Two's Complement and Fractional Arithmetic for 16-bit Processors*

Douglas L. Jones
Swaroop Appadwedula
Matthew Berry
Mark Haun
Dima Moussa
Daniel Sachs
Jason Laska

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

Abstract

Two's-complement notation is a mathematically convenient way of representing signed numbers in microprocessors. The most significant bit of a two's complement number represents its sign, and the remaining bits represent its magnitude.

Fractional arithmetic allows one to multiply numbers on an integer processor without incurring overflow. Fractional arithmetic requires sign-extension of multipliers and multiplicands, and it requires the product of two numbers to be left-shifted one bit.

1 Two's-complement notation

Two's-complement notation is an efficient way of representing signed numbers in microprocessors. It offers the advantage that addition and subtraction can be done with ordinary unsigned operations. When a number is written in two's complement notation, the most significant bit of the number represents its sign: 0 means that the number is positive, and 1 means the number is negative. A positive number written in two's-complement notation is the same as the number written in unsigned notation (although the most significant bit must be zero). A negative number can be written in two's complement notation by inverting all of the bits of its absolute value, then adding one to the result.

Example 1

Consider the following four-bit two's complement numbers (in binary form):

*Version 2.9: Jan 30, 2005 12:51 pm -0600

†http://creativecommons.org/licenses/by/1.0

$1 = 0001$	$-1 = 1110 + 1 = 1111$
$2 = 0010$	$-2 = 1101 + 1 = 1110$
$6 = 0110$	$-6 = 1001 + 1 = 1010$
$8 = 1000$	$-8 = 0111 + 1 = 1000$

Table 1

NOTE: 1000 represents -8, not 8. This is because the topmost bit (the sign bit) is 1, indicating that the number is negative.

The maximum number that can be represented with a k-bit two's-complement notation is $2^{k-1} - 1$, and the minimum number that can be represented is -2^{k-1}. The maximum integer that can be represented in a 16-bit memory register is 32767, and the minimum integer is -32768.

2 Fractional arithmetic

The DSP microprocessor is a 16-bit integer processor with some extra support for **fractional arithmetic.** Fractional arithmetic turns out to be very useful for DSP programming, since it frees us from worries about overflow on multiplies. (Two 16-bit numbers, multiplied together, can require 32 bits for the result. Two 16-bit fixed-point fractional numbers also require 32 bits for the result, but the 32-bit result can be rounded into 16 bits while only introducing an error of approximately 2^{-16}.) For this reason, we will be using fixed-point fractional representation to describe filter taps and inputs throughout this course.

Unfortunately, the assembler and debugger we are using do not recognize this fractional fixed-point representation. For this reason, when you are using the assembler or debugger, you will see decimal values (ranging from -32768 to 32767) on screen instead of the fraction being represented. The conversion is simple; the fractional number being represented is simply the decimal value shown divided by 32768. This allows us to represent numbers between -1 and $1 - 2^{-15}$.

NOTE: 1 cannot be represented exactly.

When we multiply using this representation, an extra shift left is required. Consider the two examples below:

Example 2

fractional	$0.5 \times 0.5 = 0.25$
decimal	$16384 \times 16384 = 4096 \times 2^{16} : 4096/327681/8$
hex	$4000 \times 4000 = 1000 \times 2^{16}$

Table 2

fractional	$0.125 \times 0.75 = 0.093750$
decimal	$4096 \times 24576 = 1536 \times 2^{16} : 1536/327680.046875$
hex	$1000 \times 6000 = 0600 \times 2^{16}$

Table 3

You may wish touse the MATLAB commands `hex2dec` and `dec2hex`. When we do the multiplication, we are primarily interested in the top 16 bits of the result, since these are the data that are actually used when we store the result back into memory and send it out to the digital-to-analog converter. (The entire result is actually stored in the accumulator, so rounding errors do not accumulate when we do a sequence of multiply-accumulate operations in the accumulators.) As the example above shows, the top 16 bits of the result of multiplying the fixed point fractional numbers together is half the expected fractional result. The extra left shift multiplies the result by two, giving us the correct final product.

The left-shift requirement can alternatively be explained by way of decimal place alignment. Remember that when we multiply decimal numbers, we first multiply them ignoring the decimal points, then put the decimal point back in the last step. The decimal point is placed so that the total number of digits right of the decimal point in the multiplier and multiplicand is equal to the number of digits right of the decimal point in their product. The same applies here; the "decimal point" is to the right of the leftmost (sign) bit, and there are 15 bits (digits) to the right of this point. So there are a total of 30 bits to the right of the decimal in the source. But if we do not shift the result, there are 31 bits to the right of the decimal in the 32-bit result. So we shift the number to the left by one bit, which effectively reduces the number of bits right of the decimal to 30.

Before the numbers are multiplied by the ALU, each term is **sign-extended** generating a 17-bit number from the 16-bit input. Because the examples presented above are all positive, the effect of this sign extension is simply adding an extra "0" bit at the top of the register (*i.e.*, positive numbers are not affected by the sign extension). As the following example illustrates, not including this sign-bit for negative numbers produces erroneous results.

fractional	$-0.5 \times 0.5 = -0.25$
decimal	$49152 \times 16384 = 12288 \times 2^{16} : 12288/326780.375$
hex	$C000 \times 4000 = 30000000 = 3000 \times 2^{16}$

Table 4

Note that even after the result is left-shifted by one bit following the multiply, the top bit of the result is still "0", implying that the result is incorrectly interpreted as a positive number.

To correct this problem, the ALU sign-extends negative multipliers and multiplicands by placing a "1" instead of a "0" in the added bit. This is called **sign extension** because the sign bit is "extended" to the left another place, adding an extra bit to the left of the number without changing the number's value.

fractional	$-0.5 \times 0.5 = -0.25$
hex	$1C000 \times 4000 = 70000000 = 7000 \times 2^{16}$

Table 5

Although the top bit of this result is still "0", after the final 1-bit left-shift the result is `E000 000h` which is a negative number (the top bit is "1"). To check the final answer, we can negate the product using the two's complement method described above. After flipping all of the bits we have `1FFF FFFFh`, and adding one yields `2000 0000h`, which equals 0.25 when interpreted as an 32 bit fractional number.