Application Note 33

Fixed Point Arithmetic on the ARM

Document Number: ARM DAI 0033A
Issued: September 1996
Copyright Advanced RISC Machines Ltd (ARM) 1996
All rights reserved

ENGLAND
Advanced RISC Machines Limited
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemering
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Principles of Fixed Point Arithmetic</td>
<td>3</td>
</tr>
<tr>
<td>Examples</td>
<td>5</td>
</tr>
<tr>
<td>Programming in C</td>
<td>7</td>
</tr>
<tr>
<td>Programming in Assembler</td>
<td>10</td>
</tr>
</tbody>
</table>
Introduction

1 Introduction

This application note describes how to write efficient fixed point arithmetic code using the ARM C compiler and ARM or Thumb assembler. Since the ARM core is an integer processor, all floating point operations must be simulated using integer arithmetic. Using fixed point arithmetic instead of floating point will considerably increase the performance of many algorithms.

This document contains the following sections:

• 2 Principles of Fixed Point Arithmetic on page 3 describes the mathematical concepts needed for fixed point arithmetic.
• 3 Examples on page 5 gives examples of writing fixed point code for signal processing and graphics processing, two of the most common uses.
• 4 Programming in C on page 7 covers the practical details of implementing fixed point code in C. An example program is given together with a set of macro definitions.
• 5 Programming in Assembler on page 10 deals with assembler examples.
2 Principles of Fixed Point Arithmetic

In computing arithmetic, fractional quantities can be approximated by using a pair of integers \((n, e)\): the mantissa and the exponent. The pair represents the fraction:

\[ n2^{-e} \]

The exponent \(e\) can be considered as the number of digits you have to move into \(n\) before placing the binary point.

For example:

If \(e\) is a variable quantity, held in a register and unknown at compile time, \((n, e)\) is said to be a floating point number. If \(e\) is known in advance, at compile time, \((n, e)\) is said to a fixed point number. Fixed point numbers can be stored in standard integer variables (or registers) by storing the mantissa.

The following subsections show how to perform the basic arithmetic operations on two fixed point numbers, \(a\) and \(b\), expressing the answer in the form \(c = k2^{-r}\), where \(p\), \(q\) and \(r\) are fixed constant exponents.

2.1 Change of exponent

The simplest operation to be performed on a fixed point number is to change the exponent. To change the exponent from \(p\) to \(r\) (to perform the operation \(c = a\)) the mantissa \(k\) can be calculated from \(n\) by a simple shift.

Since:

\[ n2^{-p} = n2^{-r-p} \times 2^{-r} \]

you have the formula:

\[
\begin{align*}
    k &= n << (r-p) \quad \text{if } (r>p) \\
    k &= n >> (p-r) \quad \text{if } (p>r)
\end{align*}
\]

2.2 Addition and subtraction

To perform the operation \(c = a+b\), first convert \(a\) and \(b\) to have the same exponent \(r\) as \(c\) and then add the mantissas. This method is proved by the equation:

\[ n2^{-r} + m2^{-r} = (n+m)2^{-r} \]

Subtraction is similar.
Principles of Fixed Point Arithmetic

2.3 Multiplication

The product \( c = ab \) can be performed using a single integer multiplication. From the equation:

\[ ab = n2^{-p} \times m2^{-q} = (nm)2^{-(p+q)} \]

it follows that the product \( n \times m \) is the mantissa of the answer with exponent \( p+q \). To convert the answer to have exponent \( r \), perform shifts as described above.

For example, if \( p + q \geq r \):

\[ k = (n \times m) \gg (p+q-r) \]

2.4 Division

Division, \( c = a/b \), can also be performed using a single integer division. The equation is:

\[ \frac{a}{b} = \frac{n2^{-p}}{m2^{-q}} = \left(\frac{n}{m}\right)2^{q-p} = \left(\frac{n}{m}\right)2^{(r+q-p)2^{-r}} \]

In order not to lose precision, the multiplication by \( 2^{(r+q-p)} \) must be performed before the division by \( m \).

For example, assuming that \( r + q \geq p \), perform the calculation:

\[ k = (n<<(r+q-p))/m \]

2.5 Square root

The equation for square root is:

\[ \sqrt{a} = \sqrt{n2^{-p}} = \sqrt{n2^{(2r-p)}} \times 2^{-r} \]

In other words, to perform \( c = \sqrt{a} \), set \( k = \text{isqr}(n<<(2r-p)) \) where \( \text{isqr} \) is an integer square root function.

2.6 Overflow

The above equations show how to perform the basic operations of addition, subtraction, multiplication, division and root extraction on numbers in fixed point form, using only integer operations. However, in order for the results to be accurate to the precision the answer is stored in (\( \pm 2^{-r} \)) you must ensure that overflows do not occur within the calculation. Every shift left, add/subtract or multiply can produce an overflow and lead to a nonsensical answer if the exponents are not carefully chosen.

The following examples of “real world” situations indicate how to choose the exponent.

---

Open Access
3 Examples

3.1 Signal processing

In signal processing the data often consists of fractional values, in the range -1 to +1. This example uses exponent q and 32-bit integer mantissas (so that each value can be held in a single ARM register). In order to be able to multiply two numbers without overflow, you need $2q < 32$, or $q \leq 15$. In practice $q=14$ is often chosen as this allows multiplication with several accumulates without risk of overflow.

Fix $q=14$ at compile time. Then 0xFFFFFC000 represents $-1$, 0x00004000 represents $+1$ and 0x00000001 represents $2^{-14}$, the finest division.

Suppose that $x$, $y$ are two $q=14$ mantissas and $n$, $m$ are two integers. By applying the above formulas you derive the basic operations:

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y$</td>
</tr>
<tr>
<td>$x + a \ll 14$</td>
</tr>
<tr>
<td>$xa$</td>
</tr>
<tr>
<td>$xy \gg 14$</td>
</tr>
<tr>
<td>$x/a$</td>
</tr>
<tr>
<td>$(a\ll 14)/b$</td>
</tr>
<tr>
<td>$(x\ll 14)/y$</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
</tr>
</tbody>
</table>

Table 3-2: Basic operations
3.2 Graphics processing

In this example, the \((x, y, z)\) coordinates are stored in fractional form with eight bits of fraction information \((q=8)\). If \(x\) is stored in a 32-bit register then the lowest byte of \(x\) gives the fractional part and the highest three parts the integer part.

To calculate the distance from the origin, \(d\), in \(q=8\) form:

\[
d = \sqrt{x^2 + y^2 + z^2}
\]

If you apply the above formulae directly and keep all intermediate answers in \(q=8\) form, you arrive at the following code:

\[
\begin{align*}
x &= (x\times x) >> 8 & \text{square } x \\
y &= (y\times y) >> 8 & \text{square } y \\
z &= (z\times z) >> 8 & \text{square } z \\
s &= x+y+z & \text{sum of squares} \\
d &= \text{sqr}(s<<8) & \text{the distance in } q=8 \text{ form}
\end{align*}
\]

Alternatively, if you keep the intermediate answers in \(q=16\) form, the number of shifts is reduced and the accuracy increased:

\[
\begin{align*}
x &= x\times x & \text{square of } x \text{ in } q=16 \text{ form} \\
y &= y\times y & \text{square of } y \text{ in } q=16 \text{ form} \\
z &= z\times z & \text{square of } z \text{ in } q=16 \text{ form} \\
s &= x+y+x & \text{sum of squares in } q=16 \text{ form} \\
d &= \text{sqr}(s) & \text{distance } d \text{ in } q=8 \text{ form}
\end{align*}
\]

3.3 Summary

- If you add two numbers in \(q\)-form, they stay in \(q\)-form.
- If you multiply two numbers in \(q\) form the answer is in \(2q\)-form.
- If you take the square root of a number in \(q\) form the answer is in \(q/2\)-form.
- To convert from \(q\)-form to \(r\)-form you shift left by \((r-q)\) or right by \((q-r)\), depending on which of \(q\) and \(r\) is greater.
- To get the best precision results, choose \(q\) to be the largest number such that the intermediate calculations cannot overflow.
4 Programming in C

Fixed point arithmetic can be programmed in C by using the standard integer arithmetic operations and using shifts to change $q$-form when this is necessary (usually before or after an operation to ensure that the answer is still in $q$-form).

To make programming easier to read, a set of C macros have been defined. The example program below defines these macros and illustrates their use.

```c
/* A Simple C program to illustrate the use of Fixed Point Operations */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* DEFINE THE MACROS */

/* The basic operations perfomed on two numbers a and b of fixed point q format returning the answer in q format */

#define FADD(a,b) ((a)+(b))
#define FSUB(a,b) ((a)-(b))
#define FMUL(a,b,q) (((a)*(b))>>(q))
#define FDIV(a,b,q) (((a)<<(q))/(b))

/* The basic operations where a is of fixed point q format and b is an integer */

#define FADDI(a,b,q) ((a)+((b)<<(q)))
#define FSUBI(a,b,q) ((a)-((b)<<(q)))
#define FMULI(a,b) ((a)*(b))
#define FDIVI(a,b) ((a)/(b))

#define FCONV(a, q1, q2) (((q2)>(q1)) ? (a)<<((q2)-(q1)) : (a)>>((q1)-(q2)))

#define FADDG(a,b,q1,q2,q3) (FCONV(a,q1,q3)+FCONV(b,q2,q3))
#define FSUBG(a,b,q1,q2,q3) (FCONV(a,q1,q3)-FCONV(b,q2,q3))
#define FMULG(a,b,q1,q2,q3) FCONV((a)*((b), (q1)+(q2), q3)
#define FDIVG(a,b,q1,q2,q3) (FCONV(a, q1, (q2)+(q3))/1(b))
```

/* convert to and from floating point */

```c
/* convert a from q1 format to q2 format */

#define FCONV(a, q1, q2) (((q2)>(q1)) ? (a)<<((q2)-(q1)) : (a)>>((q1)-(q2)))

/* the general operation between a in q1 format and b in q2 format returning the result in q3 format */

#define FADDG(a,b,q1,q2,q3) (FCONV(a,q1,q3)+FCONV(b,q2,q3))
#define FSUBG(a,b,q1,q2,q3) (FCONV(a,q1,q3)-FCONV(b,q2,q3))
#define FMULG(a,b,q1,q2,q3) FCONV((a)*((b), (q1)+(q2), q3)
#define FDIVG(a,b,q1,q2,q3) (FCONV(a, q1, (q2)+(q3))/1(b))
```
```c
#define TOFIX(d, q) ((int)( (d)*(double)(1<<(q)) )
#define TOFLT(a, q) ( (double)(a) / (double)(1<<(q)) )

#define TEST(FIX, FLT, STR) ( \ 
  a = a1 = randint(); \ 
  b = bi = a2 = randint(); \ 
  fa = TOFLT(a, q); \ 
  fa1 = TOFLT(a1, q1); \ 
  fa2 = TOFLT(a2, q2); \ 
  fb = TOFLT(b, q); \ 
  ans1 = FIX; \ 
  ans2 = FLT; \ 
  printf("Testing %s\n fixed point answer=%f\n floating point answer=%f\n", \ 
      STR, TOFLT(ans1, q), ans2); \ 
)

int randint(void) { 
  int i;
  i=rand();
  i = i & 32767;
  if (rand() & 1) i=-i;
  return i;
}

int main(void) { 
  int q=14, q1=15, q2=7, q3=14;
  double fa, fb, fa1, fa2;
  int a,b,bi,a1,a2;
  int ans1;
  double ans2;

  /* test each of the MACRO's with some random data */
  TEST(FADD(a,b), fa+fb, "FADD");
  TEST(FSUB(a,b), fa-fb, "FSUB");
  TEST(FMUL(a,b,q), fa*fb, "FMUL");
  TEST(FDIV(a,b,q), fa/fb, "FDIV");
  TEST(FADDI(a,bi,q), fa+bi, "FADDI");
  TEST(FSUBI(a,bi,q), fa-bi, "FSUBI");
  TEST(FMULI(a,bi), fa*bi, "FMULI");
  TEST(FDIVI(a,bi), fa/bi, "FDIVI");
  TEST(FADDG(a1,a2,q1,q2,q3), fa1+fa2, "FADDG");
  TEST(FSUBG(a1,a2,q1,q2,q3), fa1-fa2, "FSUBG");
  TEST(FMULG(a1,a2,q1,q2,q3), fa1*fa2, "FMULG");
  TEST(FDIVG(a1,a2,q1,q2,q3), fa1/fa2, "FDIVG");
  printf("Finished standard test\n");
}
```
/* the following code will calculate \( \exp(x) \) by summing the
series (not efficient but useful as an example) and compare it
with the actual value */

while (1) {
    printf("Please enter the number to be exp'ed (<1): ");
    scanf("%lf", &fa);
    printf(" x = %f\n", fa);
    printf(" \exp(x) = %f\n", exp(fa));
    q = 14; /* set the fixed point */
    a = TOFIX(fa, q); /* get \( a \) in fixed point format */
    a1 = FCONV(1, 0, q); /* \( a^0 \) */
    a2 = 1; /* 0 factorial */
    ans1 = 0; /* series sum so far */
    for (bi=0 ; bi<10; bi++) { /* do ten terms of the series */
        int j;
        j = FDIVI(a1, a2); /* \( a^n/n! \) */
        ans1 = FADD(ans1, j);
        a1 = FMUL(a1, a, q); /* increase power of \( a \) by 1 */
        a2 = a2*(bi+1); /* next factorial */
        printf("Stage %d answer = %f\n", bi, TOFLT(ans1, q));
    }
    return 0;
}
5 Programming in Assembler

5.1 ARM assembler

The ARM’s barrel shifter makes it very efficient at executing fixed point arithmetic. Suppose, as in the signal processing examples, that $x$ and $y$ are two $q=14$ numbers with mantissas held in registers, and that $a$ and $b$ are registers holding integer values. The examples in Table 5-3: ARM assembler operations show the basic operations to produce an answer $c$ in $q=14$ format.

<table>
<thead>
<tr>
<th>Operation</th>
<th>C code</th>
<th>Assembler code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = x + y$</td>
<td>$c=x+y$</td>
<td>ADD c, x, y</td>
</tr>
<tr>
<td>$c = x + a$</td>
<td>$c=x+(a&lt;&lt;14)$</td>
<td>ADD c, x, a, LSL#14</td>
</tr>
<tr>
<td>$c = xa$</td>
<td>$c=x*a$</td>
<td>MUL c, x, a</td>
</tr>
<tr>
<td>$c = xy$</td>
<td>$c=(x*y)&gt;&gt;14$</td>
<td>MUL c, x, y</td>
</tr>
<tr>
<td>$c = \frac{x}{y}$</td>
<td>$c=(x&lt;&lt;14)/y$</td>
<td>MOV R0, x, LSL#14</td>
</tr>
<tr>
<td>$c = x^2 + y^2 + x + y$</td>
<td>$c=(x<em>x+y</em>y)&gt;&gt;14+x+y$</td>
<td>MUL c, x, x</td>
</tr>
<tr>
<td>$c = \sqrt{x}$</td>
<td>$c=isqr(x&lt;&lt;14)$</td>
<td>MOV R0, c, LSL#14</td>
</tr>
</tbody>
</table>

In effect, the barrel shift on each instruction gives one free exponent conversion per instruction. Where long fractional bit accuracy is required (for example, $q=30$ rather than $q=14$), the ARM7DM’s long multiply and long multiply accumulate instructions provide an efficient implementation.

For example, to perform the operation $s = x^2 + y^2 + z^2$, where $s$, $x$, $y$, $z$ are in $q=30$ form, use the code:

- SMULL R0, R1, x, x \hspace{1cm} R1.R0 = x^2 \text{ in } q=60 \text{ form}
- SMLAL R0, R1, y, y \hspace{1cm} R1.R0 = x^2 + y^2 \text{ in } q=60 \text{ form}
- SMLAL R0, R1, z, z \hspace{1cm} R1.R0 = x^2 + y^2 + z^2 \text{ in } q=60 \text{ form}
- MOV s, R0, LSR#30
- ORR s, s, R1, LSL#2 \hspace{1cm} s = x^2 + y^2 + z^2 \text{ in } q=30 \text{ form}
### 5.2 Thumb assembler

*Table 5-4: Thumb assembler operations* duplicates the table of the previous section with Thumb code examples in place of ARM code. The code is very similar, the main difference being that separate shift instructions are needed for q format conversion.

<table>
<thead>
<tr>
<th>Operation</th>
<th>C code</th>
<th>THUMB code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = x + y$</td>
<td>c=x+y</td>
<td>ADD c, x, y</td>
</tr>
<tr>
<td>$c = x + a$</td>
<td>c=x+(a&lt;&lt;14)</td>
<td>LSL c, a, #14</td>
</tr>
<tr>
<td>$x = xa$</td>
<td>x=x*a</td>
<td>MUL x, a</td>
</tr>
<tr>
<td>$c = xy$</td>
<td>c=(x*y)&gt;&gt;14</td>
<td>MUL x, y</td>
</tr>
<tr>
<td>$c = \frac{x}{y}$</td>
<td>c=(x&lt;&lt;14)/y</td>
<td>LSL R0, x, #14</td>
</tr>
<tr>
<td>$c = x^2 + y^2 + x + y$</td>
<td>c=(x*x+y+y)&gt;&gt;14 + x+y</td>
<td>ADD R0, x, y</td>
</tr>
<tr>
<td>$c = \sqrt{x}$</td>
<td>c=isqr(x&lt;&lt;14)</td>
<td>LSL R0, c, #14</td>
</tr>
</tbody>
</table>

*Table 5-4: Thumb assembler operations*