Blackfin®处理器上快速浮点运算模拟

作者：Central Apps

Rev 4 – August 23, 2007

简介

处理器为数字信号处理所做的优化主要分为两大类：定点和浮点。一般而言，先进的定点家族趋向于速度快，功耗小，成本低而浮点处理器能够在硬件方面提供更高的精度和更广的动态范围。

尽管 Blackfin®处理器架构是为定点计算设计的，但由于它能产生足够高的时钟频率，从而能够在软件上模拟浮点运算。这为系统开发者提供了一个在浮点处理器的硬件效率和定点设备 Blackfin 处理器的低功耗低成本之间选择的机会。根据不同的目标—完全符合标准还是速度--在定点处理器上可能使用 IEEE-754 标准或快速浮点(非 IEEE)格式。

在 Blackfin 处理器平台上，IEEE-754 浮点功能都可以通过库调用在 C/C++和汇编语言中实现。这些库采用定点逻辑模拟了浮点处理过程。

为了降低运算复杂度，使用更松散更快的浮点格式更有利。用这种方法常能显著地节省时钟周期。

本文主要讲解如何在Blackfin处理器上模拟快速浮点算术。采用*双字*格式代表 short 和 long 快速浮点数据类型，包括以下操作的 C 可调用的汇编源代码：加法，减法，乘法以及定点，IEEE-754 浮点和快速浮点格式之间的转换。

概述

在定点数字表示法中，小数点总是在同一位置。虽然这可以简化数字运算，节省内存，同时也对量级和精度带来了局限。面对要求更大数字范围或更高分辨率的情况，就需要位置可变的小数点。

浮点格式能表示非常大和非常小的数字。浮点格式基于科学计数符号，一个浮点数包含尾数(或小数)以及指数。在 IEEE-754 标准中，浮点数存放在一个 32bit 的字中，其中 23bit 尾数和 8bit 指数，以及一个符号位。在本文使用的快速浮点*双字*格式中，指数是一个 16bit 有符号整数，尾数是 16 或 32bit 有符号小数(取决于所使用的数据类型是 short 还是 long 型快速浮点数)。

标准化是浮点表示法的一个重要特性。如果尾数中不包含多余的符号位，则这个浮点数是*标准化的*，也就是所有的比特都是有意义的。标准化为

有效 bit 数提供最高的精度。它还简化了数值大小的比较，因为具有较大指数的数一定数值更大，而指数相等时，只需要比较尾数的大小。本文中所有程序都假设输入标准化数值，产生标准化的结果。

ADSP-BF535 Blackfin处理器的算术标志与其它Blackfin处理器有所不同。本文汇编代码都是基于非ADSP-BF535 Blackfin处理器。

浮点数据的 Short 和 Long 类型

本文代码中对于这两个不同数据类型都采用的*双字*格式。Short 数据(*fastfloat16*)用一个 16bit 字表示指数，另一个表示小数。Long 型数据(*fastfloat32*)采用一个 16bit 字表示指数，一个 32bit 字表示小数。*fastfloat32* 数据类型是计算强度更高的类型，但是比 *fastfloat16* 具有更高的精度。有符号二进制补码符号都假设成小数和指数形式。

表 1 fastfloat16 *数据类型的格式*

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

表 2 fastfloat32 *数据类型的格式*

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

定点和快速浮点格式之间的转换

Blackfin 处理器有两条指令可用于定点与快速浮点数之间的转换。第一条指令是 signbits，返回数据中的符号 bit(例如，指数位)。另一条是 ashift，用于尾数的标准化。

short(*fastfloat16*)型和 long(*fastfloat32*)型的汇编代码如下。

表 3 定点数到 short(fastfloat16*) 快速浮点数的转换。*

```
/***********************************************
fastfloat16 fract16_to_ff16(fract16);
---------------------------
输入参数(编译器转换)
R0.L= fastfloat16
输出参数(编译器转换)
R0.L=ff16.exp
R0.H=ff16.frac
---------------------------
***********************************************/
_fract16_to_ff16:
.global _ fastfloat16_to_ff16;
r1.l=signbits r0.l; //得到数据的符号 bit
r2=-r1(v);
r2.h=ashift r0.l by r1.l; //标准化尾数
r0=r2;
rts;
_ fastfloat16_to_ff16.end:
```

表 4 定点到 long 型快速浮点数(fastfloat32)*的转换*

```
/***********************************************
Fastfloat32 fract32_to_ff32(fract32);
---------------------------
输入参数(编译器转换)
R0.L= fastfloat32
输出参数(编译器转换)
R0.L=ff32.exp
R0.L=ff32.frac
---------------------------
***********************************************/
_fract32_to_ff32:
.global _ fastfloat32_to_ff32;
r1.l=signbits r0; //得到数据的符号 bit
r2=-r1(v);
r2.h=ashift r0 by r1.l; //标准化尾数
r0=r2;
rts;
_ fastfloat32_to_ff32.end:
```

将双字快速浮点数转化为定点格式，只须使用 ashift 指令。

short(*fastfloat16*)型和 long(*fastfloat32*)型的汇编代码如下。

表 5 short 快速浮点(fastfloat16)*数到定点数的转换。*

```
/***********************************************
Fastfloat16 ff16_to_fract16(fastfloat16);
---------------------------
输入参数(编译器转换)
R0.L= ff16.exp
R0.H=ff16.frac
输出参数(编译器转换)
R0.L=fract16
---------------------------
***********************************************/
_ff16_to_ fract16:
.global _ ff16_to_ fastfloat16;
r0.h=ashift r0.h by r0.l; //移动二进制点
r0>>=16;
rts;
_ ff16_to fastfloat16_.end:
```

表 6 long 快速浮点(fastfloat32)*数到定点数的转换。*

```
/***********************************************
fastfloat32 ff32_to_fract32(fastfloat32);
---------------------------
输入参数(编译器转换)
R0.L= ff32.exp
R0.H=ff32.frac
输出参数(编译器转换)
R0.L=fract32
---------------------------
***********************************************/
_ff32_to_ fract32:
.global _ ff32_to_ fastfloat32;
r0.h=ashift r1 by r0.l; //移动二进制点
r0>>=16;
rts;
_ ff32_to fastfloat32_.end:
```

快速浮点数与 IEEE 浮点数格式的转换

将 IEEE 浮点格式转换成快速浮点格式的基本方法是首先抽取出 IEEE 浮点数的尾数，指数以及符号 bit 范围。设置快速浮点指数字之前需保证指数是无偏的。尾数连同符号 bit 一起构成二进制补码有符号小数，从而形成完整的快速浮点数双字格式。

逆序执行上述步骤，可将快速浮点数转换成 IEEE 浮点格式。

本文未涉及 IEEE-754 浮点格式的更多细节。附件中的压缩包包含转换实现的 C 代码例程。

本文所附压缩包中包含快速浮点数与IEEE浮点数间的转换例程。注意不包括诸如NaN，+∞，-∞等任何IEEE的特殊值。

浮点数加法

两个双字快速浮点数的加法算法如下：

1. 确定指数较大的数，假设该数为 X(=Ex，Fx)，另一个为 Y(=Ey，Fy)。
2. 将结果的指数设置为 Ex。
3. 按照 Ex 与 Ey 的差值右移 Fy，对齐 Fx 和 Fy 的小数点。
4. Fx 加 Fy 得到结果的小数部分。
5. 如果需要，缩放输入参数来防止溢出。
6. 标准化结果。

short(fastfloat16)和 long(fastfloat32)的汇编代码如下。

对于*fastfloat16*的算术运算(加，减，除)，使用了以下参数传递转换。默认情况下，由Blackfin处理器的编译器完成。

调用参数
r0.h=Fraction of x(=Fx)
r0.l=Exponent of x(=Ex)

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

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

*表 7 short 型快速浮点(*fastfloat16*)数加法*

```
/***********************************************
fastfloat16
add_ff16 (fastfloat16, fastfloat16);
***********************************************/
_add_ff16:
.global _add_ ff16;
r2.l = r0.l - r1.l {ns}; //1s Ex>Ey?
cc = an; //结果为负?
r2.l = r2.l << 11 (s); //  确保移位范围在[-16,15]
r2.l = r2.l >>>11;
if !cc jump _add_ff16_1; // no  移位 y
r0.h = ashift r0.h by r2.l; //yes,移位 x
jump _add_ff16_2;
```

```
_add_ff16_1:
r2 = -r2(v);
r1.h = ashift r1.h by r2.l; //对 y 的值进行移位
a0 = 0;
a0.l = r0.l; //不能用  r1.h = r2.h
r1.l = a0 (iu); //使用 a0.x  作为中间存储地
_add_ff16_2:
r2.l = r0.h + r1.h (ns); //小数部分相加
cc = v; //测试是否有溢出
if cc jump _add_ff16_3;
//标准化
r0.l = signbits r2.l; //取得符号 bit 的数目
r0.h = ashift r2.l by r0.l; //标准化尾数
r0.l = r1.l – r0.l (ns); //调整指数
rts;

//尾数溢出情况
_add_ff16_3:
r0.h = r0.h >>>1; //缩小尾数
r1.h = r1.h >>>1;
r0.h = r0.h +r1.h (ns); //小数部分相加
r2.l = 1;
r0.l = r1.l + r2.l (ns); //调整指数
rts;
_add_ff16.end:
```

对于*fastfloat32*的算术运算(加，减，除)，使用了以下参数传递转换。默认情况下，由Blackfin处理器的编译器完成。

调用参数

r1 = Fraction of x (=Fx)

r0.l = Exponent of x (=Ex)

r3 = [FP+20] = Fraction of y (=Fy)

r2.l = Exponent of y (=Ey)

返回值

r1 = Fraction of z (=Fz)

r0.l = Exponent of z (=Ez)

*表 8 长快速浮点(*fastfloat32*)加法*

```
/************************************************
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}; //1s Ex>Ey?
cc = an; //结果为负?
r4.l = r4.l << 10(s); //  确保移位范围在[-32,31]
r4.l = r4.l >>>10;
if !cc jump _add_ff32_1; // no  移位 Fy
r1 = ashift r1 by r4.l; //yes,移位 Fx
jump _add_ff32_2;
```

```
_add_ff32_1:
r4 = -r4(v);
r3 = ashift r3 by r4.l; //对 Fy 的值进行移位
r2 = r0;
_add_f32_2:
r4 = r1 + r3 (ns); //小数部分相加
cc = v; //测试是否有溢出
if cc jump _add_ff32_3;

//标准化
r0.l = signbits r4; //取得符号 bit 的数目
r1 = ashift r4 by r0.l; //标准化尾数
r0.l = r2.l – r0.l (ns); //调整指数
FF32_EPILOGUE();
rts;

//尾数溢出情况
_add_ff32_3:
r1 = r1 >>>1; //缩小尾数
r3 = r3 >>>1;
r1 = r1 +r3 (ns); //小数部分相加

r4.l = 1;
r0.l = r2.l + r4.l (ns); //调整指数
FF32_EPILOGUE;
rts;
_add_ff32.end:
```

浮点数减法

双字快速浮点数的减法算法如下：

1. 确定指数较大的数，假设该数为 X(=Ex，Fx)，另一个为 Y(=Ey，Fy)。
2. 将结果的指数设置为 Ex。
3. 按照 Ex 与 Ey 的差值右移 Fy，对齐 Fx 和 Fy 的小数点。
4. 被减数的小数部分减去减数的小数部分得到结果的小数部分。
5. 如果需要，缩放输入参数来防止溢出。
6. 标准化结果。

short(*fastfloat16*)和 long(*fastfloat32*)的汇编代码如下。

表 9 short 快速浮点(fastfloat16)*减法*

```
/************************************************
fastfloat16 sub_ff16 (fastfloat16, fastfloat16);
************************************************/
.global _sub_ ff16;
_sub_ff16:
r2.l = r0.l - r1.l {ns}; //1s Ex>Ey?
cc = an; //结果为负?
r2.l = r2.l << 11 (s); //  确保移位范围在[-16,15]
r2.l = r2.l >>>11;
if !cc jump _sub_ff16_1; // no  移位 y
r0.h = ashift r0.h by r2.l; //yes,移位 x
jump _sub_ff16_2;

_sub_ff16_1:
r2 = -r2(v);
r1.h = ashift r1.h by r2.l; //对 y 的值进行移位
a0 = 0;
a0.l = r0.l; //不能用  r1.h = r2.h
r1.l = a0 (iu); //使用 a0.x  作为中间存储器

_sub_ff16_2:
```

```
r2.l = r0.h - r1.h (ns); //小数部分相加
cc = v; //测试是否有溢出
if cc jump _sub_ff16_3;

//标准化
r0.l = signbits r2.l; //取得符号 bit 的数目
r0.h = ashift r2.l by r0.l; //标准化尾数
r0.l = r1.l – r0.l (ns); //调整指数
rts;

//尾数溢出情况
_sub_ff16_3:
r0.h = r0.h >>>1; //缩小尾数
r1.h = r1.h >>>1;
r0.h = r0.h - r1.h (ns); //小数部分相加
r2.l = 1;
r0.l = r1.l + r2.l (ns); //调整指数
rts;
_sub_ff16.end:
```

表 10 长快速浮点(fastfloat32)*减法*

```
/***********************************************
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}; //1s Ex>Ey?
cc = an; //结果为负?
r4.l = r4.l << 10 (s); //  确保移位范围在[-32,31]
r4.l = r4.l >>>10;
if !cc jump _add_ff32_1; // no  移位 Fy
r1 = ashift r1 by r4.l; //yes,移位 Fx
jump _sub_ff32_2;

_sub_ff32_1:
r4 = -r4(v);
r3 = ashift r3 by r4.l; //对 Fy 的值进行移位
r2 = r0;

_sub_ff32_2:
r4 = r1 - r3 (ns); //小数部分相加
cc = v; //测试是否有溢出
if cc jump _add_ff32_3;

//标准化
r0.l = signbits r4; //取得符号 bit 的数目
r1 = ashift r4 by r0.l; //标准化尾数
r0.l = r2.l – r0.l (ns); //调整指数
FF32_EPILOGUE();
rts;

//尾数溢出情况
_add_ff32_3:
r1 = r1 >>>1; //缩小尾数
r3 = r3 >>>1;
r1 = r1 - r3 (ns); //小数部分相加
r4.l = 1;
r0.l = r2.l + r4.l (ns); //调整指数
FF32_EPILOGUE;
rts;
_add_ff32.end:
```

浮点数乘法

因为不用对齐小数点，双字快速浮点数的乘法比加减法都简单。*x* 和 *y*(Ex，Fx 和 Ey，Fy)两数相乘的算法如下：

1. 结果的指数等于 Ex 和 Ey 的和。
2. Ex 乘以 Ey 的值得到小数部分的结果。
3. 标准化结果。

short(*fastfloat16*)和 long(*fastfloat32*)的汇编代码如下。

表 11 short 快速浮点(fastfloat16)*乘法*

```
/***********************************************
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; //取得符号 bit 的数目
a0 = ashift a0 by r2.l; //尾数标准化
r0 = a0;
r0.l = r3.l – r2.l (ns); //调整指数
rts;
_mult_ff16.end:
```

表 12 short 快速浮点(fastfloat32)*乘法*

```
/***********************************************
fastfloat32 void mult_ff32 (fastfloat32, fastfloat32);
***********************************************/
#define FF32_PROLOGUE() link 0; r3 = [fp+20]; [--sp]=r4; [--sp]=r5

#define FF32_EROLOGUE() link 0; r5 = [sp++]; r4 =[sp++];unlink

.global _mult_ff32;
_mult_ff32:
FF32_PROLOGUE();
r0.l = r0.l + r2.l (ns);//指数相加

//实现 32bit 小数乘法(VisualDSP++编译器执行)
r2 = pack(r1.l, r3.l);
cc = r2;
a1 = a1 >>16;
a1 += r1.h * r3.l (m), a0 = r1.h * r3.h;
cc &= av0;
a1 += r3.h * r1.l (m);
a1 = a1 >>>15;
r1 = (a0 += a1);
r2 = cc;
r1 = r2 + r1;

//标准化
r4.l = signbits r1; //取得符号 bit 的数目
r1 = ashift r1 by r4.l; //尾数标准化
r0.l = r0.l – r4.l(ns); //调整指数
FF32_EPILOGUE();
rts;
_mult_ff32.end:
```

总结

本文描述的双字快速浮点技术能够在定点 Blackfin 处理器平台上极大地提高浮点计算效

率。前文记述的特殊操作能作为独立流程，或特殊应用中的更先进计算细节的起始点。本文随附的压缩源代码包中提供了在新建工程中使用快速浮点方法的切入点。

参考文献

[1] *Digital Signal Processing Applications: Using the ADSP-2100 Family (Volume 1).*1992.Analog Device, Inc.

[2] *The Art of Computer Programming: Volume2 / Seminumerical Algorithms.* Knuth, D.E. Second Edition, 1969. Addison Wesley Publishing Company.

[3] *IEEE Standard for Binary Floating-Point Arithmetic: ANSI/IEEE Std 754-1985.*1985. Institute of Electrical and Electronicsd Engineers.

文档记录

Revision	**Description**
Rev 4 – August 23, 2007 *by Tom L.*	Corrected the mult_ff32 function; Updated formatting
Rev 3 – May 26, 2003 *by Tom L.*	Code updated to check for overflow conditions; New test cases added.
Rev 2 – May 12, 2003 *by Tom L.*	Updated according to new naming conventions
Rev 1 – February 19, 2003 *by Tom L.*	Initial Release