Parallel 1-D FFT Implementation With TMS320C4x DSPs

Application Report

Rose Marie Piedra
Digital Signal Processing — Semiconductor Group

SPRA108
February 1994

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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 semiconductor products or services might be or are used.

Introduction

The Fast Fourier Transform (FFT) is one of the most commonly used algorithms in digital signal processing and is widely used in applications such as image processing and spectral analysis.

The purpose of this application note is to investigate efficient partitioning/parallelization schemes for one-dimensional (1-D) FFTs on the TMS320C40 parallel processing DSP. Partitioning of the FFT algorithm is important in two special cases:

- For computation of large FFTs in which input data doesn't fit in the available processor's on-chip RAM. In this case, execution must be performed with the data off-chip, resulting in performance degradation. As a consequence, execution time grows exponentially with the FFT size.
- For FFT computation in multiprocessing systems where more than one processor is used to reduce FFT execution time. Theoretically, a maximum speed-up of P can be reached in a system with P processors. In reality, such a speed-up is never achieved, because of interprocessor communication overhead, among other factors.

This document focuses on complex FFTs; however, the concepts used can be easily applied to real FFTs.

This paper covers both Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF) methods of computation to give flexibility in programming and to demonstrate the results of parallelization on both methods.

Given the general scope of this application note, the programs have been kept as generic as possible to work for any FFT size and for any number of processsors in the system. For a specific application (fixed FFT size and fixed number of processors), a better performance is expected because of savings in programming overhead.

The programs were developed in the C language, and the core routines were implemented in 'C40 assembly language to provide a combination of C portability and assembly language performance. Compiler optimization techniques such as the use of registers for parameter passing have been used in the programs to increase performance. Even higher performance could be achieved with a total assembly language implementation.

The algorithms were tested on the Parallel Processing Development System (PPDS), a system with four TMS320C40s and with both shared and distributed-memory support.

This report is structured as follows:

Introduction	States the purpose and scope of this application note.
One-Dimensional (1-D) FFT	Gives a brief review of the FFT algorithm, discussing DIF and DIT FFT implementation methods.
Parallel 1-D FFT	Focuses on parallel 1-D FFT implementations on multiprocessing systems. DIF and DIT FFT implementations are discussed.
Partitioned 1-D FFT	Focuses on very large partitioned 1-D FFT implementations on uniprocessor systems. The DIT implementation is discussed.
TMS320C40 Implementation	Presents the results of uniprocessor and distributed-memory multiprocessor implementations on the PPDS. Gives analyses of the speed-up and efficiency achieved.
Results and Conclusions	States conclusions.
Appendices	List source code.

One-Dimensional (1-D) FFT

The Discrete Fourier Transform (DFT) of an n-point discrete signal $x(i)$ is defined by:

$$X(k) = \sum_{i=0}^{n-1} x(i) W_n^{ik}$$

where $0 \le k < n$, $j = \sqrt{-1}$, and $W_n = e^{-j2\pi/n}$ (known as the twiddle factor).

Direct DFT computation requires $O(n^2)$ arithmetic operations. A faster method of computing the DFT is the FFT algorithm. FFT computation is based on a repeated application of an elementary transform known as a "butterfly" and requires that n (FFT length) is a power of 2 (i.e., $n = 2^{m)}$ [4]. If n is not a power of 2, the sequence $x(i)$ is appended with enough zeroes to make the total length a power of 2. A more detailed analysis of 1-D FFT can be found in [3] and [6].

There are two basic variants of FFT algorithms: Decimation-in-Frequency (DIF) and Decimation-in-Time (DIT). The terminology essentially describes a way of grouping the terms of the DFT definition; see the equation above. Another parameter to consider is the radix of the FFT, which represents the number of inputs that are combined in a butterfly [4].

This application note focuses on Radix-2 Complex FFT, but the partitioning concepts stated here can also be applied to other FFT algorithms. Figure 1 and Figure 2 show complete graphs for computation of a 16-point DIF and DIT FFT, respectively. Both assume the input in correct order and the output in bit-reversed order.

Figure 1. Flow Chart of a 16-Point DIF FFT

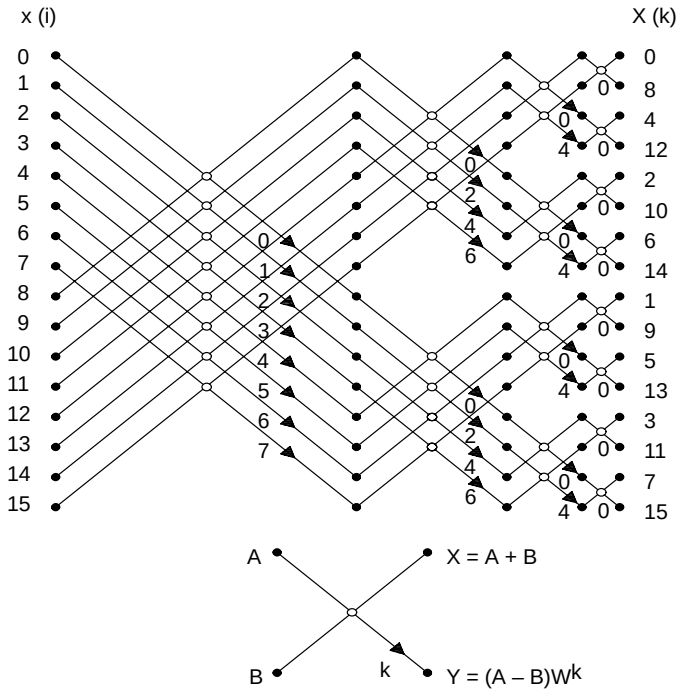
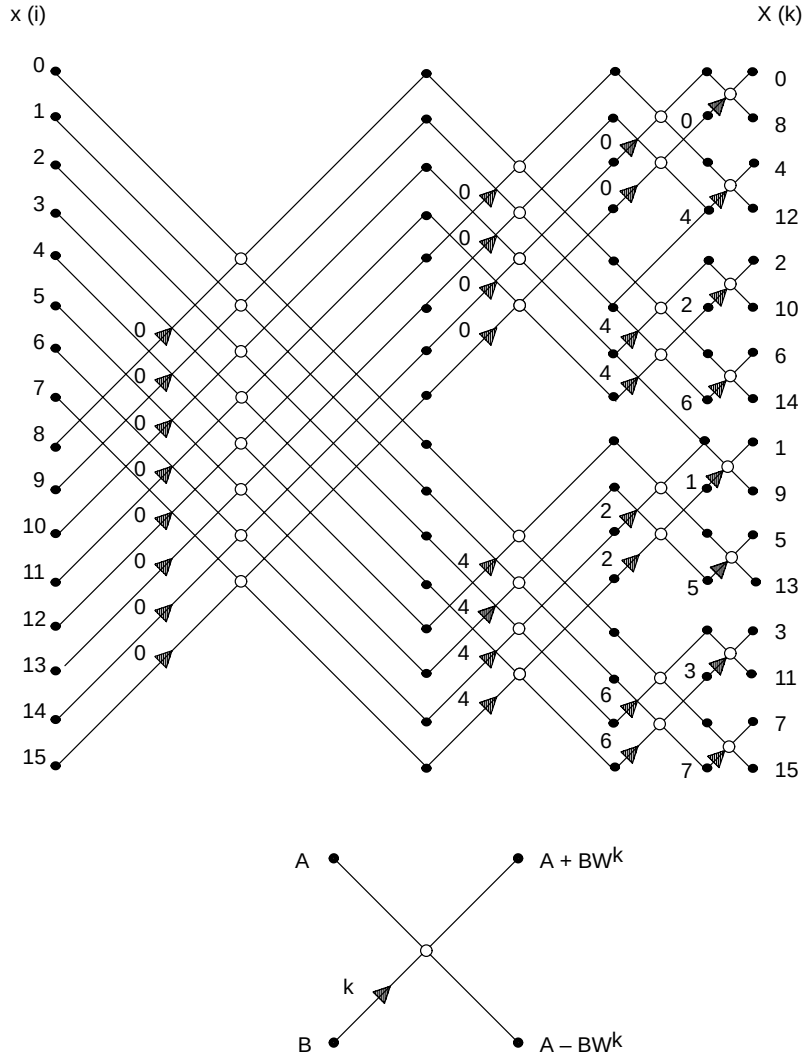


Figure 2. Flow Chart of a 16-Point DIT FFT



Timing Analysis

If FFT is used to solve an n-point DFT, ($\log_2 n$) steps are required, with $n/2$ butterfly operations per step. The FFT algorithm therefore requires approximately $(n/2)\log_2 n \sim O(n \log_2 n)$ arithmetic operations, which is $n/(\log_2 n)$ times faster than direct DFT computation.

Parallel 1-D FFT

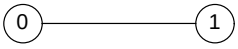
Decimation-in-Frequency (DIF) and Decimation-in-Time (DIT) decomposition schemes are investigated for parallel implementation. Parallel FFT theory is covered in [1], [11], [12], and [13].

For this parallel implementation of the 1-D FFT, a distributed-memory multiprocessing system with p processors connected in a d-dimensional hypercube network topology is required.

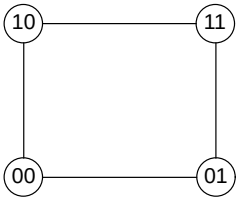
A d-dimensional hypercube is a multiprocessor system characterized by the presence of 2^d processors interconnected as a d-dimensional binary cube. Each node forms a vertex of the cube, and its node identification (node ID) differs exactly one bit from that of each of its d neighbors. Figure 3 shows the typical configuration for a 1-, 2-, and 3- dimensional hypercube.

This paper does not cover parallel shared-memory implementations, because the FFT algorithm is more suitable for distributed-memory multiprocessing systems. Solowiejczk and Petzinger have proposed an interesting approach to solve very large FFT (> 10K points) on shared-memory systems [13].

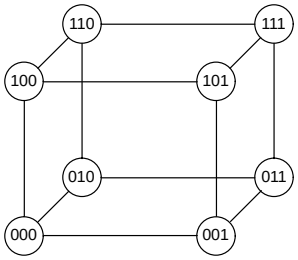
Figure 3. 1-, 2-, and 3-Dimensional Hypercubes



1-Dimensional Hypercube



2-Dimensional Hypercube

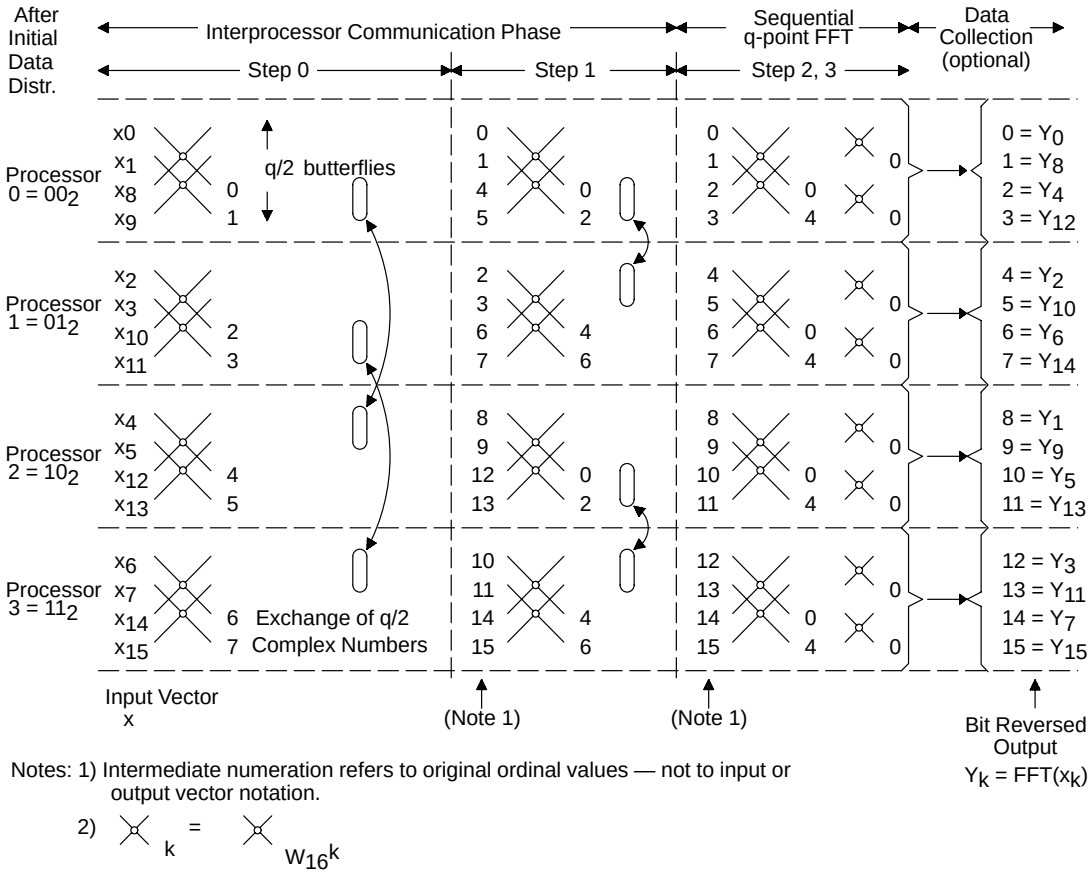


3-Dimensional Hypercube

Parallel DIF FFT

Let $n=qp$, where n is the FFT size, p is the number of processors present in a hypercube configuration , and $q \geq 1$ is an integer. FFT input is in normal order, and FFT output is in bit-reversed order. The parallel algorithm is shown in detail in Figure 4 for $n = 16$ and $p = 4$.

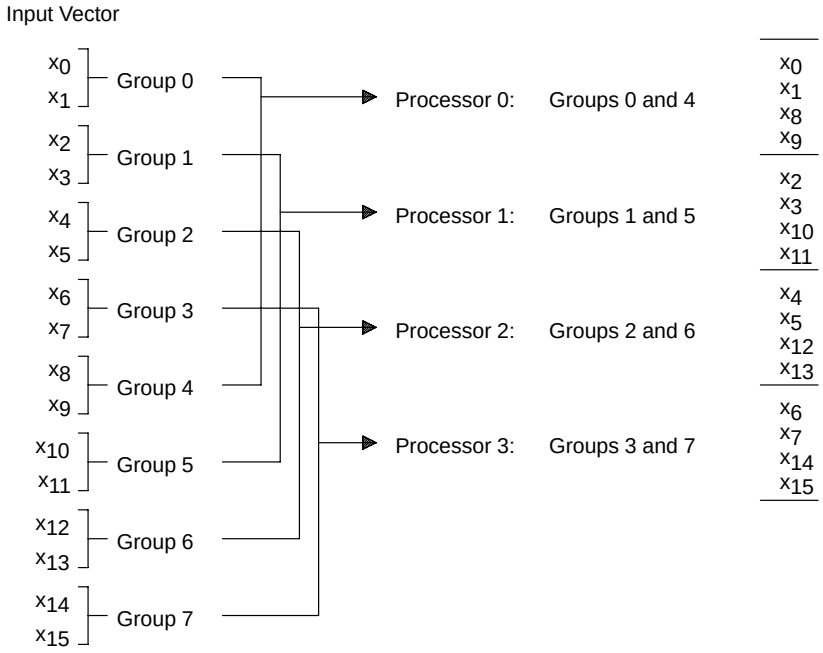
Figure 4. Parallel DIF FFT Algorithm



Phase 1. Data Distribution Phase:

Vector input x is partitioned sequentially into $2p$ groups of $q/2$ complex numbers each and assigned to processor i, groups i and $i + p$. At the end of this data distribution step, processor i contains vector elements $(i*q/2) + j$ and $(i*q/2) + n/2 + j$ where $0 \le i < p$ and $0 \le j < q/2$. This process is shown in Figure 5 for $n = 16$ and $p = 4$.

Figure 5. DIF FFT Data Distribution Step (*n*=16 and *p*=4)



Phase 2. Interprocessor Communication Phase:

Interprocessor communication is required because subsequent computations on one processor depend on intermediate results of the other processors. With this mapping scheme, d concurrent exchange communication steps are required during the first d stages ($0 \le k < d$), where $d=\log_2(p)$ is the hypercube dimension.

At each of these steps, every node :

- computes a butterfly operation on each of the butterfly pairs allocated to it and then
- sends half of its computed result ($q/2$ consecutive complex numbers) to the node that needs it for the next computation step, and waits for the information of the same length from another node to arrive for continuing computation [12].

The selection of the destination processor and the data to be sent is based on the node-id allocated to each processor as follows:

If bit j of its node ID is 0,

send $q/2$ consecutive complex numbers (lower half) to processor dnode = (node ID with bit j swapped)

else

send $q/2$ consecutive complex numbers (upper half) to processor dnode = (node ID with bit j swapped)

Variable j initially points to bit $(\log_2 p)–1$, the most significant bit of the node ID, and is right-shifted after each interprocessor communication step. Because of this bit-swapping, interprocessor communication is always carried between neighbor processors according to the hypercube definition.

Phase 3. Sequential Execution Phase:

During the remaining (n–d) stages, no interprocessor communication is required, and a sequential FFT of size q is independently performed in each of the processors. Notice in Figure 4 that steps 2 and 3 correspond to a sequential 4-point FFT because

$W_n^{\,k} = W_{n/p}^{\,k/p}$ $\quad (W_{16}^{\,4} = W_4^{\,1})$ [6].

Phase 4. Data Collection (optional):

At the end of FFT computation, processor i contains q complex elements with ordinal position $i*q + j$ in the bit-reversed FFT vector result ($0 \le j < q$ and $0 \le i < p$). If data must be collected, this will involve the linear transfer of $q*2$ consecutive memory locations. Collected results are in bit-reversed order. If required, the host processor can then execute a bit-reverse operation on a size-n vector.

This parallelization scheme reduces interprocessor communication delay (interprocessor communication is restricted to neighbor processors on a hypercube) and balances the load perfectly (each processor executes an equal number of butterfly computations assuming $q = n/p$ as an integer number).

Parallel DIT FFT

Let $n=qp$, where n is the FFT size, p is the number of processors present in a hypercube configuration, and $q \ge 1$ is an integer. FFT input is in normal order, and FFT output is bit-reversed after data collection.

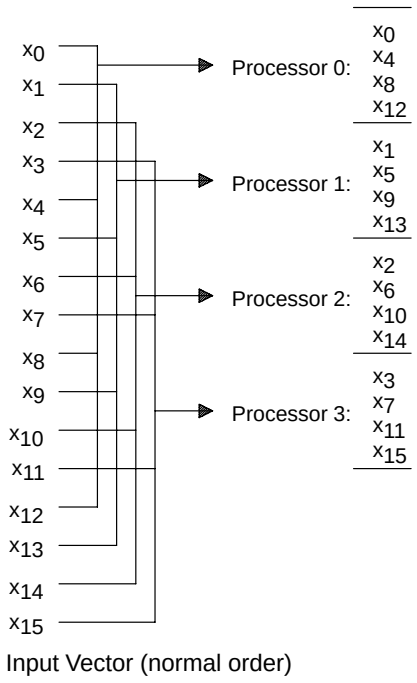
Parallel DIT requires a parallelization scheme different than the one presented for parallel DIF FFT. Sequential q-point FFT execution is required in the first ($\log_2 n - \log_2 p$) steps, and interprocessor communication is required in the last $\log_2 p$ steps. This is exactly opposite to the DIF case. However, strong similarities exist between DIT and DIF parallel approaches.

Phase 1. Data Distribution Phase:

Because applying the same DIF initial data distribution will require additional interprocessor communication during the initial q-point FFT, the data distribution scheme must be modified. Vector input x is now distributed in such a way that processor i contains elements $i + j*p$, where $0 \le j < (n/p)$ and $0 \le i < p$.

This process is shown in Figure 6 for $n = 16$ and $p = 4$.

Figure 6. DIT FFT Data Distribution Step (n=16 and p=4)



Phase 2. Sequential Execution Phase:

During the first (n–d) stages, no interprocessor communication is required, and a sequential FFT of size q is independently performed in each of the processors.

Phase 3. Interprocessor Communication Phase:

As in the DIF case, d concurrent exchange communication steps are required, where $d = \log_2(p)$ is the hypercube dimension but this time during the last d steps. At each of these steps, every node:

- sends half of its computed result (q/2 complex numbers) to the node that needs it for the next computation step,
- waits for the information of the same length from that node to arrive for continuing computation [12], and then
- computes the vector butterfly operation.

Notice that the sequence is send —> compute (not compute —> send as in the DIF case). For the send step, two approaches can be followed:

Scheme 1: Same approach as in the DIF case.

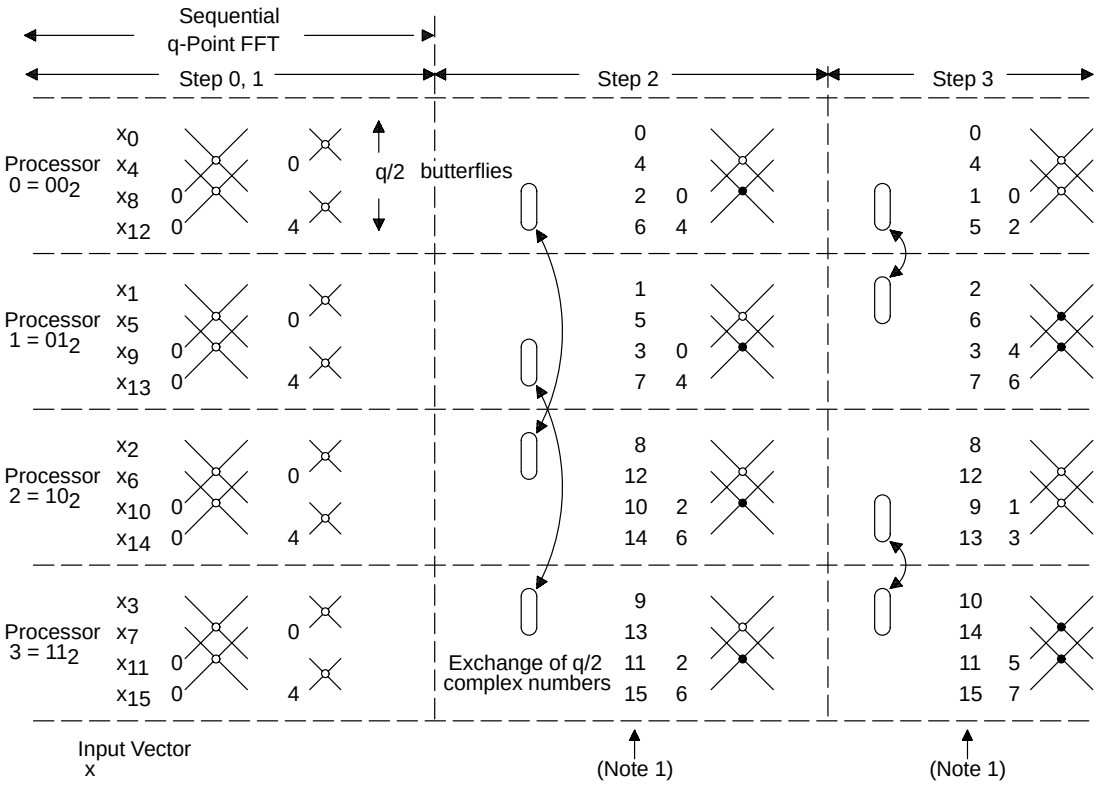
If bit j of its node ID is 0,

send q/2 consecutive complex numbers (lower half) to processor dnode = (node ID with bit j swapped)

else

send q/2 consecutive complex numbers (upper half) to processor dnode = (node ID with bit j swapped).

Figure 7. Parallel DIT FFT Algorithm (Scheme 1)



Notes: 1) Intermediate numeration refers to ordinal values — not to input or output numeric values.

2) k = W_{16}^{k}

3) Type I butterfly operation Type II butterfly operation

Variable *j* initially points to the most significant bit of node id (bit ($\log_2 p$) –*i*) and is right-shifted after each interprocessor communication step. Figure 8 illustrates this partitioning scheme. The approach is similar to the one suggested in [11] for $n = 16$ and $p = 4$ for bit-reversed data input. This scheme is useful with regular core DIT FFT routines that use a full-size sine table.

Because memory space is always a concern in real DSP applications, several highly optimized FFT routines use reduced-size sine tables. This is true of the Meyer-Schwarz Complex DIT FFT routine shown in Appendix B, which constitutes the core routine for this parallel DIT implementation. The routine offers a faster execution time and a reduced size sine table, but the programming complexity increases.

Multiplication for twiddle factors not directly available in the sine table becomes an issue during the butterfly vector operations in the interprocessor communication phase. Meyer-Schwarz FFT uses a reduced-size bit-reversed sine table of only *n*/4 complex twiddle factors.

In this case, multiplication for twiddle factors W_n, where $k \geq n/4$, must receive special treatment. In the Meyer-Schwarz FFT, the missing twiddle factors are generated with the symmetry

$W_n^{(n/4 + k)} = -j\, W_n^{k}$ (Equation 2)

This is done by changing real and imaginary parts of the twiddle factors and by negating the real part. This leads to 2 different types of butterfly operations: TYPE I (regular butterfly operation) and TYPE II (butterfly operation for missing twiddle factors). See Appendix B for a detailed explanation of the two butterfly types.

The concept of 2 types of butterflies should be extended to the butterfly vector operation for the final $\log_2 p$ steps of the parallel DIT FFT implementation. Figure 7 shows that with Scheme 1 some processors must compute only TYPE I operations, others only TYPE II operations, and others both TYPE I and TYPE II operations at each vector butterfly operation. This increases the programming complexity and makes it difficult to write a parallel program with *p* (number of processors) and *n* (FFT size) as general parameters. To solve this issue, a different interprocessor communication scheme is proposed:

Scheme 2:

If bit *j* of its node ID is 0,

- send the *q*/2 complex numbers with odd *ordinal* positions 1, 3, 5 ..(*q*–1) to processor dnode = (node ID with bit *j* swapped)
- execute vector butterfly operation (TYPE I)

else

- send the *q*/2 complex numbers with even *ordinal* positions 0, 2, 4 ..(*q*–2) to processor dnode = (node ID with bit *j* swapped)
- execute vector butterfly operation (TYPE II)

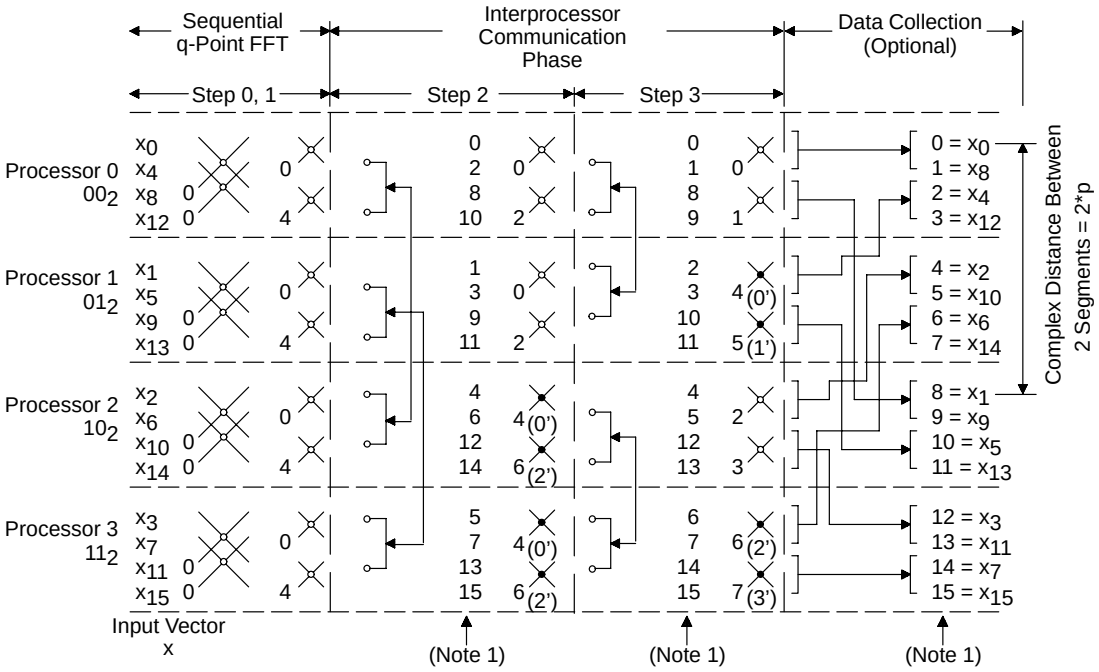
Figure 8 shows this new interprocessor communication scheme. Nonconsecutive complex numbers are exchanged between the processors. Notice that:

- Butterfly pair elements are now consecutively located.
- Each processor executes only one butterfly type at each stage. This simplifies the programming effort in trying to make the program generic.
- Based on Equation 2, a new notation has been introduced for TYPE II butterflies. It uses twiddle factor $W_n k$ and is notated by t' where t' = *k* modulo (*n*/4). This notation will be used in the rest of this documentation.

Phase 4. Data Collection (Optional):

Processor i contains *q* complex elements with ordinal positions *i*q+j*2*p and i*q+j*2*p+1* of the bit-reversed FFT vector result (*0 j ≤ q and 0 ≤ i <p*). In other words, each processor transfers consecutive pairs of complex numbers to position i**q* with a destination incremental index of 2**p*. Collected output data is in bit-reversed order. This process is illustrated in Figure 8.

Figure 8. Parallel DIT FFT Algorithm (Scheme 2)



Notes: 1) Intermediate numeration refers to ordinal values, not to input or output vector values.

2) k = W_{16}^{k}

3) Type I butterfly operation Type II butterfly operation

Partitioned 1-D FFT

Data allocation has a high impact on algorithm performance. For example, in the case of the ’C40, a program can make two data accesses to internal memory in one cycle but only one access to external memory (even with zero wait states).

This is especially important in computation-intensive algorithms like the FFT, which takes advantage of dual-access ’C40 parallel instructions [8]. The ’C40 offers 2K words of on-chip RAM that can hold up to 1k complex numbers. But for FFTs larger than 1K complex (or 2K real), the data don’t fit in on-chip RAM, and execution must be performed off-chip with the corresponding performance degradation.

For FFTs with 2K complex numbers, it is possible to compute on-chip independently a 1k-point real FFT on the real and imaginary components of the complex vector input; this solves the issue of execution on off-chip data. This approach is efficient and simple to implement, but it works only for this specific case.

A more generic solution to this problem is to apply decomposition schemes as explained in the *Parallel 1-D FFT* section. This generalization can be achieved easily: the multiprocessor environment can be replaced by looping in the code P times; the multiprocessor exchange phases are nothing more than accesses with different offsets.

Also, it’s possible to use the DMA for data I/O in a double-buffering fashion: while the CPU is working in the set of data for loop j, the DMA transfers the result from loop (j–1) and brings in the new set of data for loop (j+1). If the CPU and DMA are provided each with an independent buffer each from a separate on-chip RAM block, memory access conflict is minimized, and the DMA can run concurrently with the CPU with the corresponding cycle savings.

Scheme 2 (Figure 8) is inconvenient for DMA use. Because data transfer does not occur with the same index offset at each loop, four levels of DMA autoinitialization would be required: two to transfer the real components and two to transfer the imaginary components. Even though such implementation is possible, it will increase programming complexity and DMA transfer cycles. For this reason, a new scheme is proposed in Figure 9 for the uniprocessor case. A formal explanation follows.

Let $n=qp$, where n is the FFT size, q is the FFT size that can be executed on-chip, and $p \geq 1$ is an integer. FFT input is in normal order, and FFT output is bit-reversed.

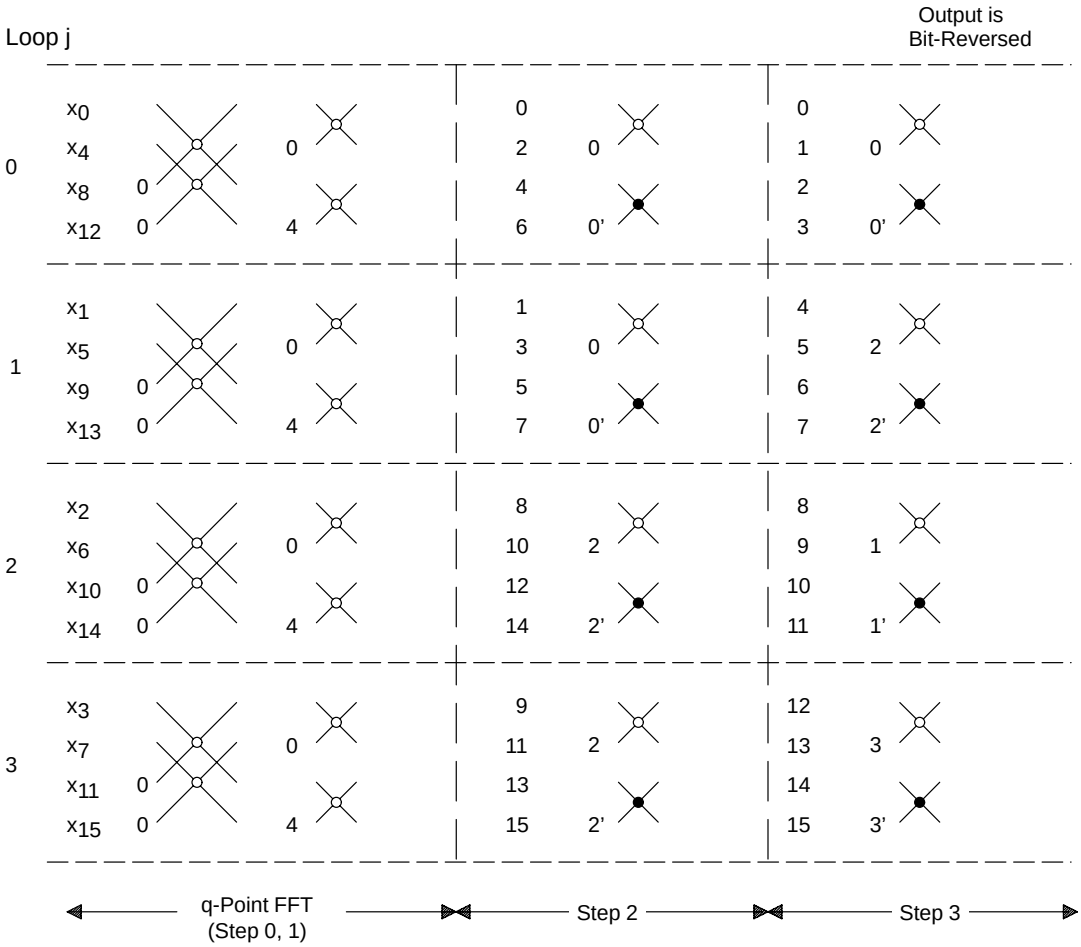
Phase 1. Execution of p q-point FFTs: elements $i + j*p$ where $0 \leq j < (n/p)$ are transferred to on-chip memory for a q-point FFT execution. The process repeats for each loop i ($0 \leq i < p$).

Phase 2. Execution of p butterfly vector operations at each of the remaining ($d = \log p$) FFT stages. Notice how the index offset between complex numbers of each input vector is constant at each stage, making it easier to implement DMA data movement. Notice also that TYPE I and II butterflies are now intercalated at each butterfly vector operation.

This scheme is not good for a parallel processing configuration, because it increases interprocessor communication delay.

The focus in this section has been on partitioned DIT FFT uniprocessor implementations because our DIT FFT core routine is substantially faster than the DIF core routine. However, partitioned DIF FFT uniprocessor implementations are feasible and even easier to implement, given the full-size sine table normally required.

Figure 9. Partitioned Uniprocessor FFT Implementation



Notes: 1) Intermediate numeration refers to ordinal values; not to input or output vector values.

2) k = W_{16}^{k}

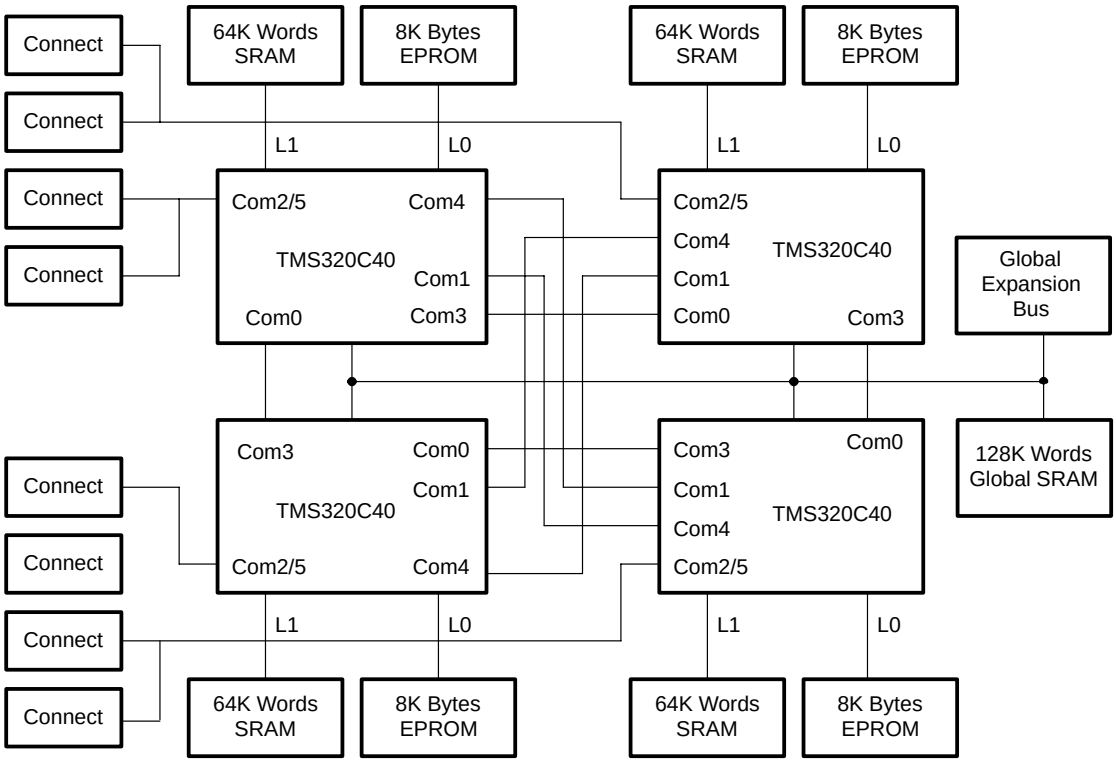
3) Type I butterfly operation Type II butterfly operation

TMS320C40 Implementation

The TMS320C40 is the world's first parallel-processing DSP. In addition to a powerful CPU with a 40- or 50-ns cycle time, the 'C40 contains six communication ports and a multichannel DMA [8]. The on-chip communication ports allow direct (glueless) processor-to-processor communication, and the DMA unit provides concurrent I/O by running parallel to the CPU. Special interlocked instructions also provide support for shared-memory arbitration. These features make the 'C40 suitable for both distributed- and shared-memory multiprocessor systems [2].

The programs presented here were tested on the TMS320C40 Parallel Processing Development System (PPDS). The PPDS includes four 'C40s, fully interconnected via the on-chip communication ports. Each 'C40 has 256KB of local memory SRAM, and all share a 512KB global memory [5]. See Figure 10.

Figure 10. TMS320C40 Parallel Processing Development System (PPDS)



Given the general scope of this application note, the programs have been written to be independent of the FFT size and the number of processors in the system. This adds to extra programming overhead. Further optimization is possible with a fixed number of processors and/or a fixed FFT size.

Appendix A and Appendix B illustrate regular serial DIF and DIT implementations, respectively, that provide comparative measures for the parallel programs.

Distributed-Memory Parallel FFT Implementations

Distributed-memory Decimation-in-Frequency (DIF) and Decimation-in-Time (DIT) parallel FFTs have been implemented. These programs can work for any FFT size and any number of processors in the system as long as $q = n/p$ stays inside the [4,1024] range (for the DIF case) and [32,1024] for the DIT case. The lower limit is due to programming specifics, and 1024 is the limit for the 2K-word ’C40 on-chip RAM. If a $q > 1024$ is required, the regular r2dif/r2dit FFT routines could be replaced by partioned FFT versions like the program presented in Appendix D (with some modifications).

Interprocessor Connections

For this parallel 1–D FFT implementation, a hypercube network is required (of course the programs can also run in a fully connected network as the PPDS). As explained in the *Parallel 1-D FFT* section, a hypercube is characterized not only by a specific comm port connectivity but also by a specific node ID allocation. In the case of the PPDS, this node ID allocation should be followed:

CPU_A = 0 CPU_B = 1 CPU_C = 3 CPU_D = 2

Node IDs can be allocated via emulator commands (e my_node = xxx) or via a predefined local memory location that can be read using the my_id function available in the parallel-runtime support library (PRTS40.LIB) available with the ’C40 compiler version 4.5 or higher.

Interprocessor Communications

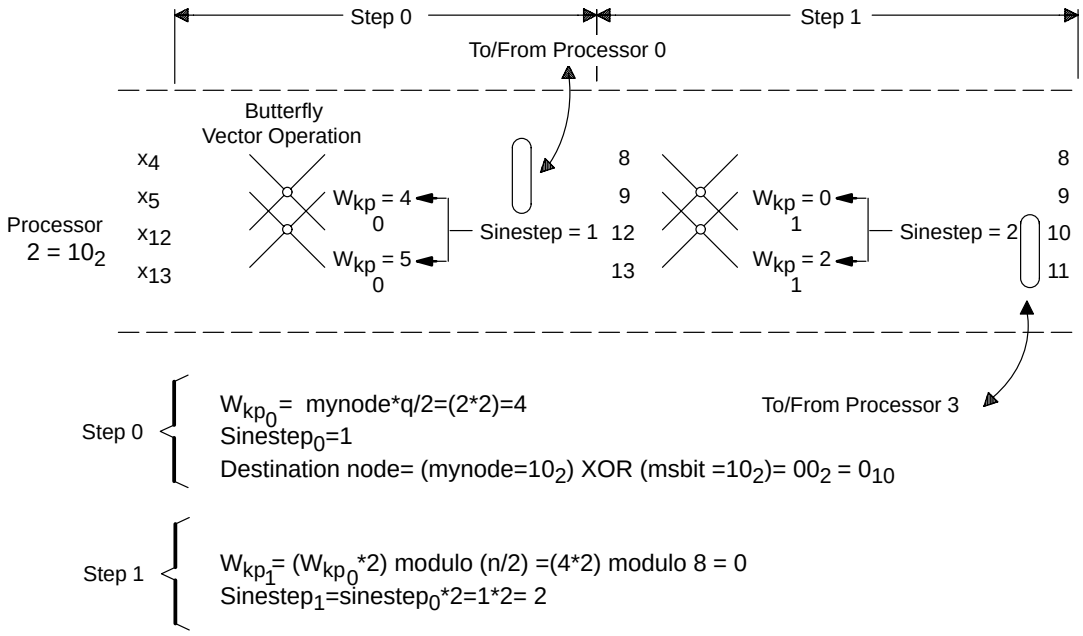
This implementation uses the CPU for interprocessor communication, even though the DMA could also be used. Because the CPU must wait for the interprocessor communication to finish before executing the next butterfly vector step, no real advantage is observed in using the DMA for data transfer. Aykanat and Dervis [11] have proposed a scheme to overlap half of the butterfly vector computation with the interprocessor communication, giving an average of 10% improvement for matrices larger than 10K points, but this almost doubles the program size. In this paper, the goal is to present a workable moderate-size parallel FFT implementation; for this reason, their approach was not used.

A) Decimation-in-Frequency (DIF) FFT

Appendix C contains the source code for the ’C40 parallel DIF FFT implementation. The Radix-2 assembly language complex DIF FFT implementation shown in Appendix F has been used as the FFT core routine. Real and imaginary components of the input data are stored in consecutive memory locations. The size of the sine table is 5*FFT_SIZE/4.

Figure 11 shows more detail of the interprocessor communication phase of DIF FFT for processor 2. These details refer to specifics of the C source implementation in Appendix C and the general diagram shown in Figure 5.

Figure 11. Interprocessor Communication Phase for DIF FFT (Processor 2)



Note: W_{kp_i}= *Wkpointer* value for step i;
$Sinestep_i$ = *sinestep* value for step i.

Notice that Phase 2 (serial *q*-point FFT execution) requires a sine table of size 5**q*/4 instead of a size 5**n*/4. The 5**q*/4 elements are part of the 5**n*/4 sine table but are not consecutively located (offset = *p*).

Two approaches can solve this issue:

1. Have an extra "move" operation to transfer the twiddle factors required for a *q*-point FFT into consecutive memory locations, or
2. Modify the assembly language FFT routine to access the twiddle factors with an offset = *p* instead of with an offset = 1.

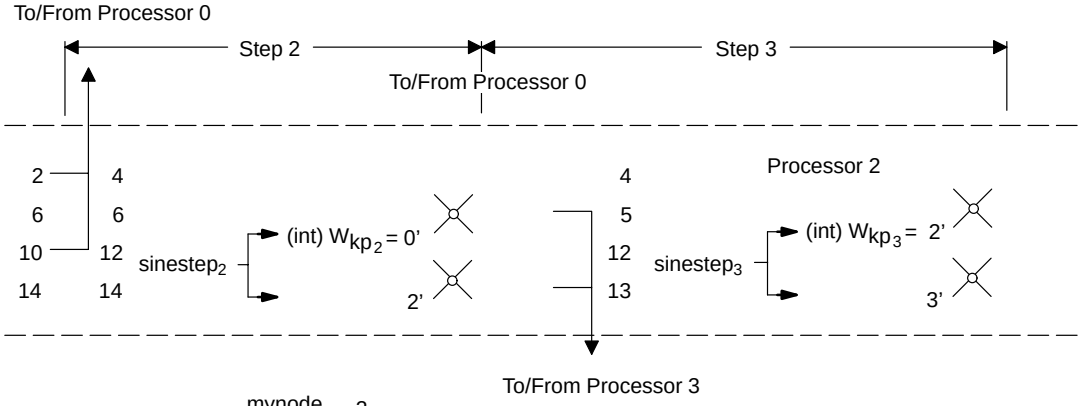
Either approach can be selected by changing the version number in DIS_DIF.C (Appendix C). The second approach (VERSION =1) is faster but requires the modification of the FFT core function. If you plan to use your own FFT routine but don't want to enter into the specifics of the parallel modification, the first approach offers a good solution.

B) Decimation-in-Time (DIT) FFT

Appendix D contains the source code for the 'C40 parallel DIT FFT implementation (Scheme 2). The Radix-2 assembly language complex DIT FFT implementation shown in Appendix F (Meyer-Schwarz FFT) has been used as the core routine. Real and imaginary components of the input data are stored in consecutive memory locations. Even though the code is larger than in the DIF core routine, Meyer-Schwarz FFT outperforms the DIF implementation in execution time and also offers a reduced-size sine table (bit-reversed).

Figure 12 shows more detail of the interprocessor communication phase of DIT FFT for processor 2. These details refer to specifics of the C source implementation in Appendix D and to the general diagram shown in Figure 8.

Figure 12. Interprocessor Communication Phase for DIT FFT (Processor 2)



Step 2

$$W_{kp_2} = \frac{mynode}{p} = \frac{2}{4} = 0.5$$

(int) $W_{kp_2} = 0$

$sinestep_2 = 2$ (complex distance =1)

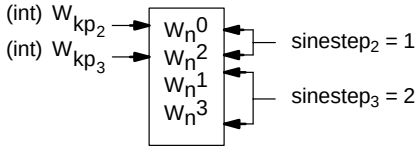
Step 3

$$W_{kp_3} = W_{kp_2} \times 2 = 0.5 \times 2 = 1$$

(int) $W_{kp_3} = 1$

$sinestep_3 = 4$ (complex distance =2)

Notes: 1) The sine table is bit reversed.



2) W_{kp_i} = Wkpointer value for step i

$Sinestep_i$ = sinestep value for step i.

The sine table is stored in bit-reversed order and with a table length of $n/2$ (n = FFT length). The table can be used for all the FFT lengths less than or equal to n. Therefore, no extra "move" operation is required to compact the sine table of a size n FFT so that it will work for a size q=(n/p) FFT.

Partitioned FFT Uniprocessor Implementation

Single buffering and double buffering have been implemented:

***Single buffering*:** For an FFT size larger than 1K point, complex data is partitioned in 1K-complex size blocks (q=1K). Initial data is in external SRAM and is transferred to on-chip RAM (0x002F F800) on blocks of size q for CPU processing.

Appendix E shows single-buffered partitioned DIT FFT implementations. Both programs (serp1.c and serp2.c) are functionally identical, but serp1.c is faster because it avoids integer divisions and moduli operations, which are costly when programming in C.

***Double buffering*:** FFT size is partitioned in 512K-complex point size blocks (q=512K). The 2K $\times$ 32-bit-word on-chip RAM constantly holds 2 buffers. Each buffer is located in a different on-chip RAM block to minimize CPU/DMA access conflict. One of the buffers is used for CPU arithmetic operations, while the other is used as source/destination address for DMA I/O operation.

While the CPU is executing one butterfly vector operation (step j) in one of the on-chip RAM blocks, the DMA transfers back the results from the previous butterfly operation (step (j–1)) to off-chip memory and brings a new set of data (step (j+1)) to the other on-chip RAM block. DMA autoinitialization is used for this purpose.

While the CPU waits for DMA to finish, it checks whether the corresponding IIF (internal interrupt flag) bit is set to 1 (DMA control register TCC (transfer counter control) bit has been set to 1). Another way of checking whether a unified DMA operation has completed is to check whether the DMA control register start bits are equal to 10_2. This is easier to implement from a C program because DMA registers are memory mapped, but the IIF checking method is preferred because it avoids DMA/CPU conflict when the 'C40 peripheral bus is accessed. It's important to remember that DMA uses the peripheral bus during autoinitialization because autoinitialization is nothing more than a regular DMA transfer operation.

A partitioned implementation of very large real FFTs for the 'C3x generation, using the same partitioning scheme explained in this application note, can be obtained from the DSP Bulletin Board Service (BBS) at (713) 274–2323, or by anonymous ftp from ti.com (Internet port address 192.94.94.1).

Results and Conclusions

Table 1 shows the 'C40 1-D FFT timing benchmarks taken in the 'C40 PPDS and using the 'C40 C compiler (version 4.5) with full optimization and registers for parameter passing. The compiler/assembler tools were run under OS/2 to avoid memory limitation problems with the optimizer, but the DOS extended-memory manager can also be used. Table 1 shows the FFT benchmark results.

Table 1. FFT Timing Benchmarks (in Milliseconds)

Type	Program	Number of Points							
		64	128	256	512	1K	2K	4K	8K
DIF	fft1.c	0.095	0.21	0.467	1.03	2.259	9.181	19.994	43.26
	dis_dif (version 0; p=2)	0.078	0.158	0.332	0.703	1.497	3.187	—	—
	dis_dif (version 1; p=2)	0.071	0.145	0.305	0.651	1.394	2.981	—	—
	dis_dif (version 0; p=4)	0.06	0.108	0.211	0.44	0.89	1.863	3.911	—
	dis_dif (version 1; p=4)	0.055	0.101	0.197	0.413	0.859	1.76	3.705	—
DIT	fft2.c	0.064	0.141	0.315	0.703	1.562	8.373	18.378	40.026
	dis_dit (p=2)	0.062	0.111	0.249	0.526	1.037	2.224	—	—
	dis_dit (p=4)	—	0.089	0.179	0.359	0.738	1.454	3.051	—
SERP	serp1	—	—	—	—	—	4.758	11.137	26.153
	serp2	—	—	—	—	—	5.65	14.228	34.319
	serpb	—	—	—	—	—	9.173	11.401	26.615

Note: 'C40 cycle time = 40 ns.

Benchmarking Considerations

1. To achieve precise benchmark measurements, a common global start of the processors is required. This feature is offered by the parallel debugger controller available with the 'C4x XDS510 emulator (version 2.20 or higher). Also a shared-memory synchronization counter can be used [2].
2. The 'C40 timer 0 and the timer routines in the parallel runtime support library (PRTS40.LIB) are used for benchmark measures. The real benchmark timing is equal to the timer counter value multiplied by 2*('C40 cycle time). For the parallel programs, the total execution time of the parallel algorithm can be defined as T= max(Ti), where Ti is the execution time required by processor i.
3. Data distribution and collection have not been included in the benchmark timings, because they are system specific. In our case, we have used the PPDS shared memory for initial data distribution/collection, but this is not the general case. It's also important to notice that data movement is not a significant timing factor in the overall algorithm execution. Data movement of n numbers is an O(n) operation [7]. As it was explained before, FFT computation is an O($n$log$n$) operation. For large n, data movement time becomes negligible. Also, DMA can be used for data distribution/collection, leaving the CPU free for some other computation in a customer-specific application. Careful analysis is required to minimize CPU/DMA memory access conflict.

PPDS Considerations

The ’C40 PPDS is a general-purpose parallel processing board. It is not optimized for distributed-memory-only type of applications, because it dedicates one of the ’C40 external buses for shared-memory interfacing. You can expect further performance improvement if you use a board that offers ’C40s with local memory in both external buses. This type of architecture takes advantage of the ’C40 dual bus architecture and reduces the memory access conflict that could exist among instruction fetches, data access, and/or DMA accesses. Modification of the linker command files for the programs is recommended to take advantage of the ’C40’s good I/O bandwidth. Allocation of program and data into different external buses is also recommended.

For the reasons explained above, in this PPDS implementation, all the linker command file sections have been allocated to the primary external bus with on-chip RAM reserved for FFT computation. This has been assumed for all the programs in order to establish a fair comparison between parallel and sequential implementations. The only exception is the regular sequential implementation of very large FFTs ($n > 1K$), in which case, most of the sections are allocated on-chip, except for the input data and sine table. See Appendix **A**.

DIF Vs. DIT Implementation

DIT implementations outperformed DIF implementations because of a faster Meyer-Schwarz complex DIT FFT core routine. The Meyer-Schwarz DIT FFT routine offers faster execution time and a reduced-size sine table at the expense of a more complex and larger implementation code. For medium and large FFTs, the overall trade-off is very positive.

Speed-Up/Efficiency Analysis

Speed-up of a parallel algorithm is defined as Sp = Ts/Tp, where Ts is the serial time (p=1) and Tp is the time of the algorithm executed using *p* processors [2]. In this application note, Ts is the execution time for programs in Appendix A. Figure 13 shows speed-up vs FFT size for the parallel 1-D FFT programs. Note that the definition of speed-up has been also applied to the partitioned serial implementation in order to have a global comparative measure.

An even more meaningful measure is efficiency defined as Ep = Sp/p with values between (0,1). Efficiency is a measure of processor utilization in terms of cost efficiency. If the efficiency is lower than 0.5, it is often better to use fewer processors because using more processors is not cost effective. Figure 14 shows efficiency versus FFT size for the parallel 1-D FFT programs.

Figure 13. FFT Speed-Up Vs. FFT Size

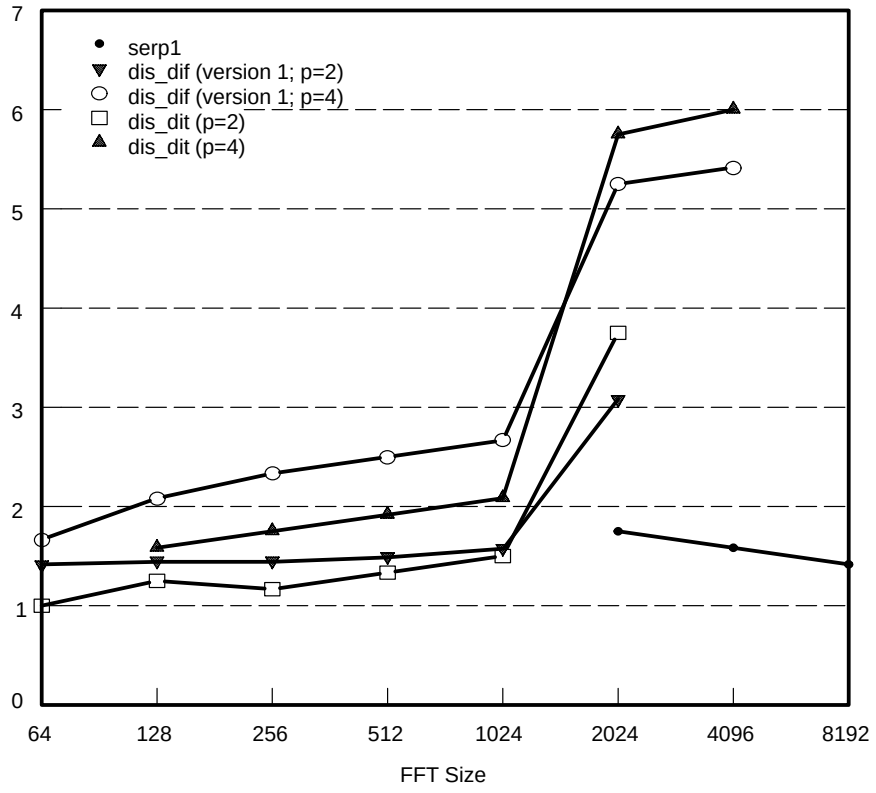
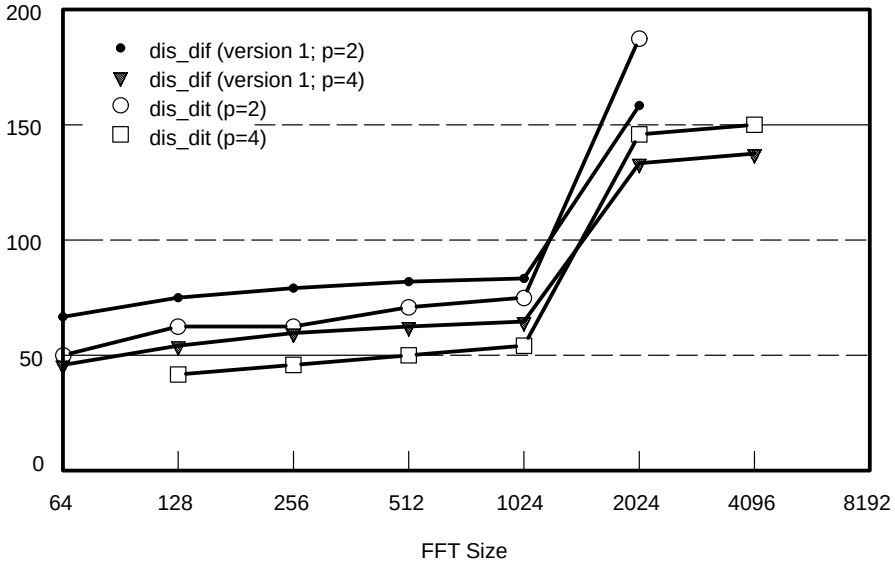


Figure 14. FFT Efficiency Vs. FFT Size



Analysis of the Results

- Speed-up is proportional to the number of processors used. However, efficiency decreases when the number of processors increases. This is normal in algorithms like the parallel FFT, which requires interprocessor communication to solve data dependencies, because eventually the communication overhead begins to dominate. Based on this, it's better to use the minimum number of processors that can still give the speed-up required.
- Parallel 1-D FFT performance improves for larger FFT size because it becomes more computationally intensive, reducing proportionally the programming overhead.
- Notice also that efficiency figures for large FFTs (complex FFT size > 1K point) go above 100%, the theoretical maximum efficiency. This extra efficiency does not come from the parallelization itself, but from savings in on-chip data execution, as explained before.
- Partitioned 1-D FFT serial implementation (complex FFT size > 1K point) shows a speed-up close to 1.8 that slightly declines as the FFT size increases. This performance improvement is due to execution of data on-chip: having the input data on-chip permits access of two data in one cycle in some 'C40 parallel instructions (in external memory will take two cycles at least). Notice also that serp1.c was considerable faster than serp2.c because of savings in the division and moduli operations.
- There was a 15 to 20% improvement using double buffering in the partitioned serial program serpb.c compared with serp2.c (the equivalent single-buffered implementation), but not with respect to serp1.c. For specific customer applications (fixed number of processors and/or FFT size), a better performance of serpb.c is expected. Also, extra performance can be obtained with a 'C40 board with dual-bus architecture because it minimizes CPU/DMA memory access conflict.

Conclusions

This application note has illustrated decomposition methods to partition the FFT algorithm in smaller FFT transforms. This is particularly useful in uniprocessor implementations of very large FFTs (> 1K point complex) or in systems where multiple processors are used for speed-up gain.

The source code and its linker command files are presented in the appendices, but they can also be downloaded from the Texas Instruments DSP Bulletin Board at (713) 274–2323 and via anonymous ftp from ti.com (Internet port address 192.94.94.1).

References

[1] Hwang, K., and F. A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[2] Piedra, R. M. *Parallel 2-D FFT Implementation with TMS320C4x DSPs*. Texas Instruments, 1991.

[3] Burrus, C. S., and T. W. Parks. *DFT/FFT and Convolution Algorithms*. New York: John Wiley and Sons, 1985.

[4] Papamichalis, P. An Implementation of FFT, DCT, and Other Transforms on the TMS320C30. *Digital Signal Processing Applications With the TMS320 Family*, Volume 3, page 53. Texas Instruments, 1990.

[5] Chen, D.C., and R. H. Price. A Real-Time TMS320C40 Based Parallel System for High Rate Digital Signal Processing. *ICASSP91 Proceedings*, May 1991.

[6] Oppenheim, A. V., and R. W. Schafer. *Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[7] Akl, S. G. *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs, New Jersey: Prentice-Hall, 1989, page 171.

[8] *TMS320C4x User's Guide*, Texas Instruments, Inc., 1991.

[9] Bertsekas, D. P., and J. N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods*. Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

[10] Kung, S. Y. *VLSI Array Processors*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[11] Aykanat, C., and A. Dervis. An Overlapped FFT Algorithm for Hypercube Multicomputers. *ICPP91 Proceedings*, August 1991.

[12] Zhu, J. P. An Efficient FFT Algorithm on Multiprocessors With Distributed-Memory. *The Fifth Distributed-Memory Computing Conference*, Vol 1, 358–363. January 1990.

[13] Solowiejczk, Y., and J. Petzinger. Large 1-D Fast Fourier Transforms on a Shared-Memory System. *ICPP91 Proceedings*, August 1991.

Appendices

Appendix A: Uniprocessor 1-D DIF FFT Implementation

- fft1.c: 1-D DIF FFT implementation
- fftl2k.cmd: linker command file for FFT < 2K
- fftg2k.cmd: linker command file for FFT ≥ 2K

Appendix B: Uniprocesor 1-D DIT FFT Implementation

- fft2.c: 1-D DIT FFT implementation

Appendix C: Parallel 1-D DIF FFT Multiprocessor Implementation

- dis_dif.c
- dis.cmd: linker command file

Appendix D: Parallel 1-D DIT FFT Multiprocessor Implementation

- dis_dit.c

Appendix E: Partitioned 1-D DIT FFT Uniprocessor Implementation

- Single-buffered implementations
  - serp1.c
  - serp2.c
  - serp.cmd: linker command file
- Double-buffered implementation
  - serpb.c

Appendix F: Library Routines (PFFT.LIB)

- bfly.asm: butterfly vector operation (type I)
- blfyr.asm: butterfly vector operation (type I&II)
- bflyr1.asm: butterfly vector operation (type I)
- bflyr2.asm: butterfly vector operation (type II)
- cmove.asm: complex numbers move operation
- exch_r.asm: interprocessor communication routine
- move.asm: real numbers move operation
- pr2dif.asm: radix-2 complex DIF routine for par FFT
- r2dif.asm: radix-2 complex DIF routine
- r2dit.asm: complex DIT routine
- waitdma.asm: routine that waits until DMA finishes

Appendix G: Input Vector and Sine Table Examples

- sintab.asm: sine table for a 64-point DIF FFT
- sintabr.asm: sine table for a 64-point DIT FFT
- input.asm: 64-point complex input vector

Appendix A: Uniprocessor 1-D DIF FFT Implementation

FFT1.C

```
/************************************************************************
  FFT1.C : Serial FFT DIF implementation
************************************************************************/
#define    N           64                  /* FFT size (n)                */
#define    LOGN         6                  /* number of rows              */
extern     void r2dif();                   /* C-callable complex DIF FFT  */
extern     float INPUT[];                  /* input  vector               */
float      *shinput    = INPUT;
int        i;
int        tcomp;                          /* for benchmarking            */
/************************************************************************/
main()
{
start:
asm(" or 1800h,st");                      /* cache enable                */
time_start(0);                            /* start timer 0 for benchmark */
r2dif (shinput,N,LOGN);                   /* FFT computation             */
tcomp= time_read(0);                      /* tcomp = execution time      */
} /*main*/
```

FFTL2K.CMD

```
/*linear command file for FFT size < 2K                                  */
-c                                  /* link using C conventions        */
fft.obj                             /* FFT C code                      */
sintab.obj                          /* sine table                      */
input.obj                           /* input data                      */
-lpfftr.lib                         /* get FFT Assembly code routine   */
-stack 0x0040                       /* set stack size                  */
-lrts40r.lib                        /* get run-time support            */
-lprts40r.lib                       /* get timer routines              */
-m fft.map                          /* generate map file               */
-o fft.out                          /* output file name                */
MEMORY
{
   ROM:  org = 0x00       len = 0x1000   /* on-chip ROM                 */
   RAM0: org = 0x002ff800 len = 0x0800   /* on-chip RAM: 2 blocks       */
   LM:   org = 0x40000000 len = 0x10000  /* local memory                */
   GM:   org = 0x80000000 len = 0x20000  /* global memory               */
}
SECTIONS
{
   .input:   {} > RAM0                   /* input vector                */
   .sintab:  {} > LM                     /* sine table                  */
   .ffttext: {} > LM                     /* FFT assembly routine (.text)*/
   .text:    {} > LM                     /* FFT C code (.text)          */
   .cinit:   {} > LM                     /* initialization table        */
   .stack:   {} > LM                     /* system stack                */
   .bss :    {} > LM                     /* global & static C variables */
   .fftdata: {} > LM                     /* FFT assembly routine (.text)*/
}
```

FFTG2K.CMD

```
/*linear command file for FFT size ≥ 2K                                  */
-c                                      /* link using C conventions      */
fft.obj                                 /* FFT C code                    */
sintab.obj                              /* sine table                    */
input.obj                               /* input data                    */
-lpfftr.lib                             /* get FFT Assembly code routine */
-stack 0x0040                           /* set stack size                */
-lrts40r.lib                            /* get run-time support          */
-lprts40r.lib                           /* get timer routines            */
-m fft.map                              /* generate map file             */
-o fft.out                              /* output file name              */
MEMORY
{
   ROM:  org = 0x00       len = 0x1000   /* on-chip ROM                  */
   RAM0: org = 0x002ff800 len = 0x0800   /* on-chip RAM: 2 blocks        */
   LM:   org = 0x40000000 len = 0x10000  /* local memory                 */
   GM:   org = 0x80000000 len = 0x20000  /* global memory                */
}
SECTIONS
{
   .input:   {} > LM                     /* input vector                 */
   .sintab:  {} > LM                     /* sine table                   */
   .ffttext: {} > RAM0                   /* FFT assembly routine (.text) */
   .text:    {} > RAM0                   /* FFT C code (.text)           */
   .cinit:   {} > RAM0                   /* initialization table         */
   .stack:   {} > RAM0                   /* system stack                 */
   .bss :    {} > RAM0                   /* global & static C variables  */
   .fftdata: {} > RAM0                   /* FFT assembly routine (.text) */
}
```

Appendix B: Uniprocessor 1-D DIT FFT Implementation

FFT2.C

```
/*********************************************************************
  FFT2.C : Serial DIT FFT implementation
*********************************************************************/
#define    N           64                  /* FFT size (n)               */
#define    LOGN        6                   /* number of rows             */
extern     void r2dit(),                   /* C–callable complex DIT FFT */
               cmove();                    /* CPU complex move           */
extern     float INPUT[];                  /* input  vector              */
float      *shinput    = INPUT;
int    i;
int        tcomp;                          /* for benchmarking           */
/*********************************************************************/
main()
{
start:
asm (" or 1800h,st");                    /* cache enable               */
time_start(0);                           /* start timer 0 for benchmark */
r2dit (shinput,N);                       /* FFT computation            */
tcomp= time_read(0);                     /* tcomp = execution time     */
} /*main*/
```

Appendix C: Parallel 1-D DIF FFT Multiprocessor Implementation

DIS_DIF.C

```
/*************************************************************************
  DIS_DIF.C :  Distributed–memory Parallel DIF FFT implementation.
               * if VERSION = 0 this program uses an extra ”move” operation
               of the sine table to avoid modification of the serial FFT
               core (r2dif.asm)
               * if VERSION = 1 this program uses pr2dif.asm as the FFT
               core routine. pr2dif.asm is a slightly modified version of
               r2dif.asm that enables a serial FFT program (size q FFT)
               to work with a sine table of size 5*n/4, where n=q*p
  Requirements: 4 <= Q <= 1024 ( minimum: because of cmove.asm requirements
                                 maximum: because of on–chip RAM limitations)
  Network topology : Hypercube
  Version : 1.0
*************************************************************************
 VERSION        DATE            COMMENT
   1.0          8/92            Original version
                                ROSEMARIE PIEDRA (TI Houston)
*************************************************************************/
#define    VERSION     1
#define    N           64                  /* FFT size (n)               */
#define    M           6                   /* Log (FFT size)             */
#define    P           2                   /* Number of processors       */
#define    D           1                   /* Log P= hypercube dimension */
#define    Q           N/P                 /* elements per processor     */
#define    LOGQ        M–D                 /* number of serial stages    */
#define    BLOCK0      0x002ff800          /* on–chip RAM buffer         */
extern     void r2dif(),                   /* C–callable complex DIF FFT */
                cmove(),                   /* CPU complex move           */
                bfly(),                    /* butterfly vector routine   */
                exchange_r();              /* interprocessor communication */
extern     float INPUT[],                  /* global input data          */
                  SINE[];                  /* sine table of size 5*N/4   */
float       *input      = (float *)BLOCK0, /* pointer to on–chip RAM     */
            *inputq     = (float *)BLOCK0+Q,
            *shinput    = INPUT;
unsigned int  n2        = N/2,             /* FFTSIZE /2                 */
              q         = Q,
              q2        = Q/2,
              q54       = 5*Q/4,
              msbit     = 1 << (D–1),      /* ”1” in most significant bit
                                              of processor id           */
         sinestep     = 1,                /* initial distance between
                                              twiddle factors of succesive
                                              butterflies                */
           my_node,dnode,comport,i,Wkpointer,sinestep;
/* Connectivity matrix : processor i is connected to processor j through
   port port[i][j]                                                       */
#if        (P==4)
int        port[P][P] = { –1,0,3,–1,
                          3,–1,–1,0,
                          0,–1,–1,3,
                          –1,3,0,–1 };
#else
int        port[P][P] = { –1,0,3,–1 };
#endif
int        tcomp;                          /* benchmarking               */
/*************************************************************************/
main()
{                                          /* cache enable               */
asm (” or 1800h,st”);
/*************************************************************************
Data distribution simulation: processor ”my_node” contains complex elements:
(my_node* Q/2)+i
(my_node* Q/2)+N/2+i    where  0<=i<Q/2
This part is optional: data distribution is system specific
*************************************************************************/
```

```
  cmove (shinput+my_node*q,input,2,2,q2);     /* move first  segment    */
  cmove (shinput+my_node*q+N,inputq,2,2,q2);  /* move second segment    */
/*************************************
 D = LOG P exchange communication steps *
*************************************/
start:
  time_start(0);                           /* start timer 0 for benchmark */
  dnode    = my_node ^ msbit;              /* select destination node     */
  comport  = port[my_node][dnode];         /* get comport to be used      */
  Wkpointer = my_node*q2;                  /* initialize offset from _SINE
                                              to first  twiddle factor    */
  for (i=0;i<D-1;i++) {                    /* loop D-1 times              */
      bfly (input,q,Wkpointer,N,sinestep); /* Butterfly vector  operation
                                              on a q-point complex input vector
                                              using twiddle factors pointed by
                                              Wkpointer with a twiddle factor
                                              offset distance = sinestep  */
      /* interprocessor data exchange:  send/receive succesive (real offset = 2)
         q complex numbers to comm port                                    */
      if (my_node & msbit) exchange_r(comport,input,q2,2);
      else                 exchange_r(comport,inputq,q2,2);
      /* parameter updates for next loop:                                  */
                                           /* twidle factor pointer :
                                            Wkpointer=(Wkpointer*2) modulo (n/2)
                                           */
      Wkpointer *= 2;
      if (Wkpointer >= n2) Wkpointer -= n2; /* substraction is faster than
                                               modulo operation           */
      msbit    >>= 1;                      /* right shift of bit selector
                                              for destination node selection */
      dnode    = my_node ^ msbit;          /* next destination node       */
      comport  = port[my_node][dnode];     /* comm port attached to dnode */
      sinestep <<= 1;                      /* distance between twiddle factors
                                              used in succesive butterflies
                                              doubles at each  stage      */
  };
  /* last loop: parameter update operations are not needed                */
  bfly (input,q,Wkpointer,N,sinestep);
  if (my_node & msbit) exchange_r(comport,input,q2,2);
     else              exchange_r(comport,inputq,q2,2);
/*********************************
Serial FFT of size Q         *
*********************************/
#if (VERSION == 0)
  move  (SINE,SINE,P,1,q54);             /* modify a size-N FFT sine table
                                            to a size Q-FFT sine table   */
  r2dif (input,q,LOGQ);                  /* regular serial DIF FFT routine */
#else
  pr2dif (input,q,LOGQ,P);               /* special FFT routine         */
#endif
/*************************************************************************
Data collection simulation: output in PPDS shared-memory is in bit-reversed
order. This part is optional: data collection is system specific
*************************************************************************/
  tcomp = time_read(0);                  /* Benchmarking                */
  cmove (input,shinput+my_node*q*2,2,2,q);
} /*main*/
```

DIS.CMD

```
-c                                      /* link using C conventions     */
dis.obj                                 /* FFT C code                   */
sintab.obj                              /* sine table                   */
input.obj                               /* input data                   */
-lpfftr.lib                             /* app. note library            */
-stack 0x0040                           /* set stack size               */
-lrts40r.lib                            /* get run-time support         */
-lprts40r.lib                           /* get timer routines           */
-m dis.map                              /* generate map file            */
-o dis.out                              /* output filename              */
MEMORY
{
   ROM:  org = 0x00       len = 0x1000  /* on-chip ROM                  */
   RAM0: org = 0x002ff800 len = 0x0800  /* on-chip RAM: 2 blocks        */
   LM:   org = 0x40000000 len = 0x10000 /* local memory                 */
   GM:   org = 0x80000000 len = 0x20000 /* global memory                */
}
SECTIONS
{
   .sintab: {} > LM                        /* SINE TABLE           */
   .ffttext:{} > LM                        /* FFT CODE             */
   .text:   {} > LM                        /* CODE                 */
   .cinit:  {} > LM                        /* INITIALIZATION TABLES*/
   .stack:  {} > LM                        /* SYSTEM STACK         */
   .bss :   {} > LM                        /* GLOBAL & STATIC VARS */
   .fftdata:{} > LM                        /* FFT DATA             */
   .input:  {} > GM                        /* INPUT VECTOR         */
}
/*
NOTE: On-chip RAM has been totally reserved for FFT execution.
       If Complex FFT size < 1K , some of the sections could be allocated
       in on-chip RAM.
*/
```

Appendix D: Parallel 1-D DIT FFT Multiprocessor Implementation

DIS_DIT.C

```
/*************************************************************************
  DIS_DIT.C : Distributed-memory Parallel DIT FFT implementation.
  Requirements: 32<= Q <=1024 (minimum: because of Meyer-Schwarz FFT limitations
                               maximum: because of on-chip RAM limitations)
  Network topology : Hypercube
  Version : 1.0
*************************************************************************
 VERSION        DATE            COMMENT
   1.0          8/92            Original version
                                ROSEMARIE PIEDRA (TI Houston)
*************************************************************************/
#define    N           64              /* FFT size (n)                      */
#define    M           6               /* Log (FFT size)                    */
#define    P           2               /* Number of processors              */
#define    D           1               /* Log P= hypercube dimension        */
#define    Q           N/P             /* elements per processor            */
#define    LOGQ        M-D             /* number of serial stages           */
#define    BLOCK0      0x002ff800       /* on-chip RAM buffer                */
extern    void r2dif(),                 /* C-callable complex DIF FFT        */
               cmove(),                /* CPU complex move                  */
               bflyr1(),               /* butterfly vector routine          */
               bflyr2(),               /* butterfly vector routine          */
               exchange_r();           /* interprocessor communication      */
extern     float INPUT[];              /* global input data                 */
float      *input       = (float *)BLOCK0, /* pointer to on-chip RAM         */
           *inputp2     = (float *)(BLOCK0+2),
           *shinput     = INPUT,
           p            = P,
           Wkpointer;
unsigned int  n2        = N/2,          /* FFTSIZE /2                        */
              q2        = Q/2,          /* FFTSIZE/(2*P)                     */
              q         = Q,
              msbit     = 1 << (D-1),   /* "1" in msbit of processor id      */
              sinestep  = 2,            /* initialize twiddle factor distance
                                           between succesive butterflies     */
              my_node,dnode,comport,i;
/* Connectivity matrix : processor i is connected to processor j through
   port port[i][j]                                                         */
#if        (P==4)
int        port[P][P]   = { -1,0,3,-1,
                            3,-1,-1,0,
                            0,-1,-1,3,
                            -1,3,0,-1 };
#else
int        port[P][P]   = { -1,0,3,-1 };
#endif
int        tcomp;                        /* benchmarking                      */
/*************************************************************************/
main()
{
asm(" or 1800h,st");
/*****************************************************************
Data distribution simulation: processor "my_node" contains elements
my_node +i*P      where 0<=i<Q
This part is optional: data distribution is system specific
*****************************************************************/
  cmove (shinput+my_node*2,input,2*P,2,q);
/*********************************
Serial FFT of size Q          *
*********************************/
start:
  time_start(0);                        /* start timer 0 for benchmark       */
  r2dit (input,q);
/*************************************
 D = LOG P communication steps         *
*************************************/
```

```
  dnode    = my_node ^ msbit;           /* select destination node           */
  comport  = port[my_node][dnode];      /* get comport to be used            */
  Wkpointer = my_node/p;                /* initialize offset from _SINE
                                           to first twiddle factor           */
  for (i=0;i<D-1;i++) {                 /* loop  D-1 times                   */
     /*
        exchange_r: interprocessor data exchange: send/receive q/2 complex
                    numbers  to com port
        bflyr2/bflyr1:  butterfly vector operation (type I/type II) on a q-point
        complex input vector using twiddle factors  pointed by Wkpointer with
        a twiddle factor offset distance = sinestep
     */
      if (my_node & msbit) {
         exchange_r(comport,input,q2,4);
         bflyr2(input,q2,(int)Wkpointer,sinestep); /* butterfly type II       */
         }
      else {
         exchange_r(comport,inputp2,q2,4);
         bflyr1(input,q2,(int)Wkpointer,sinestep); /* butterfly type I        */
         }
      /* parameters update for next loop                                     */
      msbit     >>= 1;                  /* right shift of bit selector for
                                           destination node selection        */
      dnode     = my_node ^ msbit;      /* next destination node             */
      comport   = port[my_node][dnode]; /* comm port attached to dnode       */
      Wkpointer *= 2;                   /* twiddle factor pointer update     */
      sinestep  <<= 1;                  /* distance between twiddle factors
                                           used in succesive butterflies doubles
                                           at each stage                     */
  };
  /* last loop: parameter update operations are not needed                   */
  if (my_node & msbit) {
     exchange_r(comport,input,q2,4);
     bflyr2(input,q2,(int)Wkpointer,sinestep); /* butterfly type II           */
     }
  else {
     exchange_r(comport,inputp2,q2,4);
     bflyr1(input,q2,(int)Wkpointer,sinestep); /* butterfly type I            */
     }
/*************************************************************************
Data collection simulation: output in shared-memory is in bit-reversed order.
This part is optional: data collection is system specific
*************************************************************************/
  tcomp = time_read(0);                  /* benchmarking                     */
  cmove (input,shinput+my_node*4,4,4*P,q2);
  cmove (inputp2,shinput+my_node*4+2,4,4*P,q2);
} /*main*/
```

Appendix E: Partitioned 1-D DIT FFT Uniprocessor Implementation

SERP1.C

```
/************************************************************************
  SERP1.C : Partitioned serial DIT FFT implementation(Single-buffered version)
            (This version uses the same partitioning scheme as serp2.c but
            provides cycle savings by avoiding integer divisions)
  Requirements: 32<= Q <= 1024 (minimum: because of Meyer-Schwarz FFT limitations
                                maximum: because of on-chip RAM limitations)
  Version : 1.0
************************************************************************
 VERSION        DATE            COMMENT
   1.0          8/92            Original version
                                ROSEMARIE PIEDRA (TI Houston)
************************************************************************/
#define    N           2048                /* FFT size (n)               */
#define    P           2                   /* P = N/Q                    */
#define    D           1                   /* LOG P                      */
#define    Q           N/P                 /* maximum FFT size that can
                                              be computed on-chip        */
#define    BLOCK0      0x002ff800          /* on-chip RAM buffer         */
extern      void r2dit(),                  /* C-callable complex FFT     */
                 cmove(),                  /* CPU complex move           */
                 cmoveb();                 /* CPU bit-reversed complex move */
extern      float INPUT[];                 /* Input vector = N = Q * P   */
float       *input      = (float *)BLOCK0, /* on-chip RAM                */
            *shinput    = INPUT,
            *src_addr   = INPUT;
unsigned int i,j,k,
             delta      = P,
             ngroup     = 2,
             incr_group = N,
             p2         = 2*P,
             Wkpointer  = 0,
             q          = Q,
             q2         = Q/2;
int         tcomp;                         /*  benchmarking              */
/************************************************************************/
main()
{
asm(" or 1800h,st");
start:
 time_start(0);                            /* start timer 0 for benchmark */
/*********************************
  P size-q FFT's                 *
*********************************/
for (j=0;j<P;j++,src_addr +=2) {
    cmove(src_addr,input,p2,2,q);          /* q elements are transfered to
                                              on-chip RAM for execution   */
    r2dit(input,q);                        /* q-point FFT                 */
    cmove(input,src_addr,2,p2,q);          /* FFT results are transfered back
                                              to off-chip memory          */
    }
/*********************************
  LOG P Butterfly operation steps *
*********************************/
 src_addr = shinput;
 for (i=0;i<D;i++) { /* log P steps of P butterfly vector operations each */
     for (k=0;k<ngroup;++k) {              /* at each step i there are "ngroups" of
                                              identical butterfly vector operations */
     for (j=0;j<delta;j+=2) {             /* each group contains (delta/2)
                                              butterfly vector operations          */
         cmove(src_addr+j,input,delta,2,q);  /* move data on-chip                 */
         bflyr(input,q,Wkpointer);        /* butterfly vector operation           */
         cmove(input,src_addr+j,2,delta,q); /* move result off chip               */
             }
         src_addr  += incr_group;         /* update src address base for next group */
         Wkpointer += q2;                 /* update Wk pointer for next group       */
         }
```

```
     ngroup <<=1;                          /* number of groups decrement by half
                                                 after each step                 */
     Wkpointer = 0;                        /* initialize Wk pointer= Wn(0)      */
     src_addr = shinput;
     delta >>= 1;                          /* update parameters for next step   */
     incr_group >>=1;
    }
 tcomp = time_read(0);
} /*main*/
```

SERP2.C

```
/************************************************************************
  SERP2.C : Partitioned serial DIT FFT implementation (Single-buffered version)
  Requirements: 32<= Q <= 1024 (minimum: because of Meyer-Schwarz FFT limitations
                                maximum: because of on-chip RAM limitations)
  Version : 1.0
************************************************************************
 VERSION        DATE            COMMENT
   1.0          8/92            Original version
                                ROSEMARIE PIEDRA (TI Houston)
************************************************************************/
#define    N          2048                /* FFT size (n)              */
#define    P             2                /* P = N/Q                   */
#define    D             1                /* LOG2 P                    */
#define    Q           N/P                /* Maximum FFT size that can be
                                             computed  on-chip         */
#define    BLOCK0      0x002ff800         /* on-chip RAM buffer    */
extern      void r2dit(),                 /* C-callable complex FFT    */
                 cmove(),                 /* CPU complex move          */
                 cmoveb();                /* CPU bit-reversed complex move */
extern      float INPUT[];                /* Input vector = N = Q * P  */
float       *input      = (float *)BLOCK0, /* pointer to on-chip RAM    */
            *shinput    = INPUT,          /* pointer to input vector   */
            *src_addr   = INPUT;
unsigned int i,j,k,ngroup,
             delta      = P,
             incr_group = N,
             p2         = 2*P,
             Wkpointer  = 0,
             q          = Q,
             q2         = Q/2;
int         tcomp;                        /*  benchmarking             */
/************************************************************************/
main()
{
start:
 time_start(0);                           /* start timer 0 for benchmark */
/*********************************
  P Serial FFT of size Q          *
*********************************/
for (j=0;j<P;j++,src_addr +=2) {        /* loop P times                  */
    cmove(src_addr,input,p2,2,q);       /* move q complex numbers to on-chip */
    r2dit(input,q);                     /* q-point FFT                   */
    cmove(input,src_addr,2,p2,q);       /* move FFT result to off-chip memory */
    }
/*********************************
  LOG P Butterfly operation steps *
*********************************/
 for (i=0;i<D;i++) { /* log P steps of P butterfly vector operations each    */
                                        /* first butterfly vector operation  */
     cmove(shinput,input,delta,2,q);    /* move first data on-chip           */
     bflyr(input,q,0);                  /* butterfly vector operation        */
     cmove(input,shinput,2,delta,q);    /* move vector result off-chip       */
     for (j=2;j<2*P;j+=2) {             /* (P-1) butterfly vector operations */
     ngroup   = j/delta;                /* select which group it belongs to  */
     src_addr = shinput + ngroup*incr_group + j%delta; /* data source
                                          address for butterfly operation */
     cmove(src_addr,input,delta,2,q);   /* move data on-chip                 */
     bflyr(input,q,ngroup*q2);          /* butterfly vector operation        */
     cmove(input,src_addr,2,delta,q);   /* move vector result off-chip       */
         }
     delta >>= 1;                       /* update parameters for next step   */
     incr_group >>=1;
    }
 tcomp = time_read(0);                  /* tcomp = execution time            */
} /*main*/
```

SERP.CMD

```
-c                                      /* link using C conventions      */
serp.obj                                /* FFT C code                    */
sintab.obj                              /* sine table                    */
input.obj                               /* input data                    */
-lpfftr.lib                             /* get FFT Assembly code routine */
-stack 0x0040                           /* set stack size                */
-lrts40r.lib                            /* get run-time support          */
-lprts40r.lib                           /* get timer routines            */
-m serp.map                             /* generate map file             */
-o serp.out                             /* output file name              */
MEMORY
{
   ROM:  org = 0x00       len = 0x1000   /* on-chip ROM                  */
   RAM0: org = 0x002ff800 len = 0x0800   /* on-chip RAM: 2 blocks        */
   LM:   org = 0x40000000 len = 0x10000  /* local memory                 */
   GM:   org = 0x80000000 len = 0x20000  /* global memory                */
}
SECTIONS
{
   .input:   {} > LM                     /* input vector                 */
   .sintab:  {} > LM                     /* sine table                   */
   .ffttext: {} > LM                     /* FFT assembly routine (.text) */
   .text:    {} > LM                     /* FFT C code (.text)           */
   .cinit:   {} > LM                     /* initialization table         */
   .stack:   {} > LM                     /* system stack                 */
   .bss :    {} > LM                     /* global & static C variables  */
   .fftdata: {} > LM                     /* FFT assembly routine (.text) */
}
```

SERPB.C

```
/***********************************************************************
  SERPB.C : Partitioned serial FFT algorithm (Double–buffered version)
  Requirements: 32<= Q <= 512  (minimum: because of Meyer–Schwarz FFT limitations
                                maximum: because of on–chip RAM limitations)
  Version : 1.0
***********************************************************************
 VERSION        DATE            COMMENT
   1.0          8/92            Original version
                                ROSEMARIE PIEDRA (TI Houston)
***********************************************************************/
#define    N           2048                 /* FFT size (n)              */
#define    P           4
#define    D           2
#define    Q           N/P                  /* FFT subsize (512 suggested)*/
#define    BLOCK0      0x002ff800           /* on–chip RAM buffer 1       */
#define    BLOCK1      0x002ffc00           /* on–chip RAM buffer 2       */
#define    DMA0        0x001000a0           /* DMA0 address               */
#define    CTRL0       0x00c00008           /* autoinitialization         */
#define    CTRL1       0x00c40004           /* no autoinitialization      */
#define    MASK        0x02000000           /* IIF(bit DMAINT0) = 0       */
#define    DMAGO(dma,auto)        *(dma+3)=0; *(dma+6)=(int)auto; *dma=CTRL0;
extern      void r2dit(),                   /* C–callable complex FFT     */
                 cmove(),                   /* CPU complex move           */
                 cmoveb(),                  /* CPU bit–reversed complex move */
                 wait_dma(),                /* CPU waits for DMA to finish */
                 set_dma();                 /* set–up DMA registers       */
extern      float INPUT[];                  /* Input vector = N = Q * P   */
inline      extern void wait_dma();         /* CPU waits for DMA to finish*/
float      *shinput     = INPUT,
            *src_addr   = INPUT;
/* DMA autoinitialization values */
int         dma04[7]    = {CTRL1,(int)(INPUT+5),2*P,Q,BLOCK0+1,2,0},
            dma03[7]    = {CTRL0,(int)(INPUT+4),2*P,Q,BLOCK0,2,(int)dma04},
            dma02[7]    = {CTRL0,BLOCK0+1,2,Q,(int)(INPUT+1),2*P,(int)dma03},
            dma01[7]    = {CTRL0,BLOCK0,2,Q,(int)INPUT,2*P,(int)dma02},
            dma08[7]    = {CTRL1,(int)(INPUT+3),2*P,Q,BLOCK1+1,2,0},
            dma07[7]    = {CTRL0,(int)(INPUT+2),2*P,Q,BLOCK1,2,(int)dma08},
            dma06[7]    = {CTRL0,BLOCK1+1,2,Q,(int)(INPUT+3),2*P,(int)dma07},
            dma05[7]    = {CTRL0,BLOCK1,2,Q,(int)(INPUT+2),2*P,(int)dma06};
unsigned int i,j,k0,k1,temp = 0,
            ngroup0,ngroup1,
            delta       = P,
            incr_group  = N,
            p2          = 2*P,
            Wkpointer   = 0,
            q           = Q,
            q2          = Q/2;
volatile int    *dma    = (int *)DMA0;
int         tcomp;                          /*  benchmarking              */
/***********************************************************************/
main()
{
start:
 asm(" or 1800h,st");
 time_start(0);                             /* start timer 0 for benchmark  */
/*********************************
 P Serial FFT of size Q     *
*********************************/
 DMAGO(dma,dma07);                          /* DMA transfers data block 1   */
 cmove(src_addr,BLOCK0,p2,2,q);             /* CPU transfer data block 0    */
 r2dit(BLOCK0,q);                           /* FFT on data block 0          */
 for (j=2;j<P;j+=2) {                       /* loop  (P–2)/2 times          */
```

```
    /* initialize values for DMA  autoinitialization                    */
    dma01[4] = (int)src_addr;           /* DMA transfer back butterfly result */
    dma02[4] = (int)(src_addr+1);
    dma03[1] = (int)(src_addr+4);       /* DMA brings new set of data   */
    dma04[1] = (int)(src_addr+5);
    wait_dma(MASK);                     /* wait for DMA to finish       */
    DMAGO(dma,dma01);                   /* DMA start                    */
    r2dit(BLOCK1,q);                    /* FFT on on-chip RAM block 1 data */
    dma05[4] = (int)(src_addr+2);
    dma06[4] = (int)(src_addr+3);
    dma07[1] = (int)(src_addr+6);
    dma08[1] = (int)(src_addr+7);
    wait_dma(MASK);
    DMAGO(dma,dma05);                   /* move data from/to BLOCK1     */
    src_addr = src_addr+4;              /* point to next block          */
    r2dit(BLOCK0,q);                    /* FFT on on-chip RAM block 0 data */
    }
  dma01[4] = (int)src_addr;   dma02[4] = (int)(src_addr+1); dma02[0] = CTRL1;
  wait_dma(MASK);                       /* wait for DMA to finish       */
  DMAGO(dma,dma01);                     /* start DMA                    */
  r2dit(BLOCK1,q);                      /* last FFT computation         */
  cmove(BLOCK1,src_addr+2,2,p2,q);      /* move last FFT result off-chip*/
  wait_dma(MASK);                       /* wait for DMA to finish moving
                                           of previous FFT result */
/*********************************
  LOG P Butterfly operation steps *
*********************************/
 src_addr = shinput;
 for (i=0;i<D;i++) {                    /* loop (log P) times           */
    dma02[0] = CTRL0;
    ngroup1 = 2/delta;
    k1= (int)shinput + ngroup1*incr_group + 2 % delta;
    dma07[2] = dma08[2] = dma04[2] = dma03[2] = delta; /* DMA src offset*/
    dma01[5] = dma02[5] = dma05[5] = dma06[5] = delta;
    dma07[1] = k1;       dma08[1] = k1+1;
    DMAGO(dma,dma07);                   /* section 1 --> BLOCK1         */
    k0 = (int)src_addr;
    ngroup0  = 0;
    cmove(k0,BLOCK0,delta,2,q);         /* section 0 --> BLOCK0         */
    bflyr(BLOCK0,q,0);                  /* bfly on section  0           */
    for (j=4;j<p2;j+=4) {               /* loop (P-2)/2 times           */
        dma01[4] = (int)k0;             /* move section from BLOCK0     */
        dma02[4] = (int)(k0+1);
        ngroup0  = j/delta;
        k0= (int)shinput + ngroup0*incr_group + j % delta;
        dma03[1] = k0;                  /* move new section to BLOCK0   */
        dma04[1] = k0+1;
        wait_dma(MASK);
        DMAGO(dma,dma01);
        bflyr(BLOCK1,q,ngroup1*q2);     /* bfly on current section      */
        dma05[4] = (int)k1;             /* move section from BLOCK1     */
        dma06[4] = (int)(k1+1);
        ngroup1  = (j+2)/delta;
        k1 = (int)shinput + ngroup1*incr_group + (j+2) % delta;
        dma07[1] = (int)k1;             /* move new section to BLOCK1   */
        dma08[1] = (int)k1+1;
        wait_dma(MASK);
        DMAGO(dma,dma05);
        bflyr(BLOCK0,q,ngroup0*q2);     /* bfly on current section      */
        } /* loop (j) */
    dma01[4] = k0;    dma02[4] = k0+1;    dma02[0] = CTRL1;
    wait_dma(MASK);
    DMAGO(dma,dma01);
    bflyr(BLOCK1,q,ngroup1*q2);
    cmove(BLOCK1,k1,2,delta,q);
    delta >>=1;
    incr_group >>=1;
    wait_dma(MASK);
    } /* loop (i) */
  tcomp = time_read(0);
} /*main*/
```

Appendix F: Library Routines (PFFT.SRC)

BFLY.ASM

```
*************************************************************************
*
*   BFLY.ASM :  Butterlly operation on vector input (C-callable) to be used
*               with the parallel DIF FFT program.
*
*   version : 1.0
*
*************************************************************************
*   VERSION     DATE            COMMENT
*     1.0       8/92            Original version
*                               ROSEMARIE PIEDRA (TI HOUSTON)
*************************************************************************
*   SYNOPSIS:
*
*   void bfly (input, fft_size, Wkptr, sintab_size, step)
*               ar2      r2      r3       rc         rs
*
*   float *input     : Complex vector address
*   int   fft_size   : Complex FFT size
*   int   Wkptr      : Offset from _SINE to first twiddle factor to be used.
*   int   sintab_size: Sine table size          = N (size =5*N/4)
*   int   step       : Distance between twiddle factors of succesive butterflies
*
*************************************************************************
*
*                           +
*   AR + j AI ------------------------------------- AR' + j AI'
*                       \         / +
*                         \     /
*                           \ /
*                           / \
*                         /     \
*                       /         \ +
*   BR + j BI ------------------------ COS - j SIN ---- BR' + j BI'
*                           -
*
*   AR'= AR + BR
*   AI'= AI + BI
*   BR'= (AR-BR)*COS + (AI-BI)*SIN
*   BI'= (AI-BI)*COS - (AR-BR)*SIN
*
*************************************************************************
        .global  _bfly              ; Entry point for execution
        .global  _SINE              ; pointer to sine table
        .text
SINTAB  .word    _SINE
_bfly:
        LDI     SP,AR0
        PUSH    DP                  ; save dedicated registers
        PUSH    R6
        PUSHF   R6
        PUSH    AR3
        .if .REGPARM == 0
        LDI     *-AR0(1),AR2        ; input pointer
        LDI     *-AR0(2),R2         ; fftsize
        LDI     *-AR0(4),RC         ; sintab size
        LDI     *-AR0(5),RS         ; twiddle factor step
        LDI     *-AR0(3),AR0        ; offset to first twiddle factor to be used
        .else
        LDI     R3,AR0              ; offset to first twiddle factor to be used
        .endif
        LDP SINTAB
        LDI     2,IR0               ; distance between succesive butterflies
        LDI     RS,IR1             ; distance between twiddle factors of
                                   ; succesive butterflies
        LSH3    -2,RC,R3           ; R3 = sintabsize/4, distance between sine
                                   ; and cosine pointer
```

```
        ADDI    R2,AR2,AR3          ; AR2 = A     AR3 = B = A + fftsize
        LSH3    -1,R2,R0            ; R8  = fftsize/2
        SUBI   1,R0,RC              ; RC should be one less than desired #
        RPTBD   BLK1                ; execute fftsize/2 butterfly operations
        LDI@SINTAB,R1
        ADDI   R1,AR0               ; AR0 = sine pointer
        ADDI   AR0,R3,AR1           ; AR1 = cosine pointer
        LDF     *AR0++(IR1),R6      ; R6 = SIN  ; point to next SIN
        SUBF    *AR3,*AR2,R2        ; R2 = AR-BR
        SUBF    *+AR3,*+AR2,R1      ; R1 = AI-BI
        MPYF    R2,R6,R0            ; R0 = (AR-BR)*SIN
||      ADDF    *+AR3,*+AR2,R3      ; R3 = AI+BI
        MPYF    *AR1,R1,R3          ; new R3 = (AI-BI)*COS
||      STF     R3,*+AR2            ; **** AI' = AI+BI ****
        SUBF    R0,R3,R4            ; R4 = (AI-BI)*COS - (AR-BR)*SIN
        MPYF    R1,R6,R0            ; R0 = (AI-BI)*SIN
||      ADDF    *AR3,*AR2,R3        ; R3 = AR+BR
        MPYF    *AR1++(IR1),R2,R3   ; new R3 = (AR-BR)*COS ; point to next COS
||      STF     R3,*AR2++(IR0)      ; **** AR' = AR+BR ****
                                    ; and point to next butterfly
        ADDF    R0,R3,R0            ; R0 = (AR-BR)*COS + (AI-BI)*SIN
BLK1    STF     R0,*AR3++(IR0)      ; **** BR' = (AR-BR)*COS + (AI-BI)*SIN ****
||      STF     R4,*+AR3            ; **** BI' = (AI-BI)*COS - (AR-BR)*SIN ****
        POP     AR3
        POPF   R6
        POPR6
        POPDP
        RETS
        .end
```

BFLYR.ASM

```
*************************************************************************
*   BFLYR.ASM : Butterfly operation on vector input (C-callable) to be used
*               with the parallel DIT FFT program (DIS_DIT1.C).
*   version : 1.0
*************************************************************************
*   VERSION     DATE           COMMENT
*     1.0       8/92           Original version
*                              ROSEMARIE PIEDRA (TI HOUSTON)
*************************************************************************
*   SYNOPSIS:
*   void bflyr (input, fft_size, Wkptr)
*                ar2      r2      r3
*   float *input     : Complex vector address
*   int   fft_size   : Complex FFT size
*   int   Wkptr      : Offset(Re) from _SINE to first twiddle factor to be used.
*************************************************************************
*  TYPE I  BUTTERFLY
*                                    +
*   AR + j AI ------------------------------------------ AR' + j AI'
*                                \         / +
*                                  \     /
*                                    \ /
*                                    / \
*                                  /     \
*                                /         \ +
*   BR + j BI --- COS - j SIN -------------------------  BR' + j BI'
*                                    -
*   TR = BR*COS + BI*SIN
*   TI = BI*COS - BR*SIN
*   AR'= AR + TR
*   AI'= AI + TI
*   BR'= AR - TR
*   BI'= AI - TI
*************************************************************************
*   TYPE II  BUTTERFLY
*                                    +
*   AR + j AI ------------------------------------------ AR' + j AI'
*                                \         / +
*                                  \     /
*                                    \ /
*                                    / \
*                                  /     \
*                                /         \ +
*   BR + j BI --- -SIN - j COS -------------------------  BR' + j BI'
*                                    -
*   TR = BI*COS - BR*SIN
*   TI = BR*COS + BI*SIN
*   AR'= AR + TR
*   AI'= AI - TI
*   BR'= AR - TR
*   BI'= AI + TI
*************************************************************************
*   DESCRIPTION:             ----------  <-- input    (Re + jIm)
*                Type I           |
*                       Wk   ----------  <-- input+2  (Re + jIm)
*                            ----------  <-- input+4  (Re + jIm)
*                Type II          |
*                       Wk   ----------  <-- input+6  (Re + jIm)
*                            ----------  <-- input+8  (Re + jIm)
*                Type I           |
*                       Wk+1 ----------  <-- input+10 (Re + jIm)
*                            ----------  <-- input+12 (Re + jIm)
*                Type II          |
*                       Wk+1 ----------  <-- input+14 (Re + jIm)
*************************************************************************
        .global  _bflyr             ; Entry point for execution
        .global  _SINE
        .text
SINTAB  .word    _SINE
```

```
_bflyr:
        LDI     SP,AR0
        PUSH    DP                  ; save dedicated registers
        PUSH    R6                  ; R6 lower 32 bits
        PUSHF   R6                  ; R6 upper 32 bits
        PUSH    AR3
        .if .REGPARM == 0
        LDI     *-AR0(1),AR2        ; input pointer
        LDI     *-AR0(2),R2         ; fftsize
        LDI     *-AR0(3),AR0        ; Offset to first twiddle factor to be used
        .else
        LDI     R3,AR0              ; Offset to first twiddle factor to be used
        .endif
        LDP     SINTAB
        LDI     3,IR0               ; butterfly step
        LDI     2,IR1               ; twiddle factor step (because cos-sin)
        LDI     @SINTAB,R1          ; R1  = sine table address
        ADDI    R1,AR0              ; AR0 = cosine pointer
        ADDI    2,AR2,AR3           ; AR2 = points to AR
                                    ; AR3 = points to BR

        LSH     -2,R2               ; R2  = FFTSIZE/4
        SUBI    1,R2,RC             ; RC  = FFTSIZE/4 -1
        RPTBD   BLK                 ; loop FFTSIZE/4 times
        LDF     *AR0,R6             ; R6  = COS
        MPYF    *AR3,R6,R2          ; R2  = BR*COS
        MPYF    *+AR3,*+AR0,R0      ; R0  = BI*SIN
* TYPE I Butterfly
        MPYF    *AR3,*+AR0,R1       ; R1  = BR*SIN
||      ADDF    R0,R2,R2            ; R2  = TR  = BR*COS + BI*SIN
        MPYF    *+AR3,R6,R0         ; R0  = BI*COS
||      SUBF    R2,*AR2,R3          ; R3  = AR-TR
        ADDF    *AR2,R2,R2          ; R2  = AR+TR
||      STF     R3,*AR3++           ; ******* BR = AR-TR ******
                                    ; AR3 = POINTS TO BI

        SUBF    R1,R0               ; R0  = TI = BI*COS - BR*SIN
        SUBF    R0,*+AR2,R1         ; R1  = AI-TI
||      STF     R2,*AR2++           ; ******* AR = AR+TR ******
                                    ; AR2 = POINTS TO AI

        ADDF    R0,*AR2,R3          ; R3  = AI + TI
||      STF     R1,*AR3++(IR0)      ; ******* BI = AI-TI ******
                                    ; AR3 = POINTS TO NEXT BR  (TYPE II)

        MPYF    *+AR3,R6,R0         ; R0  = NEXT BI*COS        (TYPE II)
||      STF     R3,*AR2++(IR0)      ; ******* AI = AI+TI ******
                                    ; AR2 = POINTS TO NEXT AR  (TYPE II)

* TYPE II Butterfly
        MPYF    *AR3,*+AR0,R2       ; R2  = BR*SIN
        MPYF    *+AR3,*+AR0,R1      ; R1  = BI*SIN
||      SUBF    R2,R0,R2            ; R2  = TR  = BI*COS - BR*SIN
        MPYF    *AR3,R6,R0          ; R0  = BR*COS
||      SUBF    R2,*AR2,R3          ; R3  = AR-TR
        ADDF    *AR2,R2,R2          ; R2  = AR+TR
||      STF     R3,*AR3++           ; ******* BR = AR-TR ******
                                    ; AR3 = POINTS TO BI

        ADDF    R1,R0               ; R0  = TI = BR*COS + BI*SIN
        ADDF    *+AR2,R0,R1         ; R1  = AI+TI
||      STF     R2,*AR2++           ; ******* AR = AR+TR ******
                                    ; AR2 = POINTS TO AI

        SUBF    R0,*AR2,R3          ; R3  = AI - TI
||      STF     R1,*AR3++(IR0)      ; ******* BI = AI+TI ******
                                    ; AR3 = POINTS TO NEXT BR

        LDF     *++AR0(IR1),R6      ; R6  = NEXT COSINE
        MPYF    *+AR3,*+AR0,R0      ; R0  = NEXT BI*SIN (TYPE I)
BLK     MPYF    *AR3,R6,R2          ; R2  = NEXT BR*COS (TYPE I)
||      STF     R3,*AR2++(IR0)      ; ******* AI = AI-TI ******
                                    ; AR2 = POINTS TO NEXT AR

        POP     AR3
        POPF    R6
        POP     R6
        POP     DP
        RETS
        .end
```

BFLYR1.ASM

```
*************************************************************************
*   BFLYR1.ASM : Butterlly (TYPE 1) vector operation to be used with the
*                parallel DIT FFT program (DIS_DIT2.C). C–callable routine.
*   version : 1.0
*************************************************************************
*   VERSION     DATE           COMMENT
*     1.0       8/92           Original version
*                              ROSEMARIE PIEDRA (TI HOUSTON)
*************************************************************************
*   SYNOPSIS:
*   void bflyr1 (input, fft_size, Wkptr, step)
*                  ar2      r2      r3     rc
*   float *input     : Complex vector address
*   int   fft_size   : Complex FFT size/2 = Number of butterflies
*   int   Wkptr      : Offset(complex) from _SINE to first twiddle factor to be
*                      used.
*   int   step       : Distance(real) between twiddle factors of succesive
*                      butterflies
*************************************************************************
*                                      +
*   AR + j AI ------------------------------------- AR' + j AI'
*                                 \         / +
*                                   \     /
*                                     \ /
*                                     / \
*                                   /     \
*                                 /         \ +
*   BR + j BI --- COS – j SIN -------------------------  BR' + j BI'
*                                      –
*   TR = BR*COS + BI*SIN
*   TI = BI*COS – BR*SIN
*   AR'= AR + TR
*   AI'= AI + TI
*   BR'= AR – TR
*   BI'= AI – TI
*************************************************************************
        .global  _bflyr1           ; Entry point for execution
        .global  _SINE
        .text
SINTAB  .word    _SINE
_bflyr1:
        LDI     SP,AR0
        PUSH    DP                 ; save dedicated registers
        PUSH    R6                 ; R6 lower 32 bits
        PUSHF   R6                 ; R6 upper 32 bits
        PUSH    AR3
        .if .REGPARM == 0
        LDI     *–AR0(1),AR2       ; input pointer
        LDI     *–AR0(2),R2        ; fftsize/2 = number of butterflies
        LDI     *–AR0(4),RC        ; twiddle factor step
        LDI     *–AR0(3),AR0       ; Offset to first twiddle factor to be used
        .else
        LDI     R3,AR0             ; Offset to first twiddle factor to be used
        .endif
        LDP     SINTAB
        LDI     3,IR0              ; butterfly step
        LDI     RC,IR1             ; twiddle factor step
                                   ; AR3 = lower butterfly pointer
        LSH     1,AR0              ; AR0 = 2 * (first twiddle factor offset)
        LDI     @SINTAB,R1         ; R1  = sine table address
        ADDI    R1,AR0             ; AR0 = cosine pointer
* FIRST BUTTERFLY
        ADDI    2,AR2,AR3          ; AR2 = points to AR
                                   ; AR3 = points to BR
        SUBI    1,R2,RC            ; RC = FFTSIZE/2 –1
        RPTBD   BLK1               ; loop FFTSIZE/2 times
        LDF     *AR0,R6            ; R6  = COS
        MPYF    *AR3,R6,R2         ; R2  = BR*COS
        MPYF    *+AR3,*+AR0,R0     ; R0  = BI*SIN
```

```
* BLOCK1 START : 9 instructions
        MPYF    *AR3,*+AR0,R1       ; R1  = BR*SIN
||      ADDF    R0,R2,R2            ; R2  = TR  = BR*COS + BI*SIN
        MPYF    *+AR3,R6,R0         ; R0  = BI*COS
||      SUBF    R2,*AR2,R3          ; R3  = AR-TR
        ADDF    *AR2,R2,R2          ; R2  = AR+TR
||      STF     R3,*AR3++           ; ******* BR = AR-TR ******
                                    ; AR3 = POINTS TO BI
        SUBF    R1,R0               ; R0  = TI = BI*COS - BR*SIN
        SUBF    R0,*+AR2,R1         ; R1  = AI-TI
||      STF     R2,*AR2++           ; ******* AR = AR+TR ******
                                    ; AR2 = POINTS TO AI
        ADDF    R0,*AR2,R3          ; R3  = TI + AI
||      STF     R1,*AR3++(IR0)      ; ******* BI = AI-TI ******
                                    ; AR3 = POINTS TO NEXT BR
        LDF     *++AR0(IR1),R6      ; R6  = NEXT COSINE
        MPYF    *+AR3,*+AR0,R0      ; R0  = NEXT BI*SIN
BLK1    MPYF    *AR3,R6,R2          ; R2  = NEXT BR*COS
||      STF     R3,*AR2++(IR0)      ; ******* AI = AI+TI ******
                                    ; AR2 = POINTS TO NEXT AR
        POP     AR3
        POPF    R6
        POP     R6
        POP     DP
        RETS
        .end
```

BFLYR2.ASM

```
*************************************************************************
*   BFLYR2.ASM : Butterlly (TYPE II) vector operation to be used with the
*                parallel DIT FFT program (DIS_DIT2.C)
*   version : 1.0
*************************************************************************
*   VERSION     DATE           COMMENT
*     1.0       8/92           Original version
*                              ROSEMARIE PIEDRA (TI HOUSTON)
*************************************************************************
*   SYNOPSIS:
*   void bflyr2 (input, fft_size, Wkptr, step)
*                ar2      r2      r3     rc
*   float *input     : Complex vector address
*   int   fft_size   : Complex FFT size/2 = Number of butterflies
*   int   Wkptr      : Offset (complex) from _SINE to first twiddle factor to be
*                      used.
*   int   step       : Distance (real)  between twiddle factors of succesive
*                      butterflies
*************************************************************************
*                                     +
*   AR + j AI ------------------------------------- AR' + j AI'
*                               \         / +
*                                 \     /
*                                   \ /
*                                   / \
*                                 /     \
*                               /         \ +
*   BR + j BI --- -SIN - j COS -------------------------  BR' + j BI'
*                                     -
*   TR = BI*COS - BR*SIN
*   TI = BR*COS + BI*SIN
*   AR'= AR + TR
*   AI'= AI - TI
*   BR'= AR - TR
*   BI'= AI + TI
*************************************************************************
        .global  _bflyr2           ; Entry point for execution
        .global  _SINE
        .text
SINTAB  .word    _SINE
_bflyr2:
        LDI     SP,AR0
        PUSH    DP                 ; save dedicated registers
        PUSH    R6                 ; R6 lower 32 bits
        PUSHF   R6                 ; R6 upper 32 bits
        PUSH    AR3
        .if .REGPARM == 0
        LDI     *-AR0(1),AR2       ; input pointer
        LDI     *-AR0(2),R2        ; fftsize/2 = number of butterflies
        LDI     *-AR0(4),RC        ; twiddle factor step
        LDI     *-AR0(3),AR0       ; Offset to first twiddle factor to be used
        .else
        LDI     R3,AR0             ; Offset to first twiddle factor to be used
        .endif
        LDP     SINTAB
        LDI     3,IR0              ; butterfly step
        LDI     RC,IR1             ; twiddle factor step
                                   ; AR3 = lower butterfly pointer
        LSH     1,AR0              ; AR0 = 2 * (first twiddle factor offset)
        LDI     @SINTAB,R1         ; R1  = sine table address
        ADDI    R1,AR0             ; AR0 = cosine pointer
* FIRST BUTTERFLY
        ADDI    2,AR2,AR3          ; AR2 = points to AR
                                   ; AR3 = points to BR
        SUBI    1,R2,RC            ; RC = FFTSIZE/2 -1
        RPTBD   BLK1               ; loop FFTSIZE/2 times
        LDF     *AR0,R6            ; R6  = COS
        MPYF    *+AR3,R6,R2        ; R2  = BI*COS
        MPYF    *AR3,*+AR0,R0      ; R0  = BR*SIN
```

```
* BLOCK1 START
        MPYF    *+AR3,*+AR0,R1      ; R1  = BI*SIN
||      SUBF    R0,R2,R2            ; R2  = TR  = BI*COS – BR*SIN
        MPYF    *AR3,R6,R0          ; R0  = BR*COS
||      SUBF    R2,*AR2,R3          ; R3  = AR–TR
        ADDF    *AR2,R2,R2          ; R2  = AR+TR
||      STF     R3,*AR3++           ; ******* BR = AR–TR ******
                                    ; AR3 = POINTS TO BI
        ADDF    R1,R0               ; R0  = TI = BR*COS + BI*SIN
        ADDF    *+AR2,R0,R1         ; R1  = AI+TI
||      STF     R2,*AR2++           ; ******* AR = AR+TR ******
                                    ; AR2 = POINTS TO AI
        LDF     *++AR0(IR1),R6      ; R6  = NEXT COSINE
||      STF     R1,*AR3++(IR0)      ; ******* BI = AI+TI ******
                                    ; AR3 = POINTS TO NEXT BR
        SUBF    R0,*AR2,R3          ; R3  = AI – TI
        MPYF    *AR3,*+AR0,R0       ; R0  = NEXT BR*SIN
BLK1    MPYF    *+AR3,R6,R2         ; R2  = NEXT BI*COS
||      STF     R3,*AR2++(IR0)      ; ******* AI = AI–TI ******
                                    ; AR2 = POINTS TO NEXT AR
        POP     AR3
        POPF    R6
        POP     R6
        POP     DP
        RETS
        .end
```

CMOVE.ASM

```
*****************************************************************
*
*  CMOVE.ASM : TMS320C40 C-callable routine to move a complex float
*              vector pointed by src, to an address pointed by dst.
*
*  Calling conventions:
*
*  void cmove((float *)src,(float *)dst,int src_displ,int dst_displ,int lenght)
*                  ar2            r2         r3             rc             rs
*
*  where       src      : Vector Source Address
*              dst      : Vector Destination Address
*              src_displ: Source offsset (real)
*              dst_displ: Destination offsset (real)
*              lenght   : Vector lenght (complex)
*
*  version 1.0     Rose Marie Piedra
*****************************************************************
        .global _cmove
        .text
_cmove:
        .if .REGPARM == 0
        LDI     SP,AR0
        LDI     *-AR0(1),AR2            ; Source address
        LDI     *-AR0(4),IR1            ; Destination index (real)
        LDI     *-AR0(5),RC             ; Complex lenght
        SUBI    2,RC                    ; RC=lenght-2
        RPTBD   CMOVE
        LDI     *-AR0(2),AR1            ; Destination address
        LDI     *-AR0(3),IR0            ; Source index (real)
        LDF     *+AR2(1),R0
        .else
        LDI     RC,IR1                  ; destination index (real)
        SUBI    2,RS,RC                 ; complex lenght -2
        RPTBD   CMOVE
        LDI     R2,AR                   ; source address
        LDI     R3,IR0                  ; source index (real)
        LDF     *+AR2(1),R0
        .endif
*       loop
        LDF     *AR2++(IR0),R1
||      STF     R0,*+AR1(1)
CMOVE   LDF     *+AR2(1),R0
||      STF     R1,*AR1++(IR1)
        POP     AR0
        BUD     AR0
        LDF     *AR2++(IR0),R1
||      STF     R0,*+AR1(1)
        STF     R1,*AR1
        NOP
        .end
```

EXCH_R.ASM

```
*******************************************************************************
*
*  EXCHANGE_R.ASM: TMS320C40 C–callable routine to exchange
*                  two floating point complex vectors pointed by ″src_addr″ in
*                  each processor memory. This routine uses CPU to
*                  send/receive (no port synchronization is used) ″lenght″
*                  complex numbers to ″comport″ .
*
*  Calling conventions:
*
*  void exchange_r (comport, src_addr, lenght , offset)
*                     ar2      r2        r3       rc
*
*  int    comport   : Comport number to be used
*  void   *src_addr : Source/destination address
*  int    lenght    : Complex vector lenght
*  int    offset    : Source/destination address step (real)
*
*  version 1.0     Rose Marie Piedra
*******************************************************************************
        .global _exchange_r
        .text
CP_IN_BASE  .word 0100041H
_exchange_r:
        LDI     SP,AR1          ; Points to top of stack
        PUSH    DP
        .if .REGPARM == 0
        LDI     *–AR1(1),AR2    ; comport number
        LDI     *–AR1(2),AR0    ; Source/destination adress
        LDI     *–AR1(3),R3     ; lenght (complex)
        LDI     *–AR1(4),IR0    ; offset
        .else
        LDI     R2,AR0          ; source/destination address
        LDI     RC,IR0          ; offset
        .endif
        LDP     CP_IN_BASE      ; set DP register
        ADDI    1,IR0,IR1       ; IR1 = offset + 1
        SUBI    2,R3,RC         ; RC  = complex lenght –2
        LSH3    4,AR2,R0        ; R0  = comport number << 4
        LDI     @CP_IN_BASE,AR2
        RPTBD   BLK
        ADDI    R0,AR2          ; AR2 = comport FIFO pointer
        LDI     *+AR0,R2        ; R2  = Im
        STI     R2,*+AR2        ; send Im part to OFIFO
* REPEAT BLOCK STARTS
        LDI     *AR0,R2         ; R2  = Re part
        LDI     *AR2,R0         ; R0  = receive Im part from IFIFO
        STI     R2,*+AR2        ; send Re part to OFIFO
        STI     R0,*+AR0        ; store Im part in memory
        LDI     *+AR0(IR1),R2   ; R2  = Im part (next)
        LDI     *AR2,R0         ; R0  = receive Re part from IFIFO
        STI     R2,*+AR2        ; send next Im part to OFIFO
BLK     STI     R0,*AR0++(IR0)  ; store Re part in memory
                                ; AR0 = points to next complex number
* LAST COMPLEX NUMBER TO SEND/RECEIVE
        LDI     *AR0,R2         ; R2  = last Re part
        LDI     *AR2,R0         ; R0  = read Im part from IFIFO
        STI     R2,*+AR2        ; send last Re part to OFIFO
        STI     R0,*+AR0        ; store last Im part in memory
        LDI     *AR2,R0         ; R0  = receive last Re part from IFIFO
        STI     R0,*AR0         ; store last Re part in memory
        POP     DP
        RETS
        .end
tcomp = time_read(0);
} /*main*/
```

MOVE.ASM

```
*****************************************************************
*
*  MOVE.ASM : TMS320C40 C-callable routine to move a float
*             vector pointed by src, to an address pointed by dst.
*
*  Calling conventions:
*
*  void move((float *)src,(float *)dst,int src_displ,int dst_displ,int lenght)
*                  ar2           r2         r3             rc             rs
*
*  where       src      : Vector Source Address
*              dst      : Vector Destination Address
*              src_displ: Source offsset
*              dst_displ: Destination offset
*              lenght   : Vector lenght
*
*  version 1.0    Rose Marie Piedra
*****************************************************************
        .global _move
        .text
_move:
        .if .REGPARM == 0
        LDI     SP,AR0
        LDI     *-AR0(1),AR2           ; Source address
        LDI     *-AR0(4),IR1           ; Destination index
        LDI     *-AR0(5),RC            ; Vector  lenght
        SUBI    2,RC                   ; RC=lenght-2
        RPTBD   MOVE
        LDI     *-AR0(2),AR1           ; Destination address
        LDI     *-AR0(3),IR0           ; Source index
        LDF     *AR2++(IR0),R0
        .else
        LDI     RC,IR1                 ; destination index
        SUBI    2,RS,RC                ; Vector lenght -2
        RPTBD   MOVE
        LDI     R2,AR1                 ; source address
        LDI     R3,IR0                 ; source index
        LDI     *AR2++(IR0),R0
        .endif
*       loop
MOVE    LDF     *AR2++(IR0),R0
||      STF     R0,*AR1++(IR1)
        STF     R0,*AR1
        RETS
        .end
```

PR2DIF.ASM

```
*************************************************************************
*                                                                       *
*               COMPLEX, RADIX-2 DIF FFT  :  PR2DIF.ASM                 *
*              ------------------------------------------               *
*    GENERIC PROGRAM FOR A RADIX-2 DIF FFT COMPUTATION IN TMS320C40     *
*    TO WORK WITH PARALLEL FFT PROGRAM                                  *
*    VERSION: 1.0                                                       *
*************************************************************************
*    VERSION       DATE          COMMENT                                *
*     1.0          7/92          ROSEMARIE PIEDRA (TI HOUSTON):         *
*                                modified to work with parallel FFT program  *
*************************************************************************
*    SYNOPSIS: int  pr2dif(SOURCE_ADDR,FFT_SIZE,LOGFFT,P)               *
*                           ar2        r2       r3   rc                 *
*            float   *SOURCE_ADDR    ; input address                    *
*            int     FFT_SIZE        ; 64, 128, 256, 512, 1024, ...     *
*            int     LOGFFT          ; log (base 2) of FFT_SIZE         *
*            int     P               ; number of processors             *
*                                                                       *
*    - THE COMPUTATION IS DONE IN-PLACE.                                *
*************************************************************************
*    THIS IS A SEQUENTIAL IMPLEMENTATION  OF PARALLEL FFT, ALMOST IDENTICAL  *
*    TO THE CODE AVAILABLE IN THE C40'USER'S GUIDE. THE CODE HAS BEEN   *
*    MODIFIED TO WORK WITH A SINE TABLE OF SIZE (FFT_SIZE*P)/2 INSTEAD OF    *
*    (FFT_SIZE/2)                                                       *
*************************************************************************
*    SECTIONS NEEDED IN LINKER COMMAND FILE:  .cfftext : cfft code      *
*************************************************************************
*    THE TWIDDLE FACTORS ARE STORED WITH A TABLE LENGTH OF 5*FFT_SIZE/4      *
*    THE SINE TABLE IS PROVIDED IN A SEPARATE FILE WITH GLOBAL LABEL _SINE   *
*    POINTING TO THE BEGINNING OF THE TABLE.                            *
*************************************************************************
               .global _pr2dif         ; Entry execution point.
               .global _SINE           ; address of sine table
;
; Initialize C Function.
;
               .sect   ".ffttext"
SINTAB .word    _SINE
_pr2dif:
        LDI     SP,AR0
        PUSH    DP
        PUSH    R4                      ; Save dedicated registers
        PUSH    R5
        PUSH    R6
        PUSHF   R6
        PUSH    AR4
        PUSH    AR5
        PUSH    R8
        .if .REGPARM == 0
        LDI     *-AR0(1),AR2            ; points to X(I): INPUT
        LDI     *-AR0(2),R10            ; R10=N
        LDI     *-AR0(3),R9             ; R9 holds the remain stage number
        LDI     *-AR0(4),RC   ;!!!      ; RC = P = number of processors
        .else
        LDI     R2,R10
        LDI     R3,R9
        .endif
        LDP     SINTAB
        LDI     1,R8                ; Initialize repeat counter of first loop
        LSH3    1,R10,IR0           ; IR0=2*N1 (because of real/imag)
        LSH3    -2,R10,IR1          ; IR1=N/4, pointer for SIN/COS table
        MPYI    RC,IR1   ; !!!      ; IR1=NP/4
        LDI     RC,AR5   ; !!!      ; Initialize IE index (AR5=IE)
        LSH     1,R10
        SUBI3   1,R8,RC             ; RC should be one less than desired #
```

```
*       Outer loop
LOOP:
        RPTBD   BLK1                ; Setup for first loop
        LSH     -1,R10              ; N2=N2/2
        LDI     AR2,AR0             ; AR0 points to X(I)
        ADDI    R10,AR0,AR6         ; AR6 points to X(L)
*       First loop
        ADDF    *AR0,*AR6,R0        ; R0=X(I)+X(L)
        SUBF    *AR6++,*AR0++,R1    ; R1=X(I)-X(L)
        ADDF    *AR6,*AR0,R2        ; R2=Y(I)+Y(L)
        SUBF    *AR6,*AR0,R3        ; R3=Y(I)-Y(L)
        STF     R2,*AR0--           ; Y(I)=R2  and...
||      STF     R3,*AR6--           ; Y(L)=R3
BLK1    STF     R0,*AR0++(IR0)      ; X(I)=R0  and...
||      STF     R1,*AR6++(IR0)      ; X(L)=R1 and AR0,2 = AR0,2 + 2*n
*  If this is the last stage, you are done
        SUBI    1,R9
        BZD     END
*       main inner loop
        LDI     2,AR1               ; Init loop counter for inner loop
        LDI     @SINTAB,AR4         ; Initialize IA index (AR4=IA)
        ADDI    AR5,AR4             ; IA=IA+IE; AR4 points to cosine
        ADDI    AR2,AR1,AR0         ; (X(I),Y(I)) pointer
        SUBI    1,R8,RC             ; RC should be one less than desired #
INLOP:
        RPTBD   BLK2                ; Setup for second loop
        ADDI    R10,AR0,AR6         ; (X(L),Y(L)) pointer
        ADDI    2,AR1
        LDF     *AR4,R6             ; R6=SIN
*       Second loop
        SUBF    *AR6,*AR0,R2        ; R2=X(I)-X(L)
        SUBF    *+AR6,*+AR0,R1      ; R1=Y(I)-Y(L)
        MPYF    R2,R6,R0            ; R0=R2*SIN and...
||      ADDF    *+AR6,*+AR0,R3      ; R3=Y(I)+Y(L)
        MPYF    R1,*+AR4(IR1),R3    ; R3 = R1 * COS and ...
||      STF     R3,*+AR0            ; Y(I)=Y(I)+Y(L)
        SUBF    R0,R3,R4            ; R4=R1*COS-R2*SIN
        MPYF    R1,R6,R0            ; R0=R1*SIN and...
||      ADDF    *AR6,*AR0,R3        ; R3=X(I)+X(L)
        MPYF    R2,*+AR4(IR1),R3    ; R3 = R2 * COS and...
||      STF     R3,*AR0++(IR0)      ; X(I)=X(I)+X(L) and AR0=AR0+2*N1
        ADDF    R0,R3,R5            ; R5=R2*COS+R1*SIN
BLK2    STF     R5,*AR6++(IR0)      ; X(L)=R2*COS+R1*SIN, incr AR6 and...
||      STF     R4,*+AR6            ; Y(L)=R1*COS-R2*SIN
        CMPI    R10,AR1
        BNEAF   INLOP               ; Loop back to the inner loop
        ADDI    AR5,AR4             ; IA=IA+IE; AR4 points to cosine
        ADDI    AR2,AR1,AR0         ; (X(I),Y(I)) pointer
        SUBI    1,R8,RC
        LSH     1,R8                ; Increment loop counter for next time
        BRD     LOOP                ; Next FFT stage (delayed)
        LSH     1,AR5               ; IE=2*IE
        LDI     R10,IR0             ; N1=N2
        SUBI3   1,R8,RC
 END    POP     R8
        POP     AR5                 ; Restore the register values and return
        POP     AR4
        POPF    R6
        POP     R6
        POP     R5
        POP     R4
        POP     DP
        RETS
        .end
```

R2DIF.ASM

```
*************************************************************************
*                                                                       *
*                   COMPLEX, RADIX-2 DIF FFT  :  R2DIF.ASM              *
*                 ------------------------------------------            *
*                                                                       *
*       Generic program for a radix-2 DIF FFT computation using the     *
*       TMS320C4x family. The computation is done in-place and the result*
*       is bit-reversed. The program is taken from the burrus and Parks  *
*       book, p. 111.                                                   *
*                                                                       *
*       The twiddle factors are supplied in a table put in a .data section*
*       with a global label _SINE pointing to the beginning of the table *
*       This data is included in a separate file to preserve the generic *
*       nature of the program. The sine table size  is (5*FFT_SIZE)/4.   *
*                                                                       *
*       VERSION: 3.0                                                    *
*                                                                       *
*************************************************************************
*                                                                       *
*    VERSION        DATE         COMMENT                                *
*      1.0          10/87        Original version                       *
*                                PANNOS PAPAMICHALIS (TI HOUSTON)       *
*                                                                       *
*      2.0           1/91        DANIEL CHEN (TI HOUSTON): C40 porting  *
*      3.0           7/91        ROSEMARIE PIEDRA (TI HOUSTON): made it *
*                                C-callable. Discard bit-reversed transfer*
*                                of output result (not needed for some  *
*                                applications). If bit-reversing is needed*
*                                check cmoveb.asm in "Parallel 2-D FFT  *
*                                implementation with TMS320c4x DSPs"(SPRA027)*
*                                                                       *
*************************************************************************
*    SYNOPSIS: int   r2dif(SOURCE_ADDR,FFT_SIZE,LOGFFT)                 *
*                            ar2        r2       r3                     *
*                                                                       *
*            float   *SOURCE_ADDR    ; input address                    *
*            int     FFT_SIZE        ; 64, 128, 256, 512, 1024, ...     *
*            int     LOGFFT          ; log (base 2) of FFT_SIZE         *
*                                                                       *
*    - THE COMPUTATION IS DONE IN-PLACE.                                *
*                                                                       *
*************************************************************************
*                                                                       *
*                            +                                          *
*   AR + j AI ------------------------------------- AR' + j AI'         *
*                       \         / +                                   *
*                         \     /                                       *
*                           \ /                                         *
*                           / \                                         *
*                         /     \                                       *
*                       /         \ +                                   *
*   BR + j BI ----------------------- COS - j SIN ---- BR' + j BI'       *
*                            -                                          *
*                                                                       *
*   AR'= AR + BR                                                        *
*   AI'= AI + BI                                                        *
*   BR'= (AR-BR)*COS + (AI-BI)*SIN                                      *
*   BI'= (AI-BI)*COS - (AR-BR)*SIN                                      *
*                                                                       *
*************************************************************************
        .globl _SINE                ; Address of sine/cosine table
        .globl  _r2dif              ; Entry point for execution
        .sect   ".ffttext"
SINTAB  .word  _SINE
```

```
_r2dif:
        LDI    SP,AR0
        PUSH   DP
        PUSH   R4                    ; Save dedicated registers
        PUSH   R5
        PUSH   R6                    ; lower 32 bits
        PUSHF  R6                    ; upper 32 bits
        PUSH   AR4
        PUSH   AR5
        PUSH   AR6
        PUSH   R8
       .if .REGPARM == 0
        LDI    *-AR0(1),AR2          ; points to X(I): INPUT
        LDI    *-AR0(2),R10          ; R10=N
        LDI    *-AR0(3),R9           ; R9 holds the remain stage number
        .else
        LDI    R2,R10
        LDI    R3,R9
        .endif
        LDP    SINTAB
        LDI    1,R8                  ; Initialize repeat counter of first loop
        LSH3   1,R10,IR0             ; IR0=2*N1 (because of real/imag)
        LSH3   -2,R10,IR1            ; IR1=N/4, pointer for SIN/COS table
        LDI    1,AR5                 ; Initialize IE index (AR5=IE)
        LSH    1,R10
        SUBI3  1,R8,RC               ; RC should be one less than desired #
*       Outer loop
LOOP:
        RPTBD  BLK1                  ; Setup for first loop
        LSH    -1,R10                ; N2=N2/2
        LDI    AR2,AR0               ; AR0 points to X(I)
        ADDI   R10,AR0,AR6           ; AR6 points to X(L)
*       First loop
        ADDF   *AR0,*AR6,R0          ; R0=X(I)+X(L)
        SUBF   *AR6++,*AR0++,R1      ; R1=X(I)-X(L)
        ADDF   *AR6,*AR0,R2          ; R2=Y(I)+Y(L)
        SUBF   *AR6,*AR0,R3          ; R3=Y(I)-Y(L)
        STF    R2,*AR0--             ; Y(I)=R2  and...
||      STF    R3,*AR6--             ; Y(L)=R3
BLK1    STF    R0,*AR0++(IR0)        ; X(I)=R0  and...
||      STF    R1,*AR6++(IR0)        ; X(L)=R1 and AR0,2 = AR0,2 + 2*n
*  If this is the last stage, you are done
        SUBI   1,R9
        BZD    END
*       main inner loop
        LDI    2,AR1                 ; Init loop counter for inner loop
        LDI    @SINTAB,AR4           ; Initialize IA index (AR4=IA)
        ADDI   AR5,AR4               ; IA=IA+IE; AR4 points to cosine
        ADDI   AR2,AR1,AR0           ; (X(I),Y(I)) pointer
        SUBI   1,R8,RC               ; RC should be one less than desired #
INLOP:
        RPTBD  BLK2                  ; Setup for second loop
        ADDI   R10,AR0,AR6           ; (X(L),Y(L)) pointer
        ADDI   2,AR1
        LDF    *AR4,R6               ; R6=SIN
*       Second loop
        SUBF   *AR6,*AR0,R2          ; R2=X(I)-X(L)
        SUBF   *+AR6,*+AR0,R1        ; R1=Y(I)-Y(L)
        MPYF    R2,R6,R0             ; R0=R2*SIN and...
||      ADDF   *+AR6,*+AR0,R3        ; R3=Y(I)+Y(L)
        MPYF    R1,*+AR4(IR1),R3     ; R3 = R1 * COS and ...
||      STF     R3,*+AR0             ; Y(I)=Y(I)+Y(L)
        SUBF    R0,R3,R4             ; R4=R1*COS-R2*SIN
        MPYF    R1,R6,R0             ; R0=R1*SIN and...
||      ADDF   *AR6,*AR0,R3          ; R3=X(I)+X(L)
        MPYF    R2,*+AR4(IR1),R3     ; R3 = R2 * COS and...
||      STF     R3,*AR0++(IR0)       ; X(I)=X(I)+X(L) and AR0=AR0+2*N1
        ADDF   R0,R3,R5              ; R5=R2*COS+R1*SIN
        BLK2   STF R5,*AR6++(IR0)    ; X(L)=R2*COS+R1*SIN, incr AR6 and...
||      STF    R4,*+AR6              ; Y(L)=R1*COS-R2*SIN
```

```
        CMPI   R10,AR1
        BNEAF  INLOP                ; Loop back to the inner loop
        ADDI   AR5,AR4              ; IA=IA+IE; AR4 points to cosine
        ADDI   AR2,AR1,AR0          ; (X(I),Y(I)) pointer
        SUBI   1,R8,RC
        LSH    1,R8                 ; Increment loop counter for next time
        BRD    LOOP                 ; Next FFT stage (delayed)
        LSH    1,AR5                ; IE=2*IE
        LDI    R10,IR0              ; N1=N2
        SUBI3  1,R8,RC
END     POP    R8
        POP    AR6
        POP    AR5                  ; Restore the register values and return
        POP    AR4
        POPF   R6
        POP    R6
        POP    R5
        POP    R4
        POP    DP
        RETS
        .end
```

R2DIT.ASM

```
*******************************************************************************
*                   COMPLEX, RADIX-2 DIT FFT  :  R2DIT.ASM                    *
*                  ------------------------------------                       *
*    GENERIC PROGRAM FOR A FAST LOOPED-CODE RADIX-2 DIT FFT COMPUTATION       *
*                          IN TMS320C40 VERSION: 3.0                          *
*******************************************************************************
*    VERSION        DATE         COMMENT                                      *
*     1.0           7/89         Original version                             *
*                                RAIMUND MEYER, KARL SCHWARZ                  *
*                                LEHRSTUHL FUER NACHRICHTENTECHNIK            *
*                                UNIVERSITAET ERLANGEN-NUERNBERG              *
*                                CAUERSTRASSE 7, D-8520 ERLANGEN, FRG         *
*                                                                             *
*     2.0           1/91         DANIEL CHEN (TI HOUSTON): C40 porting        *
*     3.0           7/92         ROSEMARIE PIEDRA (TI HOUSTON): made it       *
*                                C-callable and implement the same changes    *
*                                in the order of the operands in some mpyf    *
*                                instructions as it was done in the C30       *
*                                version. Also bit-reversing of output        *
*                                vector was discarded(not needed for most     *
*                                applications. If bit-reversing is needed     *
*                                check cmoveb.asm in "Parallel 2-D FFT        *
*                                implementation with TMS320c4x DSP's"(SPRA027) *
*******************************************************************************
*    SYNOPSIS: int  r2dit(SOURCE_ADDR,FFT_SIZE)                               *
*                           ar2        r2                                     *
*            float   *SOURCE_ADDR    ; Points to where data is originated     *
*                                    ; and operated on.                       *
*            int     FFT_SIZE        ; 64, 128, 256, 512, 1024, ...           *
*    - THE COMPUTATION IS DONE IN-PLACE.                                      *
*    - FOR THIS PROGRAM THE MINIMUM FFTLENGTH IS 32 POINTS BECAUSE OF THE     *
*      SEPARATE STAGES.                                                       *
*    - FIRST TWO PASSES ARE REALIZED AS A FOUR BUTTERFLY LOOP SINCE THE       *
*      MULTIPLIES ARE TRIVIAL. THE MULTIPLIER IS ONLY USED FOR A LOAD IN      *
*      PARALLEL WITH AN ADDF OR SUBF.                                         *
*******************************************************************************
*    SECTIONS NEEDED IN LINKER COMMAND FILE:  .ffttext : fft code             *
*                                             .fftdata : fft data             *
*******************************************************************************
*    THE TWIDDLE FACTORS ARE STORED IN BITREVERSED ORDER AND WITH A TABLE     *
*    LENGTH OF N/2 (N = FFTLENGTH). THE SINE TABLE IS PROVIDED IN A SEPARATE  *
*    FILE  WITH GLOBAL LABEL _SINE POINTING TO THE BEGINNING OF THE TABLE.    *
*    EXAMPLE: SHOWN FOR N=32, WN(n) = COS(2*PI*n/N) - j*SIN(2*PI*n/N)         *
*               ADDRESS                COEFFICIENT                           *
*                  0          R{WN(0)} = COS(2*PI*0/32) = 1                  *
*                  1         -I{WN(0)} = SIN(2*PI*0/32) = 0                  *
*                  2          R{WN(4)} = COS(2*PI*4/32) = 0.707              *
*                  3         -I{WN(4)} = SIN(2*PI*4/32) = 0.707              *
*                  :                    :                                    *
*                  12         R{WN(3)} = COS(2*PI*3/32) = 0.831              *
*                  13        -I{WN(3)} = SIN(2*PI*3/32) = 0.556              *
*                  14         R{WN(7)} = COS(2*PI*7/32) = 0.195              *
*                  15        -I{WN(7)} = SIN(2*PI*7/32) = 0.981              *
*    WHEN GENERATED FOR A FFT LENGTH OF 1024, THE TABLE IS FOR ALL FFT        *
*    LENGTH LESS OR EQUAL AVAILABLE.                                          *
*    THE MISSING TWIDDLE FACTORS (WN(),WN(),....) ARE GENERATED BY USING      *
*    THE SYMMETRY WN(N/4+n) = -j*WN(n). THIS CAN BE REALIZED VERY EASY, BY    *
*    CHANGING REAL- AND IMAGINARY PART OF THE TWIDDLE FACTORS AND BY          *
*    NEGATING THE NEW REAL PART.                                              *
*******************************************************************************
*   AR + j AI ------------------------------------------------ AR' + j AI'     *
*   BR + j BI ---- ( COS - j SIN ) -------------------------- BR' + j BI'     *
*   TR = BR * COS + BI * SIN                                                  *
*   TI = BI * COS - BR * SIN                                                  *
*   AR'= AR + TR                                                              *
*   AI'= AI + TI                                                              *
*   BR'= AR - TR                                                              *
*   BI'= AI - TI                                                              *
*******************************************************************************
```

```
              .global _r2dit           ; Entry execution point.
              .global _SINE
              .sect  ".fftdata"
fg2           .space  1                ; is FFT_SIZE/2
fg4m2         .space  1                ; is FFT_SIZE/4 – 2
fg8m2         .space  1                ; is FFT_SIZE/8 – 2
sintab        .word   _SINE            ; pointer to sine table
sintp2        .word   _SINE+2          ; pointer to sine table +2
inputp2       .space  1                ; pointer to input +2
inputp        .space  1
;
; Initialize C Function.
;
              .sect  ".ffttext"
_r2dit:       LDI     SP,AR0
              PUSH    R4
              PUSH    R5
              PUSH    R6
              PUSHF   R6
              PUSH    R7
              PUSHF   R7
              PUSH     AR3
              PUSH    AR4
              PUSH    AR5
              PUSH    AR6
              PUSH    AR7
              PUSH    DP
              .if .REGPARM == 0        ; arguments passed in stack
              LDI     *–AR0(1),AR2     ; input address
              LDI     *–AR0(2),R2      ; FFT size
              .endif
              LDP     fg2              ; Initialize DP pointer.
              LSH     –1,R2
              ADDI    2,AR2,R0
              STI     AR2,@inputp      ; inputp = SOURCE_ADDR
              STI     R0,@inputp2      ; inputp2= SOURCE_ADDR + 2
              STI     R2,@fg2          ; fg2 = nhalb = (FFT_size/2)
              LSH     –1,R2
              SUBI    2,R2,R0
              STI     R0,@fg4m2        ; fg4m2 = NVIERT–2 : (FFT_SIZE/4)–2
              LSH     –1,R2
              SUBI    2,R2,R0
              STI     R0,@fg8m2
*       ar0 : AR + AI
*       ar1 : BR + BI
*       ar2 : CR + CI + CR' + CI'
*       ar3 : DR + DI
*       ar4 : AR' + AI'
*       ar5 : BR' + BI'
*       ar6 : DR' + DI'
*       ar7 : first twiddle factor = 1
        ldi     @fg2,ir0               ; ir0 = n/2 = offset between SOURCE_ADDRs
        ldi     @sintab,ar7            ; ar7 points to twiddle factor 1
        ldi     ar2,ar0                ; ar0 points to AR
        addi    ir0,ar0,ar1            ; ar1 points to BR
        addi    ir0,ar1,ar2            ; ar2 points to CR
        addi    ir0,ar2,ar3            ; ar3 points to DR
        ldi     ar0,ar4                ; ar4 points to AR'
        ldi     ar1,ar5                ; ar5 points to BR'
        ldi     ar3,ar6                ; ar6 points to DR'
        ldi     2,ir1                  ; addressoffset
        lsh     –1,ir0                 ; ir0 = n/4 = number of R4–butterflies
       subi     2,ir0,rc
*******************************************************************************
* ------------ FIRST 2 STAGES AS RADIX–4 BUTTERFLY ------------------------ *
*******************************************************************************
* fill pipeline
        addf    *ar2,*ar0,r4          ; r4 = AR + CR
        subf    *ar2,*ar0++,r5        ; r5 = AR – CR
        addf    *ar1,*ar3,r6          ; r6 = DR + BR
        subf    *ar1++,*ar3++,r7      ; r7 = DR – BR
```

```
        addf    r6,r4,r0            ; AR' = r0 = r4 + r6
        mpyf    *ar7,*ar3++,r1      ; r1 = DI , BR' = r3 = r4 - r6
||      subf    r6,r4,r3
        addf    r1,*ar1,r0          ; r0 = BI + DI , AR' = r0
||      stf     r0,*ar4++
        subf    r1,*ar1++,r1        ; r1 = BI - DI , BR' = r3
||      stf     r3,*ar5++
        addf    r1,r5,r2            ; CR' = r2 = r5 + r1
        mpyf    *ar7,*+ar2,r1       ; r1 = CI , DR' = r3 = r5 - r1
||      subf    r1,r5,r3
        rptbd   blk1                ; Setup for radix-4 butterfly loop
        addf    r1,*ar0,r2          ; r2 = AI + CI , CR' = r2
||      stf     r2,*ar2++(ir1)
        subf    r1,*ar0++,r6        ; r6 = AI - CI , DR' = r3
||      stf     r3,*ar6++
        addf    r0,r2,r4            ; AI' = r4 = r2 + r0
* radix-4 butterfly loop
        mpyf    *ar7,*ar2--,r0      ; r0 = CR , (BI' = r2 = r2 - r0)
||      subf    r0,r2,r2
        mpyf    *ar7,*ar1++,r1      ; r1 = BR , (CI' = r3 = r6 + r7)
||      addf    r7,r6,r3
        addf    r0,*ar0,r4          ; r4 = AR + CR , (AI' = r4)
||      stf     r4,*ar4++
        subf    r0,*ar0++,r5        ; r5 = AR - CR , (BI' = r2)
||      stf     r2,*ar5++
        subf    r7,r6,r7            ; (DI' = r7 = r6 - r7)
        addf    r1,*ar3,r6          ; r6 = DR + BR , (DI' = r7)
||      stf     r7,*ar6++
        subf    r1,*ar3++,r7        ; r7 = DR - BR , (CI' = r3)
||      stf     r3,*ar2++
        addf    r6,r4,r0            ; AR' = r0 = r4 + r6
        mpyf    *ar7,*ar3++,r1      ; r1 = DI , BR' = r3 = r4 - r6
||      subf    r6,r4,r3
        addf    r1,*ar1,r0          ; r0 = BI + DI , AR' = r0
||      stf     r0,*ar4++
        subf    r1,*ar1++,r1        ; r1 = BI - DI , BR' = r3
||      stf     r3,*ar5++
        addf    r1,r5,r2            ; CR' = r2 = r5 + r1
        mpyf    *+ar2,*ar7,r1       ; r1 = CI , DR' = r3 = r5 - r1
||      subf    r1,r5,r3
        addf    r1,*ar0,r2          ; r2 = AI + CI , CR' = r2
||      stf     r2,*ar2++(ir1)
        subf    r1,*ar0++,r6        ; r6 = AI - CI , DR' = r3
||      stf     r3,*ar6++
blk1    addf    r0,r2,r4            ; AI' = r4 = r2 + r0
* clear pipeline
        subf    r0,r2,r2            ; BI' = r2 = r2 - r0
        addf    r7,r6,r3            ; CI' = r3 = r6 + r7
        stf     r4,*ar4             ; AI' = r4 , BI' = r2
||      stf     r2,*ar5
        subf    r7,r6,r7            ; DI' = r7 = r6 - r7
        stf     r7,*ar6             ; DI' = r7 , CI' = r3
||      stf     r3,*--ar2
*****************************************************************************
* ------------ THIRD TO LAST-2 STAGE ---------------------------------------- *
*****************************************************************************
        ldi     @fg2,ir1
        subi    1,ir0,ar5
        ldi     1,ar6
        ldi     @sintab,ar7         ; pointer to twiddle factor
        ldi     0,ar4               ; group counter
        ldi     @inputp,ar0
stufe   ldi     ar0,ar2             ; upper real butterfly output
        addi    ir0,ar0,ar3         ; lower real butterfly output
        ldi     ar3,ar1             ; lower real butterfly input
        lsh     1,ar6               ; double group count
        lsh     -2,ar5              ; half butterfly count
        lsh     1,ar5               ; clear LSB
        lsh     -1,ir0              ; half step from upper to lower real part
        lsh     -1,ir1
        addi    1,ir1               ; step from old imaginary to new
```

```
                                        ; real value
        ldf     *ar1++,r6               ; dummy load, only for address update
||      ldf     *ar7,r7                 ; r7 = COS
gruppe
* fill pipeline
*    ar0 = upper real butterfly input
*    ar1 = lower real butterfly input
*    ar2 = upper real butterfly output
*    ar3 = lower real butterfly output
*    the imaginary part has to follow
        ldf     *++ar7,r6               ; r6 = SIN
        mpyf    *ar1--,r6,r1            ; r1 = BI * SIN
||      addf    *++ar4,r0,r3            ; dummy addf for counter update
        mpyf    *ar1,r7,r0              ; r0 = BR * COS
        ldi     ar5,rc
        rptbd   bfly1                   ; Setup for loop bfly1
        mpyf    *ar7--,*ar1++,r0        ; r3 = TR = r0 + r1 , r0 = BR * SIN
||      addf    r0,r1,r3
        mpyf    *ar1++,r7,r1            ; r1 = BI * COS , r2 = AR - TR
||      subf    r3,*ar0,r2
        addf    *ar0++,r3,r5            ; r5 = AR + TR , BR' = r2
||      stf     r2,*ar3++
*   FIRST BUTTERFLY-TYPE:
*
*   TR = BR * COS + BI * SIN
*   TI = BR * SIN - BI * COS
*   AR'= AR + TR
*   AI'= AI - TI
*   BR'= AR - TR
*   BI'= AI + TI
*       loop bfly1
        mpyf    *+ar1,r6,r5             ; r5 = BI * SIN , (AR' = r5)
||      stf     r5,*ar2++
        subf    r1,r0,r2                ; (r2 = TI = r0 - r1)
        mpyf    *ar1,r7,r0              ; r0 = BR * COS , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++,r4            ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++
        addf    r0,r5,r3                ; r3 = TR = r0 + r5
        mpyf    *ar1++,r6,r0            ; r0 = BR * SIN , r2 = AR - TR
||      subf    r3,*ar0,r2
        mpyf    *ar1++,r7,r1            ; r1 = BI * COS , (AI' = r4)
||      stf     r4,*ar2++
bfly1   addf    *ar0++,r3,r5            ; r5 = AR + TR , BR' = r2
||      stf     r2,*ar3++
* switch over to next group
        subf    r1,r0,r2                ; r2 = TI = r0 - r1
        addf    r2,*ar0,r3              ; r3 = AI + TI , AR' = r5
||      stf     r5,*ar2++
        subf    r2,*ar0++(ir1),r4       ; r4 = AI - TI , BI' = r3
||      stf     r3,*ar3++(ir1)
        nop     *ar1++(ir1)             ; address update
        mpyf    *ar1--,r7,r1            ; r1 = BI * COS , AI' = r4
||      stf     r4,*ar2++(ir1)
        mpyf    *ar1,r6,r0              ; r0 = BR * SIN
        ldi     ar5,rc
        rptbd   bfly2                   ; Setup for loop bfly2
        mpyf    *ar7++,*ar1++,r0        ; r3 = TR = r1 - r0 , r0 = BR * COS
||      subf    r0,r1,r3
        mpyf    *ar1++,r6,r1            ; r1 = BI * SIN , r2 = AR - TR
||      subf    r3,*ar0,r2
        addf    *ar0++,r3,r5            ; r5 = AR + TR , BR' = r2
||      stf     r2,*ar3++
*   SECOND BUTTERFLY-TYPE:
*
*   TR = BI * COS - BR * SIN
*   TI = BI * SIN + BR * COS
*   AR'= AR + TR
*   AI'= AI - TI
*   BR'= AR - TR
*   BI'= AI + TI
```

```
*       loop bfly2
        mpyf    *+ar1,r7,r5           ; r5 = BI * COS , (AR' = r5)
||      stf     r5,*ar2++
        addf    r1,r0,r2              ; (r2 = TI = r0 + r1)
        mpyf    *ar1,r6,r0            ; r0 = BR * SIN , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++,r4          ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++
        subf    r0,r5,r3              ; TR = r3 = r5 - r0
        mpyf    *ar1++,r7,r0          ; r0 = BR * COS , r2 = AR - TR
||      subf    r3,*ar0,r2
        mpyf    *ar1++,r6,r1          ; r1 = BI * SIN , (AI' = r4)
||      stf     r4,*ar2++
bfly2   addf    *ar0++,r3,r5          ; r5 = AR + TR , BR' = r2
||      stf     r2,*ar3++
* clear pipeline
        addf    r1,r0,r2              ; r2 = TI = r0 + r1
        addf    r2,*ar0,r3            ; r3 = AI + TI
||      stf     r5,*ar2++             ; AR' = r5
        cmpi    ar6,ar4
        bned    gruppe                ; do following 3 instructions
        subf    r2,*ar0++(ir1),r4     ; r4 = AI - TI , BI' = r3
||      stf     r3,*ar3++(ir1)
        ldf     *++ar7,r7             ; r7 = COS
||      stf     r4,*ar2++(ir1)        ; AI' = r4
        nop     *ar1++(ir1)           ; branch here
* end of this butterflygroup
        cmpi    4,ir0                 ; jump out after ld(n)-3 stage
        bnzaf   stufe
        ldi     @sintab,ar7           ; pointer to twiddle factor
        ldi     0,ar4                 ; group counter
        ldi     @inputp,ar0
*****************************************************************************
* ------------ SECOND LAST STAGE ------------------------------------------ *
*****************************************************************************
        ldi     @inputp,ar0
        ldi     ar0,ar2               ; upper output
        addi    ir0,ar0,ar1           ; lower input
        ldi     ar1,ar3               ; lower output
        ldi     @sintp2,ar7           ; pointer to twiddle faktor
        ldi     5,ir0                 ; distance between two groups
        ldi     @fg8m2,rc
* fill pipeline
* 1. butterfly: w^0
        addf    *ar0,*ar1,r2          ; AR' = r2 = AR + BR
        subf    *ar1++,*ar0++,r3      ; BR' = r3 = AR - BR
        addf    *ar0,*ar1,r0          ; AI' = r0 = AI + BI
        subf    *ar1++,*ar0++,r1      ; BI' = r1 = AI - BI
* 2. butterfly: w^0
        addf    *ar0,*ar1,r6          ; AR' = r6 = AR + BR
        subf    *ar1++,*ar0++,r7      ; BR' = r7 = AR - BR
        addf    *ar0,*ar1,r4          ; AI' = r4 = AI + BI
        subf    *ar1++(ir0),*ar0++(ir0),r5   ; BI' = r5 = AI - BI
        stf     r2,*ar2++             ; (AR' = r2)
||      stf     r3,*ar3++             ; (BR' = r3)
        stf     r0,*ar2++             ; (AI' = r0)
||      stf     r1,*ar3++             ; (BI' = r1)
        stf     r6,*ar2++             ; AR' = r6
||      stf     r7,*ar3++             ; BR' = r7
        stf     r4,*ar2++(ir0)        ; AI' = r4
||      stf     r5,*ar3++(ir0)        ; BI' = r5
* 3. butterfly: w^M/4
        addf    *ar0++,*+ar1,r5       ; AR' = r5 = AR + BI
        subf    *ar1,*ar0,r4          ; AI' = r4 = AI - BR
        addf    *ar1++,*ar0--,r6      ; BI' = r6 = AI + BR
        subf    *ar1++,*ar0++,r7      ; BR' = r7 = AR - BI
* 4. butterfly: w^M/4
        addf    *+ar1,*++ar0,r3       ; AR' = r3 = AR + BI
        ldf     *-ar7,r1              ; r1 = 0 (for inner loop)
||      ldf     *ar1++,r0             ; r0 = BR (for inner loop)
        rptbd   bf2end                ; Setup for loop bf2end
```

```
        subf    *ar1++(ir0),*ar0++,r2 ; BR' = r2 = AR - BI
        stf     r5,*ar2++             ; (AR' = r5)
||      stf     r7,*ar3++             ; (BR' = r7)
        stf     r6,*ar3++             ; (BI' = r6)
* 5. to M. butterfly:
*       loop bf2end
        ldf     *ar7++,r7             ; r7 = COS , ((AI' = r4))
||      stf     r4,*ar2++
        ldf     *ar7++,r6             ; r6 = SIN , (BR' = r2)
||      stf     r2,*ar3++
        mpyf    *+ar1,r6,r5           ; r5 = BI * SIN , (AR' = r3)
||      stf     r3,*ar2++
        addf    r1,r0,r2              ; (r2 = TI = r0 + r1)
        mpyf    *ar1,r7,r0            ; r0 = BR * COS , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++(ir0),r4     ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++(ir0)
        addf    r0,r5,r3              ; r3 = TR = r0 + r5
        mpyf    *ar1++,r6,r0          ; r0 = BR * SIN , r2 = AR - TR
||      subf    r3,*ar0,r2
        mpyf    *ar1++,r7,r1          ; r1 = BI * COS , (AI' = r4)
||      stf     r4,*ar2++(ir0)
        addf    *ar0++,r3,r5          ; r5 = AR + TR , BR' = r2
||      stf     r2,*ar3++
        mpyf    *+ar1,r6,r5           ; r5 = BI * SIN , (AR' = r5)
||      stf     r5,*ar2++
        subf    r1,r0,r2              ; (r2 = TI = r0 - r1)
        mpyf    *ar1,r7,r0            ; r0 = BR * COS , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++,r4          ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++
        addf    r0,r5,r3              ; r3 = TR = r0 + r5
        mpyf    *ar1++,r6,r0          ; r0 = BR * SIN , r2 = AR - TR
||      subf    r3,*ar0,r2
        mpyf    *ar1++(ir0),r7,r1     ; r1 = BI * COS , (AI' = r4)
||      stf     r4,*ar2++
        addf    *ar0++,r3,r3          ; r3 = AR + TR , BR' = r2
||      stf     r2,*ar3++
        mpyf    *+ar1,r7,r5           ; r5 = BI * COS , (AR' = r3)
||      stf     r3,*ar2++
        subf    r1,r0,r2              ; (r2 = TI = r0 - r1)
        mpyf    *ar1,r6,r0            ; r0 = BR * SIN , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++(ir0),r4     ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++(ir0)
        subf    r0,r5,r3              ; r3 = TR = r5 - r0
        mpyf    *ar1++,r7,r0          ; r0 = BR * COS , r2 = AR - TR
||      subf    r3,*ar0,r2
        mpyf    *ar1++,r6,r1          ; r1 = BI * SIN , (AI' = r4)
||      stf     r4,*ar2++(ir0)
        addf    *ar0++,r3,r5          ; r5 = AR + TR , BR' = r2
||      stf     r2,*ar3++
        mpyf    *+ar1,r7,r5           ; r5 = BI * COS , (AR' = r5)
||      stf     r5,*ar2++
        addf    r1,r0,r2              ; (r2 = TI = r0 + r1)
        mpyf    *ar1,r6,r0            ; r0 = BR * SIN , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++,r4          ; (r4 = AI - TI , y(L) = BI' = r3)
||      stf     r3,*ar3++
        subf    r0,r5,r3              ; r3 = TR = r5 - r0
        mpyf    *ar1++,r7,r0          ; r0 = BR * COS , r2 = AR - TR
||      subf    r3,*ar0,r2
bf2end  mpyf    *ar1++(ir0),r6,r1     ; r1 = BI * SIN , r3 = AR + TR
||      addf    *ar0++,r3,r3
* clear pipeline
        stf     r2,*ar3++             ; BR' = r2 , AI' = r4
||      stf     r4,*ar2++
        addf    r1,r0,r2              ; r2 = TI = r0 + r1
        addf    r2,*ar0,r3            ; r3 = AI + TI , AR' = r3
||      stf     r3,*ar2++
        subf    r2,*ar0,r4            ; r4 = AI - TI , BI' = r3
```

```
||      stf     r3,*ar3
        stf     r4,*ar2                 ; AI' = r4
*******************************************************************************
*-------------- LAST STAGE ---------------------------------------------------*
*******************************************************************************
        ldi     @inputp,ar0
        ldi     ar0,ar2                 ; upper output
        ldi     @inputp2,ar1
        ldi     ar1,ar3                 ; lower output
        ldi     @sintp2,ar7             ; pointer to twiddle factors
        ldi     3,ir0                   ; group offset
        ldi     @fg4m2,rc
* fill pipeline
* 1. butterfly: w^0
        addf    *ar0,*ar1,r6            ; AR' = r6 = AR + BR
        subf    *ar1++,*ar0++,r7        ; BR' = r7 = AR - BR
        addf    *ar0,*ar1,r4            ; AI' = r4 = AI + BI
        subf    *ar1++(ir0),*ar0++(ir0),r5   ; BI' = r5 = AI - BI
* 2. butterfly: w^M/4
        addf    *+ar1,*ar0,r3           ; AR' = r3 = AR + BI
        ldf     *-ar7,r1                ; r1 = 0 (for inner loop)
||      ldf     *ar1++,r0               ; r0 = BR (for inner loop)
        rptbd   bflend                  ; Setup for loop bflend
        subf    *ar1++(ir0),*ar0++,r2   ; BR' = r2 = AR - BI
        stf     r6,*ar2++               ; (AR' = r6)
||      stf     r7,*ar3++               ; (BR' = r7)
        stf     r5,*ar3++(ir0)          ; (BI' = r5)
* 3. to M. butterfly:
*       loop bflend
        ldf     *ar7++,r7               ; r7 = COS , ((AI' = r4))
||      stf     r4,*ar2++(ir0)
        ldf     *ar7++,r6               ; r6 = SIN , (BR' = r2)
||      stf     r2,*ar3++
        mpyf    *+ar1,r6,r5             ; r5 = BI * SIN , (AR' = r3)
||      stf     r3,*ar2++
        addf    r1,r0,r2                ; (r2 = TI = r0 + r1)
        mpyf    *ar1,r7,r0              ; r0 = BR * COS , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++(ir0),r4       ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++(ir0)
        addf    r0,r5,r3                ; r3 = TR = r0 + r5
        mpyf    *ar1++,r6,r0            ; r0 = BR * SIN , r2 = AR - TR
||      subf    r3,*ar0,r2
        mpyf    *ar1++(ir0),r7,r1       ; r1 = BI * COS , (AI' = r4)
||      stf     r4,*ar2++(ir0)
        addf    *ar0++,r3,r3            ; r3 = AR + TR , BR' = r2
||      stf     r2,*ar3++
        mpyf    *+ar1,r7,r5             ; r5 = BI * COS , (AR' = r3)
||      stf     r3,*ar2++
        subf    r1,r0,r2                ; (r2 = TI = r0 - r1)
        mpyf    *ar1,r6,r0              ; r0 = BR * SIN , (r3 = AI + TI)
||      addf    r2,*ar0,r3
        subf    r2,*ar0++(ir0),r4       ; (r4 = AI - TI , BI' = r3)
||      stf     r3,*ar3++(ir0)
        subf    r0,r5,r3                ; r3 = TR = r0 - r5
        mpyf    *ar1++,r7,r0            ; r0 = BR * COS , r2 = AR - TR
||      subf    r3,*ar0,r2
bflend  mpyf     *ar1++(ir0),r6,r1      ; r1 = BI * SIN , r3 = AR + TR
||      addf    *ar0++,r3,r3
* clear pipeline
        stf     r2,*ar3++               ; BR' = r2 , (AI' = r4)
||      stf     r4,*ar2++(ir0)
        addf    r1,r0,r2                ; r2 = TI = r0 + r1
        addf    r2,*ar0,r3              ; r3 = AI + TI , AR' = r3
||      stf     r3,*ar2++
        subf    r2,*ar0,r4              ; r4 = AI - TI , BI' = r3
||      stf     r3,*ar3
        stf     r4,*ar2                 ; AI' = r4
*******************************************************************************
*-------------- END OF FFT ---------------------------------------------------*
*******************************************************************************
```

```
end:
;
; Return to C environment.
;
        POP     DP                      ; Restore C environment variables.
        POP     AR7
        POP     AR6
        POP     AR5
        POP     AR4
        POP     AR3
        POPF     R7
        POP      R7
        POPF     R6
        POP      R6
        POP      R5
        POP      R4
        RETS
 .end
```

WAITDMA.ASM

```
*****************************************************************
*
*  WAIT_DMA.ASM: TMS320C40 C–callable routine to check if an IIF bit
*                is set. If that is the case, the beit gets cleared.
*
*  Calling conventions:
*
*  void wait_dma(int mask)
*                  ar2
*
*  where       mask     : mask word (”1” in the corresponding IIF bit)
*
*****************************************************************
        .global _wait_dma
        .text
_wait_dma:
        .if .REGPARM == 0
        ldi     sp,ar0
        ldi     *–ar0(1),ar2               ; mask word
        .endif
wait:   tstb    ar2,iif
        bz      wait
        andn    ar2,iif
        rets
```

Appendix G: Input Vector and Sine Table Examples

SINTAB.ASM

```
*****************************************************************
*
*   SINTAB.ASM : Table with twiddle factors for a 64–point DIF FFT
*
*****************************************************************
        .global _SINE
        .sect ".sintab"
_SINE
        .float  0.000000
        .float  0.098017
        .float  0.195090
        .float  0.290285
        .float  0.382683
        .float  0.471397
        .float  0.555570
        .float  0.634393
        .float  0.707107
        .float  0.773010
        .float  0.831470
        .float  0.881921
        .float  0.923880
        .float  0.956940
        .float  0.980785
        .float  0.995185
_COS
        .float  1.000000
        .float  0.995185
        .float  0.980785
        .float  0.956940
        .float  0.923880
        .float  0.881921
        .float  0.831470
        .float  0.773010
        .float  0.707107
        .float  0.634393
        .float  0.555570
        .float  0.471397
        .float  0.382683
        .float  0.290285
        .float  0.195090
        .float  0.098017
        .float  –0.000000
        .float  –0.098017
        .float  –0.195090
        .float  –0.290285
        .float  –0.382683
        .float  –0.471397
        .float  –0.555570
        .float  –0.634393
        .float  –0.707107
        .float  –0.773010
        .float  –0.831470
        .float  –0.881921
        .float  –0.923880
        .float  –0.956940
        .float  –0.980785
        .float  –0.995185
        .float  –1.000000
        .float  –0.995185
        .float  –0.980785
        .float  –0.956940
        .float  –0.923879
        .float  –0.881921
        .float  –0.831470
        .float  –0.773010
        .float  –0.707107
```

```
        .float  –0.634393
        .float  –0.555570
        .float  –0.471397
        .float  –0.382683
        .float  –0.290285
        .float  –0.195090
        .float  –0.098017
        .float  0.000000
        .float  0.098017
        .float  0.195090
        .float  0.290285
        .float  0.382684
        .float  0.471397
        .float  0.555570
        .float  0.634393
        .float  0.707107
        .float  0.773011
        .float  0.831470
        .float  0.881921
        .float  0.923880
        .float  0.956940
        .float  0.980785
        .float  0.995185
        .end
```

SINTABR.ASM

```
*****************************************************************
*
*   SINTABR.ASM : Sine table for a 64–point DIT FFT
*
*****************************************************************
        .global _SINE
        .sect ”.sintab”
_SINE
        .float  1.000000
        .float  0.000000
        .float  0.707107
        .float  0.707107
        .float  0.923880
        .float  0.382683
        .float  0.382683
        .float  0.923880
        .float  0.980785
        .float  0.195090
        .float  0.555570
        .float  0.831470
        .float  0.831470
        .float  0.555570
        .float  0.195090
        .float  0.980785
        .float  0.995185
        .float  0.098017
        .float  0.634393
        .float  0.773010
        .float  0.881921
        .float  0.471397
        .float  0.290285
        .float  0.956940
        .float  0.956940
        .float  0.290285
        .float  0.471397
        .float  0.881921
        .float  0.773010
        .float  0.634393
        .float  0.098017
        .float  0.995185
```

INPUT.ASM

```
*************************************************************************
 INPUT.ASM: 64-point complex input vector
*************************************************************************
        .global _INPUT
        .sect   ".input"
_INPUT
        .float 10.0,26.0          ;[0]
        .float 10.0,22.0          ;[1]
        .float 37.0,16.0          ;[2]
        .float 15.0,35.0          ;[3]
        .float 6.0,28.0           ;[4]
        .float 38.0,4.0           ;[5]
        .float 39.0,11.0          ;[6]
        .float 0.0,12.0           ;[7]
        .float 1.0,12.0           ;[8]
        .float 7.0,23.0           ;[9]
        .float 1.0,39.0           ;[10]
        .float 25.0,30.0          ;[11]
        .float 29.0,14.0          ;[12]
        .float 11.0,12.0          ;[13]
        .float 16.0,19.0          ;[14]
        .float 11.0,1.0           ;[15]
        .float 33.0,35.0          ;[16]
        .float 30.0,14.0          ;[17]
        .float 35.0,19.0          ;[18]
        .float 12.0,1.0           ;[19]
        .float 8.0,9.0            ;[20]
        .float 24.0,26.0          ;[21]
        .float 23.0,12.0          ;[22]
        .float 4.0,6.0            ;[23]
        .float 31.0,39.0          ;[24]
        .float 20.0,27.0          ;[25]
        .float 12.0,35.0          ;[26]
        .float 26.0,28.0          ;[27]
        .float 2.0,27.0           ;[28]
        .float 9.0,14.0           ;[29]
        .float 23.0,29.0          ;[30]
        .float 21.0,26.0          ;[31]
        .float 38.0,30.0          ;[32]
        .float 19.0,5.0           ;[33]
        .float 33.0,30.0          ;[34]
        .float 29.0,13.0          ;[35]
        .float 22.0,5.0           ;[36]
        .float 17.0,13.0          ;[37]
        .float 28.0,36.0          ;[38]
        .float 18.0,20.0          ;[39]
        .float 0.0,16.0           ;[40]
        .float 22.0,2.0           ;[41]
        .float 35.0,27.0          ;[42]
        .float 18.0,36.0          ;[43]
        .float 39.0,36.0          ;[44]
        .float 19.0,8.0           ;[45]
        .float 17.0,1.0           ;[46]
        .float 21.0,35.0          ;[47]
        .float 0.0,35.0           ;[48]
        .float 1.0,10.0           ;[49]
        .float 15.0,17.0          ;[50]
        .float 27.0,23.0          ;[51]
        .float 31.0,32.0          ;[52]
        .float 33.0,13.0          ;[53]
        .float 33.0,34.0          ;[54]
        .float 18.0,6.0           ;[55]
        .float 10.0,6.0           ;[56]
        .float 14.0,4.0           ;[57]
        .float 39.0,31.0          ;[58]
        .float 10.0,6.0           ;[59]
        .float 11.0,24.0          ;[60]
        .float 15.0,12.0          ;[61]
        .float 6.0,23.0           ;[62]
        .float 20.0,4.0           ;[63]
```

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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 semiconductor products or services might be or are used.