of discarded is 1 we have a number half or larger and we add 1 to

// the 24-bit mantissa.

int discarded = v.mantissa & 0xff;

// Half or larger?

if ((discarded & 0x80) != 0) {

// Calculate rounded 24-bit mantissa.

int32\_t rounded = ((v.mantissa >> 8) + 1) << 8;

// If the discarded byte is exactly 0x80 we only round if

// the resultant number is even (IEEE 754 rounding rules).

if (discarded == 0x80) {

if ((rounded & 0x100) == 0) {

// Even.

v.mantissa = rounded;

}

} else {

v.mantissa = rounded;

}

}

return v;

}

Since our mantissa is 24 bits long and we have 32-bits available in the *Unpacked* struct, there are 8 bits of value that won’t be stored in the result - the lowest 8 bits. First, we extract the discarded bits and check if its highest bit is set. If so, then the discarded bits represent half or higher and we need to round. We then calculate the rounded number by adding 1 to the high 24 bits of the mantissa (shift it right by 8 bits, add 1, then shift it left again).

The wrinkle of even priority is handled by checking if the discarded bits are exactly 0x80 (only the top bit set, or exactly half). If so, we store the rounded mantissa only if it has a zero in its lowest bit (it’s an even number). If the discarded bits are not 0x80, we can store the rounded mantissa directly since the even priority doesn’t apply.

## Math

The whole point of floating point is to allow us to perform mathematical operations on the numbers. Let’s look at how to do this from first principles – only using basic integer operations (add, subtract, shifts, bitwise operations). We won’t use multiplication or division but will use 64-bit integers

### Conversions to and from binary integer.

Two of the most basic operations are conversion to and from integers and floating point numbers. Integers can be various sizes, but the most common is 32-bit.

The conversion of an integer to floating point is relatively simple. An integer is basically a fixed point number with its binary point to the right of its least significant bit. To convert to floating point we move the binary point to the left until it is to the left of its most significant bit, counting the number of times we moved it.

To perform the binary point move, we start out with a mantissa of 0 and shift the integer right into the mantissa, incrementing the exponent for each shift. We stop when we run out of non-zero bits in the integer. Shifting to the right puts the lower order bit of the integer into the high order bit of the mantissa, then shifts both to the right.[[8]](#footnote-8)

Here’s a function to perform the conversion of a 32-bit integer to floating point:

// Convert a number from integer to floating point.

float ToFloat(int32\_t v) {

if (v == 0) {

return 0;

}

Unpacked u = {0};

if (v < 0) {

u.sign = 0x80;

v = -v;

}

u.exponent = 127; // Start out at bias value for exponent.

for (;;) {

u.mantissa >>= 1;

u.mantissa |= (v & 1) << 31;

v >>= 1;

if (v == 0) {

break;

}

u.exponent++;

}

return Pack(Round(u));

}

First, we check for a value of 0, since 0 has a special encoding in IEEE 754. If the number is not zero, we check its sign and make it positive if it is negative (recording the sign in the *Unpacked* struct). We start with an exponent of 0, but we add the bias to it immediately, so the initial exponent value is 127.

We then enter a loop that terminates when the integer becomes 0. In each iteration we shift the lowest bit of the integer into the highest bit of the mantissa and then shift the integer one bit to the right. Each time we shift, the exponent is incremented as we have divided the integer by 2. The result is the bits of the integer in the high order bits of the mantissa with bit 31 always set to 1 – a normalized mantissa.

Since the integer is 32 bits wide and the mantissa is encoded in 24 bits, we may have bits to be discarded and therefore we perform a rounding operation before packing it in floating point format.

The reverse conversion is equally simple. To convert a floating point number to integer we realize that the integer can only contain 32 bits and it cannot hold any fractional value. Therefore, any exponent less than 127 (a negative exponent) cannot be represented as an integer and results in 0. Likewise, any exponent greater than 127+31 is too big for the integer and results in a zero value.

The algorithm shifts the mantissa left into the result until we’ve exhausted the exponent. Each shift decrements the exponent until it reaches 0. Basically, we are taking the top N bits of the mantissa (with the implicit 1 in place) where N is the value of the exponent. However, since 20 is 1, an exponent of 0 means to shift by 1, so we add 1 to the exponent value.

Here’s the code to convert from floating point to 32-bit integer:

int32\_t FromFloat(float f) {

if (FPIsZero(f) || FPIsNan(f) || FPIsInfinity(f)) {

return 0;

}

Unpacked u = Unpack(f);

if (u.exponent < 127 || u.exponent > (127+31)) {

// Negative exponent is < 1 so this can't be an integer.

// An exponent greater than 31 is too big.

return 0;

}

// Remove bias from exponent and increment by 1 (2^0 == 1).

int8\_t exp = u.exponent - 127 + 1;

int32\_t v = 0;

while (exp > 0) {

// Shift the mantissa into the value, increment exp for each shift.

v <<= 1;

v |= (u.mantissa >> 31);

u.mantissa <<= 1;

exp--;

}

if (u.sign == 0x80) {

v = -v;

}

return v;

}

Notice that we work with an unbiased exponent, and we also handle the sign at the end. If the sign bit is set (0x80 in the sign member of *Unpacked*), we negate the value before returning it.

Of course, if the integer size is not 32-bits, adjustments can be made to the algorithms to account for the larger or smaller numbers.

### Multiplication

If you remember your math from school, to multiple an exponential number you add the exponents and multiply the mantissas. This is exactly the same for floating point binary numbers:

(A 2B) (C 2D) = (A C) 2(B+D)

Adding the exponents is easy. Remember that the exponents in IEEE 754 have a bias of 127 added to them (to make comparison easier). To add two exponents E1 and E2 together, we first remove the bias, add them, and then reapply the bias.

(E1 – 127) + (E2 – 127) + 127

= E1 + E2 – 127 - 127 + 127

= E1 + E2 - 127

Multiplication is a little harder since we have decided not to use any multiplication operations and have restricted ourselves to simple integer manipulation. However, integer multiplication is well known and straightforward. Here’s a function to multiply two unsigned 32-bit integers, producing an unsigned 64-bit result.

// Multiply 32-bits a by b with 64 bit result.

uint64\_t Multiply32(uint32\_t a, uint32\_t b) {

uint64\_t r = 0;

uint64\_t x = a;

while (b != 0) {

if ((b & 1) == 1) {

r += x;

}

x <<= 1;

b >>= 1;

}

return r;

}

The algorithm is analogous to long multiplication you learned at school except in base 2. For multiplying **a** by **b**, we look at each bit in **b**, and if it’s 1, we add **a** to the result. For each bit in **b** we shift the result left once.

Given that we can now add the exponents and multiply two integers, we now have the components to perform floating point multiplication. Here’s the function to do it:

// To multiply we add the exponents and multiply the mantissas. The sign

// is the exclusive-OR of the two signs.

float FPMultiply(float a, float b) {

if (FPIsZero(a) || FPIsZero(b)) {

return 0;

}

if (FPIsInfinity(a) || FPIsInfinity(b)) {

return FPInfinity();

}

if (FPIsNan(a) || FPIsNan(b)) {

return FPNaN();

}

Unpacked ua = Unpack(a);

Unpacked ub = Unpack(b);

// Sign is EOR of both signs.

uint8\_t sign = ua.sign ^ ub.sign;

// Remove bias from exponents, add them and apply bias.

// (eA - 127) + (eB - 127) + 127

// = eA + eB - 127 - 127 + 127

// = eA + eB - 127

int8\_t exp = ua.exponent + ub.exponent - 127;

// Multiply the mantissas. These are both unsigned 32 bit numbers with the

// high bits set. The product will be an unsigned 64 bit number with the

// high 32 bits containing the result we need. Also, since both numbers

// were fixed point with 1 bit to the left of the binary point, the result

// will have 2 bits to the left of the binary point. We increment the

// exponent to take this into account.

uint64\_t m = Multiply32(ua.mantissa, ub.mantissa);

Unpacked r;

r.sign = sign;

r.exponent = exp + 1; // Exponent incremented.

r.mantissa = (uint32\_t)(m >> 32);

return Pack(Round(Normalize(r)));

}

It’s fairly straightforward. First, we check for special cases that have known results. If the numbers are not special, we unpack them for further processing. The sign of the result is the exclusive-OR of the two signs (a standard property of multiplication in any base). Then we add the exponents and multiply the mantissas.

Since the mantissas are both big 32-bit integers, the result of the multiplication will be a 64-bit integer where the product is in the upper 32 bits. We only need the top 32 bits of the result, so we shift it down and ignore the lower 32 bits.

Finally, we normalize the result, round it and pack it into floating point format.

### Division

Now, let’s take a look at division. It’s very similar to multiplication for exponential numbers. You subtract the exponents and divide the mantissas.

(A 2B) / (C 2D) = (A / C) 2(B-D)

To subtract the exponents, we remove the biases from each, perform the subtraction and reapply the bias.

(E1 – 127) - (E2 – 127) + 127

= E1 - E2 – 127 -+127 + 127

= E1 - E2 + 127

Since we’ve hamstrung ourselves by not allowing division by the compiler, we need to divide the integers the old-fashioned way (a.k.a. the hard way). Akin to multiplication, division is basically long division in base 2. It’s a well-known algorithm, so here’s the function to do it. We need to be able to divide unsigned 64-bit numbers since this will be used in ASCII conversions, later.

// Quotient and Remainder. This is the value of a division.

struct QR {

uint64\_t quotient;

uint64\_t remainder;

};

// Divide 64 bits a by b, producing a Quotient and Remainder (QR).

struct QR Divide64(uint64\_t a, uint64\_t b){

struct QR qr = {0,0};

for (int i = 0; i < 64; i++) {

// Shift high bit of a into rem and shift a left by one.

qr.remainder <<= 1;

if ((a & 0x8000000000000000LL) != 0) {

qr.remainder |= 1;

}

a <<= 1;

qr.quotient <<= 1;

if (qr.remainder >= b) {

qr.quotient++;

qr.remainder -= b;

}

}

return qr;

}

The function returns both the *quotient* and *remainder* of the division since both are produced naturally.

Now we can perform the floating point division with the following function.

// Division is done by subtracting the exponents and dividing the mantissas.

float FPDivide(float a, float b) {

if (FPIsZero(a)) {

return 0;

}

if (FPIsZero(b)) {

return FPInfinity();

}

if (FPIsInfinity(a) || FPIsInfinity(b)) {

return FPInfinity();

}

if (FPIsNan(a) || FPIsNan(b)) {

return FPNaN();

}

Unpacked ua = Unpack(a);

Unpacked ub = Unpack(b);

// Sign is EOR of both signs.

uint8\_t sign = ua.sign ^ ub.sign;

// Remove bias from exponents, subtract them and apply bias.

// (eA - 127) - (eB - 127) + 127

// = eA - eB - 127 + 127 + 127

// = eA - eB + 127

uint8\_t exp = ua.exponent - ub.exponent + 127;

// Divide the mantissas. The dividend is placed in the upper 32 bits of

// the division and the divisor is in the lower 32 bits.

struct QR qr = Divide64((uint64\_t)ua.mantissa << 32, ub.mantissa);

// If the quotient is greater than 32 bits, shift it right until it fits

// into 32 bits and increment exponent for each shift.

while ((qr.quotient & 0xffffffff00000000LL) != 0) {

qr.quotient >>= 1;

exp++;

}

Unpacked r;

r.sign = sign;

r.exponent = exp – 1;

r.mantissa = (uint32\_t)qr.quotient;

return Pack(Round(Normalize(r)));

}

First, the usual special cases are handled, then we unpack the arguments into *Unpacked* structs, and calculate the sign of the result (exclusive-OR of the input signs – same as multiplication). We then subtract the exponents and apply the bias to the resultant exponent.

The division of the mantissa is a little tricky. We have two large 32-bit numbers and if we divide them directly, we will get a small number that will have the wrong value. Instead, we put the dividend into the top 32-bits of a 64-bit number and the divisor in the lower 32-bits. This is the equivalent of long division where we start at the left of the dividend. We iterate 64 times so we will put the result in the lower half of the resultant 64-bit number.

After the division is done, the quotient will be 64 bits wide, but we need the lower 32 bits. However, if there are non-zero bits in the upper half, they are significant and we don’t want to lose them, so we shift the result right until it is 32-bits wide, incrementing the exponent for each shift. We lose the lower bits of the result but those are less significant so don’t affect the result. Note that we don’t use the *remainder* part of the division for this operation.

After that we normalize, round and pack the result[[9]](#footnote-9).

### Addition and subtraction

The addition of two exponential numbers is done by scaling the mantissa of one of them to make their exponents the same, then adding the mantissas. Subtraction is simply the addition of the negative of the second argument.

To perform the scaling operation, we find the higher exponent of the two numbers and divide the mantissa of the other number by 2 to the power of the difference in exponents.

Let’s take a decimal example. Say we are adding 1.5 105 and 2.5 107. This is the same as adding 0.015 107 and 2.5 107. The difference in exponents is 2 so we divide the mantissa of the smaller number by 102 before adding.

We do exactly the same thing in binary, except instead of dividing by 10 we shift the smaller number to the right (dividing by 2).

The attentive reader might notice that if the exponents are too far apart, we might shift all our bits out of the bottom and end up with 0. This is fine as we are adding a very big number to a very small number and it’s beyond the precision of a floating point number.

Here’s the function to add two floating point numbers.

// To add, we make the exponents the same by shifting the lower value to the

// right, then add the mantissas, taking sign into account.

float FPAdd(float a, float b) {

if (FPIsZero(a)) {

return b;

}

if (FPIsZero(b)) {

return a;

}

if (FPIsInfinity(a) || FPIsInfinity(b)) {

return FPInfinity();

}

if (FPIsNan(a) || FPIsNan(b)) {

return FPNaN();

}

Unpacked ua = Unpack(a);

Unpacked ub = Unpack(b);

uint8\_t exp = ua.exponent;

if (ua.exponent > ub.exponent) {

// Exponent of a is larger than b: shift b right by the difference,

// stopping if we get to zero.

int8\_t diff = ua.exponent - ub.exponent;

while (diff-- > 0 && ub.mantissa != 0) {

ub.mantissa >>= 1;

}

if (ub.mantissa == 0) {

return a;

}

} else {

int8\_t diff = ub.exponent - ua.exponent;

while (diff-- > 0 && ua.mantissa != 0) {

ua.mantissa >>= 1;

}

if (ua.mantissa == 0) {

return b;

}

exp = ub.exponent;

}

uint64\_t mA = (uint64\_t)ua.mantissa;

uint64\_t mB = (uint64\_t)ub.mantissa;

// Negate mantissas as specified by sign.

if (ua.sign == 0x80) {

mA = -mA;

}

if (ub.sign == 0x80) {

mB = -mB;

}

uint64\_t mantissa = mA + mB;

uint8\_t sign = 0;

// If result of mantissa addition is negative, set sign to 0x80 and make

// mantissa positive.

if ((mantissa & (1LL << 63)) != 0) {

// Mantissa is negative, sign is 0x80.

sign = 0x80;

mantissa = -mantissa;

}

// If we have overflowed the 32 bits of mantissa, shift it right and

// increment the exponent.

while ((mantissa & 0xffffffff00000000LL) != 0) {

mantissa >>= 1;

exp++;

}

Unpacked r;

r.sign = sign;

r.exponent = exp;

r.mantissa = (uint32\_t)mantissa;

return Pack(Round(Normalize(r)));

}

The function starts out with the usual checks for special values and unpacking the arguments. Then we scale the mantissa of the number with the smaller exponent by the difference in exponents and set the result exponent to the bigger exponent.

Since we are using 2-complement addition, and the mantissas are both sign-and-magnitude quantities, we look at the signs of the numbers and negate the negative ones. Then we add the signed numbers together giving us the sum. If the result is negative, we make it positive and set the result sign to 0x80.

Like division, we will have a 64-bit result that might have non-zero bits in the upper 32-bits. These are significant bits, so we move them down into the lower 32-bits, discarding the lowest order bits and incrementing the result exponent.

The traditional normalization, rounding and packing produces the result.

Subtraction is trivial.

// Subtraction is an addition of a negative.

float FPSub(float a, float b) {

return FPAdd(a, -b);

}

### Comparison

Floating point comparison is a tricky beast, but we are only going to do the basic 6 comparison operations:

1. Equality
2. Inequality
3. Less
4. Greater
5. Less than or equal
6. Greater than or equal

Equality is simply a matter of comparing the bit pattern in the floating point numbers.

// Equality is just a straight-forward bitwise comparison.

bool FPEqual(float a, float b) {

Alias ba = {.f = a};

Alias bb = {.f = b};

return ba.i == bb.i;

}

Inequality is trivial.

bool FPNotEqual(float a, float b) {

return !FPEqual(a, b);

}

The rest of the comparison operations can be achieved using a single function that provides a *less-than* comparison. This one is more complex and needs to take the sign of the number into account. It’s still reasonably easy to follow and uses the bit pattern of the floating point numbers to do integer comparisons.

The encoding of an IEEE 754 number, with the biased exponent in the high bits and the mantissa in the lower bits allows for ordered comparison of *like-signed* numbers using a normal integer comparison. A bigger exponent results in a bigger number. The bias lets us treat negative exponents as smaller than positive exponents.

However, this isn’t the case when comparing numbers with different signs. A larger exponent in a negative number is less than a smaller exponent. For example, -1.5 105 is less than 1.5 102, even though 5 is greater than 2. If both numbers are negative, we simply reverse the comparison.

// Less is more complex. For a positive number we just do an integer

// comparison but that won't work for negative numbers since a negative

// number with a bigger exponent is less than a smaller exponent.

bool FPLess(float a, float b) {

if (FPIsInfinity(b)) {

return true;

}

if (FPIsInfinity(a)) {

return false;

}

if (FPIsNan(a) || FPIsNan(b)) {

return false;

}

Alias ba = {.f = a};

Alias bb = {.f = b};

if (ba.si < 0) {

// If a is negative, it's less than a positive b.

if (bb.si < 0) {

// Both negative, reverse comparison.

return bb.si < ba.si;

}

return true;

}

if (bb.si < 0) {

return false;

}

// Both positive, simple integer comparison.

return ba.si < bb.si;

}

Notice the use of the *si* member of the *Alias* – a signed 32-bit integer. Once we have the less than comparison, we can derive the other 3 comparisons as follows:

1. Greater than: **a > b** is the same as **b < a**
2. Less than or equal: **a <= b** is the same as **!(a > b)** or **!(b < a)**
3. Greater than or equal: **a >= b** is the same as **!(a < b)**

Here are the functions.

bool FPGreater(float a, float b) {

return FPLess(b, a);

}

bool FPGreaterEqual(float a, float b) {

return !FPLess(a, b);

}

bool FPLessEqual(float a, float b) {

return !FPGreater(a, b);

}

## Special values

We will need some functions to check a number against one of the special values and to generate a special value. We will only handle zero, NaN and infinity.

bool FPIsZero(float f) {

Alias a = {.f = f};

return (a.i & 0x7fffffff) == 0; // Both +ve and -ve 0.

}

bool FPIsNan(float f) {

Alias a = {.f = f};

return (a.i & 0x7fffffff) == 0x7fffffff;

}

bool FPIsInfinity(float f) {

Alias a = {.f = f};

return (a.i & 0x7f800000) == 0x7f800000; // Both +ve and -ve

}

float FPZero(void) {

Alias a = {.i = 0};

return a.f;

}

float FPNaN(void) {

Alias a = {.i = 0x7fffffff};

return a.f;

}

float FPInfinity(void) {

Alias a = {.i = 0x7f800000};

return a.f;

}

## Conversions to and from ASCII decimal

If floating point numbers and how they actually work is rarely understood, there are many fewer engineers who understand how to convert them to and from ASCII. Most programmers are aware of the *libc* functions ***strtod*** and ***printf*** without knowing (or caring) how they work.

Let’s explore how to perform the conversions to and from ASCII.

The basic principle uses *fixed point numbers*. For conversion to ASCII, we first convert the floating point number into its fixed point equivalent. For the opposite direction, we generate a fixed point number and then convert it to floating point.

An IEEE 754 single precision number has an 8-bit exponent, allowing us to represent any number in the range 2-128 to 2127. In other words, a 256-bit fixed point number can accommodate any value we can represent in a single precision floating point number. The binary point location for the fixed point number is the middle, allowing us 128 bits for the integer part and 128 bits for the fractional part.

In order to manipulate 256-bit fixed point numbers we need to be able to perform arithmetic operations on them.

First, let’s choose an efficient representation for the fixed point number. We need to be able to access any byte of it and also perform efficient shifts and adds. The simplest representation is an array of 4 64-bit unsigned integers. Indices 0 and 1 will be the fractional part while 2 and 3 will represent the integer part.

Now, let’s consider the operations we need to perform.

1. Increment a 256-bit and 128-bit number
2. Add 2 256-bit or 128-bit numbers
3. Left and right shift both 128-bit and 256-bit numbers
4. Multiply 128-bit and 256-bit numbers by 10
5. Divide 128-bit and 256-bit numbers by 10
6. Check 128-bit and 256-bit numbers for zero.

### Increment

We need to increment (by 1) both 128-bit and 256-bit fixed point numbers, represented by arrays of 64-bit integers. Start at the lower end of the array and increment the 64-bit word there. If that wraps to zero, we increment the next word up. If it’s not zero, we don’t need to go any further.

void FixedIncrement128(uint64\_t a[2]) {

for (uint8\_t i = 0; i < 2; a++, i++) {

++(\*a);

if (\*a != 0) {

break;

}

}

}

void FixedIncrement256(uint64\_t a[4]) {

for (uint8\_t i = 0; i < 4; a++, i++) {

++(\*a);

if (\*a != 0) {

break;

}

}

}

## Addition

Addition for both 128-bit and 256-bit numbers is done by the carry-ripple method. The calculation is:

a = a + b

We define *FixedAdd64* that adds two 64-bit numbers with a carry input, putting the sum in the first number. It returns the carry out as a result[[10]](#footnote-10).

uint64\_t FixedAdd64(uint64\_t\* a, uint64\_t\* b, uint64\_t carry\_in) {

uint64\_t na = \*a + \*b + carry\_in;

uint64\_t carry\_out = na < \*a;

\*a = na;

return carry\_out;

}

Then we can use *FixedAdd64* for each word in the arrays from the low word to the high. We start out with a carry of 0 and each addition produces the carry for the next word.

void FixedAdd256(uint64\_t a[4], uint64\_t b[4]) {

uint64\_t carry = 0;

for (int i = 0; i < 4; a++, b++, i++) {

carry = FixedAdd64(a, b, carry);

}

}

void FixedAdd128(uint64\_t a[2], uint64\_t b[2]) {

uint64\_t carry = 0;

for (int i = 0; i < 2; a++, b++, i++) {

carry = FixedAdd64(a, b, carry);

}

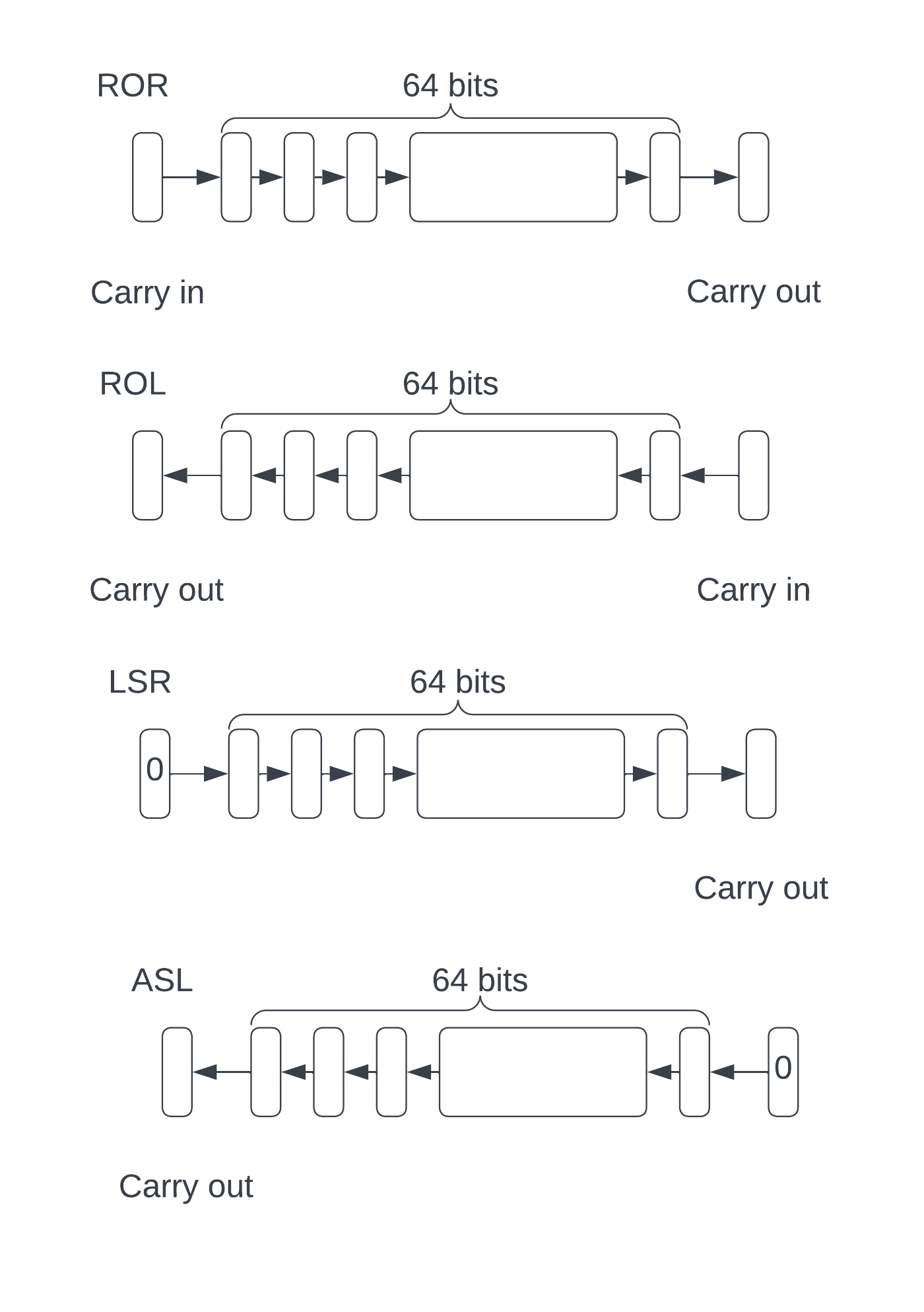
}

### Shifts

Shifting a long integer is the same as shifting a single integer except the bit shifted out from an adjacent word is shifted into the next word. We define 4 operations[[11]](#footnote-11):

1. Rotate left (ROL) – shift left and insert carry in as low bit. Return bit shifted out.
2. Rotate right (ROR) – shift right and insert carry in as high bit. Return bit shifted out.
3. Arithmetic Shift Left (ASL) - shift left returning the bit shifted out.
4. Logical Shift Right (LSR) – shift right returning the bit shifted out.

The functions we need operate on unsigned 64-bit numbers as follows:



int64\_t FixedROL64(uint64\_t\* a, uint64\_t carry\_in) {

int64\_t carry\_out = (\*a & (1LL << 63)) != 0;

\*a <<= 1;

\*a |= carry\_in;

return carry\_out;

}

int64\_t FixedASL64(uint64\_t\* a) {

int64\_t carry = (\*a & (1LL << 63)) != 0;

\*a <<= 1;

return carry;

}

int64\_t FixedROR64(uint64\_t\* a, uint64\_t carry\_in) {

int64\_t carry\_out = (\*a & 1) != 0;

\*a >>= 1;

\*a |= carry\_in << 63;

return carry\_out;

}

int64\_t FixedLSR64(uint64\_t\* a) {

int64\_t carry = (\*a & 1) != 0;

\*a >>= 1;

return carry;

}

Now to perform the actual shifts we use the defined 64-bit operations on each word in the number.

For *FixedLShift128*, we use ASL to shift the bottom 64 bits one bit to the left, keeping the carry. We then rotate the upper 64 bits left, passing the carry.

Likewise, for *FixedLShift256*, we use ASL for the bottom 64-bit word and then rotate using ROL for the upper 3 words, rippling the carry from the high bit of one word to the low bit of the next word.

We only need a *FixedRShift256*, so for this we shift the upper word right using LSR, keeping the carry and rippling it down using ROR.

void FixedLShift128(uint64\_t a[2]) {

int64\_t carry = FixedASL64(a++);

for (int i = 0; i < 1; i++) {

carry = FixedROL64(a++, carry);

}

}

void FixedLShift256(uint64\_t a[4]) {

int64\_t carry = FixedASL64(a++);

for (int i = 0; i < 3; i++) {

carry = FixedROL64(a++, carry);

}

}

void FixedRShift256(uint64\_t a[4]) {

a += 3;

int64\_t carry = FixedLSR64(a--);

for (int i = 0; i < 3; i++) {

carry = FixedROR64(a--, carry);

}

}

### Multiplication by 10

Multiplication by a constant is much easier than general variable multiplication. For decimal conversion we need to provide only multiplication by 10.

Multiplying a number by 10 is the same as multiplying it by 8 and adding the result to the number multiplied by 2.

x 10 = x 8 + x 2

Since multiplying by a factor of 2 is achieved using shifts, the multiplication by 10 can be reduced to shifts and an addition. We already have defined shifting and addition operations, so we can readily write functions to multiply 128 and 256-bit numbers by 10.

We need to multiply the original number by both 8 and 2, so we make a copy of it into a temp. We shift the temp left 3 bits (multiply by 8), then shift the original number left once (multiply by 2), then add the result and the temp, putting the product in the original location.

// Multiply by 10 by x\*8 + x\*2.

// Puts result in a.

void FixedMultiplyByTen256(uint64\_t a[4]) {

uint64\_t t[4];

memcpy(t, a, sizeof(t));

// t = a \* 8

for (uint8\_t i = 0; i < 3; i++) {

FixedLShift256(t);

}

// a = a \* 2

FixedLShift256(a);

// a = a + t

FixedAdd256(a, t);

}

// Multiply by 10 by x\*8 + x\*2.

// Puts result in a.

void FixedMultiplyByTen128(uint64\_t a[2]) {

uint64\_t t[2];

memcpy(t, a, sizeof(t));

// t = a \* 8

for (uint8\_t i = 0; i < 3; i++) {

FixedLShift128(t);

}

// a = a \* 2

FixedLShift128(a);

// a = a + t

FixedAdd128(a, t);

}

### Division by 10

Unfortunately, unlike multiplication by a constant, there is no short-cut for dividing by 10. However, it’s a little easier than dividing by an arbitrary number since we know that we are dividing by 10.

The algorithm uses standard binary division but uses our defined functions to perform the shifting and increment operations.

Both 128 and 256-bit functions take a fixed point number as an argument (by reference), divide it by 10 and put the result back in the argument. They return the remainder, which will be in the range 0...9.

An optimization for the 256-bit division checks if the top 128 bits are zero, and if so, calls the 128-bit version since the time to perform the division is linear with the number of bits in the number.

// Divide a by 10, where a is a big number. Return

// the modulus. A is modified to be a/10.

uint8\_t FixedDivideByTen128(uint64\_t a[2]){

uint64\_t quotient[2] = {0};

uint8\_t rem = 0;

const int kHiWord = 1;

for (uint16\_t i = 0; i < 2 \* 64; i++) {

// Shift high bit of a into rem and shift a left by one.

rem <<= 1;

if ((a[kHiWord] & (1LL << 63)) != 0) {

rem |= 1;

}

FixedLShift128(a);

FixedLShift128(quotient);

if (rem >= 10) {

FixedIncrement128(quotient);

rem -= 10;

}

}

memcpy(a, quotient, sizeof(quotient));

return rem;

}

// Divide a by 10, where a is a big number.

// A is modified to be a/10.

uint8\_t FixedDivideByTen256(uint64\_t a[4]) {

// If top words are zero, use 128 division.

if (FixedIsZero128(a + 2)) {

return FixedDivideByTen128(a);

}

uint64\_t quotient[4] = {0};

uint8\_t rem = 0;

const int kHiWord = 3;

for (uint16\_t i = 0; i < 4 \* 64; i++) {

// Shift high bit of a into rem and shift a left by one.

rem <<= 1;

if ((a[kHiWord] & (1LL << 63)) != 0) {

rem |= 1;

}

FixedLShift256(a);

FixedLShift256(quotient);

if (rem >= 10) {

FixedIncrement256(quotient);

rem -= 10;

}

}

memcpy(a, quotient, sizeof(quotient));

return rem;

}

### Checking for zero and one

It is also necessary to check for a number being zero or one. These are trivial functions.

bool FixedIsZero128(uint64\_t v[2]) {

uint64\_t \*p = v;

for (uint8\_t i = 0; i < 2; i++) {

if (\*p++ != 0) {

return false;

}

}

return true;

}

bool FixedIsZero256(uint64\_t v[4]) {

uint64\_t \*p = v;

for (uint8\_t i = 0; i < 4; i++) {

if (\*p++ != 0) {

return false;

}

}

return true;

}

bool FixedIsOne128(uint64\_t a[2]) {

if (\*a++ != 1) {

return false;

}

for (uint8\_t i = 1 ; i < 2; i++) {

if (\*a++ != 0) {

return false;

}

}

return true;

}

## Conversion from ASCII decimal

Now we have all the pieces we need to perform the conversion from ASCII decimal to a single precision floating point number.

First let’s take a look at the algorithm, then the function to perform it. Then we’ll see the helper functions used by the *ASCIIToFloat* function.

The input to the function is a string of ASCII characters representing a base 10 exponential number. The number is of the form:

1. A sign (+ or -)
2. An optional sequence of ASCII digits forming the integral part
3. An optional period (dot)
4. An optional sequence of ASCII digits forming the fraction
5. An optional exponent marker (e or E)
6. If the exponent marker is present
   1. An optional + or – sign for the exponent
   2. A sequence of ASCII digits forming the exponent

This should be immediately recognizable as a standard scientific number, as is used in most programming languages. For example, the following are all valid inputs:

3.1415926

-4.5e7

1e-6

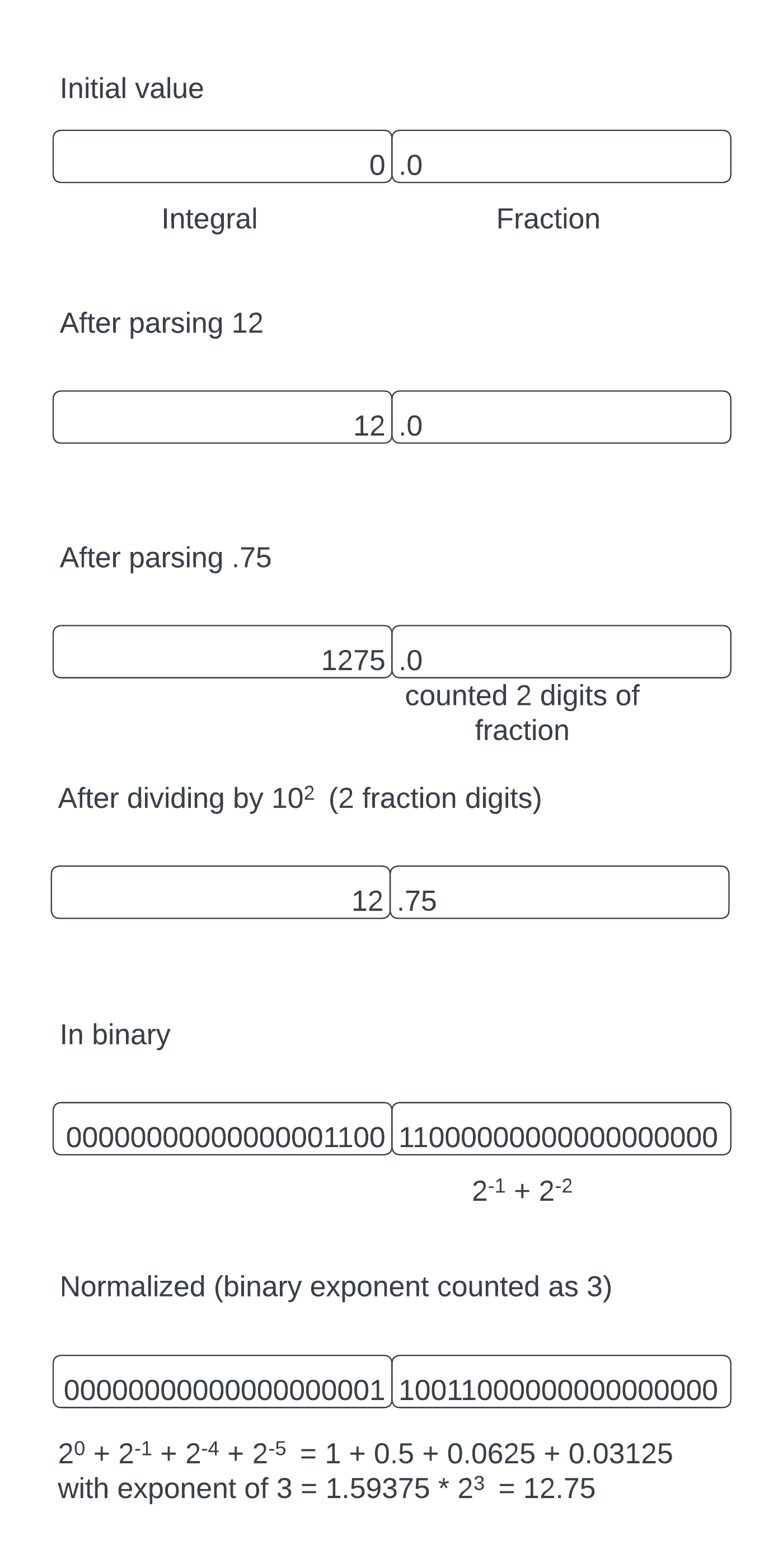
.05

.4E45

The algorithm to convert ASCII decimal to floating point has the following steps:

1. Define a 256-bit fixed point number as an array of 4 64-bit integers, with its binary point at bit 128 (the variable *fx* in the function).
2. Collect the integral part of the input into the upper 128 bits of the fixed point number in binary. This is to the left of the binary point. This calls *CollectPossibleSign* and *CollectInteger*.
3. If there is a fraction, continue collecting the digits of the fraction into the upper 128 bits. Keep a count of the number of decimal digits in the fraction (giving N), then divide the 256-bit number by 10N. This is another call to *CollectInteger*, with the division done by a looped call to *FixedDivideByTen256*.
4. The result so far will be a fixed point number with the integral part to the left of the binary point and the fraction to the right.
5. If there is an exponent:
   1. Collect the sign and digits of the exponent and convert to binary (giving value E)
   2. Multiply or divide the 256-bit number by 10E.
   3. This is handled by the *CollectAndHandleExponent* helper function.
6. If the whole number is zero, we know the result now – zero. *FixedIsZero256* determines this.
7. Normalize the fixed point number by shifting it to the right until there is a single 1 in the upper 128 bits. Count the number of times we shift, giving us the binary exponent. This is done by a call to *NormalizeAndGetExponent*.
8. Build an *Unpacked* struct from the sign, binary exponent and the second word of the fixed point number. The second word contains the most significant bits of the fraction, just to the right of the binary point. Shift this right by 1 bit and set the top bit – the implicit 1 bit in the mantissa.
9. Normalize, round and pack this into an IEEE 754 floating point number.

Let’s take an example. The input is “12.75”. The following diagram shows the integral and fractional parts after each part of the number is parsed.



Here’s the function in its entirety. See if you can follow the code steps using the algorithm above.

float ASCIIToFloat (const char\* p) {

// Fixed point 256-bit number.

uint64\_t fx[4] = {0};

// Can have a + or - prefix.

bool negative = false;

p = CollectPossibleSign(p, &negative);

// Convert integral part to binary. This is put into the top

// 128 of the fixed point number - the integral part.

p = CollectInteger(p, fx + 2);

// Any decimal point?

if (\*p == '.') {

p++;

// We have a fractional part, continue accumulating into the top 128.

const char\* fract\_start = p;

p = CollectInteger(p, fx + 2);

// Calculate number of digits in the fraction.

int fraction\_digits = (int)(p - fract\_start);

// Now shift the fractional part down to the lower half - to the right

// of the binary point. This is a base-10 division.

for (int i = 0; i < fraction\_digits; i++) {

FixedDivideByTen256(fx);

}

}

// Check for exponent. If it's present handle it, modifying the fixed

// point number.

if (\*p == 'e' || \*p == 'E') {

CollectAndHandleExponent(p + 1, fx);

}

// A zero is special case.

if (FixedIsZero256(fx)) {

return 0;

}

// We now have a fixed point binary number. Float the binary point to

// normalize the number by shifting left or right until the integer

// part has the value 1.

int exp = NormalizeAndGetExponent(fx);

// The mantissa is the upper bits of the fractional part shifted right by

// 1 bit and the top bit set.

uint32\_t mantissa = ((uint32\_t)(fx[1] >> 32) >> 1) | 0x80000000;

Unpacked u;

u.sign = negative ? 0x80 : 0;

u.exponent = exp;

u.mantissa = mantissa;

return Pack(Round(Normalize(u)));

}

Now let’s look at the helper functions. The first is *CollectInteger*, which, given a pointer to a char, collects sequential ASCII digits (determined by calling *isdigit*), accumulating them into the given fixed point result. Each digit is in the range 0x30-0x39 so we subtract 0x30 from it (‘0’) to form a number 0..9. Then we multiply the current fixed point result by 10 by calling *FixedMultiplybyTen128* and add in the converted digit using a call to *FixedAdd128*. We return the pointer to the character immediately after the last digit. Notice that this works on 128-bit quantities since we only collect the digits into the upper half of the 256-bit number. We shift them down after both the integral and fractional parts are collected.

// Collect the integer pointed to by p into the fixed point number fx.

// Returns the pointer to the character beyond the last one consumed.

const char\* CollectInteger(const char\* p, uint64\_t fx[2]) {

uint64\_t n[2] = {0};

while (isdigit(\*p)) {

n[0] = \*p - '0';

FixedMultiplyByTen128(fx);

FixedAdd128(fx, n);

p++;

}

return p;

}

The *CollectPossibleSign* function is pretty easy. It is passed a pointer to a char and if it is either ‘+’ or ‘-‘, it increments the pointer. If it’s ‘-‘, it sets the argument *\*neg* to *true* (indicating a negative value). The possibly modified character is returned.

const char\* CollectPossibleSign(const char\* p, bool\* neg) {

if (\*p == '-') {

\*neg = true;

p++;

}

if (\*p == '+') {

p++;

}

return p;

}

*CollecttAndHandleExponent* is the core of the exponent handling. It first collects a possible sign, then uses a simple loop to collect the digits of the exponent, converting them to binary. We know the exponent is a small number so we can use simple integer multiplication and addition to do the conversion[[12]](#footnote-12).

Once we have the exponent in binary, we either enter a loop to continually multiply or divide the given fixed point number by 10. The number of iterations the loop takes is the exponent value. If the exponent is negative, our loop divides the fixed point number by 10 each iteration. For a positive exponent, we multiply by 10.

The result is that the fixed point number (passed by reference) is scaled by the exponent value.

void CollectAndHandleExponent(const char\* p, uint64\_t fx[4]) {

// Exponent can have a sign.

bool negative = false;

p = CollectPossibleSign(p, &negative);

// Collect exponent and convert to binary. This is a simple integer

// as it can have a max value of 127.

int exp = 0;

while (isdigit(\*p)) {

exp = exp \* 10 + \*p++ - '0';

}

// Multiply or divide by the 10 to the power of exponent.

if (negative) {

// Exponent is negative, divide the fixed point number by 10^exp

// iteratively.

for (int i = 0; i < exp; i++) {

FixedDivideByTen256(fx);

}

} else {

// Exponent is positive, multiply by 10^exp.

for (int i = 0; i < exp; i++) {

FixedMultiplyByTen256(fx);

}

}

}

*NormalizeAndGetExponent* takes the scaled fixed point number and shifts it either left or right until the top 128 bits (the integral part) contains the value 1. The fixed point number may be much larger than 1 or much smaller. We use our shift functions to perform the shifts. We keep track of an exponent value which starts at 127 (the bias value). Each time we shift to the right we increment the exponent, and a shift to the left decrements the exponent. The fixed point number is modified in place and we return the exponent.

// Calculate the binary exponent is zero (127 with bias).

// During the normalization loops we increment or decrement the binary

// exponent until we have a binary number of 1.x.

//

// Returns the binary exponent and modifies fx.

int NormalizeAndGetExponent(uint64\_t fx[4]) {

int exp = 127;

if (FixedIsZero128(fx + 2)) {

// Number is less than one, shift left.

while (!FixedIsOne128(fx + 2)) {

FixedLShift256(fx);

exp--;

}

} else {

// Number is greater than one, shift right.

while (!FixedIsOne128(fx + 2)) {

FixedRShift256(fx);

exp++;

}

}

return exp;

}

## Conversion to ASCII decimal

The conversion of a floating point to ASCII decimal is usually accomplished using a call to one of the *printf* functions depending on where you want the output to be placed.

For example, if you want the output in a char array:

snprintf(a, sizeof(a), "%f", f);

Would put the value of *f* in non-exponential form into the array *a*. *Printf* also supports output in exponential form using “%e” and “general” form using “%g”, while also providing support for specifying the field width and precision.

So how does *printf* and its cousins actually perform the conversion from a floating point number to ASCII decimal? It should not surprise you that we convert it to fixed point first, then convert it to ASCII.

The algorithm to convert from floating point to fixed point produces the same array of 4 unsigned 64-bit integers as we used for the conversion from ASCII. The binary point is in the same place – right in the middle.

We use a state structure to hold the unpacked float, along with some flags and the fixed point number:

struct FloatPrinter {

Unpacked u;

uint8\_t naninf;

uint64\_t fx[4];

};

#define FP\_NAN 1 // Non a Number

#define FP\_INF 2 // Infinity.

The *u* member is the unpacked float (sign, exponent and mantissa). The *naninf* member holds whether the number is one of the two special values: *FP\_NAN* or *FP\_INF*. The *fx* member is the 256-bit fixed point number.

The conversion proceeds as follows:

1. Check for zero, NaN and Infinity.
2. Unpack the float into the *Unpacked* struct.
3. Put the unpacked mantissa into the high word of the fractional part of the fixed point number, just to the right of the binary point. This is actually one bit to the right of where it should be, but we correct for that later.
4. Use the value of the exponent to shift the whole fixed point number left or right to the correct position. If the exponent is less than the bias value (127) we shift the number to the right and increment the exponent until the exponent is 127. If the exponent is greater than or equal to 127 (positive exponent) we shift to the left until the exponent is equal to 126. This is where the correction for the mantissa’s position is done. Since we’ve converted to fixed point we won’t be using the exponent value any more.

The result is a 256-bit fixed point number that represents the floating point mantissa shifted left or right by the floating point exponent.

Here is the function to fix a floating point number (convert it from floating point to fixed point). Try to match it up to the algorithm above.

void FixFloat(struct FloatPrinter\* printer, float f) {

// Check for special cases.

if (FPIsZero(f)) {

return;

}

if (FPIsNan(f)) {

printer->naninf = FP\_NAN; // NaN

return;

}

if (FPIsInfinity(f)) {

printer->naninf = FP\_INF; // Infinity

return;

}

// Extract the IEEE754 single precision to sign, exponent, and mantissa.

printer->u = Unpack(f);

// Put mantissa in the high end of the fractional part. The is one less

// than the exponent says.

printer->fx[1] = (uint64\_t)printer->u.mantissa << 32;

// Shift the mantissa left or right by the exponent.

// In IEE754 single precision, the exponent has a bias

// of 127, so 127 == 0, 128=1, 126=-1

if (printer->u.exponent < 127) {

// Negative exponent - shift right until exp is 127.

while (printer->u.exponent < 126) {

FixedRShift256(printer->fx);

++printer->u.exponent;

}

} else {

// Positive exponent, shift left.

while (printer->u.exponent >= 127) {

FixedLShift256(printer->fx);

--printer->u.exponent;

}

}

}

Now, to convert the fixed point number to ASCII we write the converted decimal digits into a buffer backwards. First, we convert the fraction, then write a decimal point, then we do the integer.

The reason for going backwards is that the digits of an integer are obtained by taking the modulus of the number with 10 and then dividing the number by 10, until it is zero. This produces the digits in reverse of how they are normally written down.

After a call to *FixFloat*, we have a *FloatPrinter* struct containing the information we need to do the conversion. The conversion also takes a *precision* argument, which is the number of significant figures to which we convert. After the precision is exhausted, we round the number and stop printing.

We first check for *NaN* and *Inf* and print them as special cases. Then we check for zero and call a special function to print the number 0 to the specified precision.

Assuming we have a non-special-case, we obtain a pointer to the end of the buffer and write a terminating NUL character there. The ASCII characters will be written backwards from just before that NUL character.

The *precision* is an important parameter in the algorithm as it tells us when to stop. It’s pretty easy to figure out that we have to stop when we reach zero, but there will be situations where there are a bunch of zero bits with a one bit following them somewhere in the fixed point number. The precision tells us how may decimal digits to produce after the decimal point.

The algorithm proceeds by multiplying the fixed point number by 10*p*, where *p* is the precision. We do this in a loop calling out *FixedMultiplyByTen256* function for each loop iteration. This has the effect of “shifting” the fixed point number to the left in base 10. However, if we have zero in the integral part and leading zeroes in the fractional part of the number, multiplying it by 10 will keep the integral part as zero. We need to keep track of the number of leading zeroes we have in the fraction. Remember that the whole fixed point number cannot be zero at this point because we’ve already checked for that and handled it specially.

So, in our multiplication loop, each time we multiply by 10 and have a zero in the integral part, we increment a *leading\_zeroes* variable, telling us that we have a leading zero in the fraction. Our loop terminates when we have multiplied by 10 the number of times specified by the precision.

After we have multiplied the number up, we then need to round it. If the top bit of the remaining fractional part of the number is 1 this represents a number 0.5 or greater, so we increment the integral part[[13]](#footnote-13). We are not going to use the remaining fractional part as it is beyond our requested precision. This rounding is accomplished by calling *FixedRound*.

// Round. If top bit of the fraction part of v is set we have a

// number >= 0.5. Round up.

void FixedRound(uint64\_t v[4]) {

if ((v[1] & (1LL << 63)) != 0) {

FixedIncrement128(&v[2]);

}

}

Now we have a bit pattern in the integral part of the number that represents our full precision number rounded appropriately. All we have to do is convert this to decimal in the given buffer, inserting the decimal point character in the appropriate place. To help with this we have a boolean variable *point\_printed* that tells us that we’ve printed the decimal point (we don’t want to print it twice).

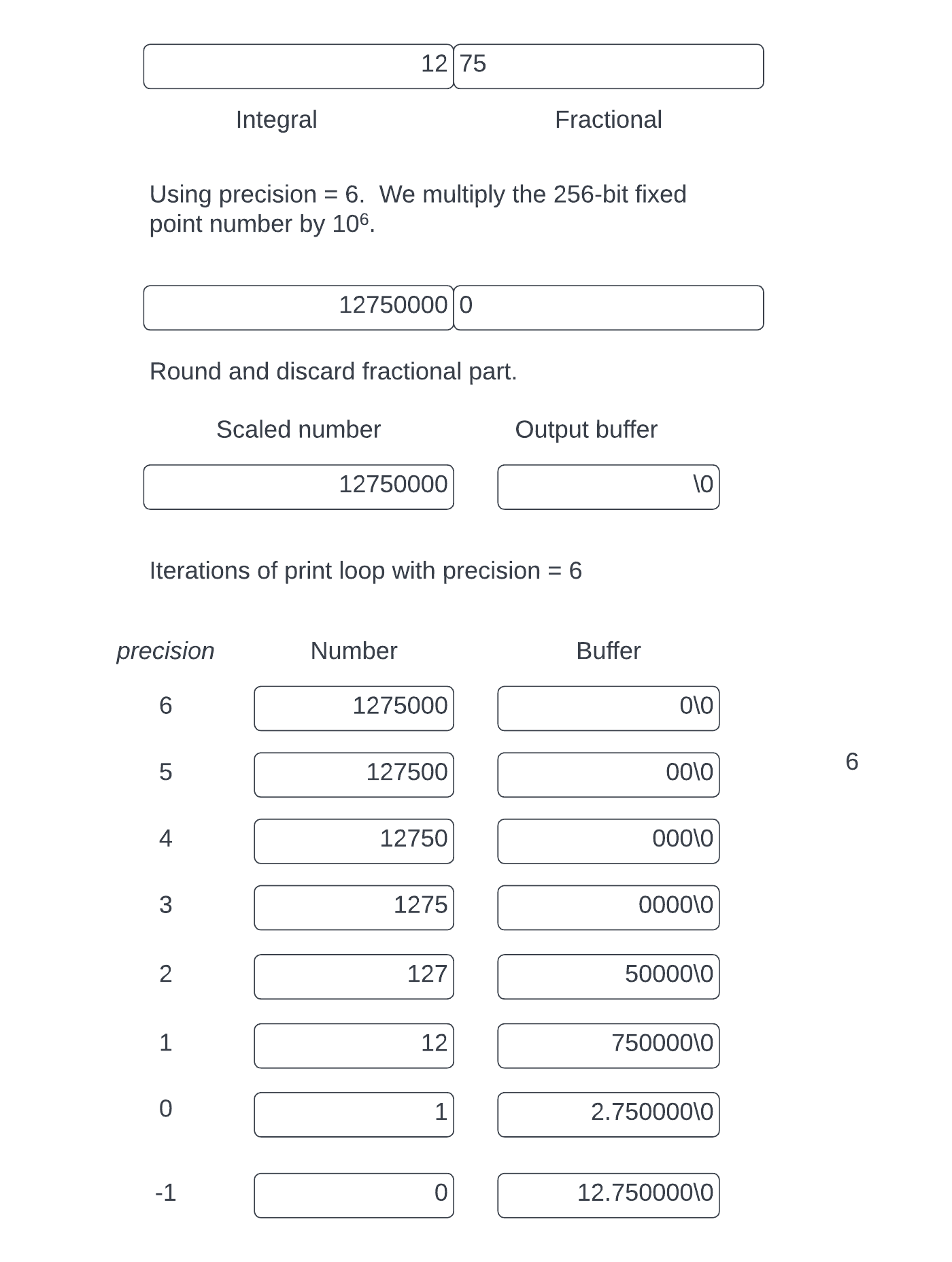
How do we know when to print the decimal point, then? Remember that we have multiplied our number by 10 to the power of the *precision* parameter? The value of precision specifies how many decimal digits to print before we print the decimal point. We are printing the number into the buffer backwards.

We enter a loop that terminates when the integral part of the fixed point number is zero. Each iteration calls *FixedDivideByTen128* to divide the 128-bit number by 10 and return the remainder. This remainder is a number between 0 and 9, so to convert it to ASCII we just add 0x30 (or the character ‘0’). We then insert this in the current position in the buffer and decrement the buffer pointer for the next iteration. In this loop, we also decrement the *precision* parameter and, if it attains the number 0, we print a decimal point character. The *precision* will continue to be decremented below zero so it will only be zero once.

We’re not quite done yet though. If the integral part of the original number was zero, we will reach zero before we print the decimal point so we will terminate the loop. In this case, we make use of the *leading\_zeroes* variable to tell us how many ‘0’ characters to print. Then we need to print “0.”.

Finally we print a ‘-‘ character if the number was negative.

Let’s look at converting the number **12.75** into ASCII with a precision of 6. The result in the output buffer will be “12.750000”. The last character in the buffer is ‘\0’ or ASCII NUL and terminates the sequence so that other functions know when it ends.



Note that when the precision reaches the value 0, we print a decimal point before we print the decimal digit.

Here’s the full *PrintFixedPoint* function. Like the others, please see if you can follow the algorithm as you read the code.

char\* PrintFixedPoint(struct FloatPrinter\* printer, int precision, char\* buf, size\_t size) {

if (printer->naninf != 0) {

return WriteNanInf(printer->u.sign, printer->naninf, buf, size);

}

if (FixedIsZero256(printer->fx)) {

return WriteZero(precision, buf, size);

}

char\* end = buf + size - 1;

char\* p = end;

\*p-- = '\0';

// Shift the number left (base 10) by the precision. This puts the

// value to be printed into the whole number part of the fixed point

// number. The first bit in the fraction is the rounding bit because

// it represents 0.5.

int8\_t leading\_zeroes = 0;

for (int8\_t i = 0; i < precision; i++) {

FixedMultiplyByTen256(printer->fx);

if (printer->fx[2] == 0) {

leading\_zeroes++;

}

}

// Round the whole part up if the fraction is >- 0.5.

FixedRound(printer->fx);

// Write the whole number in decimal, putting a decimal point

// in the appropriate place.

bool point\_printed = false;

while (!FixedIsZero128(&printer->fx[2])) {

if (precision == 0) {

\*p-- = '.';

point\_printed = true;

}

uint8\_t r = FixedDivideByTen128(&printer->fx[2]);

\*p-- = r + '0';

--precision;

}

// Write any leading zeroes.

for (uint8\_t i = 0; i < leading\_zeroes; i++) {

\*p-- = '0';

}

if (!point\_printed) {

\*p-- = '.';

\*p-- = '0';

}

if (printer->u.sign != 0) {

\*p-- = '-';

}

return p + 1;

}

The final functions to look at are the special case handling for zero, *NaN* and *Infinity*. *WriteZero* writes a bunch of zeroes to the buffer (in reverse) with a decimal point and *precision* digits after that.

// Write 0.00000 with precision number of digits after the point.

char\* WriteZero(int precision, char\* buf, size\_t size) {

char\* end = buf + size - 1;

char\* p = end;

\*p-- = '\0';

while (precision > 0) {

\*p-- = '0';

--precision;

}

\*p-- = '.';

\*p-- = '0';

return p + 1;

}

*WriteNanInf* write the string “nan” or “inf” to the buffer.

// Write nan or inf.

char\* WriteNanInf(int8\_t sign, int v, char\* buf, size\_t size) {

char\* end = buf + size - 4;

char\* p = end;

if (sign != 0) {

\*p++ = '-';

}

strcpy(p, v == FP\_NAN ? "nan" : "inf");

return end;

}

The *PrintFixedFloat* prints a floating point number in non-scientific format. The output can get very long if the number is big (with a large or small exponent). For such large numbers, it is more convenient to print using scientific format (otherwise known as exponential format). Let’s see how we can do that.

The algorithm is similar to *PrintFixedFloat* in that it multiplies the number up to the integral part of the fixed point number. The output is of the form:

X.YYYe±EE

Where X.YYY is a decimal number to a given precision, and EE is the power of 10 exponent. We start out with a 256-bit fixed point number that has been converted from floating point by *FixFloat*. The first thing we need to do is check for special values and short-circuit the effort as in *PrintFixedFloat*. Then we work out the base 10 exponent needed to make the integral part of the number in the range 1 to 10. If the number is greater than 10 we continually divide it by 10 and count the number of times we do that. Likewise, if the number is less than 1, we multiply it by 10 until it reaches a number between 1 and 10. The *CalculateExponent* function below performs this calculation. It takes the fixed point number as an argument and returns an integer representing the base 10 exponent it would take to bring the number into the required range. It doesn’t modify the fixed point number.

bool FixedIsLessThanTen128(uint64\_t v[2]) {

if (v[0] >= 10) {

return false;

}

for (int i = 1; i < 2; i++) {

if (v[i] != 0) {

return false;

}

}

return true;

}

// Given a 256 bit fixed point number, calculate the base-10 exponent

// needed to normalize it to a number between 1 and 10. Doesn't modify

// the input argument.

int CalculateExponent(uint64\_t v[4]) {

uint64\_t t[4];

memcpy(t, v, sizeof(t));

int exp = 0;

if (FixedIsZero128(&t[2])) {

// Less than 1, multiply by 10 until >= 1

while (FixedIsZero128(&t[2])) {

FixedMultiplyByTen256(t);

--exp;

}

} else {

while (!FixedIsLessThanTen128(&t[2])) {

FixedDivideByTen256(t);

++exp;

}

}

return exp;

}

Once we have the exponent required to normalize the number into the range 1..10, we shift the number (in base 10) to the correct position in the fixed point number. The naïve approach is to multiply or divide the number by 10 until the integral part is 1 and move on from there. If the number is greater than 10, we will need to use division to shift it to the right. However, division is a destructive operation in terms of accuracy, and we might lose precision. We need to avoid division if at all possible. The ultimate goal is to put all the digits of X.YYY into the integral part of the fixed point number. Like the *PrintFixedPoint* function, we do this by multiplying by 10P, where P is the precision.

To avoid unnecessary division, we use the exponent we’ve already calculated using *CalculateExponent* to determine how many powers of 10 we need to multiply or divide by to achieve the required range. The calculated exponent is positive if the number is greater than 10, and negative if the number was less than 1. Say the number is 99.8. The *CalculateExponent* function will return the value 1, meaning that we need to divide the number by 10 once in order to make it between 1 and 10 (9.98).

But we also need to multiply the number by 10P. This means we need to multiply the number by 10-exp and then multiply it by 10P. Algebra tells us we need to multiply the number by 10(P-exp). Multiplying by a negative exponent of 10 means division by the positive of the exponent.

Therefore, the algorithm to generate X.YYY is to calculate the difference *P - exp* and use that as a loop counter to multiply or divide by 10.

Once we have the fixed point number in the correct position, we can then write it out backwards into the buffer. It starts with the exponent, which is written using the *WriteExponent* helper function[[14]](#footnote-14).

// Write the exponent backwards starting at buf. Return the start of

// the result. Always writes 3 or 4 chars:

// +/-[X]XX

char\* WriteExponent(int exp, char\* buf) {

bool negative = false;

if (exp < 0) {

exp = -exp;

negative = true;

}

int ndigits = 2;

while (exp != 0 || ndigits > 0) {

uint8\_t d = exp % 10;

\*buf-- = d + '0';

exp /= 10;

--ndigits;

}

\*buf-- = negative ? '-' : '+';

return buf;

}

The value of *exp* comes from *CalculateExponent* and the *WriteExponent* function always writes 2 digits of the integral exponent, prefixed by a ‘+’ or ‘-‘ sign.

After the exponent is written into the buffer, we proceed to write out the precision-length integral part, placing the decimal point in the correct place.

Here’s the *PrintFixedPointScientific* function in its entirety.

char\* PrintFixedPointScientific(struct FloatPrinter\* printer, int precision, char\* buf, size\_t size) {

if (printer->naninf != 0) {

return WriteNanInf(printer->u.sign, printer->naninf, buf, size);

}

if (FixedIsZero256(printer->fx)) {

char\* end = buf + size - 1;

\*end-- = '\0';

char\* p = WriteExponent(0, end);

\*p-- = 'e';

return WriteZero(precision, buf, p - buf + 1);

}

int exp = CalculateExponent(printer->fx);

// We need to avoid division as much as possible to keep the accuracy

// up. The naive algorithm is to normalize the number to 1.xeEE

// and then multiply it up by 10^precision, but if the number

// is greater than 1 this will involve division by 10 then multiplication

// by 10.

//

// We already have the power-of-10 exponent but we want the result

// to be this multiplied by 10^precision.

//

// Let the fixed point number be N, exponent be 'e' and precision be P.

//

// The result we want is 10^-e \* 10^P

// Which is 10^(P-e)

//

// The number of base-10 shifts is the precision minus the exponent.

// We shift right by division and left by multiplication.

int shifts = precision - exp;

if (shifts < 0) {

for (int i = shifts; i < 0; i++) {

FixedDivideByTen256(printer->fx);

}

} else {

for (int i = 0; i < shifts; i++) {

FixedMultiplyByTen256(printer->fx);

}

}

FixedRound(printer->fx);

char\* end = buf + size - 1;

\*end-- = '\0';

char\* p = WriteExponent(exp, end);

\*p-- = 'e';

while (!FixedIsZero128(&printer->fx[2])) {

if (precision == 0) {

\*p-- = '.';

}

uint8\_t r = FixedDivideByTen128(&printer->fx[2]);

\*p-- = r + '0';

--precision;

}

if (printer->u.sign) {

\*p-- = '-';

}

return p+1;

}

## What about double precision, then?

As anyone who has worked with floating might realize, single precision numbers aren’t very precise at all. In fact, they have about 7 decimal digits of precision. The larger the integral part is, the fewer digits are available for the fraction. Most math-heavy applications use double precision numbers because of the increased precision, albeit with twice the space consumption. Given that modern computers come with floating point vector units built in and have gobs of memory to spare, the use of double precision is very attractive.

So, what’s the difference between single and double precision floating point numbers? It’s really only a matter of the number of bits allocated for the exponent and mantissa parts. Whereas a single precision number has 8 bits for its exponent and 23 bits for the mantissa, a double precision number has 11 bits for its exponent as 53 bits for the mantissa. This gives about 16 decimal digits of precision, more than enough for most applications.

I chose single precision for this article because the numbers are smaller and easier to manage, but exactly the same algorithms can be used for any length of numbers. For a double precision float, the fixed point number needs space for 4096 bits, or 512 bytes, with the binary point at the halfway point. This gives 2048 bits for the integral part and 2048 for the fraction[[15]](#footnote-15).

## Optimization

There are a number of locations in the C code where a less than optimal solution was presented in order to aid clarity. For a real floating point implementation, it would behoove the implementer to provide an optimal solution. Here are some places I can think of where it could be optimized:

1. Use the integer multiplication and division operations provided by the compiler instead of doing it from first principles
2. When multiplying or dividing by powers of 10 using loops, provide a table of pre-calculated power-of-ten constants and perform only one multiply or divide operation.
3. For the large integer operations (shifts, addition, etc.), use assembly language to obtain and use the carry flag from the processor to improve performance.
4. Instead of shifting the whole 256-bit fixed point number and keeping the binary point in a fixed location, it might be better to be able to move the binary point, keeping track of its location. However, be careful as it might make things slower.
5. The code was designed to run on a 64-bit computer. If you are targeting a smaller computer, use the native word size instead of 64-bit integers[[16]](#footnote-16).

As an example of an optimization for a 6502 processor, here’s an assembly language routine to unpack a single precision number into zero page locations. The input is an IEEE 754 single precision float in zero page, with the X index register being its offset. The result is the number unpacked into zero page locations *fexp,* *fmantissa* and *fsign*. The mantissa occupies 40 bits and we unpack the float’s mantissa into the middle 3 bytes, with bytes 0 and 4 being set to zero. This is to allow space for overflow and underflow when performing math operations on it.

The routine is actually for the 65c02, WDC’s currently available CMOS version (the original MOS 6502 is hard to find these days). You can tell by the use of the STZ instruction to store a zero into memory.

// Input:

// X: index of float in zero page.

// Output:

// fexp: exponent with bias

// fmantissa: middle 3 bytes set to mantissa

// fsign: sign (0x80 or 0)

FPUnpack:

LDA 3,X

STA fexp // Bottom 7 bits of exponent (with sign bit)

AND #0x80 // Get sign bit.

STA fsign

// Put 23 bit mantissa in middle bytes of fmantissa.

STZ fmantissa+0

LDA 0,X

STA fmantissa+1

LDA 1,X

STA fmantissa+2

LDA 2,X

STA fmantissa+3

STZ fmantissa+4

// Top bit of mantissa is bottom bit of exponent - shift it in.

ASL A // A contains top byte of mantissa. MSB is LSB of exponent.

ROL fexp

// Store implicit 1 in bit 31 of mantissa.

LDA fmantissa+3

ORA #0x80

STA fmantissa+3

RTS

And here’s the routine to pack an unpacked float in zero page into another location in zero page. In this case the input is in *fexp*, *fmantissa* and *fsign*, and the output is put into zero page at the location whose offset is passed in X. It uses an additional zero page location *t1* as a temporary scratch space.

// Pack a 32-bit float.

// fexp: 8 bit exponent with bias.

// fsign: sign bit (0x80 or 0)

// fmantissa: 40 bits of mantissa with explicit 1 in bit 31.

// X: index into zero page for result.

FPPack:

// Exponent and sign are in top byte

LDA fexp

LSR A // Carry contains bottom bit of exponent.

ORA fsign // Sign is 0x80 or 0.

STA 3,X // Store sign and top 7 bits of exponent.

// Put bottom bit of exp (in carry) in top bit of t1

STZ t1

ROR t1

// Take bytes 1,2 and 3 from mantissa scratch.

LDA fmantissa+3

AND #0x7f // Remove implicit 1.

ORA t1 // OR in bottom bit of exponent.

STA 2,X

LDA fmantissa+2

STA 1,X

LDA fmantissa+1

STA 0,X

RTS

1. This is a personal project and is unrelated to my employer. [↑](#footnote-ref-1)
2. The image above shows my 65C02 computer under development [↑](#footnote-ref-2)
3. All the code has been tested and seems to produce the same results as the floating point unit on my computer. [↑](#footnote-ref-3)
4. Or more accurately, English with a Northern Irish accent I guess. [↑](#footnote-ref-4)
5. A good description can be found at <https://en.wikipedia.org/wiki/Fixed-point_arithmetic>, albeit in base 10 and not 2. [↑](#footnote-ref-5)
6. For a real-world floating point number, we need to use something reasonable for the sizes of the mantissa and exponent. For a 32-bit representation, we could choose to use 24 bits for the mantissa and 8 bits for the exponent. [↑](#footnote-ref-6)
7. Modern compilers will use an optimization technique called Return Value Optimization (RVO) to optimize out any unnecessary copies of u. [↑](#footnote-ref-7)
8. Technically we are *shifting* the integer to the right and *rotating* the carry into the mantissa, but C doesn’t support rotation operations directly. [↑](#footnote-ref-8)
9. I’m afraid I don’t know why it’s necessary to subtract 1 from the exponent to produce the result. Perhaps someone could enlighten me. [↑](#footnote-ref-9)
10. Assembly language would be a much better choice for these operations since you get access to the carry flag without all the extra work. [↑](#footnote-ref-10)
11. The names come from the 6502 processor. [↑](#footnote-ref-11)
12. Yes, yes, I know that I said I wouldn’t use multiplication in this article, but avoiding it here adds nothing to the explanation. [↑](#footnote-ref-12)
13. This doesn’t use IEEE 754’s ‘prioritize even number’ rule. I don’t know if it should or not. [↑](#footnote-ref-13)
14. Integer division used here. Like multiplication in the conversion from ASCII, there’s no point in avoiding it for this function. [↑](#footnote-ref-14)
15. Also, since I was working on a 6502 processor, double precision just isn’t worth the effort as it uses too much memory and is too slow. Back in the day, most original 6502 floating point used a 5-byte mantissa, which gave a little more acceptable precision. [↑](#footnote-ref-15)
16. Indeed, on a 6502 it would be better to avoid shifts altogether and use byte-based operations (shifting by a multiple of 8 bits can be done by just copying bytes). Shifting by more than one bit in a 6502 is expensive. [↑](#footnote-ref-16)