Texas Instruments

Application Report
SPRA696A – April 2001

Extended-Precision Complex Radix-2 FFT/IFFT Implemented on TMS320C62x

Mattias Ahnoff
DSP Central Europe

ABSTRACT

The limited dynamic range of a fixed-point DSP causes accuracy problems in Fast Fourier Transform (FFT) calculation. This is due to quantization and the scaling that has to be applied to prevent output overflow. Hence, there is a need to perform the calculations in extended precision, especially for large FFT sizes. Highly optimized extended-precision assembly routines for the FFT and its inverse, the IFFT, have been implemented for the TMS320C62x™ (C62x™), showing that the accuracy enhancement can be performed at a very modest increase in execution time.

Contents

1 Fast Fourier Transform (FFT) 2
2 Inverse Fast Fourier Transform (IFFT) 3
3 The Need for Extended Precision 3
4 Extended-Precision Multiplication on C62x 4
5 Implementation of the FFT 6
6 Implementation of the IFFT 6
7 Scaling Issues 6
8 Error Estimation 7
9 Benchmarks 9
10 Bit Reversal of the Output 9
11 Code 9
12 References 9

Figures

Figure 1. Radix-2 FFT Decomposition of the DFT for N=8 3
Figure 2. Typical Error Distribution for FFT and IFFT of Size 64 8
Figure 3. Mean Error Magnitude as a Function of N 8

Tables

Table 1. Typical Errors in FFT Calculation 7
Table 2. Typical Errors in IFFT Calculation 7
Table 3. Benchmarks for the Extended-Precision Radix-2 FFT 9
Table 4. Code Sizes for the FFT/IFFT Routines 9

TMS320C62x and C62x are trademarks of Texas Instruments.

1 Fast Fourier Transform (FFT)

The Discrete Fourier Transform (DFT) $X(k), k = 0...N-1$ of the sequence $x(n), n = 0...N-1$ can be defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \; k = 0...N-1$$

with

$$W_N^{nk} = \exp(2\pi jnk / N).$$

Its inverse, the IDFT, is then defined as

$$x(k) = \frac{1}{N} \sum_{n=0}^{N-1} X(n) (W_N^{nk})^*, \; k = 0...N-1$$

with

$$(W_N^{nk})^* = \exp(-2\pi jnk / N).$$

Let us define an elementary operation to be a multiplication of two complex numbers, followed by the addition of two complex numbers. It is then clear that computing the DFT directly from the definition requires N^2 in such operations. The problem is that N^2 becomes enormous for large values of N. However, when N is composite, it is possible to dramatically reduce the number of operations needed. The Radix-2 FFT algorithm (which is nothing more than a fast DFT algorithm) takes advantage of the periodicity of the twiddle factors W_N^{nk} when N is a power of 2. In this case, it is possible to decompose the DFT into a sum of two DFTs, both of size N/2:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} = \sum_{n=0}^{N/2-1} \left[x(n) + (-1)^k x(n + \frac{N}{2}) \right] W_N^{nk}.$$

Recursively applying this decomposition until the trivial case $N = 1$ is reached makes the radix-2 FFT of size N requiring $N\log_2 N$ elementary operations instead of the original N^2, which is a substantial improvement. Figure 1 shows how this decomposition works in the case $N = 8$. (An upward arrow represents an addition, a downward arrow represents a subtraction, and the presence of W implies a complex multiplication by the associated twiddle factor).

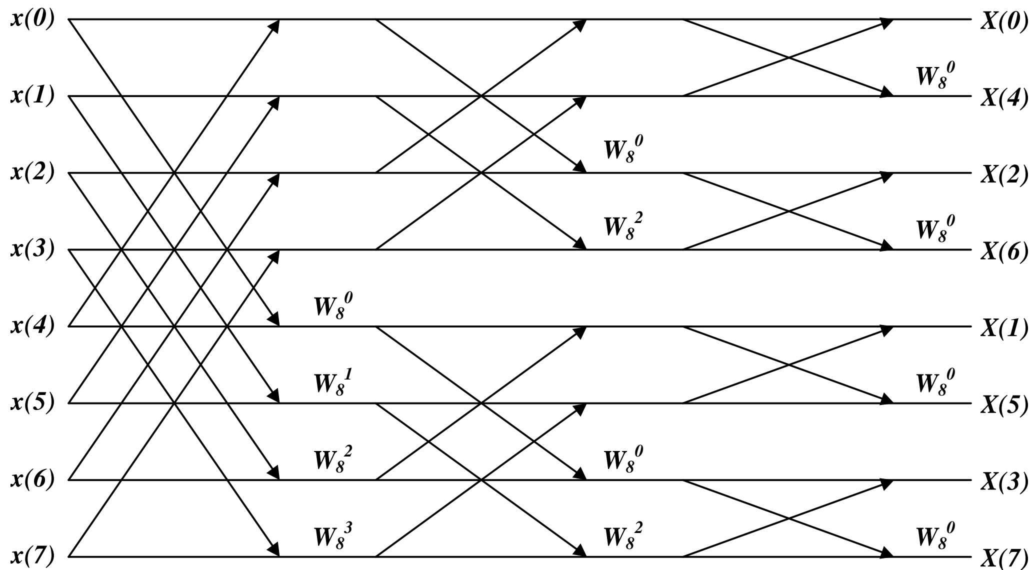

Figure 1. Radix-2 FFT Decomposition of the DFT for N=8

2 Inverse Fast Fourier Transform (IFFT)

The convention is that, if the FFT is defined without the factor $1/N$ (as it was here), then the IFFT must contain it in order to make IFFT $\circ$ FFT the unity operation.

Even though the FFT routine can be used for performing N times an IFFT (this is done by using complex conjugated coefficients), having a separate routine for the IFFT has some advantages:

- The same coefficients can be used for both the FFT and the IFFT, thus saving memory if both are needed.
- More flexibility regarding scaling issues since one routine uses the $1/N$ factor and the other does not.

3 The Need for Extended Precision

When implementing an FFT on a fixed-point DSP, some kind of scaling must be applied to prevent overflow. One could imagine two possibilities: either the input values have to be shifted before the FFT is performed to gain guard bits, or scaling is performed successively after each stage (and then the output has to be interpreted as being of a different Q format). In both cases, precision is lost quickly, as N increases, even though the latter is better in most cases.

For simplicity, consider the first option where input is scaled before applying the FFT. For example, for $N = 256$ the input has to be divided by at least 256, thus losing eight bits of accuracy. Therefore, if the input values are in 16-bit, then there is only 8-bit precision left after the necessary scaling step. In addition to this, rounding errors from the calculations within the FFT will propagate and distort the output even more. With a 32-bit input format, it would be possible to scale the input and still maintain a decent precision.

4 Extended-Precision Multiplication on C62x

Consider the multiplication of two 32-bit integers: a and b. We separate each integer into two 16-bit quantities, one signed and one unsigned:

A = A_hi <<16 + A_lo, where A_hi = A >>16, and A_lo = A & 0x0000FFFF.

B = B_hi <<16 + B_lo, where B_hi = B >>16, and B_lo = A & 0x0000FFFF.

Then we can calculate the product using four 16-bit multiplications in the following manner:

A * B = ahbh + ahbl + albh + albl, where

ahbh = (A_hi * B_hi)<<1 , signed-signed multiplication with saturation.

ahbl = (A_hi * B_lo)>>15 , signed-unsigned multiplication.

albh = (A_lo * B_hi)>>15 , unsigned-signed multiplication.

albl = (A_lo * B_lo)>>31 , unsigned multiplication.

The C62x instruction set contains mnemonics such as SMPYH, MPYHSLU, MPYLUHS etc., to easily accomplish this.

The result would have differed slightly if we had performed a 32 x 32-bit multiplication and kept the 32 MSB of the 64-bit result. For this reason, we have limited accuracy in the three terms that are shifted before they are added together. It is possible to avoid some of these errors; however, it would also be a trade-off between accuracy and execution time. Here, a technique is chosen where we get good accuracy at moderate cycle requirements.

It is quite favorable to disregard the last term albl. In the worst case, this term accounts for (0xFFFF * 0xFFFF)>>31 = 0xFFFE0001>>31 = 1, and it is zero approximately half of the time if A_lo and B_lo are equally distributed random numbers. In fact, the error introduced by the shifting/addition of the two cross-terms is larger, since it is two worst-case and one on average (0.5 + 0.5).

The following C-code illustrates how the extended-precision multiplication is implemented disregarding (A_lo * B_lo)>>31.

```
int epmpy(int a, int b)
{
 short ah, bh;
 unsigned short al, bl;
 int ahbl, albh, ahbh;

 ah = a>>16;
 al = a & 0xFFFF;
 bh= b>>16;
 bl = b & 0xFFFF;

 ahbh  = (ah*bh)<<1;
 albh = (al*bh)>>15;
 ahbl = (ah*bl)>>15;

 return ahbh+(albh+ahbl);
}
```

In the IFFT routine, we want to calculate (A*B)>>1 rather than A*B since the intermediate results are divided by 2 in each stage. Using the naming conventions given above, we have

(A * B)>>1 = ahbh + ahbl +a lbh, where

ahbh = A_hi * B_hi , signed-signed multiplication.

ahbl = (A_hi * B_lo)>>16 , signed-unsigned multiplication.

albh = (A_lo * B_hi)>>16 , unsigned-signed multiplication.

The fourth term is completely shifted out and thus disregarded. It should have been (A_lo * B_lo)>>32, but obviously this will always be zero.

It should be pointed out that the multiplications used in the FFT/IFFT algorithms are complex. Since we always operate on real data, we must rewrite a multiplication of two complex numbers X and W as

X * W = Re(X) * Re(W) - Im(X) * Im(W) + j * (Re(X) * Im(W) + Im(X) * Re(W)).

We need four (real) 32-bit multiplications to accomplish one complex 32-bit multiplication. As stated earlier, each real 32-bit multiplication is implemented using three 16-bit multiplications. Altogether, we must perform twelve 16-bit multiplications to realize one complex 32-bit multiplication.

5 Implementation of the FFT

The C-equivalent of the FFT assembly code is listed as comments in the assembly file. Each iteration of the inner loop calculates one butterfly (i.e., two complex pairs of data), and each outer loop iteration corresponds to one stage in the flow chart. This structure has been carried over to the ASM implementation. However, some changes in the variables have been made to minimize pointer overhead. With these optimizations made, it is possible to perform one iteration of the inner loop in only seven cycles. To achieve this, as much as possible of the symmetry between the calculation of the real and imaginary parts has been kept; therefore, they run in parallel with the calculations of the real parts on the A-side and the calculations of the imaginary parts on the B-side.

In the inner loop, all 32 registers are used. Thus, all variables of the outer loop that accessed after the inner loop, or in the next iteration of the outer loop, have to be pushed onto the stack.

The parallelism of the C62x is used extensively in the inner loop. Out of the 56 instructions available in the 7-cycle loop, 54 are used, and so the pipeline is filled to more than 96%.

The last iteration of the outer loop has been lifted out. In this case, the twiddle factors are all 0 or 1 so the calculations can be simplified and some complex multiplications can be avoided.

To perform an inverse FFT, the same code can be used, but the twiddle factors have to be complex-conjugated. This works even though the last outer loop iteration is hard-coded.

6 Implementation of the IFFT

The implementation of the IFFT has been done much in the same way as the IFFT. It uses an 8-cycle inner loop; therefore, it is a little slower than the FFT. Even here, the last stage was treated separately in order to gain accuracy and speed. Also, the stack was used to push and pop variables in the same way as it was used in the FFT.

In each stage, the intermediate results of the IFFT are shifted one bit to the right. Since there are $\log_2 N$ stages altogether, the pre-factor will result in being $1/2^{\log_2 N} = 1/N$, as required by the definition.

By analogy with the FFT, the IFFT routine can be used to calculate $1/N$ times an FFT. Again, complex-conjugated twiddle factors have to be used.

7 Scaling Issues

To prevent overflow, a proper scaling of the input values is required. For the IFFT, we need two guard bits to prevent overflow of the intermediate results; hence, the input must be in Q29 format, or equivalently, $[-0.25, 0.25]$ in Q31 format. For the FFT we need one bit plus one bit per stage, so the input has to be in Q(31-$\log_2 N$) format, or equivalently, $[-1/N, 1/N]$ in Q31 format.

8 Error Estimation

To determine the accuracy of the output from the FFT routines, the following tests were performed:

An input vector of random numbers in the range [-1, 1] in Q31 format was created. A reference output was calculated using MATLAB. After that, input was appropriately shifted to prevent overflow, and the FFT was calculated using the routine given. Next, the errors compared to the reference were calculated. This procedure was repeated ten times for each value of N.

A similar testing procedure was employed for the IFFT. In this case, the input was shifted two bits before the IFFT was calculated, and then the result was shifted back to represent full scale.

Table 1 and Table 2 show summations of typical errors found in FFT and IFFT calculations. The errors are given with regard to a full-scale output, i.e., [-1, 1] in Q31. Since the errors are complex-valued, the magnitude of the error has been chosen to represent the deviation from the reference. An example of the (complex) error distribution is also shown in Figure 2.

Table 1. Typical Errors in FFT Calculation

N	Mean Error	Max Error	Standard Dev.
8	1.9E-09	5.4E-09	1.3E-09
16	2.9E-09	1.0E-08	2.1E-09
32	4.1E-09	2.4E-08	3.7E-09
64	5.8E-09	4.5E-08	5.7E-09
128	7.7E-09	7.9E-08	8.0E-09
256	1.1E-08	1.6E-07	1.2E-08
512	1.8E-08	3.2E-07	1.9E-08
1024	2.7E-08	6.1E-07	2.8E-08

Table 2. Typical Errors in IFFT Calculation

N	Mean Error	Max Error	Standard Dev.
8	2.4E-09	5.6E-09	1.2E-09
16	2.6E-09	6.0E-09	1.2E-08
32	2.7E-09	7.9E-09	1.6E-09
64	2.7E-09	8.4E-09	1.7E-09
128	2.7E-09	8.5E-09	1.7E-09
256	2.7E-09	9.8E-09	1.7E-09
512	2.4E-09	1.0E-08	1.7E-09
1024	2.4E-09	1.0E-08	1.6E-09

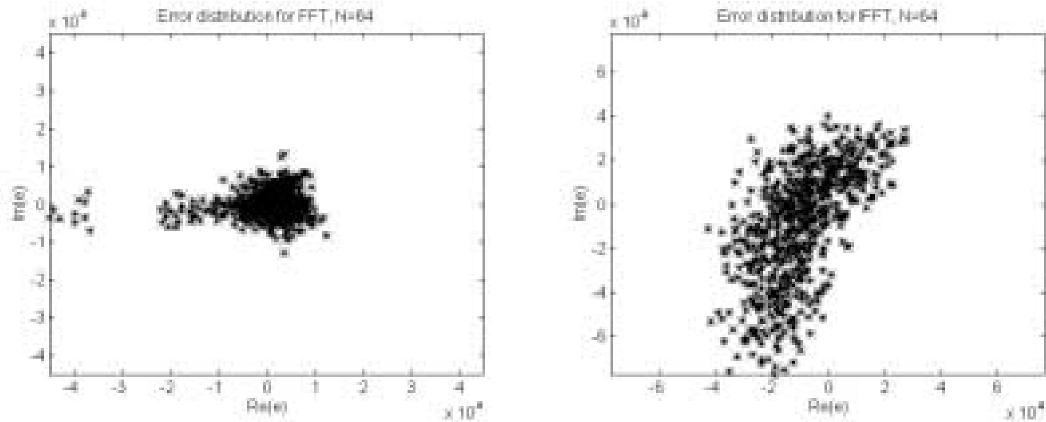

Figure 2. Typical Error Distribution for FFT and IFFT of Size 64

The input is now considered to be of infinite precision. Errors calculated here include both quantization errors and errors from the algorithm itself. Note also that a big portion of the error is added rounding errors originating from the initial scaling. The fact that the error of the FFT is increasing faster as N increases than the error of the IFFT does, is a sign of this (since the input is increasingly scaled in the FFT, whereas the scaling in the IFFT is always constant). In most practical cases, the input will not have 32 significant bits (if, for example, they come from an ADC). In such case, the scaling step may make no contribution to the error since the limited accuracy is already there.

Mean error magnitude values (from Table 1 and Table 2) are shown in Figure 3. For the IFFT, a typical error is approximately $3 \cdot 10^{-9}$. Since $3 \cdot 10^{-9} \cdot 2^{31} \approx 6.4$, this means that, on average, only approximately 4 bits of accuracy are lost, even for N as high as 1024.

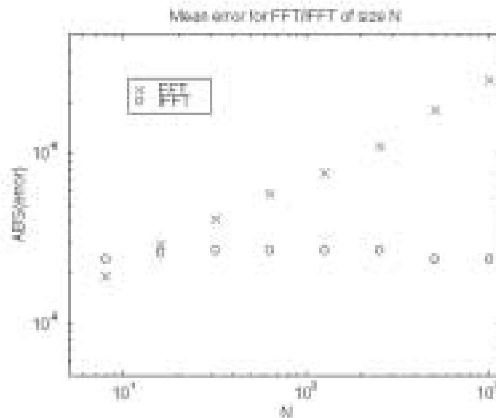

Figure 3. Mean Error Magnitude as a Function of N

9 Benchmarks

The following benchmark formula is valid for the FFT routine and for size N :

#cycles = $\log_2(N)(7N/2+22)-3N/2-8$; and

For the IFFT we have

#cycles = $\log_2(N)(4N+21)-2N-10$.

Table 3. Benchmarks for the Extended-Precision Radix-2 FFT

N	16	32	64	128	256	512	1024	2048
#cycles FFT	280	614	1372	3090	6952	15550	34561	76010
#cycles IFFT	298	671	1524	3465	7838	17587	39112	86237

Compared to the 16-bit radix-2 FFT that is available in the C62x DSP Library [1], which has a benchmark of 4225 cycles for $N=256$, this 32-bit FFT requires only 65% more cycles. This ratio stays more or less the same, regardless of N. The IFFT uses approximately 85% more cycles than the 16-bit FFT.

Code sizes of the routines are shown in Table 4.

Table 4. Code Sizes for the FFT/IFFT Routines

	FFT	IFFT
Code size (words)	624	720

10 Bit Reversal of the Output

Outputs of the FFT as well as outputs of the IFFT are in bit-reversed order. The linear assembly routine provided, together with this report, can be used to perform a bit reversal of the output array so that it is linearly ordered. A small index table of size $\sqrt{N}$ or $\sqrt{2N}$ (depending on whether $\log_2 N$ happens to be even or odd) is used to speed up the data swapping. This routine is simply an extension to 32-bit of the routine found in the C62x DSP Library.

11 Code

ASM Source code for the FFT/IFFT is included with this application report. Routines to perform the bit reversal are included as well.

12 References

1. *TMS320C62x DSP Library Programmer’s Reference* (SPRU402).

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2001, Texas Instruments Incorporated