Numeric Formats C

C.1 OVERVIEW

ADSP-2100 family processors support 16-bit fixed-point data in hardware. Special features in the computation units allow you to support other formats in software. This appendix describes various aspects of the 16-bit data format. It also describes how to implement a block floating-point format in software.

C.2 UNSIGNED OR SIGNED: TWOS-COMPLEMENT FORMAT

Unsigned binary numbers may be thought of as positive, having nearly twice the magnitude of a signed number of the same length. The least significant words of multiple precision numbers are treated as unsigned numbers.

Signed numbers supported by the ADSP-2100 family are in twos-complement format. Signed-magnitude, ones-complement, BCD or excess-n formats are not supported.

C.3 INTEGER OR FRACTIONAL

The ADSP-2100 family supports both fractional and integer data formats, with the exception that the ADSP-2100 processor does not perform integer multiplication. In an integer, the radix point is assumed to lie to the right of the LSB, so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure C.1, which can be found on the following page. Note that in twos-complement format, the sign bit has a negative weight.

C Numeric Formats

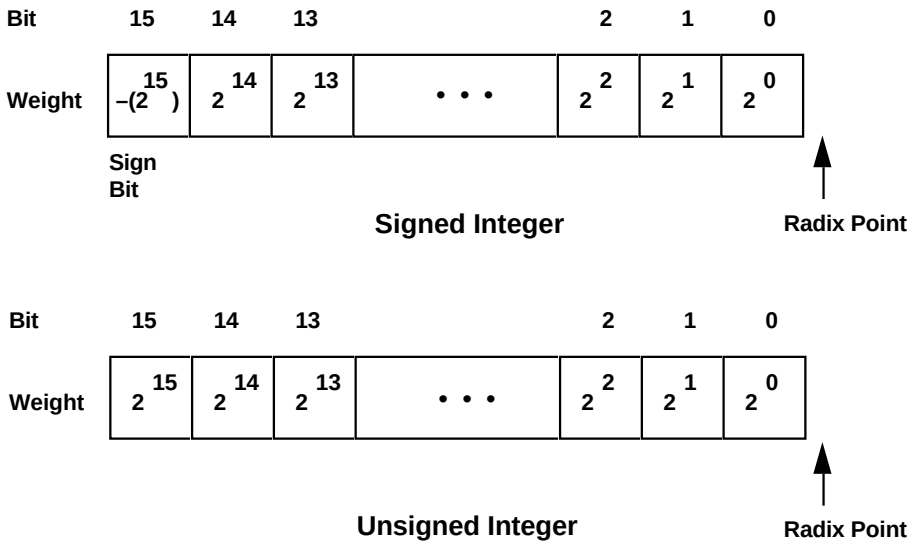


Figure C.1 Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure C.2, the assumed radix point lies to the left of the 3 LSBs, and the bits have the weights indicated.

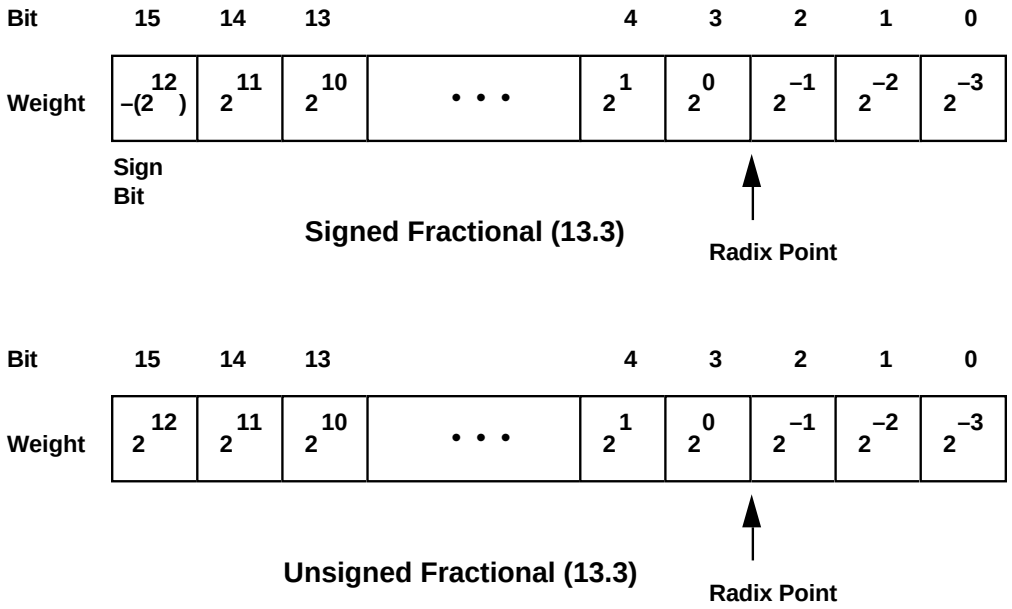


Figure C.2 Example Of Fractional Format

The notation used to describe a format consists two numbers separated by a period (.); the first number is the number of bits to the left of radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure C.2 is 13.3.

Table C.1 shows the ranges of numbers representable in the fractional formats that are possible with 16 bits.

Format	Number of Integer Bits	Number of Fractional Bits	Largest Positive Value (0x7FFF) In Decimal	Largest Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	–1.0	0.000030517578125
2.14	2	14	1.999938964843750	–2.0	0.000061035156250
3.13	3	13	3.999877929687500	–4.0	0.000122070312500
4.12	4	12	7.999755859375000	–8.0	0.000244140625000
5.11	5	11	15.999511718750000	–16.0	0.000488281250000
6.10	6	10	31.999023437500000	–32.0	0.000976562500000
7.9	7	9	63.998046875000000	–64.0	0.001953125000000
8.8	8	8	127.996093750000000	–128.0	0.003906250000000
9.7	9	7	255.992187500000000	–256.0	0.007812500000000
10.6	10	6	511.984375000000000	–512.0	0.015625000000000
11.5	11	5	1023.968750000000000	–1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	–2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	–4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	–8192.0	0.250000000000000
15.1	15	1	16383.500000000000000	–16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	–32768.0	1.000000000000000

Table C.1 Fractional Formats And Their Ranges

C.4 BINARY MULTIPLICATION

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location) and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The ADSP-2100 family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is

shown in Figure C.3. The product of two 16-bit numbers is a 32-bit number. If the inputs' formats are M.N and P.Q, the product has the format (M+P).(N+Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

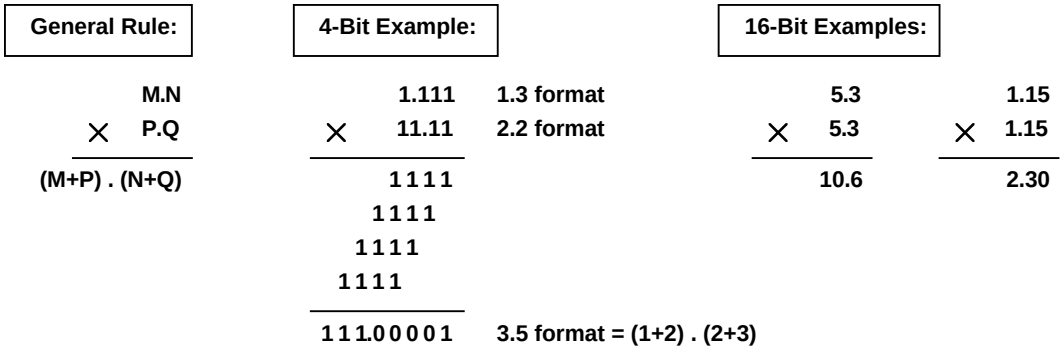


Figure C.3 Format Of Multiplier Result

C.4.1 Fractional Mode And Integer Mode

A product of 2 twos-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The ADSP-2100 family provides a mode (called the fractional mode) in which the multiplier result is always shifted left one bit before being written to the result register. (On the ADSP-2100 processor, this mode is always active; on other processors, the left shift can be omitted.) This left shift eliminates the extra sign bit when both operands are signed, yielding a correctly formatted result.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a fractional data format, it is most convenient to use the 1.15 format.

In the integer mode, the left shift does not occur. This is the mode to use if both operands are integers (in the 16.0 format). The 32-bit multiplier result is in 32.0 format, also an integer. On the ADSP-2100 only, the integer mode

is not available; the 32.0 result gets shifted to 31.1 format. Because the MSB is still available in the 40-bit accumulator, a right shift can correct the result.

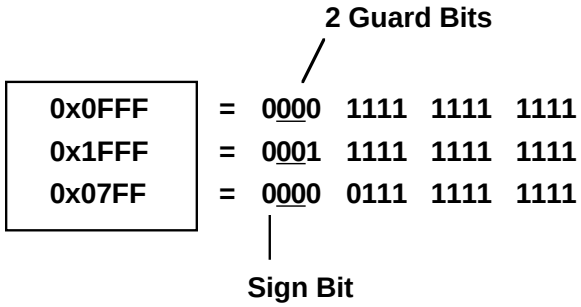
In all processors other than the ADSP-2100, fractional and integer modes are controlled by a bit in the MSTAT register. At reset, these processors default to the fractional mode, for compatibility with the ADSP-2100.

C.5 BLOCK FLOATING-POINT FORMAT

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. Some additional programming is required to maintain a block floating-point format, however.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. To convert a block of fixed-point values to block floating-point format, you would shift each value left by the same amount and store the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs, increasing the precision available in each value. You can also use block floating-point format to eliminate the possibility of a data value overflowing. Figure C.4 shows an example. The three data samples each have at least 2 non-significant, redundant sign bits. Each data value



To detect bit growth into 2 guard bits, set SB=–2

Figure C.4 Data With Guard Bits

can grow by these two bits (two orders of magnitude) before overflowing; thus, these bits are called *guard* bits. If it is known that a process will not cause any value to grow by more than these two bits, then the process can be run without loss of data. Afterward, however, the block must be adjusted to replace the guard bits before the next process.

Figure C.5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows. Initially, the value of SB is –2, corresponding to the 2 guard bits. During processing, each resulting data value is inspected by the EXPADJ instruction, which counts the number of redundant sign bits and adjusts SB is if the number of redundant sign bits is less than 2. In this example, SB=–1 after processing, indicating that the block of data must be shifted right one bit to maintain the 2 guard bits. If SB were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

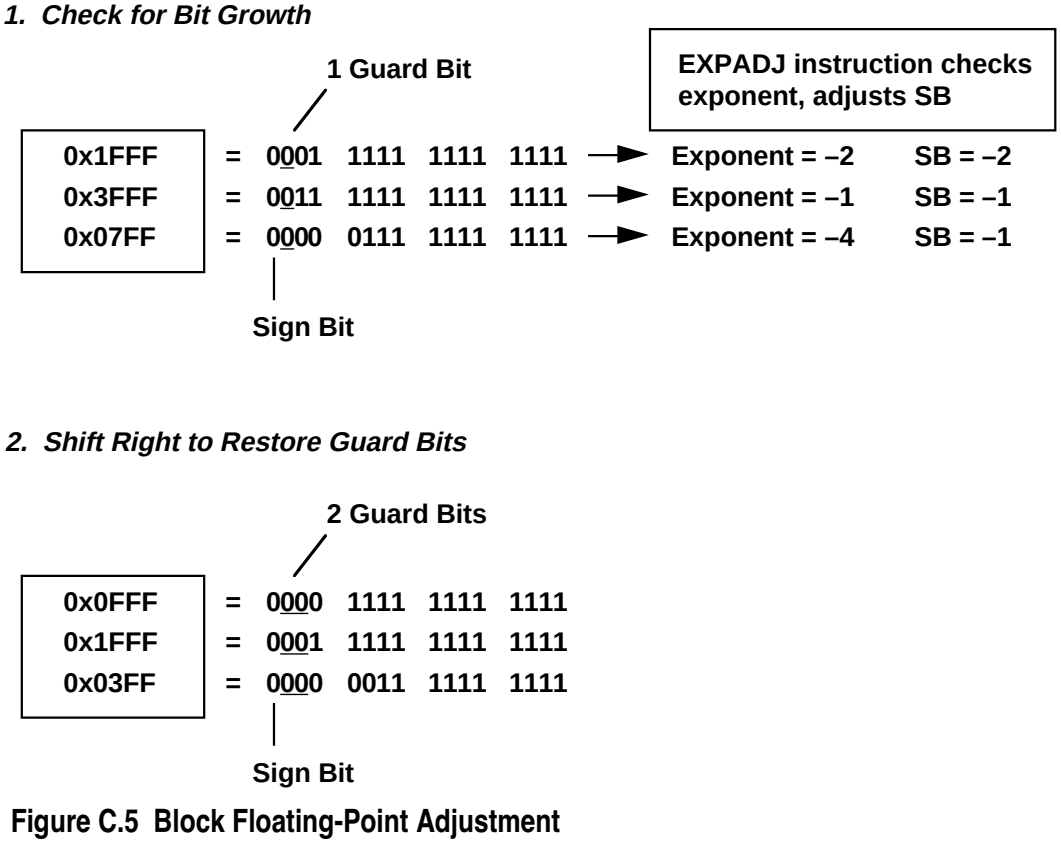


Figure C.5 Block Floating-Point Adjustment