9 Fixed-Point Processing

Contents

9.2

Introduction

Fixed-point means 'integer'

Most microprocessors are fixed-point devices – they only have support for arithmetic with integers. For example, the ARM® Cortex®-M3 processor does not have a floating-point unit (FPU). The Cortex®-M4 is a relatively special MCU because it has the option to include hardware that directly supports single-precision floating-point numbers – but at the expense of increased cost and power consumption. PC processors since the 80486DX (released in 1989) have a "math coprocessor on chip", and all subsequent generations have included an FPU. This is why PCs are fast, and expensive – a large proportion of the die area and power consumption of the CPU is taken up by the FPU.

If you do not use the FPU then compiled code can be used on another Cortex-M4 microcontroller product that does not have FPU support. Floating-point operations can be emulated in software on a fixed-point processor using special maths libraries, but the resulting overhead results in programs that run 40-100 times slower than a program that uses just fixed-point operations.

Fixed-point calculations are important when time is important

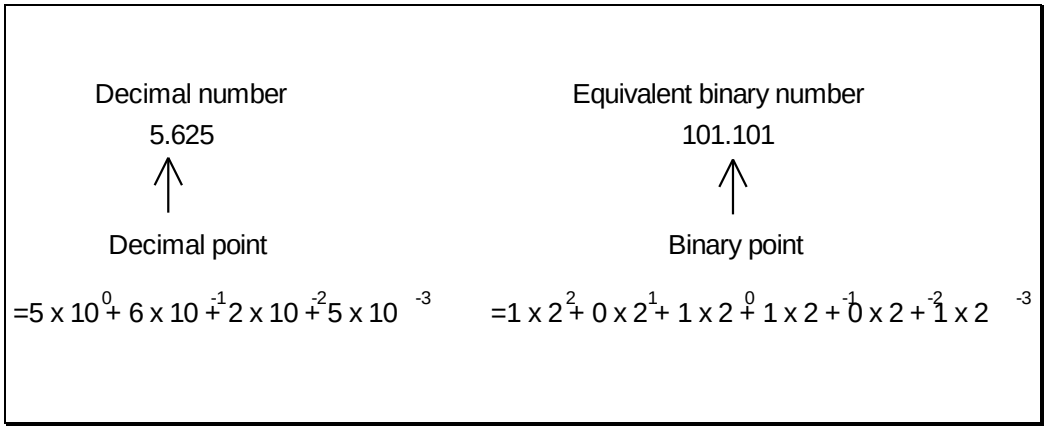
Therefore, when cost, power consumption and speed (i.e. time) is of primary importance in a design, it is necessary to perform arithmetic operations using a fixed-point processor. We therefore need to examine processing techniques that use integers but provide an interpretation of the resulting numbers as having fractional parts.

9.1 Q Notation

Fixed-point calculations are capable of performing fractional mathematics if an implied binary point is used in the *interpretation* of the integer used to represent a fractional quantity. In accordance with accepted digital signal processing (DSP) notation, we use what is called "Q notation". The "Q" stands for quotient, or a number with a fractional part.

Most quantities in signal processing use either 16 bits or 32 bits for their representation. To express a fractional part, an implied binary point is required for each quantity. It is up to us as designers to keep track of these implied binary points throughout any and all calculations. For each quantity, we express its fractional part with the notation *m*Q*n* where *n* is an integer ranging from 0 to 16 for 16-bit quantities or 0-32 for 32-bit quantities. The *m* tells how many bits are used in total, either 16 or 32. The *n* tells how many bits are to the right of the implied binary point.

Just like a decimal point, a binary point interprets digits to the right of it as being negative powers of the base. A comparison of a decimal number and its equivalent binary number is given below:



Comparison of a decimal number and equivalent binary number

Figure 9.1

Mapping integers to fractional quantities

For example, a 6Q3 number implies 3 bits to the right of the implied binary point. A mapping of the CPU's integer values to quantities that we interpret is made as follows:

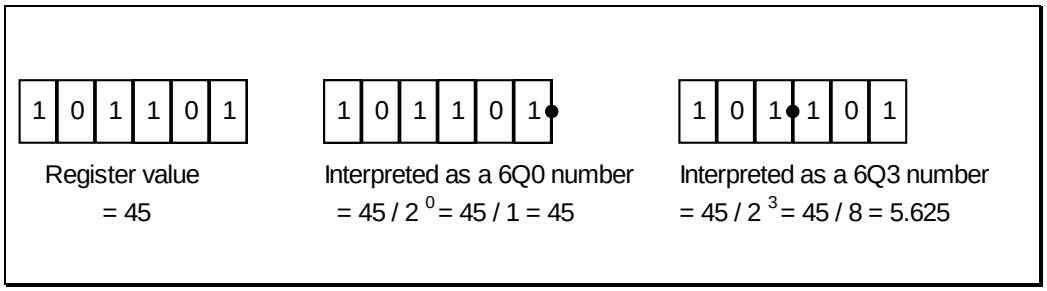


Figure 9.2

From this, it should be apparent that to interpret a register value as a *m*Q*n* value, we simply divide the raw value by 2^n. To store a fractional number in *m*Q*n* notation, we multiply it by 2^n and truncate or round the answer to an integer. This inherent round-off error cannot be prevented.

For example, if we wished to store the number 5.628 in 16Q3 notation, we get:

$$5.628 \times 2^3 = 5.628 \times 8 = 45.024 \qquad (10.1)$$
$$\therefore \text{store as } 45$$

In this case it is impossible to distinguish between 5.625 and 5.628 in 16Q3 notation.

The resolution of *m*Q*n* numbers can therefore be expressed as 2^{-n}. For example, in 16Q3 notation the resolution of the stored numbers is $2^{-3} = 0.125$. Every number in Q3 notation will be a multiple of 0.125. Clearly it is desirable to have a large *n* to store fractional values with the greatest accuracy. It is in fact impossible to store 5.628 exactly (no round-off error). The best we can do using 32-bits is to store the integer part (5 in 5.628) using the least amount of bits (3 in this case) and use the rest for the fractional part.

We therefore would use a 32Q29 number:

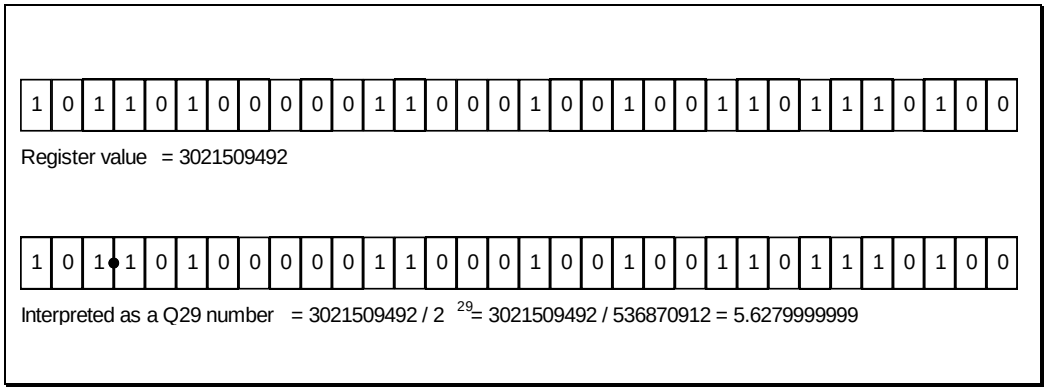


Figure 9.3

The reason we can't store this number exactly is because when we multiply 5.628 by successive powers of two to obtain an integer, the last digits form a cyclic pattern, that will never reach a multiple of 10:

$$\begin{aligned} &5.628 \times 2 = 11.256 \\ &11.256 \times 2 = 22.512 \\ &22.512 \times 2 = 45.024 \\ &45.024 \times 2 = 90.048 \\ &90.048 \times 2 = 180.096 \\ &180.096 \times 2 = 360.192 \\ &\text{etc.} \end{aligned} \qquad (10.2)$$

This shouldn't really worry us, because a 32Q29 number has a resolution of $2^{-29} = 1.8626451 \times 10^{-9}$. The error in storing the above number as shown is therefore less than 0.00000003 %.

As an aside, we should not forget that using floating-point numbers does not increase our accuracy. Accuracy is determined purely by the number of bits, not in the *way* the number is stored. It shocks some people to find that floating-point units cannot store the number 0.1, precisely because of the problem stated above. However, the floating-point number can get very close to 0.1 in the same way that we can get very close to 5.628.

9.2 Other Notations

The Q notation is convenient because it expresses a number as powers of two. It will be shown later that this provides an efficient method to convert numbers from one Q representation to another.

We can also express numbers using a base other than 2. For example, suppose we say that the number 1000 is to be interpreted as 1. We say that the number has 1000 as a base, or unity value, and that 1000 = 1 per unit (p.u.). The number 5.628 in this method would be represented as 5628, which is exact. Why don't we use this method over Q notation? The answer is because other numbers can now not be represented exactly. Remember – the fundamental limit in accuracy is set by the number of bits, and not how they are interpreted.

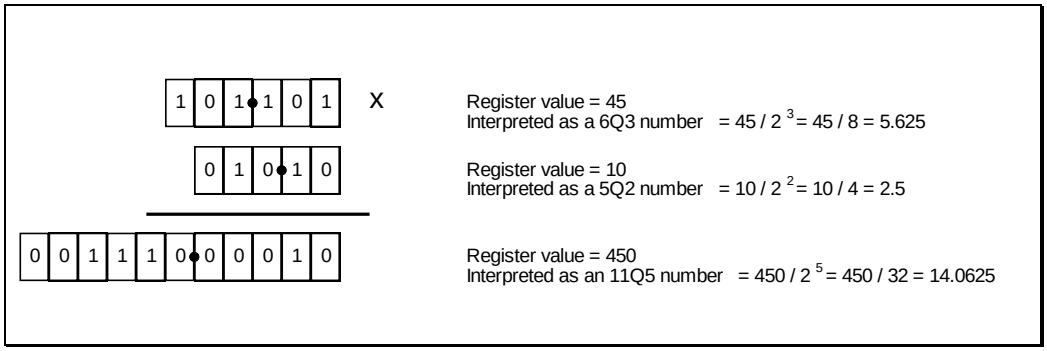
Complications arise in calculations involving multiplications and divisions. For example, multiplying two numbers with a base of 1000 produces a number whose base is 1000000. To *normalise* this result back to 1000, the result would have to be divided by 1000 – this division is expensive in terms of CPU time and is to be avoided.

It should be noted that Q notation is just representing numbers with bases that are multiples of two. For example a 16Q3 number is a number with a base or p.u. value of 8.

9.3 Fixed-Point Calculations

9.3.1 Multiplication

Multiplying two numbers together changes the base or "per unit" value. For example, consider the following multiplication:



Multiplication changes the base value

Figure 9.4

Two things happen – 1) the length of the result is equal to the sum of the lengths of the two multiplicands and 2) the Q notation of the result is equal to the sum of the individual Q notations.

We can state this formally as follows:

$$mQn \times iQj = (m+i)Q(n+j) \tag{10.3}$$

The ARM® Cortex®-M4 processor has two multiply instructions – one giving a 32-bit result, and one giving a 64-bit result. The C compiler will not automatically increase the result length – two 32-bit operands will theoretically give a 64-bit result for multiplication, but the C compiler will use the instruction with a 32-bit result to preserve "type". Even if we could arrange for a 64-bit result (we can with assembly language), we can't multiply the result by another number, because that would involve a 64-bit x 32-bit multiplication which is not directly supported by a 32-bit CPU. We have to emulate what a floating-point unit would do – *normalise*. This means the 64-bit result must be converted back to a 32-bit number that has some arbitrary Q notation. For example, if we wished to convert a result from a 64Q5 number (base 32) to a 32Q3 (base 8) number, we shift it right 2 bits (divide by 4 which is the amount the base has changed), and only keep the lower 32 bits. We should note that in

shifting, we inevitably lose accuracy. This is the price paid for maintaining successive calculation results within 32-bits.

We can see now why Q notation is efficient – normalisation is carried out by shifts which are very quick in terms of CPU time (much quicker than a divide – 1 cycle time for a shift, compared with 2-12 cycles for a divide on the K70).

9.3.2 Division

For division, we similarly have:

$$mQn \div iQj = m\,Q(n-j), \quad m > i \quad n > j \tag{10.4}$$

For example, a 32Q16 number divided by a 16Q8 number results in a 32Q8 quotient.

9.3.3 Addition

Normalisation before addition

Additions must be performed with numbers of the same Q notation. If they are different, then normalisation to the larger base is required. For example, to add a 6Q3 number and a 5Q2 number, we have to shift the 5Q2 number to the left by one to create a 6Q3 number before adding:

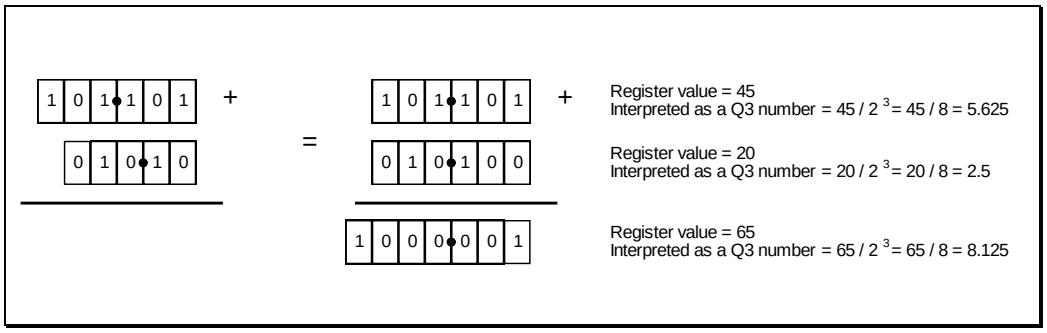


Figure 9.5

9.3.4 Subtraction

Similarly, subtraction requires normalisation of the bases so that the larger base is common.

EXAMPLE 9.1 Fixed-Point Calculations

We will develop the equations that K70 software could need to implement a digital scale. Assume the range of a position measurement system is 0 to 3 m, and the system uses the K70's ADC to perform the measurement. We will assume that the ADC has been put into single-ended 10-bit mode so that the digital output varies from 0 to 1023. Suppose also that the analog input range is 0 to +3.3 V. Let x be the distance to be measured in metres, V_{in} be the analog voltage in volts and N be the 10-bit digital ADC output. Then the equations that relate the variables are:

$$V_{in} = 3.3 * N / 1024 \qquad \text{and} \qquad x = 3\,\text{m} * V_{in} / 3.3\,\text{V}$$

Thus:

$$x = 3 * N / 1024 = 0.0029296875 * N \qquad \text{where } x \text{ is in m}$$

From this equation, we can see that the smallest change in distance that the ADC can detect is about 0.003 m. In other words, the distance must increase or decrease by 0.003 m for the digital output of the ADC to change by at least one number. It would be inappropriate to save the distance as an integer, because the only integers in this range are 0, 1, 2 and 3. To save power, we decide not to use the K70's FPU and therefore the distance data will be saved in fixed-point format. Decimal fixed-point is chosen because the distance data for this distance-meter will be displayed for a human to read. A fixed-point resolution of 0.001 m could be chosen, because it matches the resolution determined by the hardware. The table below shows the performance of the system with the resolution set to 0.001 m. The table shows us that we need to store the fixed-point number in a signed or an unsigned 16-bit variable.

x (m) distance	V_{in} (V) analog input	N ADC input	I internal representation	Approximation `(41 * N + 7) / 14`
0	0.000	0	0	0
0.003	0.003	1	3	3
0.600	0.660	205	600	600
1.500	1.650	512	1500	1499
3.000	3.300	1023	3000	2996

9.10

It is very important to carefully consider the order of operations when performing multiple integer calculations. There are two mistakes that can happen. The first error is *overflow*, and it is easy to detect. Overflow occurs when the result of a calculation exceeds the range of the number system. The following fixed-point calculation, although mathematically correct, has an overflow bug:

```
I = (3000 * N) / 1024;
```

because when `N` is greater than 21, `3000*N` exceeds the range of a 16-bit unsigned integer. If possible, we try to reduce the size of the integers. In this case, an approximate calculation can be performed without overflow

```
I = (41 * N) / 14;
```

You can add one-half of the divisor to the dividend to implement rounding. In this case:

```
I = (41 * N + 7) / 14;
```

The addition of "`7`" has the effect of rounding to the closest integer.

For example, when `N = 4`, the calculation `(41 * 4) / 14 = 11`, whereas the "`(41 * 4 + 7) / 14`" calculation yields the better answer of `12`.

No overflow occurs with this equation using unsigned 16-bit maths, because the maximum value of `41 * N` is 41943. If you cannot rework the problem to eliminate overflow, the best solution is to use promotion. Promotion is the process of performing the operation in a higher precision. For example, in C we cast the input as **`unsigned long`**, and cast the result as **`unsigned short`**:

```
I = (unsigned short)((3000 * (unsigned long)N)/1024);
```

Again, you can add one-half of the divisor to the dividend to implement rounding. In this case:

```
I = (unsigned short)((3000 * (unsigned long)N + 512) / 1024);
```

The other type of error we may experience with fixed-point arithmetic is called *drop out*. Drop out occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. It is very important to divide last when performing multiple integer calculations. If you divided first:

```
I = 41 * (N / 14);
```

then the values of `I` would be only 0, 41, 82, … or 2993.

The display algorithm for the unsigned decimal fixed-point number with 0.001 resolution is simple:

1) display `(I / 1000)` as a single digit value

2) display a decimal point

3) display `(I % 1000)` as a three-digit value

4) display the units “m”

9.3.5 Fixed-Point Operations Using a Universal 32Q16 Notation

Finding the optimum choice of Q notation for a fixed-point variable requires knowing what range of values it will have during execution. When range and resolution requirements are modest, however, a simple approach is to use 32 bits for all fixed-point numbers, with 16 bits in both the whole and the fractional parts, i.e. a 32Q16 notation:

32Q16 notation

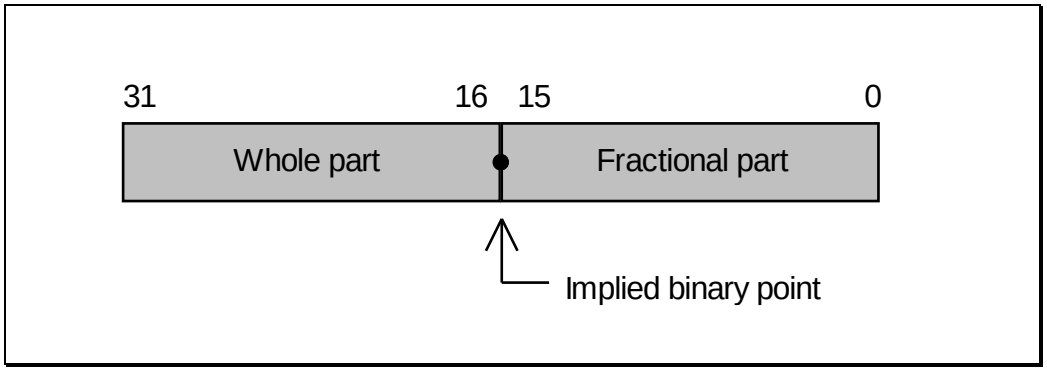


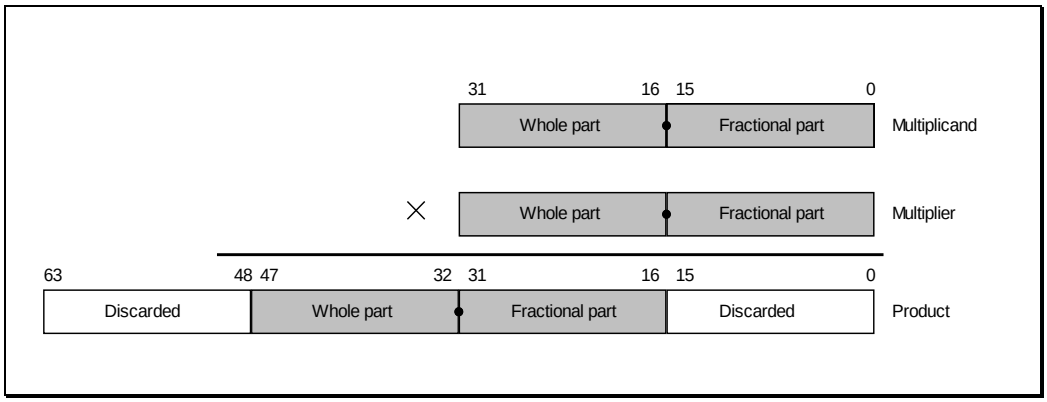
Figure 9.6

This is the method used by Sony (original Playstation) and Nintendo (DS, Gamecube, Gameboy Advance) in their 3D graphics engines to achieve fast processing performance without an FPU.

With all operands using the same notation, addition and subtraction no longer require pre-alignment of operands. Multiplication and division, however, still require *some* adjustment or else the result will not have the same notation as the operands. Remember that when you multiply two fixed-point operands together, their Q notations add:

$$32\text{Q}16 \times 32\text{Q}16 = 64\ \text{Q}32 \tag{10.5}$$

What we need is a product in 32Q16 format. In other words, the integer product needs to be right-shifted by 16 bits. Multiplying the two 32-bit integers produces a 64-bit product, with an implied binary point in the middle. Right-shifting this product by 16 bits and then putting the result back into a 32-bit location means we are discarding 16 bits from each end of the integer product:



Using integer multiplication to produce a fixed-point product

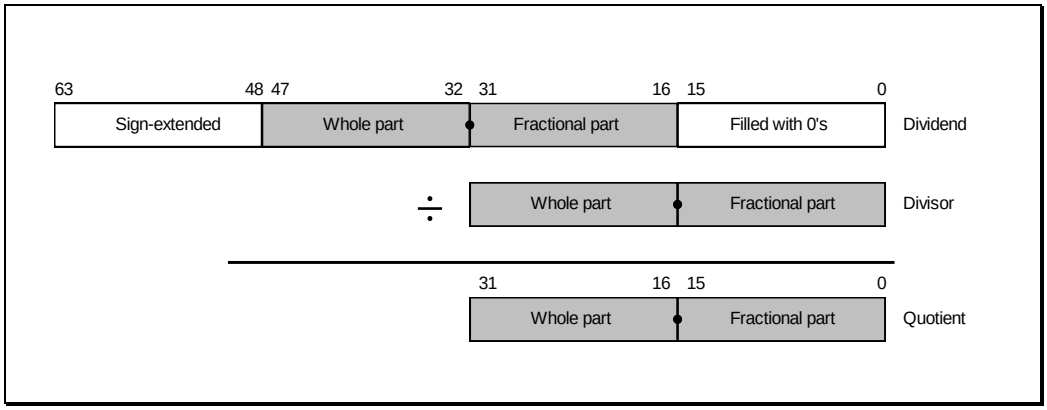
Figure 9.7

Discarding the least significant 16 bits simply causes some loss of precision; discarding the most significant 16 bits requires imposing a maximum magnitude restriction on the operands to avoid overflow.

When you *divide* one 32Q16 fixed-point operand by another, we require the result to be a 32Q16 number. We therefore need a 64Q32 dividend, since:

$$64Q32 \div 32Q16 = 326\ Q16 \tag{10.6}$$

We create a 64Q32 dividend by sign extending the original 32Q16 dividend, and then left-shifting by 16 bits. The division is then done with a 64-bit dividend and a 32-bit divisor, to give a 32-bit quotient:



Using integer division to produce a fixed-point quotient

Figure 9.8

EXAMPLE 9.2 Fixed-Point Calculations Using Universal 32Q16 Notation

We can use fixed-point algorithms to perform complex operations using the integer functions of our K70. For example, consider the following digital filter calculation:

```
y = x - 0.0532672 * x1 + x2 + 0.0506038 * y1 - 0.9025 * y2;
```

In this case, the variables `y`, `y1`, `y2`, `x`, `x1`, and `x2` are all 32Q16 fixed-point integers, and we need to express the constants in 32Q16 fixed-point format. The value -0.0532672 is approximated by $-0.0532672 \times 65536 \approx -3491$. The value 0.0506038 will be approximated by $0.0506038 \times 65536 \approx 3316$. Lastly, the value -0.9025 will be approximated by $-0.9025 \times 65536 \approx -59146$. The fixed-point implementation of this digital filter is:

```
int64_t t1, t2, t3, t4;

t1 =  -3491 * (int64_t)x1;
t2 =   3316 * (int64_t)y1;
t3 = -59146 * (int64_t)y2;
t4 = t1 + t2 - t3;
y = x + x2 + (int32_t)(t4 >> 16);
```

Note that since we are using C types, we need to allocate space for a 64-bit product, and thus the 32-bit integer variables are promoted and sign-extended to 64-bits using a typecast. If we did not do this, then the multiplication of two 32-bit quantities may overflow the 32-bit storage space.

The approximations of the constants using 32Q16 notation may be unsuitable if they do not give us enough resolution. In that case, we have to sacrifice speed and use a different non-power-of-2 base or increase the resolution of the Q notation numbers.

9.4 Square Root Algorithm for a Fixed-Point Processor

The evaluation of the square root of a number using integer arithmetic is a common operation in many embedded systems. For example, in the calculation of RMS quantities, such as voltage and current, a square root is involved. Any time a complex number is used (such as in an FFT), it is convenient to know its magnitude, which involves Pythagoras' Theorem and a square root operation.

To evaluate the square root of a number, we use Newton's method to solve the equation:

$$f(x) = R - x^2 = 0 \tag{10.7}$$

where R is the number whose square root we wish to evaluate. According to a first-order Taylor series approximation of any function, we have:

$$f(x+h) \approx f(x) + hf'(x) \tag{10.8}$$

If we have an estimate of the square root, x_*, then we can use the above formula to determine an h to add to x_*, which will hopefully be a better estimate of the square root:

$$\begin{aligned} f(x_* + h) &= 0 \\ f(x_*) + hf'(x_*) &= 0 \\ h &= \frac{-f(x_*)}{f'(x_*)} \end{aligned} \tag{10.9}$$

This process is then repeated in an iterative manner until a desired accuracy is reached:

$$x_* = x_* + h$$
$$\lim_{n\to\infty} x_* = x \tag{10.10}$$

Applying the above analysis to Eq. (10.7) gives a formula for the new estimate of the square root as:

$$\begin{aligned} x_* &= x_* - \frac{f(x_*)}{f'(x_*)} \\ &= x_* - \frac{R - x_*^2}{-2x_*} \\ &= x_* + \frac{R}{2x_*} - \frac{x_*}{2} \\ &= \frac{x_*}{2} + \frac{R}{2x_*} \\ &= \frac{\left(\frac{R}{x_*} + x_*\right)}{2} \end{aligned} \tag{10.11}$$

This is easily performed in an integer processor and involves only one division, one addition and a shift, which is very efficient.

When calculating an RMS value, we can calculate Eq. (10.11) once every sample time, and use the previous RMS value as the initial estimate. We don't need to iterate more than once since the previous RMS value will always be a good estimate of the current RMS value.

If we understand fixed-point techniques, we can optimize performance

C maths libraries provide square root routines, but when we understand their operation, we can optimise our code for performance.

EXAMPLE 9.3 Magnitude of a Complex Number

The following C function calculates the approximate magnitude of a complex number.

```
// Number of iterations to perform for square-root algorithm
const uint8_t NB_ITERATIONS = 5;

uint16_t Magnitude(int16_t real, int16_t imag)
{
  uint32_t magSquared;
  uint16_t mag;
  uint8_t i;

  magSquared = (uint32_t)((int32_t)real * (int32_t)real +
                          (int32_t)imag * (int32_t)imag);

  // Initial guess = magSquared / 2
  mag = (uint16_t)(magSquared / 2);

  // Estimate magnitude using Newton's method
  for (i = 0; i < NB_ITERATIONS; i++)
    mag = (uint16_t)((magSquared / mag + mag) / 2);

  return mag;
}
```

The function above will return an approximate result since the number of iterations is fixed. This may be acceptable in certain applications – otherwise the error between the square of the current root estimate and the original number to be squared can be used to terminate the iterations.

The function also contains two bugs:

1. The initial estimate of the magnitude may exceed the range of a `uint16_t`.
2. Division by zero is not tested for or handled. The ARM® Cortex®-M4 can produce an exception (usage fault) on division by zero, so we would need to write an exception handler.

Obviously a more robust function would need to handle these sources of potential error.