DAT093
Introduction to Electronic System Design
Fractional numbers

Introduction

When we work with fixed point digital systems (see below) involving calculations on numbers that have started as analog values and then have been sampled and analog-to-digital converted and that are later supposed to be digital-to-analog converted back to analog values it is in most cases beneficial to look at our sample values and our constants, for example filter constants, as fractional numbers. One reason for this is that the values, like filter constants, we get from DSP design tools in most cases are represented in fractional format. Now what do we mean by fractional numbers? To answer that we will start with a short sum up of number representation.

Fixed point number representation

In fixed point numbers we have a representation where the number of integer digits and the number of fractional digits in the numbers is fixed, that is the position of the decimal point within the number is fixed. If we focus on binary numbers the representation is described as I.B where I is the number of integer bits while B is the number of binals.

If we have an integer number this could, with eight bits, be represented in format 8.0. We have 8 bits to represent the integer part of the number and no bits for the binal part and that is of course OK for an integer number since there are no binals here.

For unsigned integer numbers we have

$$b_7b_6b_5b_4b_3b_2b_1b_0 = b_7\cdot 2^7+b_6\cdot 2^6+b_5\cdot 2^5+b_4\cdot 2^4+b_3\cdot 2^3+b_2\cdot 2^2+b_1\cdot 2^1+b_0\cdot 2^0$$

that is the value span is 0 to +255.

For a signed integer number with a positive value we have the representation

$$0b_6b_5b_4b_3b_2b_1b_0 = b_6\cdot 2^6+b_5\cdot 2^5+b_4\cdot 2^4+b_3\cdot 2^3+b_2\cdot 2^2+b_1\cdot 2^1+b_0\cdot 2^0$$

giving the range 0 to +127. Negative numbers are represented by the 2´s complement of this and we actually have one extra negative value so the range is -1 to -128. In total then the range is -128 to +127.

Fractional numbers

The other extreme is a number represented on 0.8 format, that have no integer bits and eight fractional bits and that is described by

$$b_7b_6b_5b_4b_3b_2b_1b_0 =$$

$$= b_7\cdot 2^{-1}+b_6\cdot 2^{-2}+b_5\cdot 2^{-3}+b_4\cdot 2^{-4}+b_3\cdot 2^{-5}+b_2\cdot 2^{-6}+b_1\cdot 2^{-7}+b_0\cdot 2^{-8}$$

The smallest number we can represent here, besides zero, is

$2^{-8} = 0.00390625$

and the largest value is

$2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8} = 0.99609375 = 1-2^{-8}$

Observe that the largest value is close to, but not quit, one (1). Also observe that since there is no sign bit we can only represent natural values, that is zero and positive values.

This representation with all values smaller than one (1) is what we call *fractional numbers* but we will change the representation a bit below to include negative fractions.

We can of course have representations in between these formats like a 4.4 format with four binal bits and four integer bits for unsigned numbers or three integer bits and a sign bit if the numbers are signed.

For unsigned numbers this would give the description

$b_7b_6b_5b_4b_3b_2b_1b_0 = b_7\cdot 2^3+b_6\cdot 2^2+b_5\cdot 2^1+b_4\cdot 2^0+b_3\cdot 2^{-1}+b_2\cdot 2^{-2}+b_1\cdot 2^{-3}+b_0\cdot 2^{-4}$

and the smallest value possible, besides zero, would be

$2^{-4} = 0.625$

And the biggest

$2^3+2^2+2^1+2^0+2^{-1}+2^{-2}+2^{-3}+2^{-4} = 15.9375 = 16-2^{-4}$

So we have the value span 0 to 15.9375.

For signed 4.4 numbers the value interval would be -8 to +7.9375. Formats like this might be beneficial for interimistic results within a calculation but seldom for the final result.

We said that 0.8 format only can represent positive numbers. We can introduce a sign bit by using 1.7 format instead and this is what we normally mean by fractional numbers.

1.7 fractional format

We will now expand our fractional description and introduce a sign bit giving a 1.7 format for eight bits.

The fractional 1.7 format means that positive values are described as

$0b_6b_5b_4b_3b_2b_1b_0 = b_6\cdot 2^{-1}+b_5\cdot 2^{-2}+b_4\cdot 2^{-3}+b_3\cdot 2^{-4}+b_2\cdot 2^{-5}+b_1\cdot 2^{-6}+b_0\cdot 2^{-7}$

and the smallest value that we can represent is

$2^{-7} = 0.0078125$

While the largest value is

$2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7} = 0.9921875 = 1-2^{-7}$

Negative numbers are represented by the 2´s complement of the magnitude and the largest number we can represent is one (1), but of course with a negative sign so we have -1.

Now how do we convert our decimal representation to fractional number? Let´s say that the decimal value is 0.37. We could convert it to a binary word on 1.7 format by trying to sum up 2^{-n} factors in the correct way, but there is a simpler way.

Fractionally the largest value is one (1), even if we cannot quit reach this on the positive side. If we convert our 1.7 format number to an integer we have one sign bit and seven integer values where our largest value would be 128 (2^7), even if we as mentioned not can reach this value on the positive side. Now this means that we can go from fractional numbers to its integer correspondence by multiplying our decimal number by 128 and then interpret the integer value as binary bits. What we are doing is relating our value to the largest integer value that can be represented with the current number of bits.
In our example we have

$$0.37 \rightarrow 0.37 \cdot 2^7 = 0.37 \cdot 128 = 47.36 \approx 47$$

We have to round or truncate to the nearest integer so we lose some accuracy in the conversion but that is unavoidable when we have a limited number of bits.

Now why is this simpler? There are lots of tools that can convert between decimal (integer) number and binary, for example the calculator in Windows if we restrict ourselves to 8, 16, 32 or 64 bits numbers, and we can use these tools to get the binary representation of the fractional numbers. Using this we get

$$47_{10} = 0010\ 1111_2$$

If we have 16 bits instead we would use the 1.15 format and the maximal value would be 2^{15} = 32768 and the conversion would be

$$0.37 \rightarrow 0.37 \cdot 32768 = 12124.16 \approx 12124$$

and this gives

$$12124_{10} = 0010\ 1111\ 0101\ 1100_2$$

Observe that the most significant bits, that are part of both representations, are the same in the 1.7 and 1.15 representations except for rounding when we have fewer bits. They should be the same since they represent the same sign bit and the same 2^{-n} factors. The difference between the formats is that the 1.15 format gives a better accuracy by including eight (8) more fractional bits.

Real world interpretation

In the beginning of this text we stated that fractional numbers could be a good choice for signals that are converted from analog values or for values that should be converted to analog values. In these cases, a good way to look at the fractional numbers is to realize that the fractional number describes the value as a fraction of the maximal signal value the ADC or DAC can handle.

Multiplication of fractional numbers

Now let´s try to multiply two numbers and first treat them as integers on 8.0 format and then treat the same value as a fractional number on 1.7 format.

Let´s pick something simple

$0100\ 0000_2 = 64_{8.0} = 0.5_{1.7}$

and

$0010\ 0000_2 = 32_{8.0} = 0.25_{1.7}$

If we multiply the binary numbers we get

$0100\ 0000 \cdot 0010\ 0000 = 0000\ 1000\ 0000\ 0000$

```
0100 0000
          ·0010 0000
____________________
           0000 0000
          0 0000 000
         00 0000 00
        000 0000 0
       0000 0000
     0 1000 000
    00 0000 00
+ 000 0000 0       _
____________________
 0000 1000 0000 0000
```

Observe that the multiplication results in a doubled number of bits, that is 8 + 8 = 16. Actually it is the sum of the number of bits in the two numbers.

What is this in integer form?

$0000\ 1000\ 0000\ 0000_2 = 2048_{8.0} = 64 \cdot 32$

So the result is correct.

Now let´s look at is as fractional numbers. The multiplication of two binary numbers is the same and correct but what do we get when we interpret it as a fractional number?

$0000\ 1000\ 0000\ 0000_2 = 2048_{8.0} \rightarrow 2048/2^{15} =$

$= 2048/32768 = 0.0625$

A direct multiplication of our original decimal numbers would give

$0.5 \cdot 0.25 = 0.125$

We compare and see that

$0.0625 \neq 0.125$

So the result is incorrect. What is going on?

If we test and multiply a number of number pairs we will notice that the result will always be the expected value divided by two and it is actually here the explanation lies.

When we multiply two numbers the format of the result will be the sum of the formats of the two multiplied numbers. That is in this case we have the format of the result

```
1.7 + 1.7 = 2.14
```

As we can see there are **two** sign bits in the result but we want to have only one in our final result so we have to shift the result one bit to the left to get 1.15 format. If we do this the result will be correct. If you think of it; the result is that the least significant bit will always be zero (0) because of the left shift so we actually have a true 1.14 format.

This will always apply so if we for example multiply an integer number on 8.0 format and a number on fractional 1.7 format the result will have the format

```
8.0 + 1.7 = 9.7
```

Addition of fractional numbers

Addition or subtraction of fractional numbers is no problem. What we need to do for these operations is to align the binal points, that is place them in the same position in the number, and this is already done when the two numbers are in the same format.

Addition of numbers on different formats

What about addition of fixed point numbers of different formats? We need to make sure to align the binal points, that is if we are to add a number on 4.4 format to a number on 1.7 format we need to place the numbers according to

```
 SIII.BBBB0000
+SSSS.BBBBBBBB
```

In the equation S is a sign bit, I is an integer bit, B is a binal bit and 0 is just a zero. As we can see the result will be on 4.7 format if we keep the full number of bit. Observe that we should add sign bits and not zeros to the left in the second number.

Changing the number of bits in fractional numbers

Sometimes we want to change the number of bits in a fractional number. For example, when we implement a FIR filter with input data on 1.7 format and filter constants in the same format the result, after shifting, will be on 1.15 format but in most cases we want to plug the filter in between two devices that are connected by just a bus with one single representation, in this case 1.7 format. This means that we have to reduce the number of bits in the result from the FIR filter from 16 to 8, that is go from 1.15 to 1.7 format. To do this we keep the sign bit and the most significant bits of the 1.15 number. That is, we take bit 8 to 15 and discard bit 0 to 7, we truncate the number. We could improve the accuracy by rounding the result instead of truncating it but that takes some extra hardware so we have to decide in each situation if this is motivated.

What we just said doesn´t mean that we should truncate to 8 bits after each multiplication in our FIR filter calculation. The result would be that each truncation (or rounding) will give a truncation (or rounding) error and these will add up at the output. In most cases, we should keep more than eight bits within the internal calculations and only truncate (or round) at the output. To double the number of bits internally might be a good choice but in more advanced calculations we may have to internally increase the number of bits even more. This is also a case where it might be good to internally have more than one integer (sign) bit for the internal results to avoid overflow.

On the other hand if we like to increase the number of bits we should still keep the position of the binal point and add the required number of zeroes to the right of the LSB.