IEEE 754 Floating point standard

Professor Paul J Conrad

FIXED POINT BINARY

We can represent real numbers in terms of fixed point binary (this is really easy but will not be as “accurate” due to how many bits we use to the left and right of the radix point).

Consider the decimal real number: 3.75_{10}

We can represent it in fixed point binary as: 0011.1100_2

The radix point (the ’.’) is what we use to separate the integer part (to the left of the point), and the fractional part to the right. You should be familiar with this ‘.’ being known as the decimal point when naturally working with decimal numbers in math or science. In this document, it will be referred to as the radix point, decimal point and ”binary point” depending on the number base we are working with (pay attention closely to any subscripts you see!!!).

FIXED POINT BINARY

A fixed point binary number can typically be 8 bits, 16 bits, 32 bits, 64 bits, etc., depending your program’s requirements. The layout can be as follows:

xxxxxx.yyyyyy

Where xxxxxx is the integer part to the left of the radix point in incrementing powers of N, where N is the number base you are working with, starting with N^0 and incrementing by 1 as you go left. To the right of the radix point is yyyyyy, or the fractional part in decrementing powers of N, starting at N^{-1} and decrementing by 1 as you go to the right.

Example: $x_3x_2x_1x_0.y_{-1}y_{-2}y_{-3}y_{-4}$ (base 2) can be expanded as:

$x_3*2^3 + x_2*2^2 + x_1*2^1 + x_0*2^0 + y_{-1}*2^{-1} + y_{-2}*2^{-2} + y_{-3}*2^{-3} + y_{-4}*2^{-4}$

The number of bits to the left and right of the radix point can be predetermined based on your program requirements and is often “programmatically fixed”. For example: you can have a scheme for a 32 bit fix binary as 20 bits to left and 12 bits to the right, or 8 bits to the left and 24 bits to the right (the first will not have as many decimal places of accuracy as the second).

FIXED POINT BINARY

Here are some examples (all are with varying precision to the left and right denoted as NxM, where N is number of bits to the left, and M is the number of bits to the right, sum of N and M must be a power of 2, ie. 8, 16, 32, 64, etc):

Example 1 (4x4):

$7.375_{10} = ?_2$

We apply the division algorithm we covered for the integer side of the radix point, and replace division with multiplication from our division algorithm for the right hand side of radix point.

7_{10} is 0111_2 from what we've done with converting to integers. The right side for the fractional part is doing the opposite of division (aka multiplication), we will take the integer part of our multiplication and put it in our result and zero out the integer from our result for the next multiplication. Repeat until result is 0.00000…. Or the number of fractional bits is filled.

FIXED POINT BINARY

Working on the $.375_{10}$ part is as follows with the multiplication algorithm described on the last slide:

1) $0.375_{10} * 2 = 0.75_{10}$, take out the 0 to the left of radix and add to our fixed point binary (denoted as bold and underlined), we should have: 0111.**<u>0</u>**
2) $0.75_{10} * 2 = 1.50_{10}$, take out the 1 to the left of radix and add to our fixed point binary (denoted as bold and underlined), we should have: 0111.0**<u>1</u>**, note for step #3 we do not include the leading 1, we "took it out" and put it in our fixed point binary number.
3) $0.5_{10} * 2 = 1.00_{10}$, take out the 1 to the left of radix and add to our fixed point binary (denoted as bold and underlined), we should have: 0111.01**<u>1</u>**, now when we "took out" the 1, we have 0.00 for step #4, and since we have our new number in the form of 0.0000… we are done and can fill any remaining fractional bits to zero as 0*2=0, our number in binary is: 0111.0110_2

We can verify this as:

$0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-2} + 0*2^{-3} =$

$4 + 2 + 1 + 0.25 + 0.125 = 7.375_{10}$

IEEE 754 FLOATING POINT

The Fixed Point Binary is pretty simple but does not address several issue:

1. Easy, intuitive approach for signed values
2. What about exponents?
3. What about degree of precision?

To address these three key issue, we will redirect our attention to IEEE 754 Floating Point Standard (also known as the IEEE 754 Standard).

Q: Why must we have a standard? A: So everyone in the industry can be in agreement for compatibility across various architectures, compilers, programming languages, etc.

The IEEE 754 Standard states the arrangement of bits must be in the following from left most bit to rightmost:

Sign Bit (1 bit)	Exponent*	Mantissa (fractional part)**

* = 8 bits for float, 11 bits for double

– 23 bits for float, 52 bits for double

IEEE 754 FLOATING POINT

Sign Bit (1 bit)	Exponent*	Mantissa (fractional part)**

* = 8 bits for float, 11 bits for double

** = 23 bits for float, 52 bits for double

The sign bit, 0 = positive number, 1 = negative number

The exponent is stored as an unsigned biased value. The bias value is an offset that is approximately halfway between the minimum negative exponent and maximum positive exponent (IEEE decided to use this technique so an extra bit doesn't have to be used for the exponent sign bit).

The formula for calculating the bias value is as follows:

$$\text{bias} = \left(2^{(\text{number\_bits\_for\_exponent}-1)} \right) - 1$$

So, for 32 bit float, having 8 bit exponent, the bias is 127; and for 64 bit double, having 11 bit exponent, the bias is 1023.

IEEE 754 FLOATING POINT

Perhaps the easiest way to explain the bias (or offset) is to consider the relationship between Celsius degrees and Kelvin degrees.

In science, 0 degrees Kelvin is “absolute zero” and 0 degrees Kelvin is approximately -273.15 degrees Celsius. To convert degrees Celsius to degrees Kelvin, we simply add 273.15 to the Celsius value to get the Kelvin value. We could say that 273.15 is the “offset” or the bias.

degrees_Kelvin = 273.15 + degrees_Celsius

Using this information, we could “store Celsius values” in the realm of unsigned values by adding 273.15 to the value; to get the Celsius value back, we simply subtract 273.15 from the Kelvin value. Here are some values, note how all Kelvin values are “unsigned”:

-273.15 Celsius = 0 Kelvin

0 Celsius = 273.15 Kelvin

-40 Celsius = 233.15 Kelvin

100 Celsius = 373.15 Kelvin

IEEE 754 FLOATING POINT

With this idea, we can use bias or offset to store positive and negative exponents in “the realm of unsigned values” and hence, save one extra bit from being used as a sign bit. This savings of one extra bit will later translate to one extra bit of precision for the mantissa. We will add to our bias the value of our exponent when converting from real/decimal to IEEE 754 Standard, and subtract the bias from our stored value when converting back from IEEE 754 to real/decimal.

Other important math to cover:

1. 32 bit float has min/max exponent of -38, +38, respectively
2. 64 bit double has min/max exponent of -308, +308, respectively

These just are not some numbers ”magically” pulled out of the air, but they can be calculated by the following idea:

- In binary (base 2), it takes approximately 3.32192 bits to represent all digits from 0 to 9 in decimal ($2^{3.32192}$ = 9.99994)

Calculate the following: 127/3.32192 and -127/3.32192 (truncate the fraction), also do the same calculations for: 1023/3.32192 and -1023/3.32192

IEEE 754 FLOATING POINT

You have also seen that 32 bit float has about 7 to 8 digits of accuracy, and 64 bit double has about 14 to 16 digits of accuracy. Knowing how many bits you have for your mantissa, you can calculate as follows:

32 bit float: 23 bits / 3.32192 = 6.924 or about 7

32 bit float*: 25 bits / 3.32192 = 7.52 or about 8

- We consider 2 extra bits due to rounding the last bit and also the implicit storage of the leading 1 in our binary scientific number (more on this in next slide).

64 bit double: 52 bits / 3.32192 = 15.6 or about 16

64 bit double*: 54 bits / 3.32192 = 16.3 or about 16

IEEE 754 FLOATING POINT

Binary Scientific notation is key to putting everything together and is not really any different than decimal scientific notation. Consider the following decimal scientific notation:

13.568×10^2

Would make math and science audience pause for a moment because the value is unnormalized. We have to normalize the value to be properly correct, the normalization is moving the decimal point left or right until the only number to left is any number between 1 and 9 (for decimal, for binary, can only be 1). So the correct representation of the above number in decimal scientific notation is:

1.3568×10^3 (we moved radix point, aka decimal point left 1 digit, increment exponent by 1).

We can do binary scientific notation such as:

1.1010×2^2 (this is normalized because to the left of the binary point or radix point is 1, if we have a 0, then it is unnormalized). This number in decimal would be $1.625 \times 4 = 6.5_{10}$

IEEE 754 FLOATING POINT

Okay, we got all the math and theory stuff out of the way, let’s convert 1.25_{10} to the IEEE 754 Standard:

1. Sign bit is going to be 0 since this is a positive number.
2. Convert 1.25 to binary scientific notation (left of decimal we will use division algorithm, right of decimal – the multiplication algorithm shown in the Fixed Point Binary section)
3. Left of the decimal point in binary is 0001 (padded leading zeros for clarity), right of decimal point is: 0.0100...
4. Here’s the math for finding right side:
    1. 0.25 * 2 = 0.50, take the 0 from left side and place it after the binary point, having: 0001.**<u>0</u>** (where the bold, underline is what was concatenated)

    2. 0.50 * 2 = 1.00, take the 1 from left side and place it after the last value we concatenated, having now: 0001.0**<u>1</u>** (since we now have 0.00 – remember the 1 was taken out and added to end of our number), we can pad the rest of the number with zeros if need (you will need to in your work), we have now:

    1.01000… x 2^0 as our binary scientific notation of 1.25_{10}

    In 32 bit float IEEE 754 we have: 0 0111 1111 01000000000000000000000

    First 0 is our sign bit, the bit pattern 01111111 is 127, which is our bias of 127 plus the exponent of 0, the last 23 bits (01000000000000000000000) is our 1.0100000000000… with the leading 1 excluded as it is “implicitly” stored – we do this to gain one extra bit of accuracy, in the computer’s memory this 32 bit float will look like: 0011 1111 1010 0000 0000 0000 0000 0000 (I broke it into nibbles for it to be easier to see), in hexadecimal it is: $3FA0\ 0000_{16}$

IEEE 754 FLOATING POINT

Another example, let's convert -121.1$_{10}$ to the IEEE 754 Standard:

1. Sign bit is going to be 1 since this is a negative number.
2. Convert 121.1 (we make it positive since we already counted the sign bit to be negative) to binary scientific notation (left of decimal we will use division algorithm, right of decimal – the multiplication algorithm shown in the Fixed Point Binary section)
3. Left of the decimal point in binary is 0111 1001 (padded leading zero for clarity), right of decimal point is:
4. Here's the math for finding right side:
    1. 0.10 * 2 = 0.20, take the 0 from left side and place it after the binary point, having: 0111 1001.**<u>0</u>** (where the bold, underline is what was concatenated)

    2. 0.20 * 2 = 0.40, take the 0 from left side and place it after the last value we concatenated, having now: 01111001.0**<u>0</u>**

    3. 0.04 * 2 = 0.80, take the 0 from left side and place it after the last value we concatenated, having now: 01111001.00**<u>0</u>**

    4. 0.80 * 2 = 1.60, take the 1 from left side and place it after the last value we concatenated, having now: 0111 1001.000**<u>1</u>**

    5. 0.60 * 2 = 1.20, take the 1 from left side and place it after the last value we concatenated, having now: 0111 1001.0001 **<u>1</u>**

    6. 0.20 * 2 = 0.40, WAIT A MOMENT!!! We've seen this 0.20 * 2 in step #2, this number is repeating (all negative powers of 10, 0.1, 0.2 … are repeaters in binary), we can carry out this repeating for the number of digits we need once we see the pattern.

    we have now:

    0111 1001.0001 1001 1001 1001 1001 1001… x 2^0 as our binary scientific notation of 121.1$_{10}$

    But we have a problem, it's not normalized, so normalize it!

IEEE 754 FLOATING POINT

0111 1001.0011 0011 0011 0011 0011 0011… x 2^0 is normalized as:

01.11 1001 0011 0011 0011 0011 0011 0011… x 2^6 (we moved radix point left 6 bits)

In 32 bit float IEEE 754 we have: 1 10000101 11100110011001100110011

First 1 is our sign bit, the bit pattern 10000101 is 133, which is our bias of 127 plus the exponent of 6, the last 23 bits (11100011001100110011001) is our 1.11100011001100110011001… with the leading 1 excluded as it is "implicitly" stored. Note we have a 24th bit that I have bolded and underlined, if we are rounding up, it is a 1 and we can round up the 22nd and 23rd bits to 10_2 – if not rounding, we simply truncate that 24th bit. In the computer's memory this 32 bit float will look like: 1100 0010 1111 0010 0011 0011 0011 0011 (with rounding up assumed), in hexadecimal it is: C2F1 $999A_{16}$