



Fast Floating-Point Arithmetic Emulation on the Blackfin® Processor Platform

Contributed by DSP Apps

May 26, 2003

Introduction

Processors optimized for digital signal processing are divided into two broad categories: fixed-point and floating-point. In general, the cutting-edge fixed-point families tend to be fast, low power, and low cost, while floating-point processors offer high precision and wide dynamic range in hardware.

While the Blackfin® Processor architecture was designed for native fixed-point computations, it can achieve clock speeds that are high enough to emulate floating-point operations in software. This gives the system designer a choice between hardware efficiency of floating-point processors and the low cost and power of Blackfin Processor fixed-point devices. Depending on whether full standard conformance or speed is the goal, floating-point emulation on a fixed-point processor might use the IEEE-754 standard or a fast floating-point (non-IEEE-compliant) format.

On the Blackfin Processor platform, IEEE-754 floating-point functions are available as library calls from both C/C++ and assembly language. These libraries emulate floating-point processing using fixed-point logic.

To reduce computational complexity, it is sometimes advantageous to use a more relaxed and faster floating-point format. A significant cycle savings can often be achieved in this way.

This document shows how to emulate fast floating-point arithmetic on the Blackfin Processor platform. A *two-word* format is employed for representing short and long fast floating-point data types. C-callable assembly source code is included for the following operations: addition, subtraction, multiplication and conversion between fixed-point, IEEE-754 floating-point, and the fast floating-point formats.

Overview

In fixed-point number representation, the radix point is always at the same location. While this convention simplifies numeric operations and conserves memory, it places a limit the magnitude and precision. In situations that require a large range of numbers or high resolution, a changeable radix point is desirable.

Very large and very small numbers can be represented in floating-point format. Floating-point format is basically scientific notation; a floating-point number consists of a mantissa (or fraction) and an exponent. In the IEEE-754 standard, a floating-point number is stored in a 32-bit word, with a 23-bit mantissa, an 8-bit exponent, and a 1-bit sign. In the fast floating-point *two-word* format described in this document, the exponent is a 16-bit signed integer, while the mantissa is either a 16- or a 32-bit signed fraction (depending on whether the

short of the long fast floating-point data type is used).

Normalization is an important feature of floating-point representation. A floating-point number is *normalized* if it contains no redundant sign bits in the mantissa; that is, all bits are significant. Normalization provides the highest precision for the number of bits available. It also simplifies the comparison of magnitudes, because the number with the greater exponent has the greater magnitude; only if the exponents are equal is it necessary to compare the mantissas. All routines presented in this document assume normalized input and produce normalized results.



There are some differences in arithmetic flags between the ADSP-BF535 Blackfin Processor and the ADSP-BF531/2/3 Blackfin Processor platforms. All of the assembly code in this document was written for the ADSP-BF531/2/3 Blackfin Processor devices.

Short and Long Fast Floating-Point Data Types

The code routines in this document use the *two-word* format for two different data types. The short data type (*fastfloat16*) provides one 16-bit word for the exponent and one 16-bit word for the fraction. The long data type (*fastfloat32*) provides one 16-bit word for the exponent and one 32-bit word for the fraction. The *fastfloat32* data type is more computationally intensive, but provides greater precision than the *fastfloat16* data type. Signed twos-complement notation is assumed for both the fraction and the exponent.

Listing 1 Format of the fastfloat16 data type

```
typedef struct
{
    short exp;
    fract16 frac;
} fastfloat16;
```

Listing 2 Format of the fastfloat32 data type

```
typedef struct
{
    short exp;
    fract32 frac;
} fastfloat32;
```

Converting Between Fixed-Point and Fast Floating-Point Formats

There are two Blackfin Processor instructions used in fixed-point to fast floating-point conversion. The first instruction, *signbits*, returns the number of sign bits in a number (i.e. the exponent). The second, *ashift*, is used to normalize the mantissa.

Assembly code for both the short version (*fastfloat16*) and the long version (*fastfloat32*) is shown below.

Listing 3 Fixed-point to short fast floating-point (fastfloat16) conversion

```
/******
fastfloat16 fract16_to_ff16(fract16);
-----
Input parameters (compiler convention):
R0.L = fract16
Output parameters (compiler convention):
R0.L = ff16.exp
R0.H = ff16.frac
-----
*****/
_fract16_to_ff16:
.global _fract16_to_ff16;
    r1.l = signbits r0.l; // get the number of
sign bits
    r2 = -r1 (v);
    r2.h = ashift r0.l by r1.l; // normalize
the mantissa
    r0 = r2;
    rts;
_fract16_to_ff16.end;
```

Listing 4 Fixed-point to long fast floating-point (fastfloat32) conversion

```

/*****
fastfloat32 fract32_to_ff32(fract32);
-----

Input parameters (compiler convention):
R0 = fract32
Output parameters (compiler convention):
R0.L = ff32.exp
R1 = ff32.frac
-----

*****/
_fract32_to_ff32:
.global _fract32_to_ff32;
    r1.l = signbits r0; // get the number of
                        // sign bits

    r2 = -r1 (v);
    r1 = ashift r0 by r1.l; // normalize the
                        // mantissa

    r0 = r2;
    rts;
_fract32_to_ff32.end:

```

Converting *two-word* fast floating-point numbers to fixed-point format is made simple by using the `ashift` instruction.

Assembly code for both the short version (*fastfloat16*) and the long version (*fastfloat32*) is shown below.

Listing 5 Short fast floating-point (fastfloat16) to fixed-point conversion

```

/*****
fract16 ff16_to_fract16(fastfloat16);
-----

Input parameters (compiler convention):
R0.L = ff16.exp
R0.H = ff16.frac
Output parameters (compiler convention):
R0.L = fract16

```

```

-----

*****/
_ff16_to_fract16:

.global _ff16_to_fract16;

    r0.h = ashift r0.h by r0.l; // shift the
                        // binary point

    r0 >>= 16;

    rts;

_ff16_to_fract16.end:

```

Listing 6 Long fast floating-point (fastfloat32) to fixed-point conversion

```

/*****
fract32 ff32_to_fract32(fastfloat32);
-----

Input parameters (compiler convention):
R0.L = ff32.exp
R1 = ff32.frac
Output parameters (compiler convention):
R0 = fract32
-----

*****/
_ff32_to_fract32:

.global _ff32_to_fract32;

    r0 = ashift r1 by r0.l; // shift the
                        // binary point

    rts;

_ff32_to_fract32.end:

```


Converting Between Fast Floating-Point and IEEE Floating-Point Formats

The basic approach to converting from the IEEE floating-point format to the fast floating-point

format begins with extracting the mantissa, exponent, and sign bit ranges from the IEEE floating-point word. The exponent needs to be unbiased before setting the fast floating-point exponent word. The mantissa is used with the sign bit to create a twos-complement signed fraction, which completes the fast floating-point *two-word* format.

Doing these steps in reverse converts a fast floating-point number into IEEE floating-point format.

It is outside the scope of this document to provide more detail about the IEEE-754 floating-point format. The attached compressed package contains sample C code to perform the conversions.


 The compressed package that accompanies this document contain sample routines that convert between the fast floating-point and the IEEE floating-point numbers. Note that they do not treat any special IEEE defined values like NaN, $+\infty$, or $-\infty$.

Floating-Point Addition

The algorithm for adding two numbers in two-word fast floating-point format is as follows:

1. Determine which number has the larger exponent. Let's call this number X (= Ex, Fx) and the other number Y (= Ey, Fy).
2. Set the exponent of the result to Ex.
3. Shift Fy right by the difference between Ex and Ey, to align the radix points of Fx and Fy.
4. Add Fx and Fy to produce the fraction of the result.
5. Treat overflows by scaling back the input parameters, if necessary.
6. Normalize the result.

Assembly code for both the short version (*fastfloat16*) and the long version (*fastfloat32*) is shown below.

 For the *fastfloat16* arithmetic operations (add, subtract, divide), the following parameter passing conventions are used. These are the default conventions used by the Blackfin Processor compiler.

Calling Parameters

```
r0.h = Fraction of x (=Fx)
r0.l = Exponent of x (=Ex)
```

```
r1.h = Fraction of y (=Fy)
r1.l = Exponent of y (=Ey)
```

Return Values

```
r0.h = Fraction of z (=Fz)
r0.l = Exponent of z (=Ez)
```

Listing 7 Short fast floating-point (fastfloat16) addition

```

/*****
fastfloat16
add_ff16(fastfloat16,fastfloat16);
*****/
_add_ff16:
.global _add_ff16;

    r2.l = r0.l - r1.l (ns); // is Ex > Ey?
    cc = an; // negative result?
    r2.l = r2.l << 11 (s); // guarantee shift
                        // range [-16,15]

    r2.l = r2.l >>> 11;
    if !cc jump _add_ff16_1; // no, shift y
    r0.h = ashift r0.h by r2.l; // yes,
                        // shift x

    jump _add_ff16_2;

_add_ff16_1:
    r2 = -r2 (v);
    r1.h = ashift r1.h by r2.l; // shift y
    a0 = 0;

```

```

a0.l = r0.l; // you can't do r1.h = r2.h
r1.l = a0 (iu); // so use a0.x as an
                // intermediate storage
                // place

_add_ff16_2:

    r2.l = r0.h + r1.h (ns); // add fractional
                            // parts

    cc = v; // was there an overflow?
    if cc jump _add_ff16_3;

// normalize

    r0.l = signbits r2.l; // get the number of
                        // sign bits

    r0.h = ashift r2.l by r0.l; // normalize
                            // the mantissa

    r0.l = r1.l - r0.l (ns); // adjust the
                            // exponent

rts;

// overflow condition for mantissa addition
_add_ff16_3:

    r0.h = r0.h >>> 1; // shift the mantissas
                    // down

    r1.h = r1.h >>> 1;

    r0.h = r0.h + r1.h (ns); // add fractional
                            // parts

    r2.l = 1;

    r0.l = r1.l + r2.l (ns); // adjust the
                            //exponent

    rts;

_add_ff16.end:

```



For the *fastfloat32* arithmetic operations (add, subtract, divide), the following parameter passing conventions are used. These are the default conventions used by the Blackfin Processor compiler.

Calling Parameters

```

r1 = Fraction of x (=Fx)
r0.l = Exponent of x (=Ex)

r3 = [FP+20] = Fraction of y (=Fy)
r2.l = Exponent of y (=Ey)

```

Return Values

```

r1 = Fraction of z (=Fz)
r0.l = Exponent of z (=Ez)

```

Listing 8 Long fast floating-point (fastfloat32) addition

```

/*****
fastfloat32 add_ff32(fastfloat32,
fastfloat32);
*****/

#define FF32_PROLOGUE() link 0; r3 =
[fp+20]; [--sp]=r4; [--sp]=r5

#define FF32_EPILOGUE() r5=[sp++];
r4=[sp++]; unlink

.global _add_ff32;
_add_ff32:

    FF32_PROLOGUE();

    r4.l = r0.l - r2.l (ns); // is Ex > Ey?
    cc = an; // negative result?
    r4.l = r4.l << 10 (s); // guarantee shift
                        // range [-32,31]

    r4.l = r4.l >>> 10;
    if !cc jump _add_ff32_1; // no, shift Fy
    r1 = ashift r1 by r4.l; // yes, shift Fx
    jump _add_ff32_2;

_add_ff32_1:

    r4 = -r4 (v);

    r3 = ashift r3 by r4.l; // shift Fy
    r2 = r0;

_add_ff32_2:

    r4 = r1 + r3 (ns); // add fractional parts
    cc = v; // was there an overflow?
    if cc jump _add_ff32_3;

// normalize

    r0.l = signbits r4; // get the number of
                        // sign bits

    r1 = ashift r4 by r0.l; // normalize the
                        // mantissa

```

```

r0.l = r2.l - r0.l (ns); // adjust the
                        // exponent

FF32_EPILOGUE();
rts;

// overflow condition for mantissa addition
_add_ff32_3:
    r1 = r1 >>> 1;
    r3 = r3 >>> 1;
    r1 = r1 + r3 (ns); // add fractional parts

    r4.l = 1;
    r0.l = r2.l + r4.l (ns); // adjust the
                        // exponent

    FF32_EPILOGUE();
    rts;
_add_ff32.end:

```

Floating-Point Subtraction

The algorithm for subtracting one number from another in two-word fast floating-point format is as follows:

1. Determine which number has the larger exponent. Let's call this number X (= Ex, Fx) and the other number Y (= Ey, Fy).
2. Set the exponent of the result to Ex.
3. Shift Fy right by the difference between Ex and Ey, to align the radix points of Fx and Fy.
4. Subtract the fraction of the subtrahend from the fraction of the minuend to produce the fraction of the result.
5. Treat overflows by scaling back the input parameters, if necessary.
6. Normalize the result.

Assembly code for both the short version (*fastfloat16*) and the long version (*fastfloat32*) is shown below.

Listing 9 Short fast floating-point (fastfloat16) subtraction

```

/*****
fastfloat16 sub_ff16(fastfloat16,
fastfloat16);
*****/

.global _sub_ff16;
_sub_ff16:
    r2.l = r0.l - r1.l (ns); // is Ex > Ey?
    cc = an; // negative result?
    r2.l = r2.l << 11 (s); // guarantee shift
                        // range [-16,15]

    r2.l = r2.l >>> 11;
    if !cc jump _sub_ff16_1; // no, shift y
    r0.h = ashift r0.h by r2.l; // yes, shift
                        // x

    jump _sub_ff16_2;

_sub_ff16_1:
    r2 = -r2 (v);
    r1.h = ashift r1.h by r2.l; // shift y
    a0 = 0;
    a0.l = r0.l; // you can't do r1.h = r2.h
    r1.l = a0 (iu); // so use a0.x as an
                        // intermediate storage place

_sub_ff16_2:
    r2.l = r0.h - r1.h (ns); // subtract
                        // fractions

    cc = v; // was there an overflow?
    if cc jump _sub_ff16_3;

// normalize
    r0.l = signbits r2.l; // get the number of
                        // sign bits

    r0.h = ashift r2.l by r0.l; // normalize
                        // mantissa

    r0.l = r1.l - r0.l (ns); // adjust
                        // exponent

    rts;

// overflow condition for mantissa
subtraction
_sub_ff16_3:
    r0.h = r0.h >>> 1; // shift the mantissas
                        // down

```

```

r1.h = r1.h >>> 1;
r0.h = r0.h - r1.h (ns); // subtract
                        // fractions

r2.l = 1;
r0.l = r1.l + r2.l (ns); // adjust the
                        //exponent

rts;
_sub_ff16.end:

```

Listing 10 Long fast floating-point (fastfloat32) subtraction

```

/*****
fastfloat32 sub_ff32(fastfloat32,
fastfloat32);
*****/

#define FF32_PROLOGUE() link 0; r3 =
[fp+20]; [--sp]=r4; [--sp]=r5

#define FF32_EPILOGUE() r5=[sp++];
r4=[sp++]; unlink

.global _sub_ff32;
_sub_ff32:
    FF32_PROLOGUE();
    r4.l = r0.l - r2.l (ns); // is Ex > Ey?
    cc = an; // negative result?
    r4.l = r4.l << 10 (s); // guarantee shift
                        // range [-32,31]

    r4.l = r4.l >>> 10;
    if !cc jump _sub_ff32_1; // no, shift Fy
    r1 = ashift r1 by r4.l; // yes, shift Fx
    jump _sub_ff32_2;

_sub_ff32_1:
    r4 = -r4 (v);
    r3 = ashift r3 by r4.l; // shift Fy
    r2 = r0;

_sub_ff32_2:
    r4 = r1 - r3 (ns); // subtract fractions
    cc = v; // was there an overflow?
    if cc jump _sub_ff32_3;

```

```

// normalize
r0.l = signbits r4; // get the number of
                    // sign bits
r1 = ashift r4 by r0.l; // normalize the
                    // mantissa
r0.l = r2.l - r0.l (ns); // adjust the
                    //exponent

FF32_EPILOGUE();
rts;

// overflow condition for mantissa
// subtraction
_sub_ff32_3:
    r1 = r1 >>> 1;
    r3 = r3 >>> 1;
    r1 = r1 - r3 (ns); // subtract fractions

    r4.l = 1;
    r0.l = r2.l + r4.l (ns); // adjust the
                    // exponent

    FF32_EPILOGUE();
    rts;
_sub_ff32.end:

```

Floating-Point Multiplication

Multiplication of two numbers in two-word fast floating-point format is simpler than either addition or subtraction, because there is no need to align the radix points. The algorithm to multiply two numbers x and y (E_x , F_x and E_y , F_y) is as follows:

1. Add E_x and E_y to produce the exponent of the result.
2. Multiply F_x by F_y to produce the fraction of the result.
3. Normalize the result.

Assembly code for both the short version (*fastfloat16*) and the long version (*fastfloat32*) is shown below.

Listing 11 Short fast floating-point (fastfloat16) multiplication

```

/*****
fastfloat16 void mult_ff16(fastfloat16,
fastfloat16);
*****/

.global _mult_ff16;
_mult_ff16:
    r3.l = r0.l + r1.l (ns);
    a0 = r0.h * r1.h;

    r2.l = signbits a0; // get the number of
                        // sign bits
    a0 = ashift a0 by r2.l; // normalize the
                        // mantissa
    r0 = a0;
    r0.l = r3.l - r2.l (ns); // adjust the
                        // exponent

    rts;
_mult_ff16.end:

```

Listing 12 Long fast floating-point (fastfloat16) multiplication

```

/*****
fastfloat32 void mult_ff32(fastfloat32,
fastfloat32);
*****/

#define FF32_PROLOGUE() link 0; r3 =
[fp+20]; [--sp]=r4; [--sp]=r5

#define FF32_EPILOGUE() r5=[sp++];
r4=[sp++]; unlink

.global _mult_ff32;
_mult_ff32:
    FF32_PROLOGUE();
    r0.l = r0.l + r2.l (ns); // add the
                        // exponents

    // perform 32-bit fractional multiplication
    a1 = a0 = 0;
    r5 = 0;
    r5.l = (a0 = r1.l * r3.l) (fu);
    a1 = r5;

```

```

    r2 = (a0 = r1.h * r3.h),
    a1 += r1.h * r3.l (m);

    r5 = (a1 += r3.h * r1.l) (m);
    r5 = r5 >>> 15;
    r1 = r2 + r5;

    // normalize
    r4.l = signbits r1; // get the number of
                        // sign bits
    r1 = ashift r1 by r4.l; // normalize the
                        // mantissa
    r0.l = r0.l - r4.l (ns); // adjust the
                        // exponent

    FF32_EPILOGUE();
    rts;
_mult_ff32.end:

```

Summary

The *two-word* fast floating-point technique described in this document can greatly improve floating-point computational efficiency on the fixed-point Blackfin Processor platform. The specific operations described above can be used as standalone routines, or as starting points for more advanced calculations specific to a particular application. The compressed source code package that accompanies this document provides a starting for using the fast floating-point method in new projects.

References

Analog Devices (DSP Division). *Digital Signal Processing Applications: Using the ADSP-2100 Family (Volume 1)*. NJ: Prentice Hall, 1992.

Knuth, D. E. *The Art of Computer Programming: Volume 2 / Seminumerical Algorithms*. Second Edition. Reading, MA: Addison-Wesley Publishing Company, 1969.

IEEE. *IEEE Standard for Binary Floating-Point Arithmetic: ANSI/IEEE Std 754-1985*. New York: The Institute of Electrical and Electronics Engineers, 1985.

Document History

Version	Description
May 26, 2003 by Tom L.	Code updated to check for overflow conditions; New test cases added
May 12, 2003 by Tom L.	Updated according to new naming conventions
February 19, 2003 by Tom L.	Initial release