FFT - fRISCy Fourier transforms?

M R Smith

This is an applications tutorial oriented towards the practical use of the discrete Fourier transform (DFT) implemented via the fast Fourier transform (FFT) algorithm. The DFT plays an important role in many areas of digital signal processing, including linear filtering, convolution and spectral analysis. The first part of the article is a practical industrial example and takes the reader through the thought process an engineer might take as DFT familiarity is gained. If standard software packages did not provide the necessary performance, the engineer would need to port the application to specialized hardware. The second part of the tutorial discusses the theoretical concepts behind the FFT algorithm and a processor architecture suitable for high speed FFT handling. Rather than examining the standard digital signal processors (DSP) in this situation, the final section looks at how the reduced instruction set (RISC) processors perform. The Advanced Micro Devices scalar Am29050 and the super-scalar Intel i860 processors are detailed. Comparison of the DSP and RISC processors is given showing that the more generalized RISC chips do well, although changes in certain aspects of the RISC architecture would provide for considerable improvements in performance.

Keywords: FFT, digital signal processing, RISC, Am29050 processor

Many digital signal processing (DSP) algorithms are now available in application packages. The 'standard' engineer (such as myself) would approach any new DSP problem by starting with a quick glance in the user manual to get a flavour of how to solve the problem and use the package. Next would come testing with a few simple examples to ensure that the concepts were understood. This would be followed by more complex examples where the expected 'results' were known and then the algorithm would be tried on the 'real' data. The success of this approach depends on the expertise and judgement of the engineer. These factors must be tempered by the following descriptions of 'judgement' and 'experts'.

> Good judgement is gained from experience.
> Experience is gained from bad judgement.
> An expert is made of two parts: 'X' - a has-been, and 'spurt' - a drip* under pressure.

This tutorial examines the implementation of a major digital signal processing technique, the discrete Fourier transform (DFT). It is intended to provide a person unfamiliar with using the DFT with the experience to avoid any unnecessary pitfalls. The DFT plays an important role in many areas of digital signal processing, including linear filtering, correlation analysis and spectrum analysis. A major reason for its importance is the existence of efficient algorithms for computing the DFT and a thorough understanding of the problems in its application.

The first part of the article is a practical industrial example involving digital filtering to remove an unwanted signal. This section discusses the importance of data windowing and the trick of deliberate synchronous sampling to avoid problems when using the efficient fast Fourier transform (FFT) algorithm to calculate the DFT. This section would be useful for the engineer intending to use the FFT algorithm from a standard package. (An excellent source of algorithms and their theoretical background can be found in the book by Press *et al.*[1] A diskette with (debugged) source code is available.)

If using a standard package did not provide the required performance, the engineer would need a more thorough understanding of the FFT algorithm and how current processors match the required resources for this algorithm. The second part of this tutorial provides a brief analysis of the fast Fourier transform. The characteristics needed for the efficient implementation of the FFT are discussed in terms of chip architecture in general. The architecture of specialized DSP chips (Texas Instrument

Department of Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada. Email: smith@enel.ucalgary.ca
Paper received: 22 June 1992. Revised: 11 February 1993

*Drip is a colloquialism for idiot.

0141-9331/93/090507-15

TMS32020, 32025 and 32030[†], Motorola DSP56001 and DSP56002[‡] and Analog Devices ADSP-2100 family[§]) and a number of RISC chips (the superscalar Intel i860 and the scalar Advanced Micro Devices Am29050 RISC[¶]) are compared.

Since the FFT performance of DSP processors is well documented in application notes, the final part of the tutorial provides a detailed analysis of the ease (and problems) of implementing DSP algorithms on RISC chips. The RISC performances are compared to those of the DSP processors.

The reason for examining the RISC chip in DSP applications is that many systems already have a high performance RISC as the main engine or a coprocessor. What has to be taken into account to get maximum (DSP) performance from these processors? In addition, there are a number of low-end RISC processors appearing on the market – the Intel RISC processor i960SA and the Advanced Micro Devices RISC Am29240 controller. Although not yet up to the top-end DSP chips in terms of performance, future variants of these chips, based around the same RISC core, may be.

INDUSTRIAL APPLICATION OF THE DFT

In this section, we discuss a simple practical application through the eyes of an engineer gaining greater familiarity with using the DFT. The first part sets the scene of how the data were gathered. The unwanted signal or 'noise' on the data has a particular frequency characteristic. This suggests that if the data were transformed into the frequency domain using a DFT, this 'noise' would appear at one particular location in the frequency spectrum. It could then be 'cut out' (filtered) and the spectrum transformed back into the original data domain. The resulting 'noiseless' data could then be analysed.

The problem

Beta Monitors and Controls Ltd. is a typical small company servicing the oil and gas industry in Alberta, Canada. One particular problem they handle on a daily basis is monitoring the performance of the heavy reciprocating compressors used in the natural gas industry. This analysis requires the determination of the compressor's effective input and exhaust strokes. This is obtained by measuring the 'pressure' as a function of 'crankshaft angle' for a complete stroke. This crankshaft angle is then converted by a non-linear transform to a 'stroke volume'. The compressor's efficiency is determined from the area under the pressure versus volume curve.

Experimental measurement technique

The pressure is obtained by attaching a transducer to the compressor (*Figure 1*). The transducer's output is fed through a low-pass anti-aliasing analogue filter before being digitized by an analogue-to-digital (A/D) converter. The analogue filter is an important part of the measurement system as it removes all frequency components (such as high frequency random noise) greater than half the digital sampling frequency. This ensures that the digitized signal accurately represents the analogue signal being converted[2] and avoids the problems of signal 'aliasing'. Signal aliasing is when one signal appears, on sampling, as another. For example a 7 kHz signal sampled at an 8 kHz rate will be indistinguishable from a 1 kHz signal sampled at the same rate. An unnecessary degradation of the signal-to-noise ratio occurs if high frequency noise is aliased on sampling.

Actual measurements are shown in *Figure* 2. Although

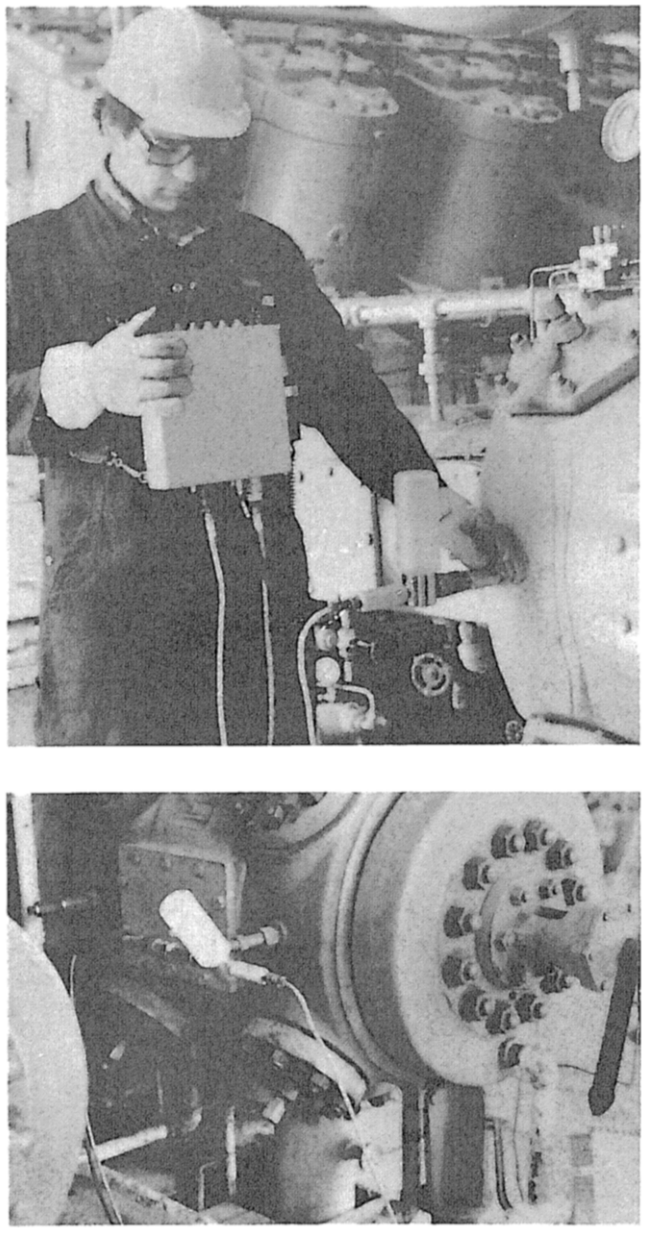

Figure 1 Measurement of the pressure is obtained by attaching a transducer to the heavy compressor cylinder (Figure courtesy of Beta Monitors and Controls Ltd., Calgary, Alberta, Canada)

[†]TMS32020, TMS32025, TMS32030 are trademarks of Texas Instruments Ltd.
[‡]DSP56001 and DSP96002 are trademarks of Motorola Ltd.
[§]ADSP-2100 is a trademark of Analog Devices.
[¶]Am29050 and Am29240 are trademarks of Advanced Micro Devices Ltd.

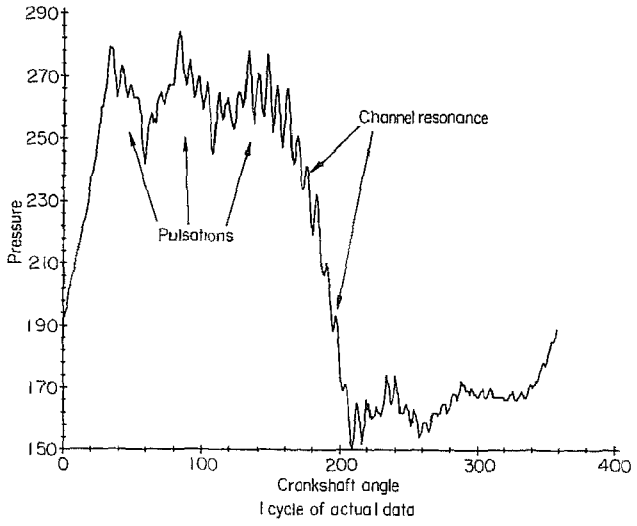

Figure 2 The pressure as a function of the crankshaft angle is measured for a Natural Gas reciprocating compressor. There are data, compressor-related pulsations and unwanted channel noise components

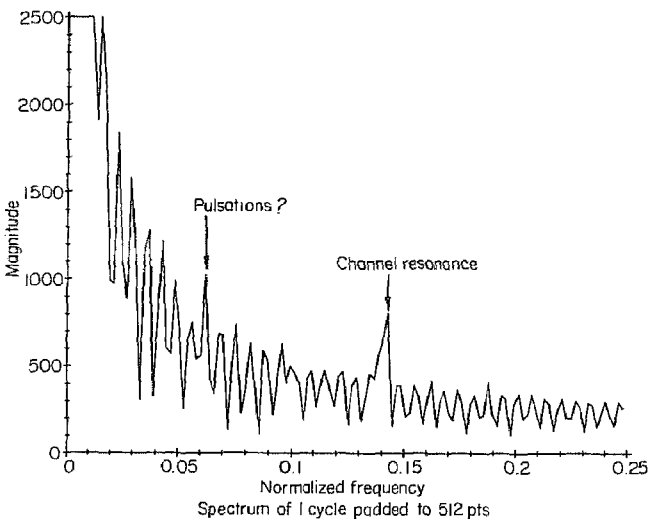

Figure 3 The transform of the data from *Figure 2*. The noise or channel resonance frequency components are indicated

the basic curve is simple and has a high signal-to-noise ratio, the measurements are distorted by important (wanted) low frequency 'compressor related pulsations' and (unwanted) high frequency noise components. The unwanted noise arises from the transducer which is attached to the cylinder via a small channel or pipe (see *Figure 1*). Just as a bottle will whistle if you blow across the top, this channel will resonate during the cylinder stroke, appearing as rapid vibrations during one part of the measured cycle. Even a 2% error in the measurement of the compressor performance can mean under-production (loss of profits) or over-load (premature failure of the compressor). The problem is made more difficult by the non-linear transform from 'angle' to 'volume' which distorts the noise oscillations.

The way to remove this noise is to transform the data using the DFT into the frequency domain where its frequency components can be identified and removed. By inverse transforming the modified spectrum, it should be possible to get the data without the noise component. This data can be converted to 'volume' and analysed as required.

Attempt 1 - Bull-at-the-gate approach

The scientific computer libraries and applications packages typically include an efficient implementation of the DFT, the fast Fourier transform (FFT) algorithm, based on a data length M that is a power of 2 (M = 16, 32, . . . , 512, 1024 etc.). Since the pressure versus angle data have a length of 360 points (1 cycle), it seems appropriate to pad the data with zeros to size 512 and then apply the FFT.

Transforming the original data (*Figure 2*) produces a spectrum (*Figure 3*) with the channel resonance frequency components fairly evident above a background signal. The frequency scale has been normalized (frequency/DFT-points) so that spectra from different sized DFTs can be compared. Large frequency components are displayed so that the smaller components are more easily seen. The noise frequency components can be zeroed (filtered out) for normalized frequencies 0.12 to 0.16, and the modified spectrum transformed back into the data domain for the compressor analysis. It can be seen from the resulting data (*Figure 4*) that the majority of the noise oscillations have been removed by the filtering, but there are now different distortions that were not there before.

There are many books that will explain the problems during this simple filtering[2,3]; however, the following argument outlines the underlying principles. The new distortions can be understood at a number of levels. Because of the finite amount of data, there are discontinuities at its boundaries on padding with zeros*. These 'sharp edges' have frequency components all across the spectrum (the background signal of *Figure 3*). This means that the data is no longer confined to the low frequencies. When the noise frequency components are removed, so is a significant part of the 'spread-out' data components. On inverse transforming, the removed data components mean that the filtered signal will be incorrect, particularly near the discontinuities. In addition, since the noise components were also spread out, they are more difficult to identify and (correctly) remove.

There is also a second, less obvious problem. If discontinuities in the data lead to a range of frequencies in the spectrum, then discontinuities in the spectrum will lead to a range of original data values. When we removed the noise frequency components by setting them to zero, this created discontinuities in the spectrum which can lead to additional distortions in the filtered data. Proper application of the DFT can remove or reduce many of these artifacts.

Attempt 2 - DFT with windowing

The previous section described the problems associated with blindly applying the DFT. The difficulties are deeper

*To make the problem more obvious, the data's DC offset was deliberately increased.

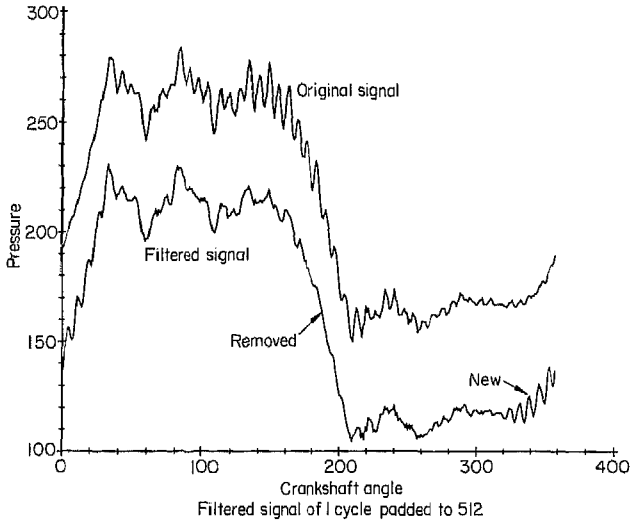

Figure 4 The original data is shown as the upper trace. When the data is padded with zeros before filtering, the resulting filtered signal has new distortions at its edges

and more complicated than what appears on the surface. We 'think' we are trying to transform the signal shown in *Figure* 2. However, when we use the DFT what we are actually trying to do is to transform an infinitely long signal of which we only know a small part. This subtle effect is known as 'windowing' and has a very pronounced effect on a signal's spectrum. If we had an 'infinite' amount of data taken from the compressor, there would be no discontinuities and no distortions introduced when calculating its spectrum.

Figure 5 (top) shows an 'infinitely-long' complex sinusoid and its spectrum, a single spike. *Figure* 5 (middle) shows a 'windowed' sinusoid and its (magnitude) spectrum. It can be seen that the single spike has spread into a wide centre lobe and there are a number of high side-lobes. Every frequency component in the data shown in *Figure* 2 undergoes a similar spreading. The discontinuities associated with the effective 'rectangular' sampling window applied to get the 'finite' data record lead to a wide range of frequency components – 'spectral-leakage'. When we filter the noise components, we will remove some of the spread-out data frequency components. This will produce distortion in the filtered signal when an inverse transform is performed.

Windowing is a fundamental limitation to the DFT. There is no way around it; the best that you can do is to apply different windows to minimize distortion. The secret is to modify the window on your data so that the spectral leakage (side-lobes) of the window is reduced. This means that you will be able to better discern small signals (the channel resonance) in the presence of larger signals (the frequency components associated with the data edges). However, applying the window to reduce the sidelobes must be balanced against the fact that each spectral peak is widened, resulting in a loss of resolution. An excellent paper on the properties of the DFT and windows has been given by Harris[4].

Applying the window to reduce distortions introduces different distortions. By gathering additional data (say 2.5 cycles padded to 1024 points, *Figure* 6) it is possible to minimize the effect of these new distortions. A window with smoothly changing edges is applied to this extended data before calculating the DFT. This window will allow the noise frequency components to be more clearly identified and filtered. The spectrum can then be inverse transformed and the window removed.

Suppose that you have M data points, $x(m)$; $0 \leqslant n \leqslant M$, which you intend to filter. The steps in generating the filtered signal are:

$$x_{\text{windowed}}(m) = x(m) \times w_{\text{window}}(m); \quad 0 \leqslant m \leqslant M \quad (1)$$

$$X(f) = FT[x_{\text{windowed}}(m)]; \quad 0 \leqslant m,\ f < M \quad (2)$$

$$X_{\text{filtered}}(f) = X(f) \times F_{\text{filter}}(f) \quad (3)$$

$$x_{\text{filtered}}(m) = FT^{-1}[X_{\text{filtered}}(f)] \quad (4)$$

$$x_{\text{corrected}}(m) = x_{\text{filtered}}(m)/w_{\text{window}}(m) \quad (5)$$

First the windowed data points ($x_{\text{windowed}}(m)$) are generated from the original data points using one of the

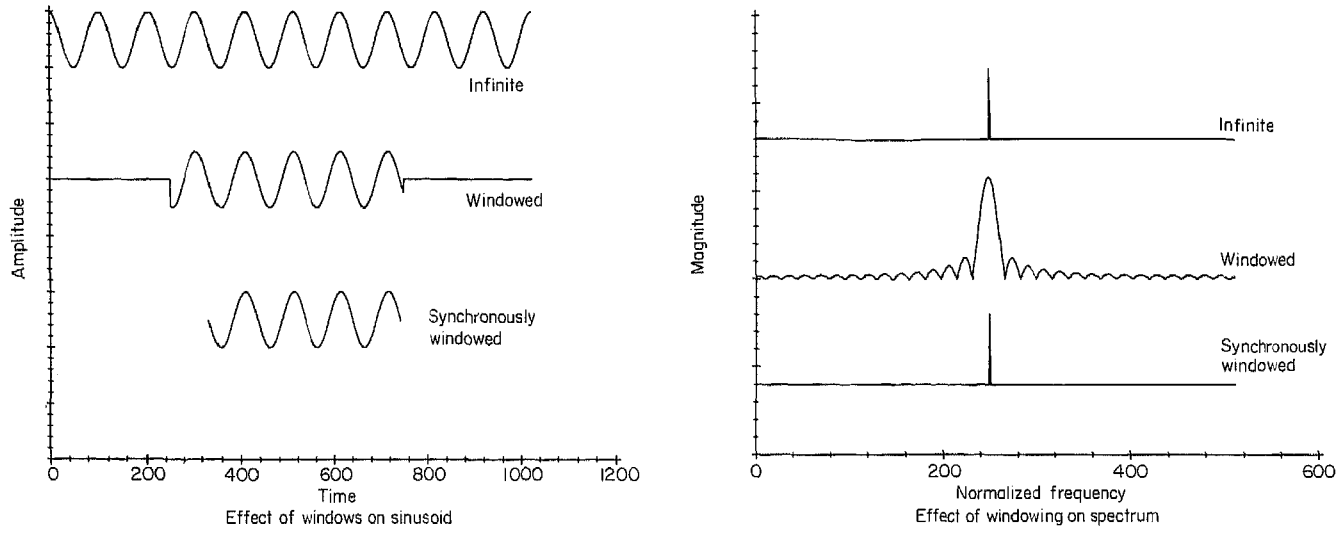

Figure 5 (Top) An 'infinitely-long' sinusoid and its spectrum; (middle) a 'windowed' sinusoid and its spectrum. Note how the windowing produces 'spectral leakage'; (bottom) a 'synchronously sampled' sinusoid. Note how this spectrum appears not to have any 'spectral leakage'

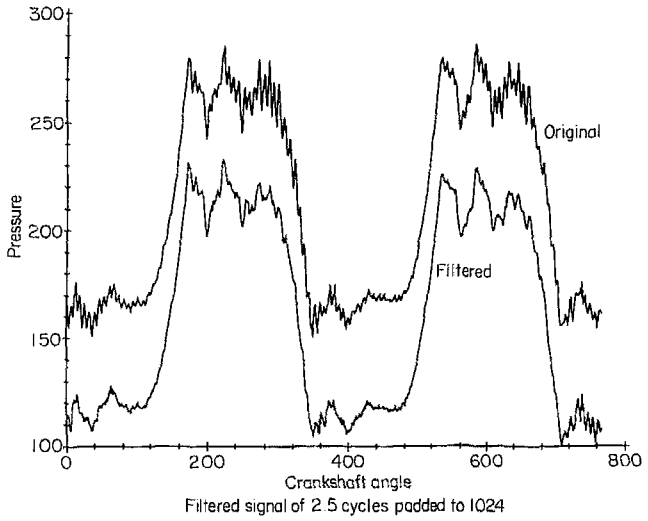

Figure 6 By applying windowing techniques to a number of cycles of data (top) before using the DFT it is possible to generate the filtered signal (bottom). By throwing away the distorted ends, the analysis can be performed on the undistorted centre cycle

filter window shapes ($w_{window}(m)$) suggested by Harris. The windowed data is then transformed ($FT[]$) into the frequency domain ($X(f)$), where the unwanted noise components are removed using a band-stop filter ($F_{filter}(f)$). The filtered frequency domain signal ($X_{filtered}(f)$) is inverse transformed ($FT^{-1}[]$) back to the data domain ($x_{filtered}(m)$) where the original window is removed to produce the required signal ($x_{corrected}(m)$).

There are a number of popular windows, chosen because they are simple to remember and because applying any window is often better than none. A simple window is given by:

$$F(m) = a_0 + a_1 \cos\left(\frac{2\pi}{M} m\right); \quad 0 \leqslant m < M \tag{1}$$

where

$$a_0 = 0.54; \quad a_1 = -0.46 \tag{2}$$

I prefer to use a slightly more complex window known as the 'Blackmann-Harris 3 term window':

$$F(m) = a_0 + a_1 \cos\left(\frac{2\pi}{M} m\right) + a_2 \cos\left(\frac{2\pi}{M} 2m\right); \quad 0 \leqslant m < M \tag{3}$$

where

$$a_0 = 0.44959; \quad a_1 = -0.49364; \quad a_2 = 0.05677 \tag{4}$$

which was designed to have a reasonable main lobe width and minimum side-lobe height.

It should also be remembered that it is often important to filter out the noise frequency components rather than zeroing them out, again to avoid discontinuities. Suppose that the spectrum $X(f)$ has been evaluated using M points and that the noise frequency components are centred around location P with a bandwidth of B, then a suitable bandstop filter with which to multiply the spectrum would be:

$$F(f) = 1; \quad 0 \leqslant f < P - B/2$$

$$F(f) = 1 - a_0 - a_1 \cos\left(\frac{2\pi}{B}(f - (B + P)/2)\right) - a_2 \cos\left(\frac{2\pi}{B} 2(f - (B + P)/2)\right); \quad P - B/2 \leqslant f < P + B/2$$

$$F(f) = 1; \quad P + B/2 \leqslant f < M$$

Note that for most data the noise will have components at two locations (P and at $N - P$), because of the way the DFT generates the data spectrum, so that two bandstop filters must be applied. There is also some reasonable argument to recommend (smoothly) removing all the frequency components $P - B/2 \leqslant n < N - P + B/2$ as these will mainly contain unwanted random noise. However, if there are some valid high-frequency components present in the data, then removing them will degrade any sharp edges actually present in the data.

Figure 7 shows the un-windowed spectrum for 2.5 cycles of data padded with zero out to 1024 points. The spreading of the DC components is very evident. By comparison, in the widowed data spectrum, the channel resonance and pulsations become very clear as the side-lobes are removed. When the noise is removed, the modified spectrum inverse transformed and the window removed, the filtered signal shown in *Figure* 6 is obtained.

It might be assumed that multiplying by a window $w_{window}(m)$ and later dividing by the same window, cancels out the effect of the window. This is not the case because the frequency domain filtering changes the window so that the division and multiplication effects no longer cancel. Filtering in the frequency domain is equivalent to performing a convolution on the data. The simple 'non-widowed' filtering can be expressed as:

$$x_{simple}(m) = \sum_{v} x(v)\, f_{bandstop}(m - v) \tag{5}$$

where $f_{bandstop}(m)$ is the (inverse) discrete Fourier transform of the frequency domain notch filter. Windowing,

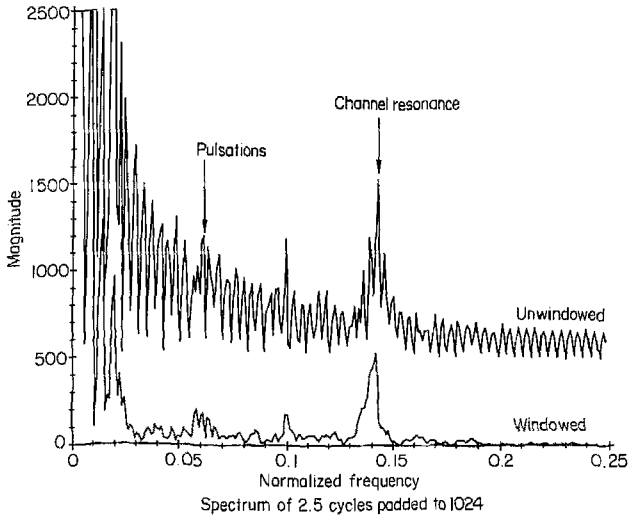

Figure 7 The upper spectrum is from 2.5 cycles of data padded to 1024 points. Note the large background because of 'spectral leakage' from the main data components. The lower spectrum is obtained by windowing the data before filtering

applying the notch filter, and inverse windowing is equivalent to:

$$x_{corrected}(m) =$$

$$\sum_{v} x(v)\, w_{window}(v)\, f_{bandstop}(m - v)/w_{window}(m) \quad (6)$$

which is not the same expression. Since $1/w_{window}(m)$ is very large near the edge of the window, the division operation will amplify any noise on the data, leading to possible distortion near the data edges. This means that it is necessary to use a number of cycles of the data and 'throw away' the parts (near the edges) that remain distorted.

Attempt 3 - DFT with deliberate synchronous sampling

When Beta Monitors discussed with me their filtering problems, they had already empirically attempted the approaches suggested in Attempts 1 and 2. However, they wanted more. They wanted to remove the windowing problems but, at the same time, reduce the number of points measured. Normally, this is not possible as a fundamental DFT limitation is the window effects of some form or another. You must simply decide which of the distortions (resolution loss or side-lobes) you can best live with in practice.

One way of reducing the effect of the window can be taken for data of the form obtained by Beta Monitors. By removing the large DC offset and shifting the start of the sampling, it can be seen that the signal is 'naturally' windowed with very few edge discontinuities (*Figure 8*). Filtering the spectrum produced by the naturally windowed signal is also shown in *Figure 8*. Note that there are edge effects still present, but they are less evident.

Most of the time, the data cannot be naturally windowed and you must live with the effects. However, there is one very special and unusual situation where something can be done. Fortunately for Beta Monitors, their data could be manipulated into the required format. *Figure 5* (lower) shows a 'windowed' sine wave and its spectrum obtained by using a DFT. This spectrum appears to contradict all that was said in the previous section. Where are the side-lobes from the window?

When you are applying the DFT you do not calculate the continuous spectrum as suggested in Figure 5 (top and middle). Instead you determine only certain parts of that continuous spectrum. The 'windowing' in *Figure 5* (bottom) has been chosen so that a whole number of cycles of the sinusoid are included in the sampling period, called synchronous sampling. This has the effect that when you apply the DFT you only sample the 'spectral leakage' at the central maximum and at all places where the 'leakage' is zero (*Figure 9*). This means that if you can achieve synchronous sampling, then it as if there was not any leakage when using the DFT.

True synchronous sampling of all components of your data is not something that can be readily achieved. It is normally only done by mistake when students choose a poor example for use with their spectral analysis programs in DSP courses. By accidentally synchronously sampling, their algorithms will appear to perform much better than they would in real life. However, in the case of Beta Monitor's data, both the data and the noise were repeatable every 360 points as the fixed speed compressor made one rotation. By performing a 720 point DFT rather than a standard 1024 point DFT, it was possible to achieve synchronous sampling and generate the spectrum shown in *Figure 10*. The spectral components just jump out at you. It is now very easy to identify and remove the noise frequency components without disturbing the main data components and achieve 'perfect filtering'. After inverse transforming, the signal shown in *Figure 11* was obtained. A further improvement in the signal could be obtained by adding two cycles to average out the effect of random noise.

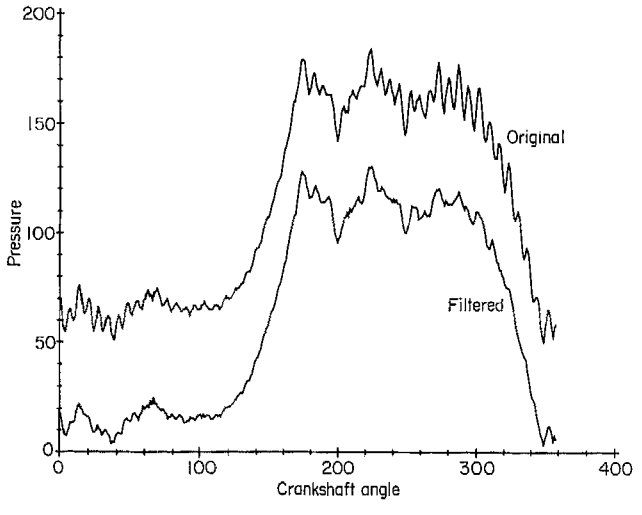

Figure 8 Less distortion arises from filtering a 'naturally widowed' signal obtained by removing the zero offset and adjusting the position of the signal in the window

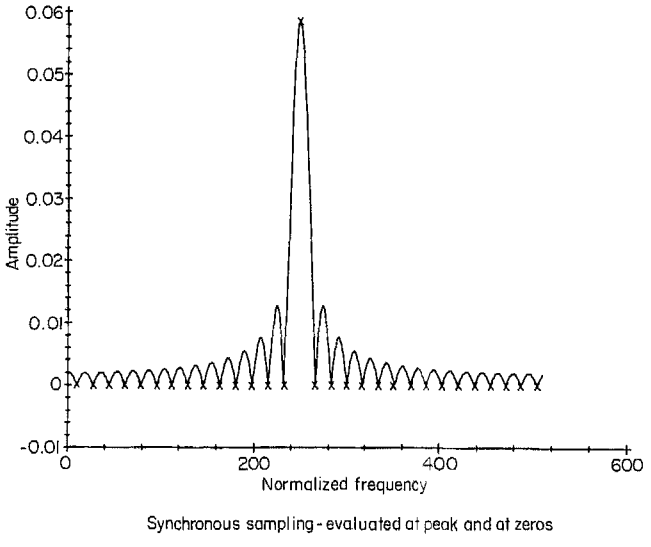

Figure 9 When windowing a synchronously sampled signal, the wide main lobe and the side-lobes are in fact present. However the 'spectral leakage' signal is sampled only at the centre of the main lobe and at the zero-crossing points between the side-lobes

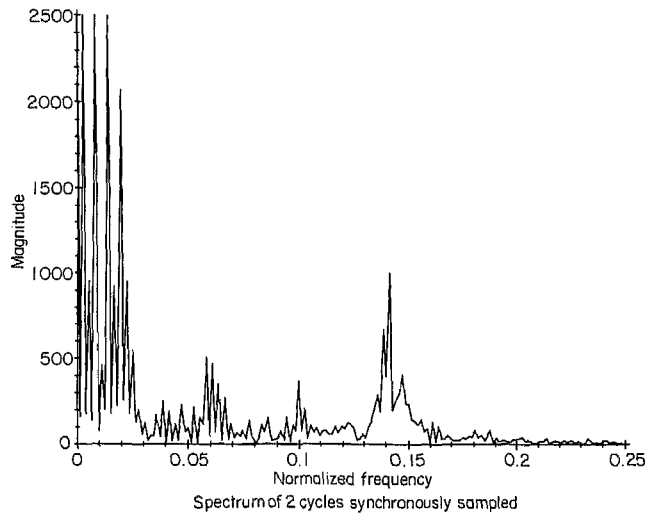

Figure 10 *The spectrum from a 720 point 'synchronously sampled'* data set. Note the sharp spectral spikes

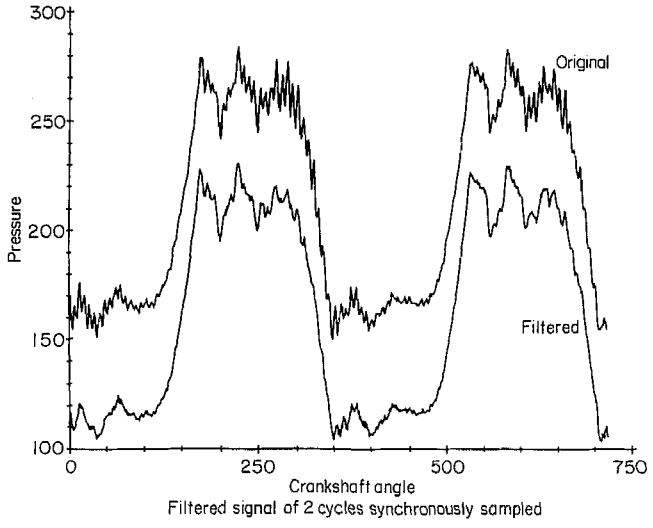

Figure 11 This filtered data set was achieved by using a 720 point DFT on two data cycles

THEORY BEHIND THE FFT ALGORITHM

Before moving on to detail a suitable processor upon which to implement the DFT efficiently, we need to determine what the processor must handle. The basic computational problem for the DFT is to compute the (complex number) spectrum $X(f)$; $0 \leqslant f < M$, given the input sequence $x(m)$; $0 \leqslant m < M$ according to the formula:

$$X(f) = \sum_{m=0}^{M-1} x(m) W_M^{fm}, \quad 0 \leqslant f < M \tag{7}$$

where

$$W_M^{fm} = e^{-j2\pi fm/M} = \cos(2\pi fm/M) - j\sin(2\pi fm/M) \tag{8}$$

In general the data sequence $x(m)$ may also be complex valued. The inverse DFT is given by:

$$x(m) = \frac{1}{M} \sum_{f=0}^{M-1} X(f) W_M^{-mf}, \quad 0 \leqslant m < M \tag{9}$$

Since these equations are basically equivalent, we shall discuss only the DFT implementation.

A number of steps can be taken to speed the direct implementation of the DFT. First the coefficients W_M^{fm} can be pre-calculated and stored for reuse (in ROM or RAM). The calculation time for all the sine and cosine values can take almost as long as the DFT calculation itself. If the input values $x(m)$ are real, a further time saving of 2 can be made using the DFT since half the spectrum can be derived from the other half rather than being calculated.

$$X(f) = a_f + jb_f \qquad 0 \leqslant f < M/2$$
$$X(M-f) = a_f - jb_f \qquad 0 \leqslant f < M/2$$

Despite all these 'special' fixes, basically the direct implementation of the DFT is a real number-cruncher. For each value of f, we require M complex multiplications ($4M$ real multiplications) and $M-1$ complex additions ($4M-2$ real additions). This means that to compute all M values of the DFT requires M^2 complex multiplications and $M^2 - M$ complex additions. Take $M = 1024$ as a realistic data size and you have over 4 million multiplications and 4 million additions, which is a lot of CPU time, even with the current high-speed processors.

There are a number of ways around this problem, based on a divide and conquer concept which leads to a group of FFT algorithms. One such algorithm is the M point decimation-in-frequency* radix 2 FFT.

Consider computing the DFT of the data sequence $x(m)$ with $M = 2^r$ points. These data can be split into odd and even series:

$$g_{even}(m) = x(2m)$$
$$g_{odd}(m) = x(2m+1); \quad 0 \leqslant m\, M/2$$

These series can be used in calculating the DFT of $x(m)$.

$$X(f) = \sum_{m=0}^{M-1} x(m) W_M^{fm}, \quad 0 \leqslant f < M$$

$$= \sum_{m\ \text{even}} x(m) W_M^{fm} + \sum_{m\ \text{odd}} x(m) W_M^{fm}$$

$$= \sum_{n=0}^{(M/2)-1} x(2n) W_M^{2nf} + \sum_{n=0}^{(M/2)-1} x(2n+1) W_M^{f(2n+1)}$$

$$= \sum_{n=0}^{(M/2)-1} g_{even}(n) W_{M/2}^{fm} + W_M^f \sum_{n=0}^{(M/2)-1} g_{odd}(n) W_{M/2}^{fn}$$

$$= G_{even}(f) + W_M^f G_{odd}(f)$$

*DSP and gratuitous violence - the word 'decimate' comes from the ancient Roman method of punishing mutinous legions of soldiers by lining them up and killing every 10th soldier.

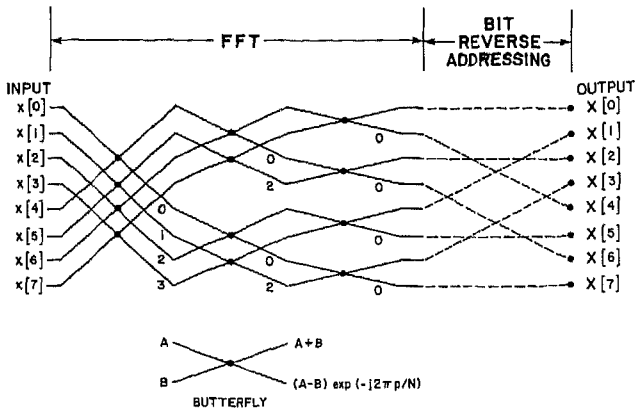

Figure 12 The flow chart for an eight point Radix 2 FFT algorithm

where we have used the property that $W_M^2 = W_{M/2}$. This analysis means that the M-point DFT $X(f)$ can be calculated from the $M/2$-point DFTs $G_{even}(f)$ and $G_{odd}(f)$. This does not seem much until you realize that $g_{even}(m)$ and $g_{odd}(m)$ can also be broken up into their odd and even components, and these components into theirs. This breaking up and calculating a higher DFT from a set of other DFTs is demonstrated in *Figure 12* for an eight-point DFT.

The advantage of this approach can be seen by the fact that calculating an M-point DFT from known $M/2$-point DFTs requires only $M/2$ additional complex computations.

$$X(f) = G_{even}(f) + W_M^f G_{odd}(f); \quad 0 \leqslant f < M/2$$

$$X(f + M/2) = G_{even}(f) - W_M^f G_{odd}(f)$$

This operation is known as an FFT butterfly and, as can be seen from *Figure 12*, forms the basis of the FFT calculation. Calculating the $M/2$-point $G_{even}(f)$ and $G_{odd}(f)$ from their $M/4$-point components takes a similar number of multiplications. Thus by using this divide and conquer approach, it is possible to calculate an M-point DFT using $(M/2)\log_2 M$ complex multiplications rather than the M^2 required for the direct method.

The time saving that this new approach provides is enormous as can be seen from *Table 1*, which compares the number of complex multiplications for the direct and Radix 2 DFT algorithms.

The speed improvements rapidly increase as the number of points increases. The C-code for implementing this Radix 2 algorithm is given in *Figure 13* (modified from Reference 2). Breaking up the data into four components (Radix 4) also provides some speed improvement, which for 1024 points gives an additional 30% advantage.

The divide-and-conquer approach is most often used for data numbers that are a power of 2. However, the approach is more general. For example, the 720 point FFT needed for the Beta Monitor data discussed in the last section might be obtained by decimating the data into four groups of two, two groups of three and one group of five*. For further information on Radix 2, 4, 8, split radix or 720 point FFT algorithms see References 2 and 3.

As can be seen from *Figure 2*, a disadvantage to this approach to the FFT algorithm is that the results are stored at a location whose address is 'bit-reversed' to the correct address. Thus the data value $X(\%011)$ is stored at location %110. This requires a final pass through the data to sort them into the correct order. As will be shown in the final part of this tutorial, this plays an important role in the efficient implementation of the FFT on RISC chips.

Table 1 Comparison of complex operations required for direct and radix 2 DFT algorithms

Points	Direct	Radix 2	Speed improvement
4	16	4	400%
32	1024	80	1280%
128	16384	448	2130%
1024	1048576	5120	20488%

PROCESSOR ARCHITECTURE FOR THE FFT ALGORITHM

A recent article[5] discusses in detail how DSP and RISC chips handle various DSP algorithms. This article points out that although the FFT is a specialized algorithm, it makes a fairly good test-bed for investigating the architecture required in DSP applications. Examining the Radix 2 'C-code' shown in *Figure 13* provides a good indication of what the processor should handle in an efficient way:

- The algorithm is multiplication and addition intensive.
- The precision should be high to avoid round-off errors as values are reused.
- There are many accesses to memory. These accesses should not compete with instruction access to avoid bottlenecks.
- The algorithm uses complex arithmetic.

```
/*******************
        xr -- array of real part of data
        xi -- array of imag part of data
        wr - array of precalculated cosine values for n points
        wi - array of precalculated sine values for n points
        m = log (n)
              2
                                        *************************/
DoFFT(xr, xi, wr, wi, n, m)
float xr[], xi[], wr[], wi[];
int n, m;
{
        int i, j, k, m, inc, ie, ia, n1, n2;
        float xrt, xit, c, s;

        n2 = n;
        for (k = 0; k < m; k++) {       /* outer-loop */
                n1 = n2;        n2 = n2 / 2;    ie = n / n1;
                ia = 1;

                for (j = 0; j < n2; j++) {
                                /* sine and cosine values */
                        c = wr[ia];     s = wi[ia];
                        ia = ia + ie;   /* next address offset */

                        for (i = j; i < n; i += n1) {
                                m = i + n2;                     /* offset */

                                xrt = xr[i] - xr[m];            /* common */
                                xit = xi[i] -xi[m];

                                xr[i] += xr[m];                 /* upper */
                                xi[i] += xi[m];

                                xr[m] = c * xrt + s * xit;      /* lower */
                                xi[m] = c * xit - s * xrt;

                        }
                }
        }
}
```

Figure 13 The 'C-code' for a simple three-loop non-custom N point Radix 2 FFT algorithm

*On the basis of 'if it ain't broke, don't fix it', if I had a very fast processor and only had to do a 720 point DFT a very few times, I would be tempted to simply code a straight DFT. That was the approach I took for this paper. I also forgot to turn on the maths coprocessor and was really reminded just how slow a 'slow DFT' algorithm is.

- The algorithm has a number of loops, which should not cause 'dead-time' in the calculations.
- There are many address calculations, which should not compete with the data calculations for the use of the arithmetic processor unit (APU) or the floating point unit (FPU).
- There are a number of values that are reused. Rather than storing these out to a slower external memory, it should be possible to store these values in fast on-board registers (or data cache).
- There are fixed coefficients associated with the algorithm.
- Speed, speed and more speed.

The DSP processor is designed with this sort of problem in mind – all the resources needed are provided on the chip. Typically, DSP processors run at one instruction every two clock cycles. In that time, they might perform an arithmetic operation, read and store values to memory and calculate various memory addresses. By comparison, RISC chips are more highly pipelined and can complete one instruction every clock cycle. When there is an equivalent instruction (for example the highly pipelined multiply and accumulate instruction or a basic add) this gives the RISC processor the edge. It loses the edge when many RISC instructions are needed to imitate a complex DSP instruction. Depending on the algorithm and the architecture of the particular chips, the DSP and RISC processors come out fairly even in DSP applications[5].

None of the processors on the market has true 'complex arithmetic' capability with data paths and ALUs duplicated to simultaneously handle the real and imaginary data values. Since complex arithmetic is not that common, adding the additional resources to a single chip is not worthwhile. This is the realm of expensive custom chip fabrication, microprogrammable DSP parts[6] or multi-processor systems.

Many DSP applications have extensive looping. This can be handled by hardware zero overhead loop(s) (Texas Instruments TMS320 DSP family and Motorola DSP96002). On RISC processors, the faster instruction cycle, the delayed branch and unfolding loops (straight line coding) remove the majority of the delay problems with using branches. This is particularly true for algorithms, such as the FFT, where the loops are long.

Nor is there significant difference in the available precision on the DSP and the RISC processors. For example, the DSP56001 has a 24 bit wide on-chip memory but uses 56 bits for 'sum-of-products' operations to avoid loss of precision. The i860 and Am29050 processors have 32 bit wide data buses and can use 64 bits for single cycle sum-of-products operations. Many of the RISC and DSP chips now come with floating point capability at no time penalty. Although not strictly necessary, the availability of floating point makes DSP algorithms easier to develop and can provide improved accuracy in many applications.

There is one area in which the DSP chips appear to have a significant advantage, and that is in the area of address calculation. The FFT algorithm requires 'bit reversal' addressing to correct the positions of the data in memory, a standard DSP processor mode. This mode must be implemented in software on the RISC chips. However, as was pointed out to me at a recent DSP seminar*, it is possible to avoid the overhead for bit-reverse addressing by modifying the FFT algorithm so that it does not do the calculation in place.

Auto-incrementing addressing modes are also standard as part of the longer DSP processor instruction cycle. On the Am29050 processor this must be done in software (at the faster instruction cycle) or by taking advantage of this processor's ability to bring in bursts of data from memory (see the section on 'Efficient handling of the RISC's pipelined FPU'). The super-scalar i860 RISC is almost a CISC chip in some of its available addressing modes.

When comparing the capabilities of RISC and DSP processors it is important to consider the possibility that the processor is about to run out of resources. For example the TMS32025 has sufficient data resources on board to perform a 256 point complex FFT on-chip in 1.8 ms with a 50 MHz clock. The available resources are insufficient for 1024 complex points, which takes 15 ms rather than the 9 ms if the 256 point timing is simply scaled[7]. The various processors have different break points depending on the algorithm chosen[5]. The evaluation is, however, difficult because of the different parallel operations that are possible on the various chips, some of the time.

There are as many solutions to the problem of instruction/data fetch bus clashes as there are processors. The DSP chips have on-chip memory while the RISC chips have caches (i860), Harvard architecture and large register files (Am29050). However, it is often possible to find a (practical) number of points that will cause any particular processor to run out of resources and reintroduce bus clashes.

Many DSP chips conveniently have the FFT coefficients (up to size $M = 256$ or 1024) stored 'on-chip'. By contrast, the RISC chips must fetch these coefficients from main memory (ROM or RAM). Provided the fetching of these coefficients can be done efficiently, there is no speed penalty. Again there are problems of 'running out of resources' if the appropriate number of points being processed is sufficiently large on either type of processor.

Changes in the architecture of RISC and DSP chips can be expected in the next couple of years as the fabricators respond to the market and what they perceive as advantages present in other chips. For example the TMS32030 can perform a 1024 complex FFT in 3.23 ms with a 33 MHz clock[8] compared to 1.58 ms with a 40 MHz clock for a DSP96002[9]. The advantage of the DSP96002 is not just in clock speed. It has the capability of simultaneously performing a floating point add and a subtract on two registers. This is particularly appropriate for the FFT algorithm, which is made up of many such operations. The advantage that it gives the DSP96002 can be seen from the fact that it performs an FFT butterfly in four instructions compared to 7–30 instructions on the other RISC and DSP processors. With this sort of advantage, can it be long before the same feature is seen on other chips?

The FFT implementation on the DSP processors is well documented in the data books and will not be further discussed here. In the next section, we shall examine in

*Analog Devices mini-DSP seminar, Calgary, January 1993.

some detail the less familiar problems associated with efficiently implementing the FFT on RISC chips.

EFFICIENT FFT IMPLEMENTATION ON THE RISC PROCESSORS

One of the reasons that DSP processors perform so well in DSP applications is that full use is made of their resources. To get maximum performance out of a RISC processor a similar programming technique must be taken. Although in due time, good 'DSP-intelligent' C compilers will become available for RISC processors, best DSP performance is currently obtained in Assembler by a programmer familiar with the chip's architecture.

In terms of available instructions there is little major difference between the scalar and super-scalar chip. Both the scalar Am29050 and super-scalar i860 RISCs have an integer and a floating point pipeline that can operate in parallel. The major advantage of super-scalar chips is that they can initiate (get started) both a floating point and an integer operation at the same time. However, other architectural features, such as the large floating point register window on the Am29050 processor, can sometimes be used to avoid the necessity of issuing dual instructions. The relative advantages depends on the DSP algorithm. For example, in a real-time FIR application, the scalar Am29050 processor outperformed the super-scalar i860, and both RISCs outperformed the DSP chips[11].

The practical considerations of using RISC chips for the DFT algorithm could equally be explained using the i860 and the Am29050 processors. However, for a person unfamiliar with FPU pipelining and RISC assembler language code, the Am29050 processor has the more user-friendly assembly language[12] and is the easier to understand. The information on the i860 performance is based on References 13 and 14. The Am29050 processor FFT results are based on my own research in modelling and spectral analysis in medical imaging[6,15–17] and the use of the Am29050 processor in a fourth year computer engineering course on comparative architecture which discusses CISC, DSP and RISC architectures.

Efficient handling of the RISC's pipelined FPU

The basic reasons that DSP and RISC chips perform so well are associated with the pipelining of all the important resources. However, there is an old saying 'you don't get something for nothing'. This FPU speed is available if and only if the pipeline can be kept full. If you cannot tailor your algorithm to keep the pipelines full, then you do not get the speed. A 95 tap finite impulse filter (FIR) DSP application[11] is basically 95 multiply-and-accumulate instructions one after the other and is fairly easy to custom program for maximum performance. The FFT algorithm on the other hand is a miscellaneous miss-modge of floating point add (FADD), multiplication (FMUL), LOAD and STORE instructions, which is far more difficult to code efficiently.

The problems can be demonstrated by considering a butterfly from a decimation-in-time FFT algorithm. Similar problems will arise from the decimation-in-frequency FFT discussed earlier.

The butterfly is given by:

$$C = A + WB$$

$$D = A - WB$$

which must be split into the real and imaginary components before being processed:

$$C_{re} = A_{re} + W_{re}B_{re} - W_{im}B_{im} \quad (10)$$

$$C_{im} = A_{im} + W_{re}B_{im} + W_{im}B_{re}$$

$$D_{re} = A_{re} + W_{re}B_{re} + W_{im}B_{im} \quad (11)$$

$$D_{im} = A_{im} - W_{re}B_{im} - W_{im}B_{re}$$

The values A and B are the complex input values to the butterfly and W the complex Fourier multiplier. The output values C and D must be calculated and then stored at the same memory address from which A and B were obtained.

For the sake of initial simplicity we shall assume that all the components of A, B and W are present in the RISC registers. The Am29050 processor has 192 general-purpose registers available and, more importantly, directly wired to the pipelined FPU. The problems of efficiently getting the information into those registers will be discussed later.

At first glance, calculation of the butterfly requires a total of eight multiplications and eight additions/subtractions. Since the Am29050 processor is capable of initiating and completing a floating point operation every cycle, it appears that the FFT butterfly would take 16 FPU operations and therefore 16 cycles to complete. However by rearranging the butterfly terms the number of instructions can be reduced to 10 instructions.

$$Tmp_{re} = (W_{re} * B_{re}) - (W_{im} * B_{im})$$

$$Tmp_{im} = (W_{re} * B_{im}) + (W_{im} * B_{re})$$

$$C_{re} = A_{re} + Tmp_{re}$$

$$D_{re} = A_{re} - Tmp_{re}$$

$$C_{im} = A_{im} + Tmp_{im}$$

$$D_{im} = A_{im} - Tmp_{im}$$

which can be implemented in Am29050 RISC code as:

```
FMUL TR, WR, BR  ; Tmp_re = W_re *B_re
FMUL T1, WI, BI  ; T1 = W_im *B_im
FMUL TI, WR, BI  ; Tmp_im = W_re *B_im
FMUL T2, WI, BI  ; T2 = W_im *B_re
FSUB TR, TR, T1  ; Tmp_re -= T1
FADD TI, TI, T2  ; Tmp_im += T2
FADD CR, AR, TR  ; C_re = A_re + Tmp_re
FSUB DR, AR, TR  ; D_re = A_re - Tmp_re
FADD CI, AI, TI  ; C_im = A_im + Tmp_im
FSUB DI, AI, TI  ; D_im = A_im - Tmp_im
```

Figure 14 shows the floating point unit architecture of the Am29050 processor. The multiplier unit is made up of a three-stage pipeline (MT, PS and RU). The adder unit is also a three-stage pipeline (DN, AD and RU). *Figure 15* shows the passage through the Am29050 FPU of the register values used in the butterfly (based on staging information provided in Reference 12). The fact that the two pipelines overlap and that the steps are interdependent means the butterfly is a mixture of very efficient pipeline usage intermingled with stalls. These stalls are completely

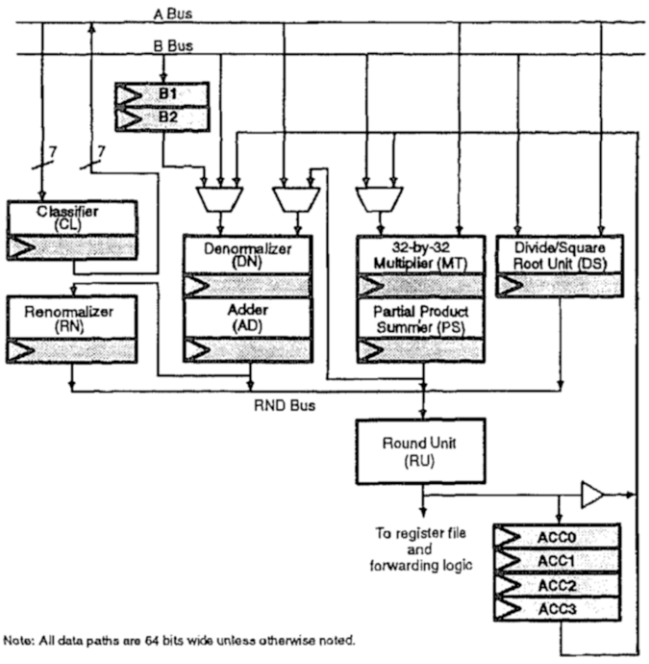

Figure 14 The floating point unit architecture of the Am29050 processor. © 1991 Advanced Micro Devices, Inc*

transparent to the programmer, but that does not make the algorithm execute any faster. The 10 FPU instructions take 15 cycles to complete. However, it is possible to fill the stall positions with address calculations, memory fetches or additional FPU instructions from another butterfly. Although this example was for the Am29050 processor, a similar analysis holds for the Motorola scalar MC88100 RISC processor†. The problem is slightly more complicated for the MC88100 as the pipelines are deeper (four and five steps) and there are only 32 registers.

By comparison with the 192 registers on the Am29050 processor, the i860 has only a few floating point registers (32). However, for the FFT algorithm, the dual instruction capability of this processor allows integer operations (such as memory fetches) to be peformed in parallel with FPU operations. Does this give the super-scalar processor an advantage over the scalar RISC processor? The general answer is a definite 'maybe' for many algorithms as it is not always possible to find suitable parallel operations. However, the FFT can be performed with great advantage since the super-scalar dual instruction capability allows floating point calculations to be moved into the stalls with the integer operations (memory moves) occurring for 'free'. In addition, although the Am29050 processor has some capability of simultaneously performing FADD and FMULT instructions, it does not have the depth of instructions available on the super-scalar i860. Using all the i860 resources to overlap integer/floating point and FADD/FMULT operations, a custom (overlapped) FFT butterfly effectively takes seven cycles[13].

Efficient management of the RISC's registers

The previous section showed how to efficiently handle the Am29050 RISC FPU based on the assumption that the necessary values for the FFT butterflies were stored in the on-board registers. While the super-scalar i860 can bring in four 32-bit registers at a time from the (limited) data cache in parallel with floating point operations, this is not the situation for the Am29050 and MC88100 processors. The i960SA and Am29240 processor variants suggested in the introduction for low end embedded DSP applications are also scalar rather than super-scalar. The following discussion explains how to handle the situation on the Am29050 processor.

Consider again the butterfly equations (10) and (11). Scalar RISC chips have essentially very simple memory access instructions. The Am29050 processor has the capability of a LOAD or a STORE of a register value using the address stored in another register. For simplicity, we shall assume that these instructions can be written as:

```
LOAD RegValue, RegAddress ; RegValue = Memory[RegAddress]
STORE RegValue, RegAddress ; Memory[RegAddress] = RegValue
```

The actual Am29050 syntax is a little more complex as the LOAD and STORE instructions are also used to

*Advanced Micro Devices reserves the right to make changes in its products without notice in order to improve design or performance characteristics. This publication neither states nor implies any warranty of any kind, including but not limited to implied warranties of merchantability or fitness for a particular application.
†MC88100 is a registered trademark of Motorola Ltd.

```
fpuflow.fig        Wed Jan 27 09:39:09 1993        1

INSTR          FPU BUSES                INTERNAL FPU REGISTERS                    LOCAL REGISTERS

               BusA    BusB     MT      PS      DN      AD      RU       DEST     Tr      Ti      T1      T2
               -------- -------- ------  ------  ------  ------  -------  -------  ------  ------  ------  ------
FMUL           Wr      Br       WrBr
FMUL           Wi      Bi       WiBi    WrBr
FMUL           Wr      Bi       WrBi    WiBi                    WrBr
FMUL           Wi      Br       WiBr    WrBi                    WiBi             WrBr
FSUB           Tr      T1               WiBr    TrT1            WrBi                                     WiBi
-s-                                                     TrT1    WiBr                     WrBi
FADD           Ti      T2                       TiT2            TrT1                                             WiBr
-s-                                                     TiT2                             TrT1
FADD           Ar      Tr                       ArTr            TiT2
FSUB           Ar      Tr                       ArTr    ArTr                             TiT2
FADD           Ai      Ti                       AiTi    ArTr    ArTr
FSUB           Ai      Ti                       AiTi    AiTi    ArTr     Cr
-s-                                                     AiTi    AiTi     Dr
-s-                                                             AiTi     Ci
-s-                                                                      Di
```

Figure 15 Passage of the register values through the Am29050 processor FPU during the execution of a single decimation-in-time FFT butterfly. The transparent stalls (-s-) must be filled with other instructions to get maximum performance from the RISC

communicate with coprocessors and various memory spaces[12].

The real and imaginary components of *A*, *B* and *W* would be stored in adjacent memory locations (complex array) since this is more efficient than the separate real and imaginary components given earlier. It can be seen from *Figure 12* that the FFT memory accesses follow a very definite pattern. We assume that the addresses for *A*, *B* and *W* are stored in registers *Aaddress* etc., and the increments needed to move onto the next butterfly addresses are stored in registers *Ainc* etc.. It can be seen that a basic requirement for efficient FFT implementation on a RISC chip is either a multitude of registers (e.g. Am29050 processor) or the ability to be able to reload the registers on the fly (e.g. i860).

A simple 'bull-at-the-gate' approach to fetching and storing the values for the butterfly of Equations (10) and (11) would generate code something like *Listing 1*. How does this match up against the DSP processor with all the necessary resources to handle DSP algorithms (particularly the address calculations)? We can get a rough comparison by supposing that the DSP chip takes the same 10 FPU instructions as does the RISC chip operation and assume it requires no additional cycles to handle the addressing. On a RISC chip the same 10 FPU instructions plus associated memory handling require 33 instructions. At first sight, things do not look promising for the RISC chip as it executes nearly 3.3 times more instructions.

```
Prepare to load the registers
ADD Atmp, Aaddress, 0      ; make a copy of the starting addresses
ADD Btmp, Baddress, 0
ADD Wt, Waddress, 0

LOAD Are, Atmp
ADD Atmp, Atmp, 4          ; Are = M[Atmp++]
LOAD Aim, Atmp             ; Aim = M[Atmp]
LOAD Bre, Btmp
ADD Btmp, Btmp, 4          ; Bre = M[Btmp++]
LOAD Bim, Btmp             ; Bim = M[Btmp]
LOAD Wre, Wtmp
ADD Wtmp, Wtmp, 4          ; Wre = M[Wtmp++]
LOAD Wim, Wtmp             ; Wim = M[Wtmp]

Now handle the butterfly calculations using the FPU

Prepare to store the results
ADD Atmp, Aaddress, 0      ; make a copy of the starting addresses
ADD Btmp, Baddress, 0

STORE Cre, Atmp
ADD Atmp, Atmp, 4          ; M[Atmp++] = Cre
STORE Cim, Atmp            ; M[Atmp++] = Cim
STORE Dre, Btmp
ADD Btmp, Btmp, 4          ; M[Btmp++] = Dre
STORE Dim, Btmp            ; M[Btmp++] = Dim

Prepare for next butterfly
ADD Aaddress, Aaddress, Ainc
ADD Baddress, Baddress, Binc
ADD Waddress, Waddress, Winc
```

Listing 1 Bull-at-gate approach to developing FFT algorithm for Am29050 RISC processor

At second glance, things become more promising. DSP chips run at one instruction every two clock cycles, RISC at one instruction every clock cycle if the pipeline can be kept full. The addressing instructions can be placed as useful instructions in the RISC FPU pipeline stalls. Thus the 10 instructions on the DSP take 20 clock cycles and the 33 instructions on the RISC take 33 clock cycles. The time required is now only off by a factor of 1.65.

The reason why the DSP chips perform well is that their FFT implementation takes full advantage of the available resources. A similar thing must be done to get the best from the Am29050 RISC architecture. The major problem with the address calculations is all the time manipulating pointers using software. This has to be moved to hardware to achieve any speed improvement.

The first problem to fix is the fact that it is necessary to calculate the address for A_{im} and C_{im} despite the fact that they are the same address. Let us use additional temporary registers to store these calculated addresses (see *Listing 2*). Since the same thing can be done for the *B* and *D* addresses, this reduces the addressing calculations by six out of 23 cycles: performance is now down only by 1.35×.

```
LOAD Are, Aaddress              ; Are = M[Aaddress]
ADD Aimaddress, Aaddress, 4     ; Aim address = Aaddress+4
LOAD Aim, Aimaddress            ; Aim = M[Aim address]
...
STORE Cre, Aaddress             ; M[Aaddress] = Cre
STORE Cim, Aimaddress           ; M[Aim address] = Cim
```

Listing 2 Reusing addresses stored in the large Am29050 register cuts calculation time

The scalar Am29050 has a LOADM (load multiple) instruction which will bring in adjacent locations of memory into adjacent registers automatically. This is nothing more than another name for an auto-incrementing addressing mode. Thus the code to bring in the real and imaginary parts of *A* (*Listing 3*) can be replaced by the instructions shown in *Listing 4**.

```
LOAD Are, Aaddress              ; Are = M[Aaddress]
ADD Aimaddress, Aaddress, 4     ; Aim address = Aaddress+4 ||
LOAD Aim, Aimaddress            ; Aim = M[Aim address]
```

Listing 3 One approach to bringing in real and imaginary data components from memory to the Am29050 register window

```
LoadMemoryCounter 2   ; prepare to fetch 2 memory values
LOADM Are, Aaddress   ; Are = M[Aaddress]; Aim = M[Aaddress+4]
```

Listing 4 An alternative approach to bringing in data components into the Am29050 registers

This looks like a further improvement to two cycles from three, but is not. With the LOAD instruction it is possible to bring in one register while another is used. The LOADM instruction, however, makes more extensive use of all the Am29050 processor resources and therefore stalls the execution of other instructions for (*COUNT* − 1) cycles while the data is brought in (an Am29050 processor weakness in my opinion).

However, suppose that instead of bringing in enough data to perform one butterfly, we take further advantage of the 192 Am29050 registers and bring in and store enough data to perform four butterflies. This means that there will be only one *LoadMemoryCounter* and one *Adjust Aaddress* calculation for all the *A* value fetches

*'LoadMemoryCounter value' is a macro for 'mtsrim cnt, (value − 1)'.

during those butterflies, a considerable saving. The code then becomes[11] as shown in *Listing 5* †.

```
LoadMemoryCounter 8
LOADM Are, Aaddress                ; Bring in 4 complex A numbers
LoadMemoryCounter 8
LOADM Bre, Baddress                ; Bring in 4 complex B numbers

.set n = 0
.rep 4
ADD Wreaddress, Wreaddress, Winc
LOADoffset (Wre + n), Wreaddress
ADD Wimaddress, Wimaddress, Winc
LOADoffset (Wim + n), Wimaddress   ; Bring in the four complex W numbers
.set n = n + 1
.endrep

**FPU usage***                     ; Do 4 intermingled FFT butterflies

LoadMemoryCounter 8
STOREM Cre, Aaddress               ; Output 4 complex C numbers
LoadMemoryCounter 8
STOREM Dre, Baddress               ; Output 4 complex D numbers

ADD Aaddress, Aaddress, Ainc       ; Get ready for next butterfly set
ADD Baddress, Baddress, Binc
```

Listing 5 An efficient approach to loading the Am29050 registers from memory

This reduces the total time for four butterflies from 132 clock cycles to 94, or approximately 24 per butterfly. With the faster instruction cycle of the Am29050, it is performing within 1.2× of a DSP chip for the FFT algorithm: not bad for a general purpose scalar RISC processor. It has been shown[5] that for equivalent clock rates, the Am29050 handled DSP algorithms with between 50% and 200% of the efficiency of a specialized DSP chip depending on the algorithm chosen.

Taking the same FFT programming approach with the i860 RISC processor would just make it curl up and die. Its registers are just not designed to be used the same way as those of the Am29050 chip. The Am29050 processor has 192 registers attached to the FPU, the i860 only 32. A different approach that makes use of the dual instruction capability of the super-scalar i860 must be taken. The i860 has the capability of overlapping the fetching of four registers from a data cache over a 128 bit wide bus (an integer operation) with the use of a different bank of registers in conjunction with the FPU (a floating point operation). By combining two butterflies, and making good use of its ability to fetch four registers at a time and its really convoluted (flexible) FPU architecture, this gives an extremely efficient seven cycles per butterfly. This and the faster instruction cycle gives the i860 a 2× performance edge over most DSP chips for the FFT algorithm.

Full details on implementing the FFT algorithm on the Am29050 processor are beyond the scope of this article. Further information can be found in Reference 18.

How can you improve the DSP performance of RISC chips?

In the previous section, we indicated that the RISC chips can already give equivalent or better performance than DSP chips in providing an efficient platform for implementing the FFT algorithm. *Table 2* provides details of the FFT performance of a number of integer and floating point DSP and RISC processors.

The FFT algorithm is not just 'butterflies' in the FPU. There are loop and bit-reverse addressing overheads to be considered. The performance figures are gleaned from the various processor data books with the timings scaled up to the latest announced clock rates. The Am29050 timings are based on my own research using a 25 MHz YARC card and an 8 MHz STEB board scaled up to a 40 MHz clock. The standalone STEB board is an inexpensive evaluation board configured for single cycle memory but, to keep costs down, with overlapped instruction and data buses so that Am29050 performance will be degraded by bus conflicts. The PC processor YARC card avoids the bus conflicts but uses multi-cycle data memory.

Comparison of the timings for the FFT on the DSP and RISC processors is rather like comparing apples and oranges. Some of the code is more custom than others and the full details on the limitations or set up conditions are not always obvious – so take the timings with a heavy pinch of salt. If the timings differ by 10%, then there is probably nothing much to choose between the chips in terms of DSP performance. If the difference is more than 50% then perhaps the processor has something that will very shortly be stolen and added to the other chips (or the conditions were non-obviously different).

In Reference 5 the (fictitious) DSP-oriented comprehensive reduced instruction set processor (the Smith's CRISP) was introduced. This is a scalar RISC because of the cost associated with using the current super-scalar RISCs in embedded DSP applications. It was essentially an Am29050 processor, with its large register file, combined with some elements from the i860. The major improvement recognized was the need to have sufficient resources to allow the memory and FPU pipelines to be operated in parallel for more of the Am29050 processor instructions. Improvements were also suggested in allowing more instructions where the FADD and FMULT operations were combined (*à la* i860). The FFT performance for the CRISP was simulated and is given in *Table 2*.

If you read the literature on RISC and DSP chips, you will notice that the inverse DFT takes longer than the straight DFT. This is because the inverse DFT requires that each data point be normalized by dividing by M, the number of data points. Division takes a long time when doing floating point numbers. What is needed is a fast floating point divide by 2.0 equivalent to the integer shift operations. This is available as an FSCALE instruction on the DSP96002 and as a one or two cycle software operation on the Am29050 and i860 processors[19]. However, in many applications the scale factor is just ignored as being irrevelant to the analysis.

Reference 5 suggested that there was one major flaw with both the Am29050 and the i860 chips as DSP processors made evident by the implementation of the FFT algorithm. Neither of the architectures will support bit-reversed memory addressing which is required in the last pass of the FFT to correctly reorder the output values. Done in software, this requires an additional 20% overhead. It was suggested that the overhead be reduced by adding an external address (bit reverse) generator. If you

† 'LOADoffset REG1, REG2' is a macro equivalent to 'LOAD %%(&REG1+*n*), REG2'.

Table 2 Comparison of the timings for Radix 2 and Radix 4 FFT algorithms on a number of current DSP and RISC processors. Based on Reference 5

	DSP						RISC		
	TMS32025	TMS32030	DSP56001	DSP96002	ADSP2100	ADSP21000	i860	Am29050	CRISP
Type	Integer	FP	Integer	FP	Integer	FP	FP	FP	FP
Clock speed (MHz)	50	40	33	40	16.7	33.3	40	40	50
Instr. cycle (ns)	80	50	60	50	60	30	25	25	20
Radix 2									
256 Complex (ms)	1.8	0.68	0.94	0.32	0.85	0.135	0.18	0.69	0.36
256 Bit rev. (ms)							0.22	0.79	
1024 Complex (ms)	15.6	1.97	1.04				1.11	3.74	
1024 Bit rev. (ms)							1.11	3.74	
Radix 4									
256 Complex (ms)	1.2	0.53			0.45	0.121		0.44	0.26
256 Bit rev. (ms)							0.54		
1024 Complex (ms)		2.53		1.81	2.23	0.577		2.13	1.2
1024 Bit rev. (ms)								2.52	

are in need for a fix 'now', it would not be that difficult for a fixed size FFT application to add a little external hardware to 'flip' the effective address line positions at the appropriate time during the final FFT pass.

However, it was recently pointed out to me that the flaw was not in the processors but in the algorithm. There are FFT algorithms other than the decimation-in-time/frequency ones discussed here. This includes algorithms that use extra memory (not-in-place FFT) and avoid the necessity to perform bit-reverse addressing. These algorithms seem to have dropped out of sight over the last 20 years. My next task will be to investigate both DSP and RISC processors using these algorithms to determine if anything is to be gained by revisiting them. After all, the FFT algorithm was known to many authors over many hundreds of years before Cooley and Tukey 'discovered' it!

CONCLUSION

In this applications oriented tutorial article, we have covered a very wide range of topics associated with implementing the discrete Fourier transform (DFT) using the efficient fast Fourier transform (FFT) algorithm. (Make sure you can explain the difference between the DFT and the FFT, if you want to make a DSP expert happy.)

We first examined an industrial application of the DFT and explained the importance of correctly handling the data if the results were to mean anything. The theoretical aspects of generating an efficient algorithm to implement the DFT were then discussed and a class of FFT algorithms detailed. We examined the architectural features required in a processor to handle the FFT algorithm and discussed how the current crop of processors meet these requirements.

This final section was dedicated to details of implementing the FFT on a scalar (Am29050) and a super-scalar (i860) RISC chip. It was shown that by taking into account the architecture of the RISC chips, it was possible to get FFT performance out of these chips approaching or better than current DSP chips. Methods for improving the DSP performance of current RISC processors were indicated. It has been stated in the literature[5] that the lack of a bit-reversed addressing mode penalized the RISC chips as this was a standard DSP processor capability. However, this is not a problem with the RISC processors but rather with the choice of FFT algorithm. There are many FFT algorithms that do not require a bit-reverse address correction pass, although these have been ignored in the literature for many years. By making proper use of the RISC processor's deep pipelined FPU, large register bank or equivalent dual instruction capability and specialized MMU functionality, it was quite obvious that FFT really did stand for fRISCy Fourier transforms!

ACKNOWLEDGEMENTS

The author would like to thank the University of Calgary and the Natural Sciences and Engineering Research Council (NSERC) of Canada for financial support. Bryan Long of Beta Monitors and Control Ltd, Calgary, Canada was kind enough to provide the data for the 'Industrial Application' section and to make constructive comments on this note. The Am29050 processor evaluation timings were generated using a YARC card and a STEB board supplied courtesy of D Mann and R McCarthy under the auspices of the Advanced Micro Devices University Support Program.

NOTE ADDED IN PROOF

AMD has recently announced (June 1993) an integer RISC microcontroller, the Am29240, which has an on-board single cycle hardware multiplier. This processor performs the FFT algorithm in cycle times close to that of the Am29050 floating point processor (preliminary results). Integer RISC processors will be the subject of a future article.

REFERENCES

1 Press, W H, Flannery, B P, Teukolsky, S A and Vettrling, W T, *Numerical Recipes in C – The Art of Scientific Computing* Cambridge University Press, Cambridge (1988)
2 Burrus, C S and Parks, T W *DFT/FFT and Convolution Algorithms – Theory and Implementation* John Wiley, Toronto (1985)
3 Proakis, J G and Manolakis, D G *Digital Signal Processing – Principles, Algorithms and Applications* Maxwell Macmillan, Canada, Toronto (1992)
4 Harris, F J 'On the use of windows for harmonic analysis with the discrete Fourier Transform' *Proc. IEEE*, Vol 66 (1978) pp 51–83
5 Smith, M R 'How RISCy is DSP?' *IEEE Micro Mag.* (December 1992) pp 10–23
6 Smith, M R, Smit, T J, Nichols, S W, Nichols, S T, Orbay, H and Campbell, K 'A hardware implementation of an autoregressive algorithm' *Meas. Sci. Technol.* Vol 1 (1991) pp 1000–1006
7 Papamichalis, P and So, J 'Implementation of fast fourier transform on the TMS32020' in *Digital Signal Processing Applications with TMS320 Family* Vol 1, Texas Instruments (1986) pp 69–92
8 Papamichalis, P 'An implementation of FFT, DCT and other transforms on the TMS320C30' in *Algorithms and Implementations* Vol 3, Texas Instruments (1990)
9 Sohie, G *Implementation of Fast Fourier Transforms on Motorola's DSP56000/DSP56001 and DSP96002 Digital Signal Processors* Motorola Inc. (1989)
10 'Analog devices' in *Digital Signal Processing Applications using the ADSP-2100 family* Vol 1, Prentice-Hall, Englewood Cliffs, NJ (1992)
11 Smith, M R 'To DSP or not to DSP' *Comput. Appl. J.* No 28 (August/September 1992) pp 14–25
12 *Am29050 32-Bit Streamlined Instruction Processor, User's Manual* (1991)
13 Atkins, M *FAST Fourier Transforms in the i860 Microprocessor* Intel Application Note Ap-435, Intel Corporation (1990)
14 Margulis, N *i860 Microprocessor Architecture* Osbourne McGraw-Hill (1990)
15 Smith, M R, Nichols, S T, Henkelman, R M and Wood, M L 'Application of autoregressive moving average parametric modeling in magnetic resonance image reconstruction *IEEE Med. Imag.* Vol MI5 No 3 (1986) pp 132–139
16 Mitchel, D K, Nichols, S T, Smith, M R and Scott, K 'The use of band selectable digital filtering in magnetic resonance image enhancement' *Magn. Reson. Med.* Vol 19 (1989) pp 353–368
17 Smith, M R, Nichols, S T, Constable, R T and Henkelman, R M 'A quantitative comparison of the TERA modeling and DFT magnetic resonance image reconstruction techniques' *Magn. Reson. Med.* Vol 21 (1991) pp 1–19
18 Smith, M R *The FFT on the Am29050 Processor* AMD Application note PID 17245 (to be published)
19 Smith, M R and Lau, C 'Fast floating point scaling operations on RISC and DSP processors' *Comput. Appl. J.* (to be published)

Dr Michael Smith emigrated from the UK to Canada in 1966 to undertake a PhD in physics, during which he was introduced to computer programming. After spending several years in digital signal processing applications in the pathology area, he left research to teach junior and high school science and mathematics. He joined the Electrical and Computer Engineering Department at the University of Calgary, Canada in 1981. His current interests are in the software and hardware implementation of ARMA modelling algorithms for use in resolution enhancement and artifact reduction of magnetic resonance (MRI) images.