

FIXED POINT ARITHMETIC

Version 2.2: 2003/02/24

Hyeokho Choi

This work is produced by The Connexions Project and licensed under the Creative Commons Attribution License *

Abstract

Fixed Point Arithmetic

1 Fixed-point arithmetic

This handout explains how numbers are represented in the fixed point TI C6211 DSP processor. Because hardware can only store and process *bits*, all the numbers must be represented as a collection of bits. Each bit represents either "0" or "1", hence the number system naturally used in microprocessors is the binary system. This handout explains how numbers are represented and processed in DSP processors for implementing DSP algorithms.

1.1 How numbers are represented

A collection of N binary digits (bits) has 2^N possible states. This can be seen from elementary counting theory, which tells us that there are two possibilities for the first bit, two possibilities for the next bit, and so on until the last bit, resulting in

$$2 \times 2 \times 2 \cdots = 2^N$$

possibilities or states. In the most general sense, we can allow these states to represent anything conceivable. The point is that there is no meaning inherent in a binary word, although most people are tempted to think of them as positive integers. However, the meaning of an N -bit binary word depends entirely on its *interpretation*.

1.1.1 Unsigned integer representation

The **natural binary** representation interprets each binary word as a positive integer. For example, we interpret an 8-bit binary word

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

as an integer

$$x = b_72^7 + b_62^6 + \cdots + b_12 + b_0 = \sum_{i=0}^7 (2^i b_i)$$

* <http://creativecommons.org/licenses/by/1.0>

This way, an N -bit binary word corresponds to an integer between 0 and $2^N - 1$. Conversely, all the integers in this range can be represented by an N -bit binary word. We call this interpretation of binary words **unsigned integer** representation, because each word corresponds to a positive (or unsigned) integer.

We can add and multiply two binary words in a straightforward fashion. Because all the numbers are positive, the results of addition or multiplication are also positive.

However, the result of adding two N -bit words in general results in an $N + 1$ bits. When the result cannot be represented as an N -bit word, we say that an **overflow** has occurred. In general, the result of multiplying two N -bit words is a $2N$ bit word. Note that as we multiply numbers together, the number of necessary bits increases indefinitely. This is undesirable in DSP algorithms implemented on hardware. So, later (Section 1.1.3) we will introduce the fractional interpretation of binary words, to overcome this problem.

Another problem of the unsigned integer representation is that it can only represent positive integers. To represent negative values, naturally we need a different interpretation of binary words, and we introduce the **two's complement** representation and corresponding operations to implement arithmetic on the numbers represented in the two's complement format.

1.1.2 Two's complement integer representation

Using the natural binary representation, an N -bit word can represent integers from 0 to $2^N - 1$. However, to represent negative numbers as well as positive integers, we can use the **two's complement** representation. In 2's complement representation, an N -bit word represents integers from $(-2)^{N-1}$ to $2^{N-1} - 1$.

For example, we interpret an 8-bit binary word

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

as an integer

$$x = -(b_72^7) + b_62^6 + \cdots + b_12 + b_0 = -(b_72^7) + \sum_{i=0}^6 (2^i b_i)$$

in the 2's complement representation, and x ranges from -128 ($-(2^7)$) to 127 ($2^7 - 1$). Several examples:

| binary | decimal |
|----------|---------|
| 00000000 | 0 |
| 00000001 | 1 |
| 01000000 | 64 |
| 01111111 | 127 |
| 10000000 | -128 |
| 10000001 | -127 |
| 11000000 | -64 |
| 11111111 | -1 |

When x is a positive (negative) number in 2's complement format, $-x$ can be found by inverting each bit and adding 1. For example, 01000000_2 is 64 in decimal and -64 is found by first inverting the bits to obtain 10111111_2 and adding 1, thus -64 is 11000000_2 as shown in the above table. Because the MSB indicates the sign of the number represented by the

binary word, we call this bit the **sign bit**. If the sign bit is 0, the word represents positive number, while negative numbers have 1 as the sign bit.

In 2's complement representation, subtraction of two integers can be accomplished by usual binary summation by computing $x - y$ as $x + (-y)$. We investigate the operations on the 2's complement numbers later (Section 1.2). However, when you add two 2's complement numbers, you must keep in mind that the 1 in MSB is actually -1.

Exercise 1:

(2's complement): What are the decimal numbers corresponding to the 2's complement 8-bit binary numbers; 01001101_2 , 11100100_2 , 01111001_2 , and 10001011_2 ?

Solution:

Intentionally left blank.

Sometimes, you need to convert an 8-bit 2's complement number to a 16-bit number. What is the 16-bit 2's complement number representing the same value as the 8-bit numbers 01001011_2 and 10010111_2 ? The answer is 0000000001001000_2 and 1111111100101111_2 . For nonnegative numbers (sign bit = 0), you simply add enough 0's to extend the number of bits. For negative numbers, you add enough 1's. This operation is called **sign extension**. The same rule holds for extending a 16-bit 2's complement number to a 32-bit number.

For the arithmetic assembly instructions, C62x CPU has different versions depending on how it handles the signs. For example, the load instructions LDH and LDB load halfword and byte value to a 32-bit register with sign extension. That is, the loaded values are converted to 32-bit 2's complement number and loaded into a register. The instructions LDHU and LDBU do not perform sign extension. They simply fill zeros for the upper 16- and 24-bits, respectively.

For the shift right instructions SHR and SHRU, the same rule applies. The ADDU instruction simply treats the operands as unsigned values.

1.1.3 Fractional representation

Although using 2's complement integers we can implement both addition and subtraction by usual binary addition (with special care for the sign bit), the integers are not convenient to handle to implement DSP algorithms. For example, If we multiply two 8-bit words together, we need 16 bits to store the result. The number of required word length increases without bound as we multiply numbers together more. Although not impossible, it is complicated to handle this increase in word-length using integer arithmetic. The problem can be easily handled by using numbers between -1 and 1 , instead of integers, because the product of two numbers in $[-1, 1]$ are always in the same range.

In the 2's complement fractional representation, an N bit binary word can represent 2^N equally space numbers from $\frac{(-2)^{N-1}}{2^{N-1}} = -1$ to $\frac{2^{-(N-1)}}{2^{N-1}} = 1 - 2^{N-1}$.

For example, we interpret an 8-bit binary word

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

as a fractional number

$$x = \frac{-(b_72^7) + b_62^6 + \cdots + b_12 + b_0}{2^7} = -(b_7) + \sum_{i=0}^6 (2^{i-7}b_i) \in [-1, 1 - 2^{-7}]$$

This representation is also referred as **Q-format**. We can think of having an implied binary digit right after the MSB. If we have an N -bit binary word with MSB as the sign bit,

we have $N - 1$ bits to represent the fraction. We say the number has Q- $(N - 1)$ format. For example, in the example, x is a Q-7 number. In C6211, it is easiest to handle Q-15 numbers represented by each 16 bit binary word, because the multiplication of two Q-15 numbers results in a Q-30 number that can still be stored in a 32-bit wide register of C6211. The programmer needs to keep track of the implied binary point when manipulating Q-format numbers.

Exercise 2:

(Q format): What are the decimal fractional numbers corresponding to the Q-7 format binary numbers; 01001101_2 , 11100100_2 , 01111001_2 , and 10001011_2 ?

Solution:

Intentionally left blank.

1.2 Two's complement arithmetic

The convenience of 2's compliment format comes from the ability to represent negative numbers and compute subtraction using the same algorithm as a binary addition. The C62x processor has instructions to add, subtract and multiply numbers in the 2's compliment format. Because, in most digital signal processing algorithms, Q-15 format is most easy to implement on C62x processors, we only focus on the arithmetic operations on Q-15 numbers in the following.

1.2.1 Addition and subtraction

The addition of two binary numbers is computed in the same way as we compute the sum of two decimal numbers. Using the relation $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$ and $1 + 1 = 10$, we can easily compute the sum of two binary numbers. The C62x instruction ADD performs this binary addition on different operands.

However, care must be taken when adding binary numbers. Because each Q-15 number can represent numbers in the range $[-1, 1 - 2^{-15}]$, if the result of summing two Q-15 numbers is not in this range, we cannot represent the result in the Q-15 format. When this happens, we say an **overflow** has occurred. Unless carefully handled, the overflow makes the result incorrect. Therefore, it is really important to prevent overflows from occurring when implementing DSP algorithms. One way of avoiding overflow is to scale all the numbers down by a constant factor, effectively making all the numbers very small, so that any summation would give results in the $[-1, 1)$ range. This *scaling* is necessary and it is important to figure out how much scaling is necessary to avoid overflow. Because scaling results in loss of effective number of digits, increasing quantization errors, we usually need to find the minimum amount of scaling to prevent overflow.

Another way of handling the overflow (and underflow) is **saturation**. If the result is out of the range that can be properly represented in the given data size, the value is saturated, meaning that the value closest to the true result is taken in the range representable. Such instructions as SADD, SSUB perform the operations followed by saturation.

Exercise 3:

(Q format addition, subtraction): Perform the additions $01001101_2 + 11100100_2$, and $01111001_2 + 10001011_2$ when the binary numbers are Q-7 format. Also compute $01001101_2 - 11100100_2$ and $10001011_2 - 00110111_2$. In which cases, do you have overflow?

Solution:

Intentionally left blank.

1.2.2 Multiplication

Multiplication of two 2's complement numbers is a bit complicated because of the sign bit. Similar to the multiplication of two decimal fractional numbers, the result of multiplying two $Q-N$ numbers is $Q-2N$, meaning that we have $2N$ binary digits following the implied binary digit. However, depending on the numbers multiplied, the result can have either 1 or 2 binary digits before the binary point. We call the digit right before the binary point the **sign bit** and the one proceeding the sign bit (if any) the **extended sign bit**.

The following is the two examples of binary fractional multiplications:

```

0.110 0.75 Q-3
X 1.110 -0.25 Q-3
-----
0000
0110
    0110
    1010
-----
1110100 -0.1875 Q-6

```

Above, all partial products are computed and represented in $Q-6$ format for summation. For example, $0.110 * 0.010 = 0.01100$ in $Q-6$ for the second partial product. For the 4th partial product, care must be taken because in $0.110 * 1.000$, 1.000 represents -1 , so the product is $-0.110 = 1.01000$ (in $Q-6$ format) that is 2's complement of 0.11000 . As noticed in this example, it is important to represent each partial product in $Q-6$ (or in general $Q-2N$) format before adding them together. Another example makes this point clearer:

```

1.110 -0.25 Q-3
X 0.110 0.75 Q-3
-----
0000
    111110
    11110
    0000
-----
11110100 -0.1875 Q-6

```

For the second partial product, we need $1.110 * 0.010$ in $Q-6$ format. This is obtained as 1111100 in $Q-6$ (check!). A simple way to obtain it is to first perform the multiplication in normal fashion as $1110 * 0010 = 11100$ ignoring the binary points, then perform *sign extension* by putting enough 1s (if the result is negative) or 0s (if the result is nonnegative), then put the binary point to obtain a $Q-6$ number. Also notice that we need to remove the extra sign bit to obtain the final result.

In C62x, if we multiply two Q-15 numbers using one of multiply instruction (for example MPY), we obtain 32 bit result in Q-30 format with 2 sign bits. To obtain the result back in Q-15 format, (i) first we remove 15 trailing bits and (ii) remove the extended sign bit.

Exercise 4:

(Q format multiplication): Perform the multiplications 01001101*11100100, and 01111001*10001011 when the binary numbers are Q-7 format.

Solution:

Intentionally left blank.

1.3 Assembly language implementation

When A0 and A1 contain two 16-bit numbers in the Q-15 format, we can perform the multiplications using MPY followed by a right shift.

```

1    MPY    .M1    A0,A1,A2
2    NOP
3    SHR    .S1    A2,15,A2    ;lower 16 bit contains result
4                                ;in Q-15 format

```

Rather than throwing away the 15 LSBs of the multiplication result by shifting, you can round up the result by adding 0x4000 before shifting.

```

1    MPY    .M1    A0,A1,A2
2    NOP
3    ADDK   .S1    4000h,A6
4    SHR    .S1    A2,15,A2    ;lower 16 bit contains result
5                                ;in Q-15 format

```

1.4 C language implementation

Let's suppose we have two 16-bit numbers in Q-15 format, stored in variable x and y as follows:

```

short x = 0x0011;    /* 0.000518799 in decimal */
short y = 0xfe12;    /* -0.015075684 in decimal */
short z;             /* variable to store x*y */

```

The product of x and y can be computed and stored in Q-15 format as follows:

```
z = (x * y) >> 15;
```

The result of x*y is a 32-bit word with 2 sign bits. Right shifting it by 15 bits ignores the last 15 bits, and storing the shifted result in z that is a short variable (16 bit) removes the extended sign bit by taking only lower 16 bits.