Application Note 33

Fixed Point Arithmetic on the ARM

Document Number: ARM DAI 0033A

Issued: September 1996

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-Riemerling
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

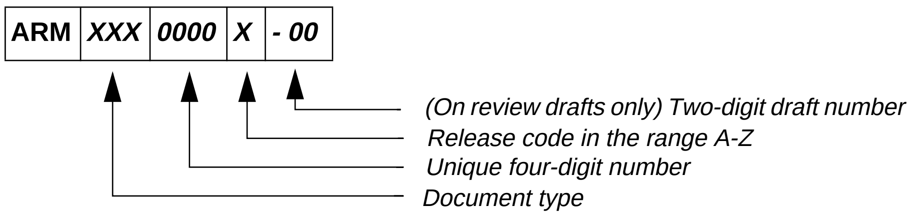
World Wide Web address: http://www.arm.com

ARM
Advanced RISC Machines

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.


ARM
XXX
0000
X
- 00
(On review drafts only) Two-digit draft number
Release code in the range A-Z
Unique four-digit number
Document type

Document Status

The document’s status is displayed in a banner at the bottom of each page. This describes the document’s confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	September 1996	SKW	First release

Table of Contents

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:

Mantissa (*n*)	Exponent (*e*)	Binary	Decimal
01100100	-1	011001000.	200
01100100	0	01100100.	100
01100100	1	0110010.0	50
01100100	2	011001.00	25
01100100	3	01100.100	12.5
01100100	7	0.1100100	0.78125

Table 2-1: Principles of fixed point arithmetic

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.

For fixed point numbers, the exponent *e* is usually denoted by the letter *q*. The following subsections show how to perform the basic arithmetic operations on two fixed point numbers, $a = n2^{-p}$ and $b = m2^{-q}$, 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:

```
k = n << (r-p)       if (r>=p)
k = n >> (p-r)       if (p>r)
```

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.

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 $n*m$ 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*m) >> (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=isqr(n<<(2r-p))` where `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.

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 0xFFFFC000 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:

Operation	Code to produce mantissa of answer in q=14 format
$x + y$	`x+y`
$x + a$	`x+(a<<14)`
xa	`x*a`
xy	`(x*y)>>14`
$\frac{x}{a}$	`x/a`
$\frac{a}{b}$	`(a<<14)/b`
$\frac{x}{y}$	`(x<<14)/y`
$\sqrt{x}$	`sqr(x<<14)`

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:

```
x = (x*x)>>8        square x
y = (y*y)>>8        square y
z = (z*z)>>8        square z
s = x+y+z           sum of squares
d = sqrt(s<<8)      the distance in q=8 form
```

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

```
x = x*x             square of x in q=16 form
y = y*y             square of y in q=16 form
z = z*z             square of z in q=16 form
s = x+y+x           sum of squares in q=16 form
d = sqr(s)          distance d in q=8 form
```

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.

```
/* 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))

/* 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))/(b))

/* convert to and from floating point */
```

```
#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, "FSUBI");
  TEST(FDIVI(a,bi), fa/bi, "FSUBI");
  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 to power 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.

Operation	C code	Assembler code
$c = x + y$	`c=x+y`	`ADD c, x, y`
$c = x + a$	`c=x+(a<<14)`	`ADD c, x, a, LSL#14`
$c = xa$	`c=x*a`	`MUL c, x, a`

$c = xy$	`c=(x*y)>>14`	`MUL c, x, y`
		`MOV c, c, ASR#14`

$c = \frac{x}{y}$	`c=(x<<14)/y`	`MOV R0, x, LSL#14`
		`MOV R1, y`
		`BL  divide`
		`MOV c, R0`

$c = x^2 + y^2 + x + y$	`c=((x*x+y*y)>>14)+x+y`	`MUL c, x, x`
		`MLA c, y, y, c`
		`ADD c, x, c, ASR#14`
		`ADD c, c, y`

$c = \sqrt{x}$	`c=isqr(x<<14)`	`MOV R0, c, LSL#14`
		`BL  isqr`
		`MOV c, R0`

Table 5-3: ARM assembler operations

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`	$R1.R0 = x^2$ in *q=60* form
`SMLAL R0, R1, y, y`	$R1.R0 = x^2 + y^2$ in *q=60* form
`SMLAL R0, R1, z, z`	$R1.R0 = x^2 + y^2 + z^2$ in *q=60* form
`MOV   s, R0, LSR#30`	
`ORR   s, s, R1, LSL#2`	$s = x^2 + y^2 + z^2$ in *q=30* 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.

Operation	C code	THUMB code
$c = x + y$	`c=x+y`	`ADD c, x, y`

$c = x + a$	`c=x+(a<<14)`	`LSL c, a, #14`
		`ADD c, x`

$x = xa$	`x=x*a`	`MUL x, a`

$c = xy$	`c=(x*y)>>14`	`MUL x, y`
		`ASR c, x, #14`

$c = \frac{x}{y}$	`c=(x<<14)/y`	`LSL R0, x, #14`
		`MOV R1, y`
		`BL  divide`
		`MOV c, R0`

$c = x^2 + y^2 + x + y$	`c=((x*x+y*y)>>14)+x+y`	`ADD R0, x, y`
		`MOV R1, x`
		`MOV R2, y`
		`MUL R1, x`
		`MUL R2, y`
		`ADD R1, R1, R2`
		`ASR R1, R1, #14`
		`ADD c, R0, R1`

$c = \sqrt{x}$	`c=isqr(x<<14)`	`LSL R0, c, #14`
		`BL  isqr`
		`MOV c,R0`

Table 5-4: Thumb assembler operations