Attention:

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Let's Get to the (Floating) Point

Chris Hecker

Question: Is floating-point math inherently evil? Answer: Maybe not, if the instruction timings of modern chips are to be believed. Question: But are they to be believed?

When I sat down to write this article, it was supposed to be the last installment in our epic perspective texture mapping series. Part of the way through, I realized I needed to cover what ended up as the actual topic of this article—floating-point optimizations—to complete the optimizations on the perspective texture mapper's inner loop. So we'll learn some generally cool tricks today and apply them to the texture mapper in the next issue. In fact, these techniques are applicable to any high-performance application that mixes floating-point and integer math on modern processors. Of course, that's a long and drawn-out way of saying the techniques are eminently suitable to cool games, whether they texture map or not.

The Real Story

A few years ago, you couldn't have described a game as an "application that mixes floating-point and integer math," because no games used floating-point. In fact, floating-point has been synonymous with slowness since the beginning of the personal computer, when you had to go out to the store, buy a floating-point coprocessor, and stick it into a socket in your motherboard by hand. It's not that game developers didn't want to use real arithmetic, but the original PCs had enough trouble with integer math, let alone dealing with the added complexities of floating-point.

We don't have enough space to cover much of the history behind the transition from integer math, through rational (Bresenham's line-drawing algorithm, for example) and fixed-point arithmetic, and finally to floating-point, but here's the quick summary: For a long time, game developers only used floating-point math in prototype algorithms written in high-level languages. Once prototyped, the programmers usually dropped the code down into fixed-point for speed. These days, as we'll see, floating-point math has caught up with integer and fixed-point in terms of speed and in some ways has even surpassed it.

To see why, let's look at the cycle timings of the most common mathematical operations used by game developers (addition, subtraction, multiplication, and—hopefully relatively rarely—division) for fixed-point, integer, and floating-point. Table 1 shows the cycles times on three generations of Intel processors, the PowerPC 604, and a modern MIPS. We see that the floating-point operations, with the exception of additions and subtractions, are actually faster than their integer counterparts on the modern processors (the 386, a decidedly nonmodern processor without an integrated floating-point unit, lags far behind, and the transitional 486 has mixed results).

Of course, these numbers alone don't tell the whole story. The table doesn't show that, although the cycle times are still slow compared to single-cycle instructions like integer addition, you can usually execute other integer instructions while the longer floating-point operations are running. The amount of cycle overlap varies from processor to processor, but on really long instructions, like floating-point

Table 1. Various Instruction Timings (Parentheses Indicate Single Precision)

	Integer Add/Subtract	Float Add/Subtract	Integer Multiply	Float Multiply	Integer Divide	Float Divide
Intel 386/387	2	23-34	9-38	27-35	43	89
Intel i486	1	10	12-42	11	43	62 (35)
Intel Pentium	1	3	10	3	46	33 (19)
PowerPC 604	1	3	4	3	20	31 (18)
MIPS R4x00	1	4	10	8 (7)	69	36 (23)

division, you can usually overlap all but a few cycles with other integer (and sometimes even floating-point) instructions. In contrast, the longer integer instructions allow no overlap on the current Intel processors and limited overlap on the other processors.

On the other hand, the floating-point operations are not quite as fast as Table 1 implies because you have to load the operands into the floating-point unit to operate on them, and floating-point loads and stores are usually slower than their integer counterparts. Worse yet, the instructions to convert floating-point numbers to integers are even slower still. In fact, the overhead of loading, storing, and converting floating-point numbers is enough to bias the speed advantage towards fixed-point on the 486, even though the floating-point instruction timings for the actual mathematical operations are faster than the corresponding integer operations.

Today, however, the decreased cycle counts combined with the tricks and techniques we'll cover in this article give floating-point math the definite speed advantage for some operations, and the combination of floating-point and fixed-point math is unbeatable.

If It Ain't Float, Don't Fix It

As usual, I'm going to have to assume you know how fixed-point numbers work for this discussion. Mathematically speaking, a fixed-point number is an integer created by multiplying a real number by a constant positive integer scale and removing the remaining fractional part. This scale creates an integer that has a portion of the original real number's fraction encoded in its least significant bits. This is why fixed-point was the real number system of choice for so many years; as long as we're consistent with our scales, we can use fast integer operations and it just works, with a few caveats. It has big problems with range and is a mess to deal with, for the most part. You need to be very careful to avoid overflow and underflow with fixed-point numbers, and those "fast" integer operations aren't as fast as the same floating-point operations anymore.

Floating-point math, on the other hand, is a breeze to work with. The main idea behind floating-point is to trade some bits of precision for a lot of range.

For the moment, let's forget about floating-point numbers and imagine we have really huge binary fixed-point numbers, with lots of bits on the integer and fractional sides of our binary point. Say we have 1,000 bits on each side, so we can represent numbers as large as 2^{1000} and as small as 2^{1000}. This hypothetical fixed-point format has a huge range and a lot of precision, where range is defined as the ratio between the largest and the smallest representable number, and precision is defined as how many significant digits (or bits) the representation has. So, for example, when we're dealing with incredibly huge numbers in our galaxy simulator, we can still keep the celestial distances accurate to within subatomic particle radii.

However, most applications don't need anywhere near that much precision. Many applications need the range to represent huge values like distances between stars or tiny values like the distance between atoms, but they don't often need to represent values from one scale when dealing with values of the other.

Floating-point numbers take advantage of this discrepency between precision and range to store numbers with a very large range (even greater than our hypothetical 2,000-bit fixed-point number, in fact) in very few bits. They do this by storing the real number's exponent separately from its mantissa, just like scientific notation. In scientific notation, a number like 2.345×10^{35} has a mantissa of 2.345 and an exponent of 35 (sometimes you'll see the terms significand and characteristic instead of mantissa and exponent, but they're synonymous). This number is only precise to four significant digits, but its exponent is quite big (imagine moving the decimal point 35 places to the right and adding zeros after the mantissa runs out of significant digits).

The way the precision scales with the magnitude of the value is the other important thing. When the exponent is 35, incrementing the first digit changes the value by 10^{35}, but when the exponent is 0, incrementing the first digit only changes the value by 1. This way you get angstrom accuracy when you're on the scale of angstroms, but not when you're on the scale of galaxies (when you really don't need it).

The IEEE floating-point standard specifies floating-point representations and how operations on those representations behave. The two IEEE floating-point formats we care about are "single" and "double" precision. They both share the same equation for conversion from the binary floating-point representation to the real number representation, and you'll recognize it as a binary form of scientific notation:

$$-1^{sign} \, x2^{exponent-bias} \, x1.mantissa \qquad (1)$$

The only differences between the two formats are the widths of some of the

named fields in Equation 1, so we'll go over each part of it in turn and point out the differences when they pop up.

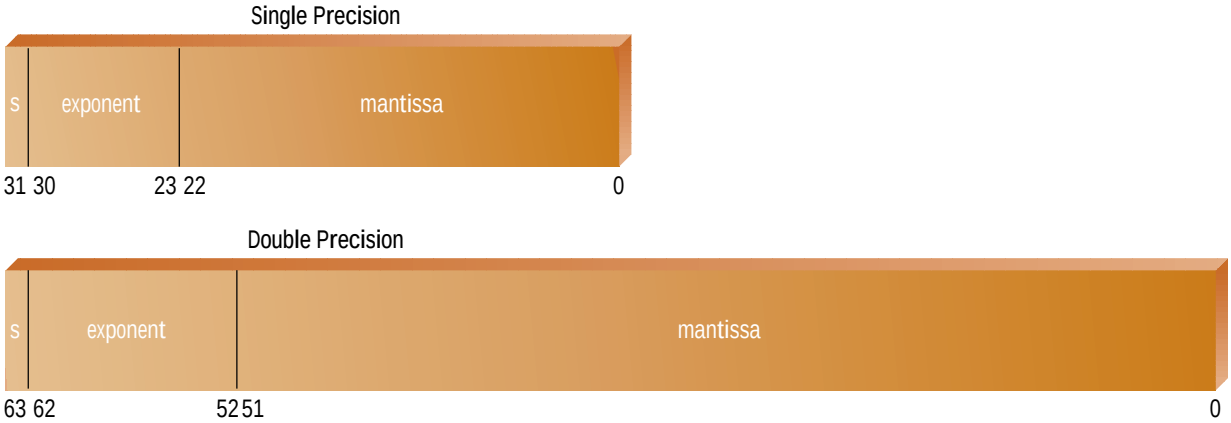
We'll start on the right-hand side of Equation 1. The mantissa expression (the 1.mantissa part of the equation) is somewhat strange when you first look at it, but it becomes a little clearer when you realize that mantissa is a binary number. It's also important to realize that it is a normalized, binary number. "Normalized" for scientific numbers means the mantissa is always greater than or equal to 1 and less than 10 (in other words, a single, non-zero digit). Our previous example of scientific notation, 2.345×10^{35}, is normalized, while the same number represented as $234{,}500 \times 10^{30}$ is not. When a binary number is normalized, it is shifted until the most significant bit is 1. Think about this for a second (I had to). If there are leading zeros in the binary number, we can represent them as part of the exponent, just as if there are leading 0 digits in our decimal scientific notation. And because the most significant bit is always 1, we can avoid storing it altogether and make it implicit in our representation. So, just like normalized decimal scientific notation keeps its mantissa between 1 and 10, the binary mantissa in a floating-point number is always greater than or equal to 1 and less than 2 (if we include the implicit leading 1).

Next in line, the exponent expression shifts the binary point right or left based on a positive or negative exponent, respectively. This is exactly analogous to the base-10 decimal scientific notation, where the exponent shifts the decimal point right or left, inserting zeros as necessary. The exponent field is the key to the range advantage floating-point numbers have over fixed-point numbers of the same bit width. While a fixed-point number has an implicit binary point and all the bits are, in essence, a mantissa, a floating-point number reserves an exponent field to shift the binary point around (hence the term, "floating-point"). It's clear that 8 bits reserved for an exponent from a 32-bit word allows a range from about 2^{127} to 2^{-127}, while the best a fixed-point number could do would be a 32-bit range, for example 2^{32} to 0, 2^{16} to 2^{-16}, or 0 to 2^{-32}, but not all at the same time. However, there's no such thing as a free lunch; the 32-bit fixed-point number actually has better precision than the floating-point number, because the fixed-point number has 32 significant bits, while the floating-point number only has 24 significant bits left after the exponent is reserved.

You'll notice the exponent expression is actually exponent - bias. The bias is a value set so that the actual bits of the exponent field are always positive. In other words, assuming the exponent is 8 bits and the bias is 127, if you want your unbiased exponent to be -126 you set your biased exponent bits to 1. Likewise, a biased exponent field value of 254 yields an unbiased exponent of 127. Exponent values of all 0s and all 1s are reserved for special numerical situations, like infinity and zero, but we don't have space to cover them.

Finally, the sign bit dictates whether the number is positive or negative. Unlike two's complement representations, floating-point numbers that differ only in sign also differ only in their sign bit. We'll discuss the implications of this further. Table 2 contains the field sizes for single and double precision IEEE floating-point values, and Figure 1 shows their layout, with the sign always at the most significant bit.

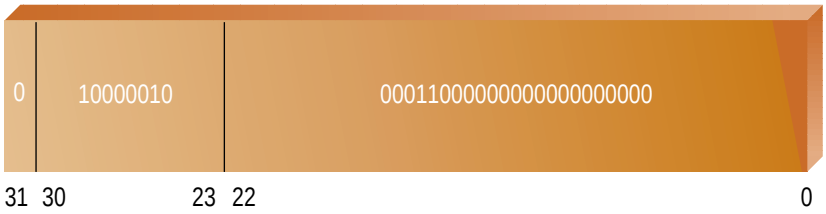
Figure 1. IEEE Floating-Point Layouts



An Example

Let's run through an example by converting a decimal number into a single precision, binary, floating-point number. We'll use the number 8.75 because it's easy enough to do by hand, but it still shows the important points. First, we turn it into a binary fixed-point number, 1000.11, by figuring out which binary bit positions are 1 and 0. Remember, the bit positions to the right of the binary point go 2^{-1}, 2^{-2}, 2^{-3}, and so on. It should be clear that I chose .75 for the fractional part because it's $2^{-1} + 2^{-2}$, so it's easy to calculate. Next, we shift the binary point three positions to the left to normalize the number, giving us 1.00011×2^{3}. Finally, we bias this exponent by adding 127 for the single precision case, leaving us with 130 (or 10000010 bina-

Figure 2. 8.75 As a IEEE Single Precision Value



ry) for our biased exponent and 1.00011 for our mantissa. The leading 1 is implicit, as we've already discussed, and our number is positive, so the sign bit is 0. The final floating-point number's bit representation is shown in Figure 2.

Now that we're familiar with floating-point numbers and their representations, let's learn some tricks.

Conversions

I mentioned that on some processors the floating-point to integer conversions are pretty slow, and I wasn't exaggerating. On the Pentium, the instruction to store a float as an `int`, `FIST`, takes six cycles. Compare that to a multiply, which only takes three, and you see what I mean. Worse yet, the `FIST` instruction stalls the floating-point pipeline and both integer pipelines, so no other instructions can execute until the store is finished. However, there is an alternative, if we put some of the floating-point knowledge we've learned to use and build our own version of `FIST` using a normal floating-point addition.

In order to add two floating-point numbers together, the CPU needs to line up the binary points before doing the operation; it can't add the mantissas together until they're the same magnitude. This "lining up" basically amounts to a left shift of the smaller number's binary point by the difference in the two exponents. For example, if we want to add 2.345×10^{35} to 1.0×10^{32} in decimal scientific notation, we shift the smaller value's decimal point 3 places to the left until the numbers are the same magnitude, and do the calculation: $2.345 \times 10^{35} + 0.001 \times 10^{35} = 2.346 \times 10^{35}$. Binary floating-point works in the same way.

We can take advantage of this alignment shift to change the bit representation of a floating-point number until it's the same as an integer's bit representation, and then we can just read it like a normal integer. The key to understanding this is to realize that the integer value we want is actually in the floating-point bits, but it's been normalized, so it's shifted up to its leading 1 bit in the mantissa field. Take 8.75, as shown in Figure 2. The integer 8 part is the implicit 1 bit and the three leading 0s in the mantissa. The following 11 in the mantissa is .75 in binary fractional bits, just waiting to be turned into a fixed-point number.

Imagine what happens when we add a power-of-two floating-point number, like 2^8=256, to 8.75, as in Figure 3. In order to add the numbers, the CPU shifts the 8.75 binary point left by the difference in the exponents (8 - 3 = 5, in this example), and then completes the addition. The addition itself takes the implicit 1 bit in the 256 value and adds it to the newly aligned 8.75, and when the result is normalized again the implicit 1 from the 256 is still in the implicit 1-bit place, so the 8.75 stays shifted down. You can see it in the middle of the mantissa of the result in Figure 3. What happens if we add in 223, or the width of the mantissa? As you'd expect, the 8.75 mantissa is shifted down by 23 - 3 = 20, leaving just the 1,000 for the 8 (because we shifted .75 off the end of the single precision mantissa, the rounding mode will come into play, but let's assume we're truncating towards zero). If we read in the resulting single precision value as an integer and mask off the exponent and sign bit, we get the original 8.75 floating-point value converted to an integer 8!

This trick works for positive numbers, but if you try to convert a negative number it will fail. You can see why by doing the aligned operation by hand. I find it easier to work by subtracting two positive numbers than by adding a positive and a negative. Instead of 2^{23} + (-8.75), I think of 2^{23} - 8.75. The single sign bit representation lends itself to this as well (using a piece of paper and a pen will really help you see this in action). So, when we do the aligned subtraction, the 8.75 subtracts from the large value's mantissa, and since that's all 0s (it's a power-of-two), the subtract borrows from the implicit 1 bit. This seems fine at first, and the mantissa is the correct value of -8.75 (shifted down), but the normalization step screws it up because now that we've borrowed from the implicit 1 bit, it's no longer the most significant bit, so the normalization shifts everything up by one and ruins our integer.

But wait, all is not lost. All we need is a single bit from which to borrow in the mantissa field of the big number so that the subtraction will leave the implicit 1 bit alone and our number will stay shifted. We can get this 1 bit simply by multiplying our large number by 1.5. 1.5 in binary is 1.1, and the first 1 becomes the implicit 1 bit, and the second becomes the most significant bit of the mantissa, ready to be used for borrowing. Now negative and positive numbers will stay shifted after their normalization. The masking for negative numbers amounts

Table 2. Floating-Point Field Widths and Parameters

	Total Width	Sign	Mantissa	Exponent	Bias Value
Single	32 bits	1 bit	23 bits	8 bits	+127
Double	64 bits	1 bit	52 bits	11 bits	+1023

Figure 3. Adding Single Precision Floating-Point Numbers

to filling in the top bits with 1 to complete the two's-complement integer representation.

The need to mask for positive and negative values is bad enough when you are only dealing with one or the other, but if you want to transparently deal with either, figuring out how to mask the upper bits can be slow. However, there's a trick for this as well. If you subtract the integer representation of our large, floating-point shift number (in other words, treat its bits like an integer instead of a float) from the integer representation of the number we just converted, it will remove all the high bits properly for both types of numbers, making the bits equal zero for positive values and filling them in with ones for negative values.

You'll notice that this technique applied to single precision values can only use a portion of the full 32 bits because the exponent and sign bits are in the way. Also, when we use the 1.5 trick we lose another bit to ensure both positive and negative numbers work. However, we can avoid the range problems and avoid masking as well by using a double precision number as our conversion temporary. If we add our number as a double (making sure we use the bigger shift value—2^{52} x 1.5 for integer truncation) and only read in the least significant 32 bits as an integer, we get a full 32 bits of precision and we don't need to mask, because the exponent and sign bits are way up in the second 32 bits of the double precision value.

In summary, we can control the shift amount by using the exponent of a large number added to the value we want to convert. We can shift all the way down to integer truncation, or we can shift part of the way down and preserve some fractional precision in fixed-point.

This seems like a lot of trouble, but on some processors with slow conversion functions, like the x86 family, it can make a difference. On the Pentium with `FIST` you have to wait for six cycles before you can execute any other instructions. But using the addition trick, you can insert three cycles worth of integer instructions between the add and the store. You can also control how many bits of fractional precision you keep, instead of always converting to an integer.

What's Your Sign?

Before I wrap this up, I'd like to throw out some other techniques to get you thinking.

The exponent bias is there for a reason: comparing. Because the exponents are always positive (and are in more significant bits than the mantissa), large numbers compare greater than small numbers even when the floating-point values are compared as normal integer bits. The sign bit throws a monkey wrench in this, but it works great for single-signed values. Of course, you can take the absolute value of a floating-point number by masking off the sign bit.

I've already hinted at the coolest trick—overlapping integer instructions while doing lengthy divides—but I haven't gone into detail on it. It will have to wait until next time, when we'll discuss this in depth.

Two people introduced me to the various tricks in this article and got me interested in the details of floating-point arithmetic. Terje Mathisen at Norsk Hydro first showed me the conversion trick, and Sean Barrett from Looking Glass Technologies made it work on negative numbers.

If you want to learn more about floating-point, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (Addison-Wesley, 1981) by D. Knuth is a good source for the theory. Most CPU programming manuals have fairly detailed descriptions as well. You can get Adobe Portable Document Format versions of the PowerPC manuals on http://www.mot.com. If you really want to understand floating-point and its implications for numerical programming, you'll want to pick up a numerical analysis textbook that deals with computers.

You also might want to look at the *Graphics Gems* series from AP Professional. The series covers a number of floating-point tricks like the ones discussed here. A good example calculates a quick and dirty square root by halving the exponent and looking up the first few bits of the mantissa in a table. Another takes advantage of the format to do quick absolute values and compares for clipping outcode generation. Once you understand the floating-point format, you can come up with all sorts of tricks of your own. ■

Chris Hecker tries to stay normalized, but he can be biased at checker@bix.com.