# Lab 6: Fixed-Point Implementation

EE 352 DSP Laboratory (Spring 2019)

**Lab Goals**

- Understand fixed-point representation of numbers and fixed-point computations.
- Compare fixed-point and floating-point implementations of an FIR filter in terms of cycle count and accuracy.
- Handling overflow issues.

## 1 Introduction

TMS320C5515 is a fixed-point processor. Fixed-point arithmetic is generally used when hardware resources are limited and we can afford a reduction in accuracy in return for higher execution speed. Fixed-point processors are either 16-bit or 24-bit devices, while floating point processors are usually 32-bit devices. A typical 16-bit processor such as the TMS320C55x, stores data as a 16-bit integer or a fraction format in a fixed range. Although signals are only stored with 16-bit precision, intermediate values during the arithmetic operations may be kept at 32-bit precision. This is done using the internal 40-bit accumulator, which reduces cumulative round-off errors. Fixed-point DSP devices are usually cheaper and faster than their floating-point counterparts because they use less silicon, have lower power consumption, and require fewer external pins. Most high volume low cost embedded applications, such as appliance control, cellular phones, hard disk drives, modems, audio players, and digital cameras use fixed-point processors.

Floating-point arithmetic greatly expands the dynamic range of numbers. A typical 32-bit floating point DSP processor, such as the TMS320C67x, represents number with a 24-bit mantissa and an 8-bit exponent. The mantissa represents a fraction in the range -1.0 to +1.0, while the exponent is an integer that represents the number of places that the binary point must be shifted left or right in order to obtain the true value. For example, in decimal number system we have some number like 10.2345. We can write this in the form of $0.102345 \times 10^2$. So in this format, 0.102345 is called mantissa and 2 is called exponent.

A 32-bit floating-point format covers a large dynamic range, thus the data dynamic range restriction may be virtually ignored in a design using floating-point DSP processors. But in fixed-point format, the designer has to apply scaling factors and other techniques to prevent arithmetic overflow. This is usually a difficult and time consuming process. As a result, floating-point DSP processors are generally easy to program and use, but are more expensive and have higher power consumption. In this session, you will learn the issues related to fixed-point arithmetic.

## 2 Fixed-point Formats

### Two's complement Integer Representation

This representation is most common method for representing a signed integer. The two's complement of a binary number is defined as the value obtained by subtracting the number from a large power of two (specifically, from $2^N$ for an N-bit two's complement). As far as the hardware is concerned, fixed-point number systems represent data as $B$-bit integers. The two's-complement number system usually used is:

$$k = \begin{cases} binary\ integer\ representation, & \text{if } 0 \le k \le 2^{B-1} - 1 \\ bitwise\ complement\ of\ k+1, & if - (2^{B-1}) \le k \le 0 \end{cases} \tag{1}$$

The most significant bit is known as the sign bit. It is 0 when the number is non-negative and 1 when the number is negative.

Figure 1 is an easy way to visualize two's complement representation.

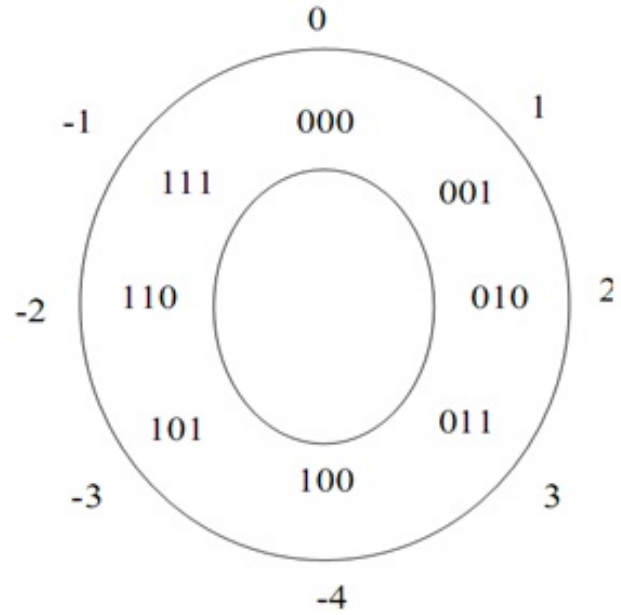


Figure 1: Two's complement representation

## Fractional fixed-point number representation

For the purposes of signal processing, we often regard the fixed-point numbers as binary fractions in the range [-1, 1), by implicitly placing a decimal point after the sign bit. Fixed-point representation of a fractional number x is illustrated in figure 2.

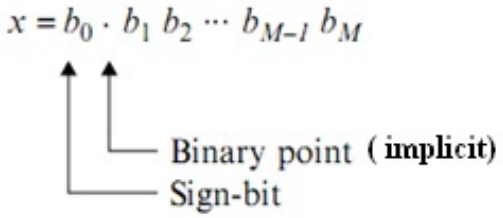


Figure 2: Fixed-point representation of binary fractional numbers

The word-length is $B(= M + 1)$ bits, i.e. $M$ magnitude bits and one sign bit. The most significant bit (MSB) is the sign bit, which represents the sign of the number as follows.

$$b_0 = \begin{cases} 0, & \text{if } x \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{2}$$

The remaining $M$ bits give the magnitude of the number. The rightmost bit $b_M$ is called the least significant bit (LSB), which represents precision of the number.

The decimal value corresponding to a binary fractional number x can be expressed as,

$$x = b_0 + b_1 2^{-1} + b_2 2^{-2} + \cdots + b_M 2^{-M} \tag{3}$$

$$x = b_0 + \sum_{m=1}^{M} b_m 2^{-m} \tag{4}$$

Figure 3 provides a visual representation for a 3 bit fractional representation.

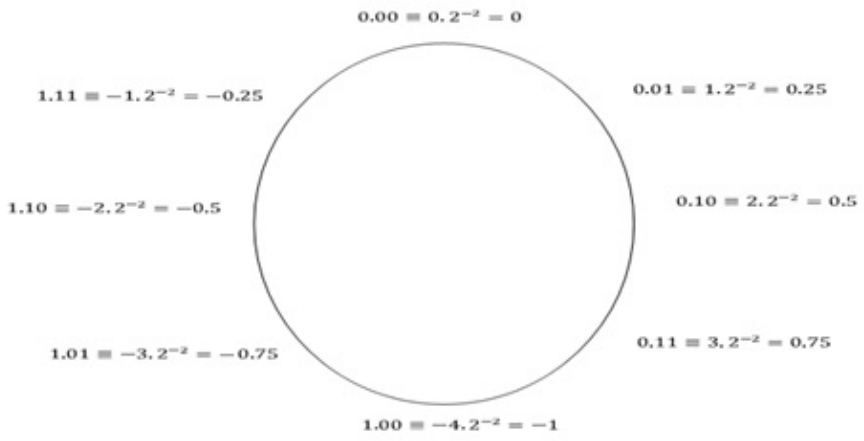


Figure 3: Fixed-point representation of binary fractional numbers that uses 3 bits

For example, from Figure 2, the easiest way to convert a normalized 16-bit fractional binary number into an integer that can be used by C55x assembler is to move the binary point to the right by 15 bits (at the right of $b_M$). Since shifting the binary point 1 bit right is equivalent to multiplying the fractional number by 2, moving the binary point to the right by 15 bits can be done by multiplying the decimal value by $2^{15} = 32768$.

### Q-format

Q-format is a formal mechanism to keep track of radix or fixed point. Format Qnm is illustrated in Figure 4, where $m = M = B - 1$ . There are $n$ bits to the left of the binary point representing integer portion, and $m$ bits to the right, representing fractional value.

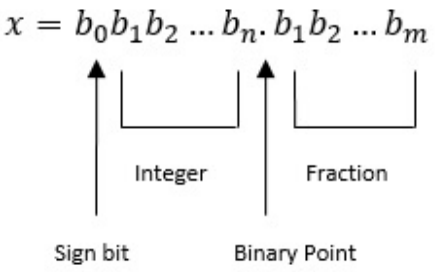


Figure 4: general binary fractional number

The most popular fractional number format is Q0.15 format ($n = 0$ and $m = 15$) , which is simply referred to as Q15 format since there are 15 fractional bits.

## 3 Integer Word-Length (IWL)

The number of bits required to express the integer part of a floating point number is called as the *integer word-length* of that number. IWL of a number gives us an idea of the maximum precision with which it can be represented in a finite sized shift register. For example, the IWL of 56.25 is 6 (Why?). Therefore, it can be stored in a 16 bit register just by using the 6 LSBs. But, this results in wastage of the 9 MSBs which all are zeros. Thus, to store 56.25 in a 16 bit register, we will have to multiply it by $2^{15-6}$, i.e., $2^9 = 512$, so that

it occupies the whole 16 bit register. Note that, the decimal point is implicit while storing the floating point number. To understand this better, carry out the multiplication yourself and represent the product in signed binary format. Compare this representation with that of 56. This concept of IWL and its use for effective storage is used in section 5.

# 4 Round-off Error in Fixed Point Implementation

Fixed-point arithmetic is often used with DSP hardware for real-time processing because it offers fast operation and relatively economical implementation. Its drawbacks include a small dynamic range and low resolution. A fixed-point representation also leads to arithmetic errors, which if not handled correctly will lead to erroneous operations. These errors have to be handled in such a way as to have some trade-off among accuracy, speed and cost.

Consider the multiplication of two binary fractions, as shown in figure4. From above calculation, we see that

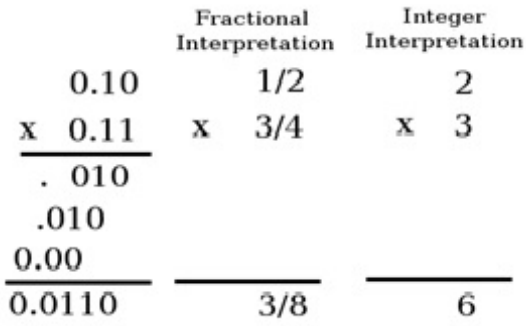


Figure 5: Demonstration of round-off error

full-precision multiplication almost doubles the number of bits. If we wish to return the product to a $b$-bit representation, we must truncate the $(b-1)$ least significant bits. However this introduces truncation error. Truncation error is also known as round off error (the number is rounded to the nearest $b$-bit fractional value rather than truncated). This type of error occurs after the multiplication.

# 5 Example code

Open the `fixed_float.c` file uploaded on the wiki-page. Two arrays `x_floating` and `y_floating` of size 100 each, are defined globally. We want to perform the operation `y_floating[i] = Ax_floating[i]+B` where $A = 3.2$ and $B = 2.7$. The function `compute_floating()` carries out this operation with the help of floating point variables. On the other hand, the `compute_fixed()` deals with only the finite precision data-types such as `short int`, `long int` etc. It demonstrates the steps involved in the conversion of floating-point computations to fixed-point. This code is self explanatory. For better understanding, Refer `https://www.youtube.com/watch?v=om7Bvk7WzJg&list=PLV5Sz2W8vMyagrItexjFRw2X_X99qpX6v`.

## 5.1 Learning checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

You will be using `fixed_float.c` file uploaded on the lab wiki-page for both the checkpoints.

1. Function `compute_floating()` computes $y[i] = Ax[i]+B$ using regular floating-point arithmetic. Firstly, call only this function in `main` and perform profiling to record the clock-cycle count of its execution. Repeat this exercise with `compute_fixed()` function (Call only one of the two functions at a time in `main`). Which one is faster? Now, call both the above functions in main and compare the output arrays `y_floating` and `y`. Which one is closer to the actual answer?

2. Modify the given C code (`compute_floating()`) with the values of `x_floating[i]` taking floating point values between 4000 and 4100 (as specified below) and convert this code to fixed point inside the function `compute_fixed()`. The floating point values may be assigned to `x_floating[]` as shown in the following code snippet.

```
for (i=4000; i<4100; i++)
{
     x_floating[i-4000] = i + 0.875;
}
```

   So, instead of $[2000, 2099]$, `x[i]` will take values from the range $[4000.875, 4099.875]$ and you will have to calculate $y[i] = Ax[i] + B$. Note that, since `x_floating[i]` itself contains floating point values, we cannot use the trick of calculating `xf[i]` by just bitwise left shifting the values in `x[i]`. You will have to actually multiply the array `x_floating[i]` by correct value to get `xf[i]`.

3. Now, store the output values of the exercises carried out in checkpoint 2 from section 5.1 in separate dat files called `floating.dat` and `fixed.dat`, respectively. Write a **function** in MATLAB named `compare.m` which will

   (a) receive the names of these two dat files as input arguments,

   (b) read the values stored in them, and

   (c) output the absolute values of the difference between them in an array.

   The syntax of the function should be `abs_diff = compare('fixed.dat','floating.dat')`. If you are getting all zeros in `abs_diff` when displayed, then check the current setting of the floating point value display by typing `get(0,'format')` in the command window. If it is `short`, then you will have to change it to `long`. Type `doc format` for information about `format` and changing the floating point number display settings.

*Note: Try to keep the scaling of variables down to the minimum while avoiding overflow.*

# 6 Overflow issues

Many signal processing algorithms mainly involve convolution i.e. multiply and accumulate (MAC) operation. During these operations there are chances that the result might go above the available range leading to an overflow. This overflow result can be made to either saturate to a fixed value or wrapped-around depending on the requirement.

Saturated output is the case wherein overflowed value is represented by a fixed value which is the maximum value possible without overflow. In the wrap-around case, values exceeding the maximum value "wraps around" to the minimum value and vice-versa.

Thus, to deal with these cases, the TMS320C55x DSPs have a 40 bit accumulator which has 8 guard bits in addition to the 32 bits. In most of the cases, the result of the convolution operation can be well accommodated in the 32 bits. On the other hand, there are possibilities that the result of MAC operation can go beyond 32 bits and thus having a 40 bit accumulator proves beneficial. We will try to practically learn these aspects using the C55xx DSP. For this, we will convolve two predefined sequences in a way to witness a 32 bit overflow. Further we will see how a 40 bit accumulator will help us to avoid overflow and proper results.

**Further reading:** `https://en.wikipedia.org/wiki/Saturation_arithmetic`

### Overflow Error

Consider the addition of two binary fractions as shown next,

From the above calculation, we can see that the sign bit of the output has been changed indicating a wrap-around or overflow. It may be possible that if signals or input have been properly normalized in the range of -1 to +1 for fixed point arithmetic, the sum of two $B$-bit numbers may fall outside the range of -1 to +1. The

| | Fractional Interpretation | Integer Interpretation |
|---|---|---|
| 0.10 | 1/2 | 2 |
| + 0.11 | + 3/4 | + 3 |
| 1.01 | 5/4 = -3/4 | 5 = -1 |

term overflow is a condition in which the result of arithmetic operation exceeds the capacity of the register used to hold that result. When using a fixed point processor, the range of numbers must be carefully examined and adjusted in order to avoid overflow. This may be achieved by using different Qnm format with desired dynamic range.

### Steps to demonstrate a 32 bit overflow

1. Make use of the filtering code (lab3) using linear buffering to perform the convolution operation using **stored inputs** and filter coefficients.
2. Predefine your own set of inputs and coefficient values such that the output magnitude gets beyond 31 bits. (For ex: both the input and impulse response sequence can be a train of impulses having identical and maximum possible magnitude. Also pad zeros at the end of input to get a triangular shape output)
3. Next in the assembly code prior to the convolution. Clear the `M40` bit (a status bit) using `BCLR M40` instruction to change the data operation mode to 32. Accumulator will now perform as a 32 bit register and any overflow in $32^{nd}$ bit will be shown by the overflow flag ($11^{th}$ bit from LSB in the `ST0` register in case we use `AC0` as the accumulator).
4. Performing the filtering will result in an overflow if appropriate inputs as mentioned in step 2 have been selected.
5. The overflow can be set to either wrap or saturate by clearing (`BCLR SATD`) or setting (`BSET SATD`) the `SATD` register respectively. You can see the overflow by looking at the graph of output and also monitoring the ST0 register mentioned in above step 3.

Once we have witnessed this overflow we will try to do away with it by using all 40 bits in the accumulator.

### Steps to demonstrate a way to avoid 32 bit overflow

1. Keep the input and the coefficient sequences same as above.
2. In assembly code, set the `M40` bit (`BSET M40`) for 40 bit data operation. Now the above mentioned overflow flag i.e. the 11th bit of `ST0` will be set in case of a 40 bit overflow. 40 bit overflow is very unlikely to happen.
3. After the MAC operation is executed, we know that, after certain iterations, the output is getting overflowed above 32 bits and therefore we need to consider the guard bits to get proper output with some loss in precision.
4. Since in our filtering code we are taking the higher 16 bits of accumulator as our result (with some loss in precision), output in this case should consist of 8 guard bits + 8 MSBs.
5. To extract the 8 guard bits, shift the accumulator content to the right by 8 bits (`SFTL AC0, #-8`).
6. Now looking at the output graph, you can see a proper output which in previous case without guard bits was distorted due to overflow.

### 6.1 Learning checkpoints

- For this exercise, use the `overflow.c` file uploaded on EE352 wiki page. The file consists of simple convolution operation on specific kind of input. Inputs are chosen such that there are chances of overflow while performing 32 bit arithmetic operation (MAC operation). Firstly, paste the `asm` file, in which you wrote your code for convolution using linear buffering in lab 3, into the project folder. `overflow.c` file is written such that it adheres to all the naming conventions we used in lab3. Check the output of convolution in watch-window/Graph and corresponding overflow bit in Status Register (`ST0`). And give your comments on overflow for following two cases,

  `M40` = 0, `SATD` = 0

  `M40` = 0, `SATD` = 1

  Singed bit considered

- For this exercise, implement same linear buffer with 40 bit operation mode as discussed in previous sections. This program solves the problem of overflow. See the output of convolution on watch-window/ Graph and corresponding overflow bit in Status Register (`ST0`). Give your comments on the output values. What should be the scaling factor to get the output values close to original (Real-world values)?

### 6.2 Learning Checkpoint

- Try your own set of inputs to get overflow on 40 bit accumulator after convolution operation. Check the overflow using Status Register (`ST0`).