Texas Instruments

# Parallel Processing With the TMS320C4x

## Application Guide

1994 Digital Signal Processing Products

TEXAS
INSTRUMENTS

Printed in U.S.A., February 1994

SPRA031

Application
Guide

# Parallel Processing With the TMS320C4x

1994

# Parallel Processing With the TMS320C4x

## Application Guide

SPRA031
February 1994

PRINTED WITH SOY INK™

TEXAS INSTRUMENTS

Printed on Recycled Paper

# Part 1

# Introduction to Parallel Processing

# Part 2

# Hardware Applications

# Part 3

# Software Algorithms

# Part 4

# End Applications

## IMPORTANT NOTICE

Texas Instruments Incorporated (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 current specifications 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.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

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.

## WARNING

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

## Preface

The exponentially increasing demands for computationally intensive realtime signal processing applications outpace the performance improvements of each new generation of processors. With each passing year, more applications require tremendous computing capabilities that current processors cannot achieve. By coordinating the activities of individual processors, parallel processing offers the performance required by these applications. DSP applications are particularly suited for parallel processing because they are computationally intensive, highly parallel, highly structured, and often, periodic. Moreover, parallel multiprocessor systems have many benefits over single-processor systems. Parallel multiprocessor systems have virtually unlimited performance, better fault tolerance, scalability, flexibility, and upgradability. For these reasons, many high-end DSP applications, such as imaging, graphics, and data processing already benefit from parallel processing.

Witnessing this evolution, Texas Instruments specifically developed its general-purpose TMS320C4x parallel digital signal processors and their development tools. The TMS320C4x not only has a high-performance floating-point/integer CPU that achieves 275 MOPS and 50 MFLOPS, but also has high data throughput peripherals that can transfer 320 MBytes/s. To facilitate interprocessor communication, the ’C4x has two 32-bit external buses and six byte-wide communication ports that can exchange data and programs. For ease of use, the ’C4x architecture supports an efficient C compiler. Furthermore, you can program and debug any size multiprocessor system with one set of development tools.

This book introduces you to parallel processing applications with the ’C4x and is divided into four parts:

**PART I . . . . . . . . . . . . . Introduction to Parallel Processing**

**PART II . . . . . . . . . . . . . Hardware Implementations**

**PART III . . . . . . . . . . . . Software Algorithms**

**PART IV . . . . . . . . . . . . End Applications**

The editor and authors hope that you find this application book useful and gain valuable information to assist you in designing parallel processing systems with the TMS320C4x. In addition, the editor thanks all of the authors and reviewers for their contribution to this volume of application reports.

Michael D. Luczak
Digital Signal Processing
Texas Instruments Incorporated

# Contents

*Page*

## PART I — INTRODUCTION TO PARALLEL PROCESSING

## PART II — HARDWARE APPLICATIONS

## PART III — SOFTWARE ALGORITHMS

## PART IV — END APPLICATIONS

# List of Figures

## List of Tables

*Table* *Title* *Page*

# Parallel Digital Signal Processing: An Emerging Market

Mitch Reifel and Daniel Chen
Digital Signal Processing Products — Semiconductor Group
Texas Instruments Incorporated

## Introduction

During the past decade, while CPU performance increased from 5 MIPS in the early 1980s to over 40 MIPS today, applications performance developed exponentially, especially in imaging, graphics, and high-end data processing. This “Malthusian” effect, in conjunction with the “silicon wall”, has created a situation in which application needs have vastly outpaced the ability of single processors to keep up.

This condition inspired rapid development in parallel processing, especially in digital signal processing (DSP). Currently, 75 to 80% of all 32-bit, floating-point DSP applications use multiple processors in their design for several reasons. First, DSP algorithms are inherently suited to task partitioning and, thus, to parallel processing solutions. Second, as the cost of single-chip DSPs decrease, using multiple DSPs in a system becomes increasingly cost effective. Third, the high data throughput, real-time processing capability, and intrinsic on-chip parallelism of DSPs make them especially suitable for multiprocessing systems.

Simply put, parallel processing uses multiple processors working together to solve a single task. Processors can either solve different portions of the same problem simultaneously or work on the same portion of a problem concurrently.

This paper discusses digital signal parallel processing as well as the reasons why DSP and parallel processing have become a natural match:

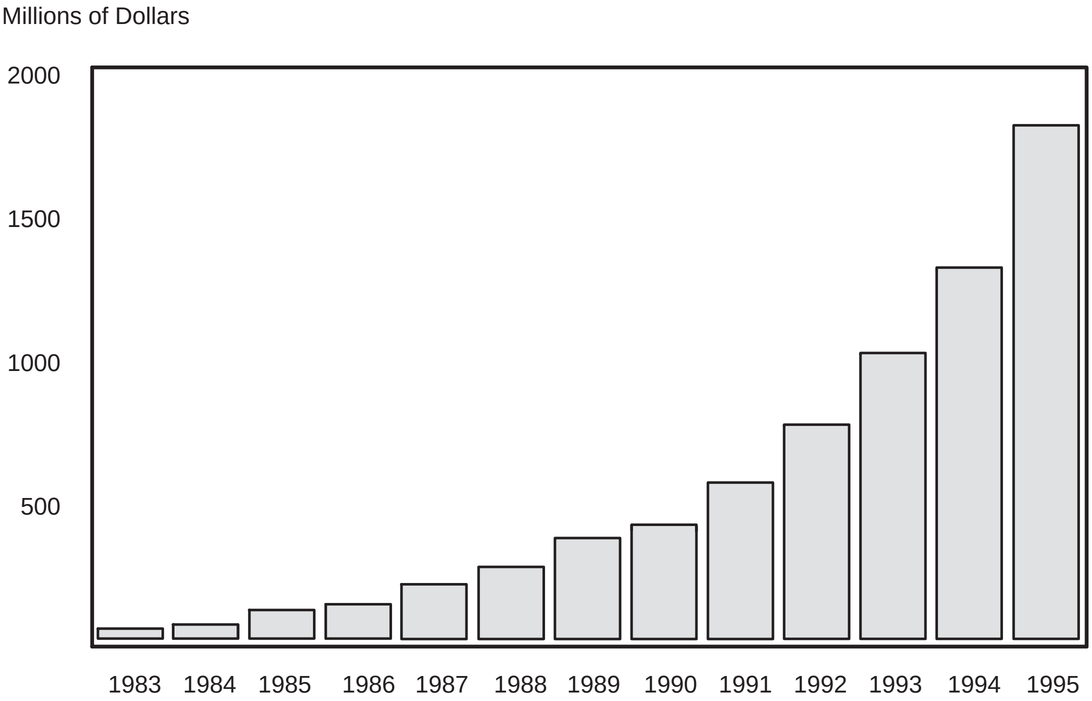
- Advances in CPU architectures.
- New developments in hardware development tools.
- The emergence of software languages and operating systems for multiprocessing.

This paper looks at solutions from different vendors as well as trends in the industry as a whole.

## The Technology Merge

The first practical single-chip DSPs were introduced in the early 1980s. Because of their real-time processing capability, high throughput, and intensive math-processing capability, DSPs began to replace general-purpose processors in many applications. These applications were well suited for real-time processing such as speech processing, telecommunications, and high-speed control. They also pushed DSP to the forefront of technology and created one of the fastest going markets of the decade (see Figure 1).

**Figure 1. Worldwide Single-Chip DSP Market**

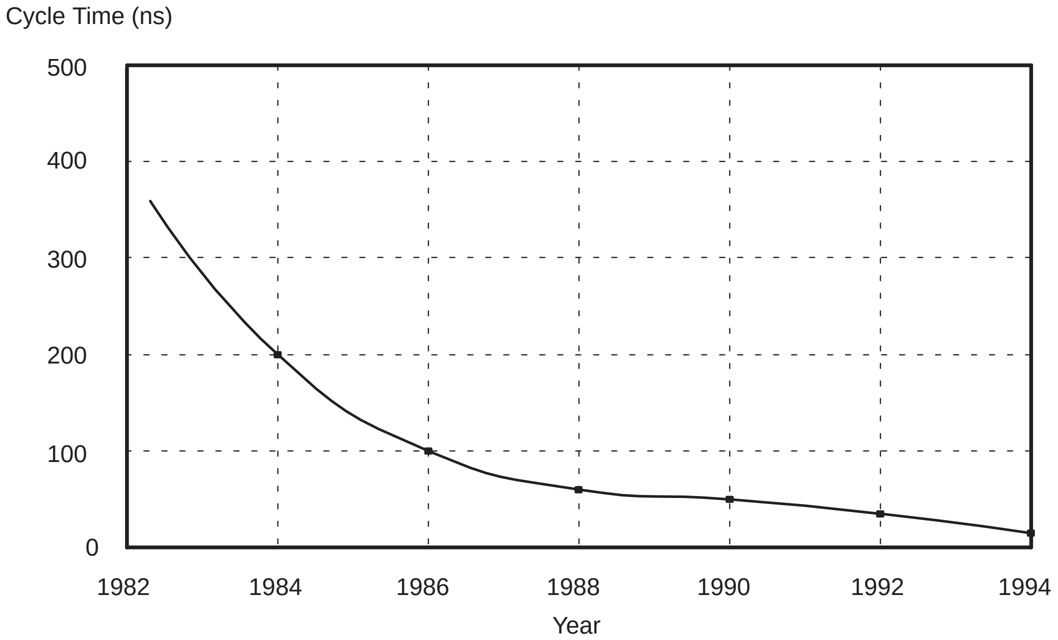


*The DSP market was one of the fastest growing markets of the 1980s. Parallel processing is predicted to follow a similar pattern in the 1990s.*

DSPs are now used in a broad range of nontraditional applications, such as graphics, imaging, and servo-control, that were not originally thought of as part of the signal processing domain. Application designers turned to DSPs because their cycle times were faster than those of general-purpose and RISC architectures. By the middle 1980s, however, cycle time improvements in each new generation became smaller.

In the 1990s, processor manufacturers are approaching the physical limitations of silicon and can no longer rely on smaller geometries alone for increasing processor performance for next generation products, as shown in Figure 2.

**Figure 2. DSP Performance Evolution**

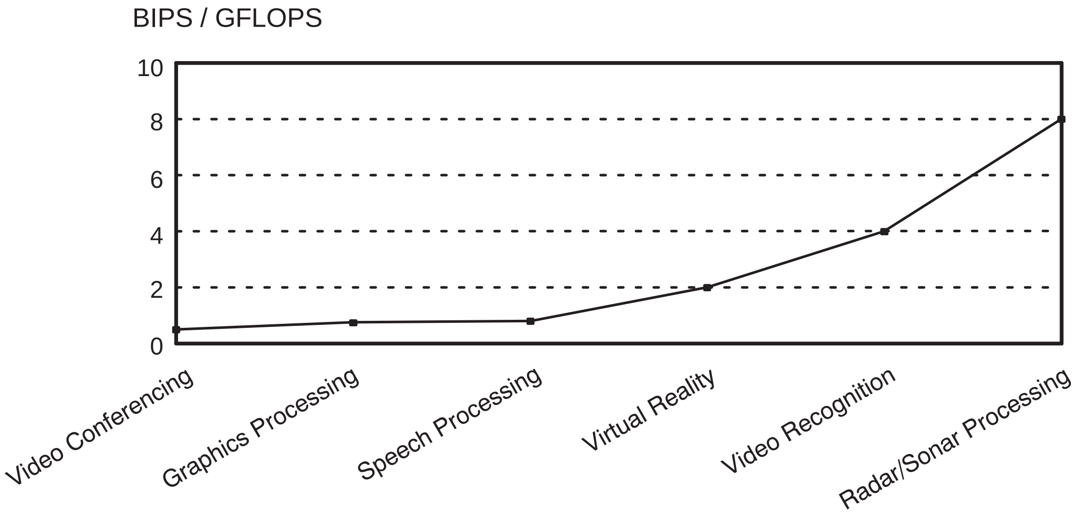


*All semiconductor manufacturers are approaching the “silicon wall” and are looking at different multiprocessor solutions to get around the problem.*

In the meantime, tasks that were unheard of just a few years ago — such as virtual reality and video recognition — are pushing the envelope of performance requirements. Figure 3 shows the trend with actual designs that use TMS320 DSPs.

Multiprocessing meets these challenges. However, multiprocessing comes in different forms. Some manufacturers gain performance improvements with on-board architectural enhancements, but this technique alone cannot meet every need.

**Figure 3. Performance Requirements (Actual TMS320 Designs)**



*Continued growth in application requirements demands intensive development in processor technology.*

## On-Chip Vs. Off-Chip Parallel Processing

Parallel processing enhancements can be divided into two broad categories: on-chip and off-chip. On-chip parallelism relies on architectural enhancements for improved performance, while off-chip parallelism incorporates additional processors.

### On-Chip Parallel Processing

Architectural enhancements on RISC processors can be grouped into three distinct categories: superpipelining, superscaling, and multi-CPU integration.

**Superpipelining** This technique breaks the instruction pipeline into smaller pipeline stages, allowing the CPU to start executing the next instruction before completing the previous one. The processor can run multiple instructions simultaneously, with each instruction being at a different stage of completion.

The main drawbacks of this technique are the increased level of control logic on the processor, difficulty in programming, and difficulty in task switching. Real-time multitasking on a superpipelined processor can become impossible if the pipeline grows too deep.

Processors that use superpipelining are the Intel i860 and the MIPS R4000.

**Superscaling** Instead of breaking the pipeline into smaller stages, superscaling creates multiple pipelines within a processor, allowing the CPU to execute multiple instructions simultaneously.

However, when multiple instructions are executed simultaneously, any data dependency between the instructions (such as a conditional branch) increases the complexity of the programming. Programmers must make certain that simultaneously executed instructions don't need the same on-chip resource, or that one executing instruction doesn't need the result of another whose result is not yet available.

Digital's Alpha processor is one example of a CPU that uses superscaling.

**Multi-CPU Integration** This technique goes a step further than the preceding techniques and integrates multiple CPUs into a single piece of silicon. The number of processors may vary, depending on chip size, power dissipation, and pin count.

Star Semiconductor's SPROC and the soon-to-be-announced MVP (Multimedia Video Processor) advanced imaging processor from TI implement this technique.

All three of these parallel processing techniques increase processor performance without the need for dramatic cycle time improvement. None of the techniques, however, can achieve the BIPS performance required by today's applications. If an application demands higher performance than on-chip processors can deliver, the solution must be multiple processors.

### Off-Chip Parallel Processing

Off-chip parallel processing is not necessarily better — it's inevitable. No single processor, no matter how it is pipelined, how it is scaled, or how many CPUs it has on board, can handle all applications. Recognizing this, manufacturers developed techniques to integrate multiple processors efficiently. Like building blocks, off-chip parallel processors connect easily to form expandable systems of virtually infinite size and variety.

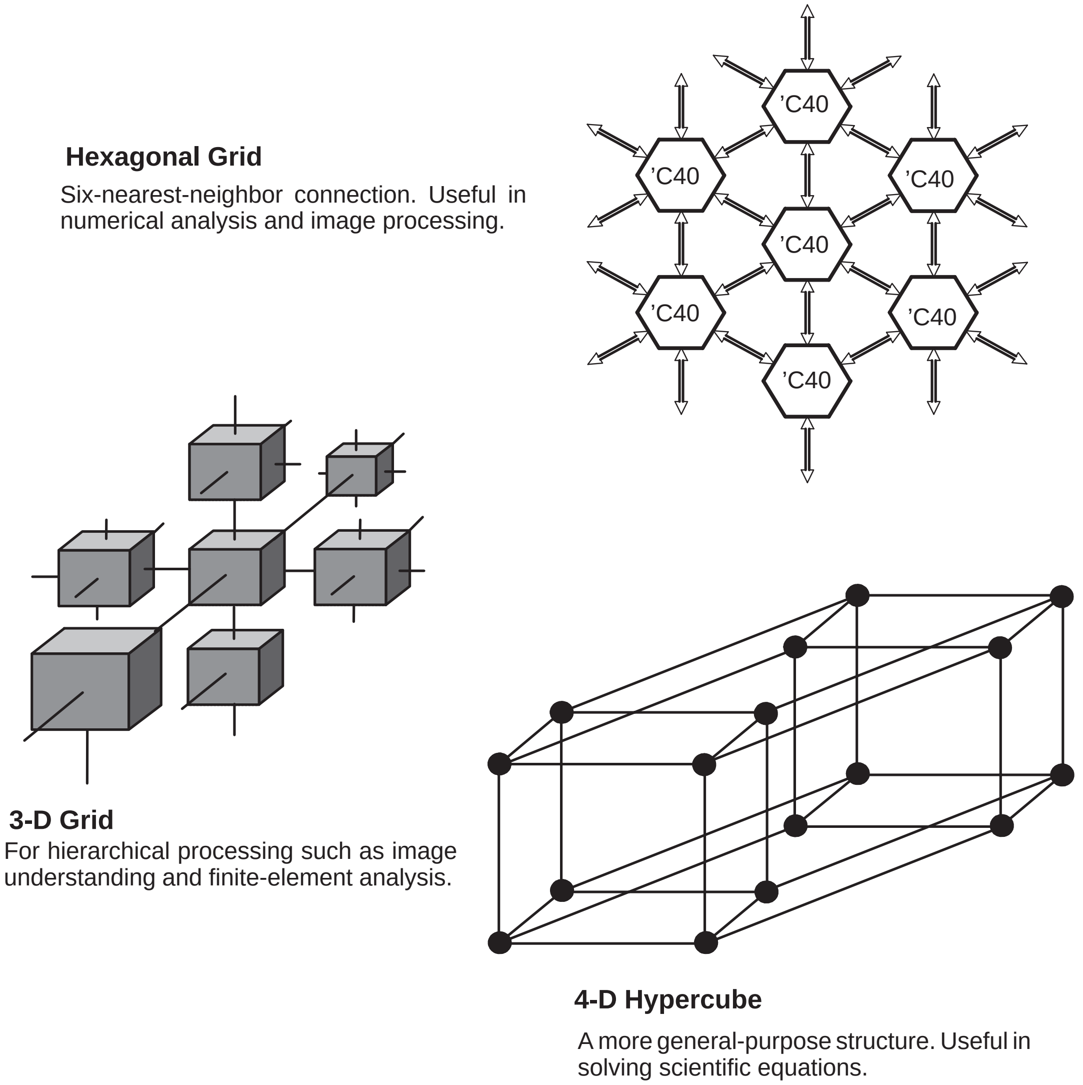
Two processors employ this technique: the Inmos Transputer and the Texas Instruments TMS320C40. Both of these processors also incorporate on-chip parallel processing features to achieve high individual performance. The latest generation Transputer, T9000, uses superpipelining, while the 'C40 uses superscaling. These processors offer both the high performance of the on-chip parallel processing architectural enhancements and the extra features of off-chip expansion.

Off-chip expansion is achieved by connecting multiple processors together with zero glue logic for direct processor-to-processor communication. While methods are different, (TI uses six 8-bit parallel communication ports; Inmos uses four serial links), the concept is the same: connect multiple processors together to create a topology or array of virtually any size to achieve the performance needed by high-end applications (see Figure 4). The communication ports (or links) on the devices are supplemented by parallel memory buses and other support peripherals, allowing designers broad flexibility in designing their systems.

These are some benefits of off-chip parallelism:

- **Expandability** — You can easily add more processors to your system to meet performance requirements.
- **Flexibility** — You can implement a wide array of processor topologies that best fit your application needs. Unlike hardwired multi-CPU integration, off-chip processing can implement everything from 1D pipelines to 4D hypercubes.
- **Upgradability** — With processors that connect like building blocks, systems can be designed in a modular fashion, allowing extra processing power to be added at a later date to meet expanding processing needs.

**Figure 4. TMS320C40 System Architectures**



**Hexagonal Grid**

Six-nearest-neighbor connection. Useful in numerical analysis and image processing.

**3-D Grid**

For hierarchical processing such as image understanding and finite-element analysis.

**4-D Hypercube**

A more general-purpose structure. Useful in solving scientific equations.

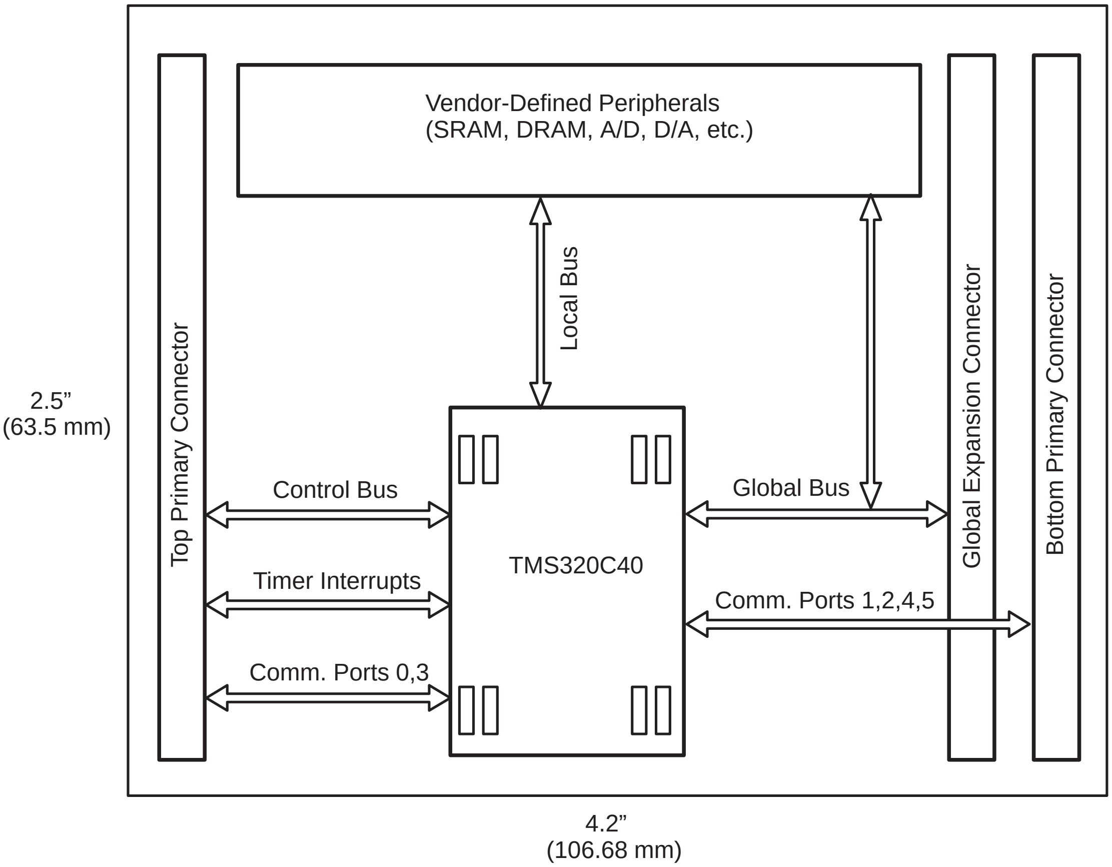
*The TMS320C40 has six interprocessor communication ports for creating topologies of virtually any size and type.*

## Processing Modules

Upgrading, expanding, and integrating parallel systems is even easier with processing modules than with processors. TRAMs (Transputer Modules) for the Transputer and TIMs (TI Modules) for the ’C40 provide an open standard, easy-to-use approach that saves time.

The TIM-40 and TRAM describe modular building blocks for prototyping and manufacturing parallel-processing systems. Both standards consist of a daughterboard module that can include a parallel processor, memory, A/D-D/A conversion, and other functions as required. System designs can contain any number of modules, limited only by the amount of room in the system (see Figure 5).

**Figure 5. TIM-40 Module**



*The architecture of the TIM-40 gives you both a standard interface to build parallel processing systems and also the flexibility to add support peripherals and features that best fit your application.*

Designs based on the modules can be scaled and upgraded easily as system performance requirements increase. Furthermore, modules used in development activities can be reused in new programs.

The modular approach helps designers enhance system reliability. In a massively parallel system that requires 100 ’C40s, TIM-40 modules can reduce the challenge of more than 3,000 pin connections to a task of only 200 daughterboard-to-motherboard connections.

These architectural enhancements make hardware design and integration of multiple processors easy, but they do not address debugging and programming the large parallel systems that result. This is where the ’C40 and Transputer differ. The designers of the ’C40, realizing the problems in debugging large parallel systems, built into the processor features that allow unique multiprocessing debugging capabilities.

## Multiprocessing Emulation

Programming and debugging single, serial processors has always been difficult. Programming the enhanced processors in multiprocessor systems is even more difficult. Prior to the availability of the ’C40 and its development tools, developers used tools intended for uniprocessor architectures to design and debug multiprocessor systems. While such tools were satisfactory for their original purpose, they were difficult to use with embedded processors in parallel architectures. Designers used multiple emulators and/or complicated software monitors to debug their parallel systems. These tools provided neither system synchronization, unintrusive real-time operation, or the fine detail required to design and debug embedded parallel processors.

The ’C40 XDS510 in-circuit, scan-based emulator incorporates the same cutting-edge tasks that are used for parallel supercomputing. It supports global starting, stopping, and single-stepping of multiple ’C40s in a target system. It also has the capability to halt all the ’C40s in a system if a single ’C40 hits a breakpoint.

This parallel debug capability of the XDS510 is supported by the on-chip analysis logic designed into the ’C40. The XDS510 can access the analysis module to efficiently debug, monitor, and analyze the internal operation of the device. The analysis module consists of an analysis control block, an analysis input block, and a JTAG test/emulation interface block. The module features program, data, and DMA breakpoints, a program counter trace-back buffer, and a dedicated timer for profiling.

A single XDS510 emulator can perform mouse-driven, windowed debugging of C and assembly language for all the ’C40 processors in your system, regardless of the complexity of the topology. It also determines whether the system load is balanced across the processors.

The TMS320C40 is the only parallel processor that has this emulation and debugging feature.

## Software Development

One of the largest problems facing developers of multiprocessing systems is programming. Issues such as program partitioning, load balancing, and program routing present unique difficulties. Various solutions have been offered:

**Graphical Programming Languages** — Comdisco Systems recently introduced Multiprox — the first graphical programming environment for developing systems that employ multiple TMS320C40s. Multiprox lets you partition a signal flow block diagram into regions for separate processors to execute. Multiprox automatically generates code for each processor, then compiles and downloads the code with all the necessary interprocessor communication. As a result, you can develop algorithms in less time, and the development process is simplified for those who are not parallel processing experts. Topologies of any size and variation can be used with the system.

**Operating Systems (OS)** — Various operating systems are available to help designers implement realtime multiprocessor systems:

- Helios is a distributed parallel operating system designed to run on multiple-instruction/multiple-data (MIMD) architectures, making it ideal for use in processing modules. After the OS is distributed across the network, each processor runs the Helios nucleus, and they all operate together as a single processing resource. The UNIX-like interface and Posix programming interface allows developers familiar with these environments to program on the ’C40 quickly and easily.
- SPOX offers a hardware development platform and run-time support for real-time systems, thereby simplifying the development of embedded multitasking applications. ’C40 SPOX provides comprehensive sets of parallel DSP operations and includes a high-level software interface that makes it easy to utilize the ’C40’s communication ports and DMA coprocessor. SPOX supports both multiprocessing and multitasking applications.
- RTXC/MP for the ’C40 is designed for complex distributed systems with large arrays of processors and has support for fault-tolerant systems.

**Parallel Programming Languages** — Programming languages are emerging to help the programmer implement software across multiple processors. Parallel C for the ’C40 has been introduced by 3L Ltd. Parallel C is a full implementation of C with many additional features that support parallel processing. The compatibility with C allows existing single processor applications to be ported easily and quickly to parallel systems while the parallel processing features facilitate easier network programming and communications. Other languages available on the ’C40 include ANSI C and Ada, both of which come with multiprocessing support.

## Emerging Trends

One of the questions usually asked about the flattening in performance of silicon speed is, “What about gallium arsenide?” (also called GaAs). To date, no semiconductor manufacturer has planned mass production of GaAs-based processors, and it will probably be another decade before GaAs processors make it onto the market. When GaAs processors do appear, their performance by itself still won’t meet the requirements of the newest applications. Multiprocessing, even with GaAs processors, will be a necessity.

A more imminent trend is multichip modules (MCMs). This is simply an extension of the off-board processing theory that puts multiple processors into a single package, thus requiring smaller pin count and board area than if the processors were used separately. MCMs provide the best of off-chip and on-chip parallel processing. They offer the improved thermal management, power distribution, and signal integrity of signal processors, as well as the flexibility, upgradability, and expansibility of off-chip parallel processing. TI has already announced dual and quad ’C40 MCMs. Even higher integration with new packaging advancements, such as 3-D packaging, are planned.

## Conclusion

The inability of single-chip processors to keep up with the expanding needs of emerging applications makes parallel processing potentially one of the most rapidly growing technologies of the 1990s.

On-chip parallelism can improve performance only to a certain degree. Off-chip parallel processing can increase the performance almost infinitely. Three key factors of parallel processing have been identified: interprocessor communication, parallel debugging, and parallel programming. Two processors, the Texas Instruments ’C40 and Inmos Transputer, were discussed. While both processors incorporate features for high-speed processing and off-chip interprocessor communication, only the ’C40 has the on-chip debug capability and the programming tools needed for programming arrays of processors of arbitrary size and complexity.

## Bibliography

Peterson, Robert, and John Scoggan, *Electronic Packaging in DSEG*, Texas Instruments Technical Journal, Volume 9. No. 3, May–June 1992.

Simar, Ray, *The TMS320C40 and Its Application Development Environment: A DSP for Parallel Processing*, International Conference on Parallel Processing, Volume 1, p. 149–151.

Weiss, Ray, “Third Generation RISC Processors,” *EDN*, March 30, 1992, p. 96–108.

# Parallel Processing With the TMS320C40 Parallel Digital Signal Processor

**Yogendra Jain**
**Sonitech International Inc.**

## Introduction

This paper examines parallel processing using the Texas Instruments TMS320C40 floating-point processor. It demonstrates popular parallel architecture topologies such as hypercube, mesh, ring, and pyramid with the ’C40 and discusses the tradeoffs and performance of these ’C40-based architectures. This paper is divided into the following sections:

- **Overview**

  Tells why the ’C40 architecture is ideal for parallel processing and describes a VME-based ’C40 board.
- **Parallel Processing Topologies**

  Describes different parallel architectures such as hypercube, pyramid, mesh, ring, and tree. Also discusses designing of massively parallel systems using the principle of reconfigurability.
- **TMS320C40-Based AT/ISA and VME Architecture**

  Discusses the architecture of VME and AT/ISA TMS320C40-based boards that are expandable from two nodes capable of 100 MFLOPS (million floating-point operations per second) to hundreds of nodes.
- **System-Level Design Issues With the TMS320C40 Communication Ports**

  Discusses the ’C40 node reset and offers design suggestions.
- **Matrix Multiplication Application of Parallel Processing**

  Explains matrix multiplication.
- **Parallel Processing Architecture Topologies**

  Describes the hypercube and its properties and mesh topologies.
- **Reconfigurable Massively Parallel Processors**

  Discusses software approaches and links.
- **Benchmarks, Analysis, and Data Composition**

  Explains benchmarking and evaluation of parallel processing systems, as well as algorithm efficiency and data decomposition strategies.
- **Conclusion**
- **Definitions**
- **References**

## Overview

Computational demands continue to outpace readily available technology. In recent years, fiber optics technology has revolutionized the rate at which data can be carried between any two points. The theoretical communication bandwidth offered by fiber optic channels is of the order of $10^{16}$ Mbytes/s. Satellite-generated data must be processed at a rate of $10^{10}$ Hz. Before and during surgery, when a team of surgeons require a 3-D view of human body parts on a TV screen using tomography technology, information must be processed at speeds on the order of $10^{15}$ Hz. Many more applications demand increased processing speeds: speech recognition, spatial and temporal pattern recognition, modeling fusion rectors, oil explorations, astronomy, robotics, and the solutions of large differential equations for numerical simulations of earthquakes and for atomic and nuclear physics with thousands of variables. Next generation communication technology will support and link ISDN, FDDI, and ATM data bases; live surgery across continents; and intricate defense networks. No sequential computer or existing supercomputer can meet processing demands today and certainly not in the future.

The silicon technology barrier has almost been reached; pushing closer may reduce reliability and increase cost. One of the few options for meeting the computation needs is to exploit parallel processing. The concept of parallel processing is illustrated in the simple analogy of a farmer plowing his field. Alone, it will take him several months, by which time the harvest season will be gone, and his efforts will have been in vain. Instead, the farmer brings in his four sons, all able to do the same quality of work at the same speed. Each one of them starts from four different directions—say north, east, west, and south—and the father coordinates the activities. Ideally, the total plowing time is reduced by a factor of four. This is precisely how parallel processing works; each processor is similar in clock speed, memory size, and communication rate, and they divide any task among themselves to speed up execution.

### Need for a Dedicated Parallel Processing System

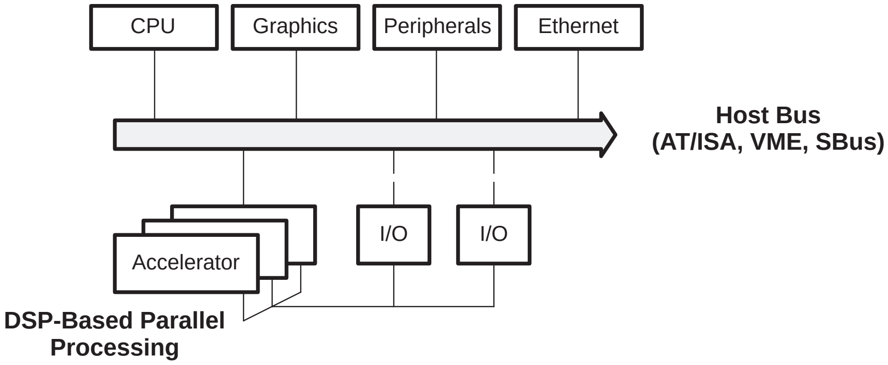
Workstations such as the Sun SPARC, HP 9000 series, and Digital Alpha offer 50 to 200 MIPS (million instructions per second) processing. However, using a workstation as a high-performance computation engine has several drawbacks:

- Complex operating systems can occupy as much as 50% of a CPU's processing time.
- Sophisticated graphics displays and graphical user interfaces demand extensive CPU processing.
- Real-time applications require an additional software layer, which must coordinate the disk, graphics, and peripheral I/O, host-to-data acquisition, and data-to-host transfers.

These drawbacks can be easily bypassed by integrating the workstation with dedicated parallel processing host-based hardware. Such hardware can accelerate an application by several orders of magnitude over the workstation. For example, a dual TMS320C40-based processor board can accelerate the performance of a '486 processor by a factor of 20 to 40 for signal processing applications.

Computer manufacturers have also realized the limited I/O and processing capability of workstations and are gravitating toward providing such standard host buses as the ISA, EISA, Microchannel, SBus, and Futurebus for application-specific hardware. As shown in Figure 1, a host can have numerous subsystems, including a dedicated accelerator and a data I/O.

**Figure 1. Architecture for Real-Time Processing**



Workstation buses are easily saturated by the various subsystems that interface to the bus. I/O systems with dedicated links to an accelerator can significantly reduce the data transfer burden on the bus as well as off-load the CPU from real-time computation processing.

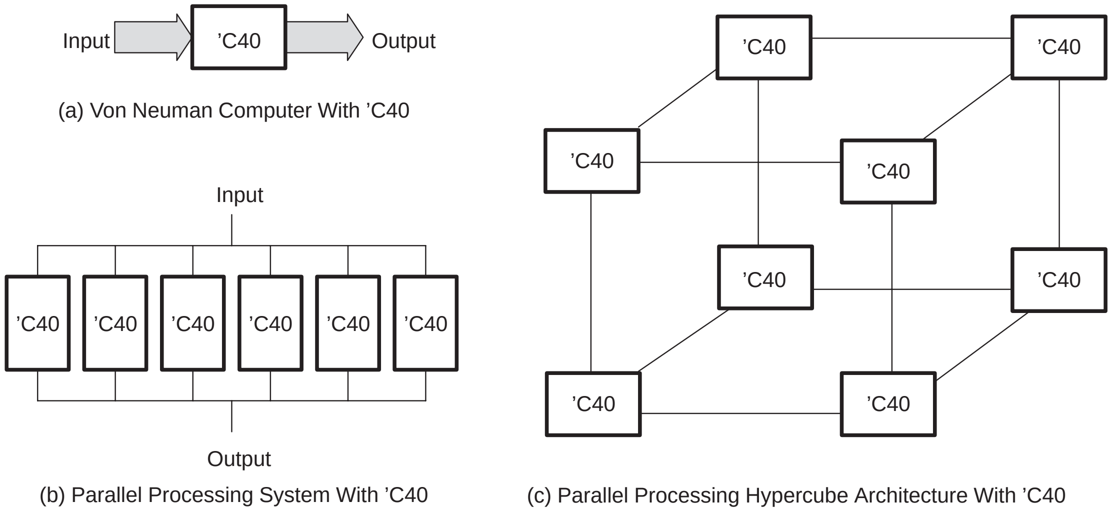
### Need for a Dedicated I/O Link

Although these standard host buses are sufficient for nonreal-time data and program transfers and for uploading and downloading of compressed and processed data, they fall short for high-speed data transfers and processing. For example, the ISA bus transfers peak at 2 Mbytes/s, EISA at 5–10 Mbytes/s, Microchannel at 10–20 Mbytes/s, VME at 40 Mbytes/s, and SBus at 40–80 Mbytes/s. An image processing system running at 40 frames per second with color image sizes of 1024 × 1024, 32 bits/pixel, and about 100 operations/pixel requires more than 320 Mbytes/s of data transfer and 4 GFLOPS (billion operations per second). Such high speeds and massive amounts of data require a dedicated I/O link to the processing node (see Figure 1). Furthermore, a real-time application requires determinism on the I/O bus, which is not available on the standard buses. Dedicated parallel processing systems can overcome limitations of host bus speed and I/O throughput bottleneck and offer more performance per unit cost compared to a sequential supercomputer.

Parallel computing is the second fastest growing market next to network computing. In 1991, about 50% of the supercomputers sold were parallel computers. By 1996, approximately 70% of all supercomputers will be parallel computers [1]. A similar trend is emerging in minisupercomputers, superminicomputers, and workstations. Cray and IBM are targeting a performance of 1 TFLOPS (trillion floating-point operations per second).

A parallel computer consists of a collection of processing elements that cooperate to solve a problem by working simultaneously on different parts of the problem (similar to the work sharing done by the farmer's sons). The difference between the conventional Von Neuman computer, which consists of a single processor, and a parallel processing system is shown in Figure 2.

**Figure 2. Conventional Vs. Parallel Processing Systems**



(a) Von Neuman Computer With 'C40

(b) Parallel Processing System With 'C40

(c) Parallel Processing Hypercube Architecture With 'C40

Figure 2(a) shows a conventional Von Neuman setup; Figure 2(b) shows a parallel processing system with multiple 'C40s; view (c) shows a hypercube architecture.

## Evaluating a Node for a Parallel Processing Architecture

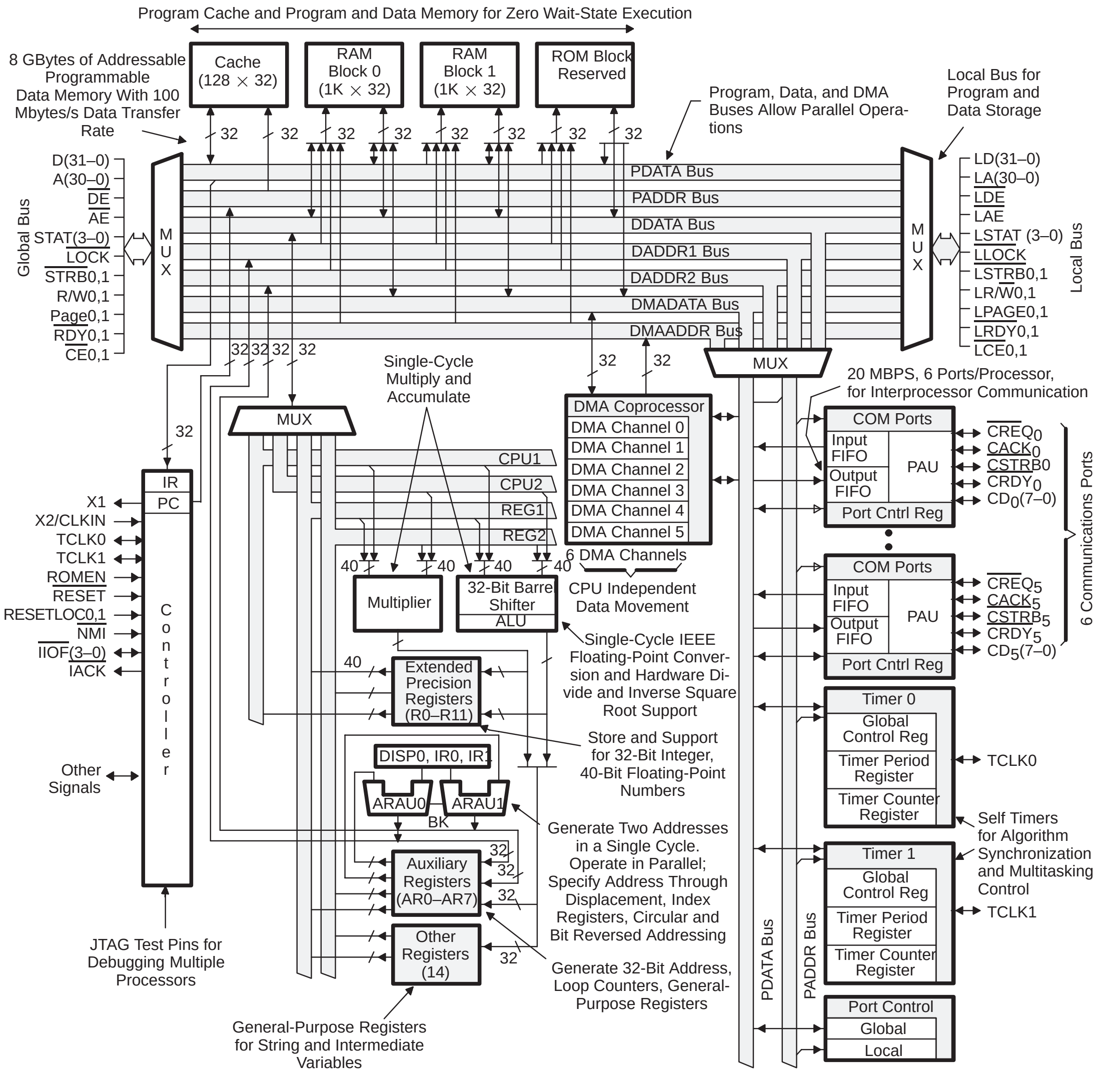
A high-performance cost-effective parallel processing system for a nonspecific application must have nodes that meet these criteria:

- High processing speed,
- A large number of high-speed DMA-supported links,
- Easily configurable and incrementally expandable architecture,
- Ease in load balancing (even processing distribution over all the processors),
- Low cost per node or low cost per MIPS/MFLOPS/MOPS,
- Ease in programming via multitasking kernels and multiprocessing program support,
- High speed I/O, and
- Such development tools as compilers, assemblers, linkers, simulators, and emulation hardware.

### *Why ’C40 as a Node*

The selection of a node is critical for a parallel processing system. Many high-speed processors, such as the 80486, i860, and 68040, are capable of number crunching. None, however, offer parallel processing support. Some designers opt for a custom silicon solution for a node. The TMS320C40 from Texas Instruments, as shown in Figure 3, meets all the above defined criteria and is ideally suited for parallel processing.

### Figure 3. TMS320C40 Parallel Processor Block Diagram


Program Cache and Program and Data Memory for Zero Wait-State Execution
8 GBytes of Addressable Programmable Data Memory With 100 Mbytes/s Data Transfer Rate
Cache (128 × 32)
RAM Block 0 (1K × 32)
RAM Block 1 (1K × 32)
ROM Block Reserved
Program, Data, and DMA Buses Allow Parallel Operations
Local Bus for Program and Data Storage
32
PDATA Bus
PADDR Bus
DDATA Bus
DADDR1 Bus
DADDR2 Bus
DMADATA Bus
DMAADDR Bus
Global Bus
D(31–0)
A(30–0)
DE
AE
STAT(3–0)
LOCK
STRB0,1
R/W0,1
Page0,1
RDY0,1
CE0,1
MUX
LD(31–0)
LA(30–0)
LDE
LAE
LSTAT (3–0)
LLOCK
LSTRB0,1
LR/W0,1
LPAGE0,1
LRDY0,1
LCE0,1
Local Bus
Single-Cycle Multiply and Accumulate
20 MBPS, 6 Ports/Processor, for Interprocessor Communication
DMA Coprocessor
DMA Channel 0
DMA Channel 1
DMA Channel 2
DMA Channel 3
DMA Channel 4
DMA Channel 5
6 DMA Channels
CPU Independent Data Movement
CPU1
CPU2
REG1
REG2
40
Multiplier
32-Bit Barrel Shifter
ALU
Single-Cycle IEEE Floating-Point Conversion and Hardware Divide and Inverse Square Root Support
Extended Precision Registers (R0–R11)
Store and Support for 32-Bit Integer, 40-Bit Floating-Point Numbers
DISP0, IR0, IR1
ARAU0
ARAU1
BK
Generate Two Addresses in a Single Cycle. Operate in Parallel; Specify Address Through Displacement, Index Registers, Circular and Bit Reversed Addressing
Auxiliary Registers (AR0–AR7)
Other Registers (14)
Generate 32-Bit Address, Loop Counters, General-Purpose Registers
General-Purpose Registers for String and Intermediate Variables
IR
PC
Controller
X1
X2/CLKIN
TCLK0
TCLK1
ROMEN
RESET
RESETLOC0,1
NMI
IIOF(3–0)
IACK
Other Signals
JTAG Test Pins for Debugging Multiple Processors
PDATA Bus
PADDR Bus
COM Ports
Input FIFO
Output FIFO
PAU
Port Cntrl Reg
CREQ0
CACK0
CSTRB0
CRDY0
CD0(7–0)
CREQ5
CACK5
CSTRB5
CRDY5
CD5(7–0)
6 Communications Ports
Timer 0
Global Control Reg
Timer Period Register
Timer Counter Register
TCLK0
Self Timers for Algorithm Synchronization and Multitasking Control
Timer 1
TCLK1
Port Control
Global
Local

Texas Instruments, along with DSP board vendors, realized that over half of its customers were using multiple TMS320C30/31 DSP processors. The processors communicated via shared memory or from serial port to serial port. Such a configuration required additional hardware, software, and, most of all, degraded processor performance. The ’C40 was designed specifically to facilitate system configurability and provide massive parallelism with very little or no glue logic and without additional software overhead. The instruction set of the ’C40 is compatible with the popular ’C30 DSP processor. The parallel processing capability of the ’C40 [2], as illustrated in Figure 3, makes the ’C40 ideal for a parallel processing node and particularly for scientific and DSP applications. Table 1 summarizes the ’C40’s key parallel-processing features.

### Table 1. ’C40 Parallel Processing Features

| Criteria | Description |
|---|---|
| Node Speed | 40 to 50 MFLOPS, 275 MOPS (up to 11 operations per cycle throughput), 40/32-bit single-cycle floating-point/integer multiplier, hardware divide, and inverse square root support. |
| Node Link | Six 20-Mbyte/s node-to-node communication ports. Configuration for parallel processing topologies via the communication ports:<br>a. mesh up to three dimensions<br>b. pyramid of any dimension<br>c. hypercube of up to six dimensions |
| DMA | Six-channel DMA coprocessor supports 20-Mbyte/s transfers at each communication port. Compiler and operating system support for message passing and processor-to-processor communication. |
| Expandability | Up to six links with no glue logic.<br>Two identical external buses (32 bits data, 31 bits address).<br>Shared memory support. |
| Cost per Node | $500 to $1000 per node at 40 MFLOPS. |
| I/O per Node | 320 Mbytes (100 Mbytes/s on global bus, 100 Mbytes/s on the local bus, and 120 Mbytes on the six communication ports). |
| Programming Tools | ANSI C compiler, ADA, simulator, off-the-shelf boards for PC and VME, off-the-shelf DSP libraries. |
| Operating Systems | Currently supported by five manufacturers. See Conclusion section for details. |
| Hardware Debugging Tools | Debugging of multiple ’C40s on the Sun SPARCstation (Sonitech’s JTAG Brahma SBus) or the TI XDS510 emulator. |
| Manufacturer Support | Support from TI and over 50 board and software vendors. |

The TMS320C40 is the processing node for Sonitech’s ISA, VME, and SBus family of board-level products. The architecture of these boards is discussed below in *Parallel Processing Topologies*.

## Parallel Processing Topologies

This section presents several architectures that improve processing speeds. A matrix multiplication example is discussed to demonstrate the application of parallel processing. Several parallel processing architectures, such as hypercube, pyramid, ring, and mesh of trees topologies are explained in detail.

### Approaches to Performance/Throughput Enhancement

In the Von Neuman computer model, the steps of decoding, fetch, execution, and memory operations are performed in a sequential manner. Only one task is handled at a time; hence, the core CPU is utilized only during the execution phase. Newer approaches, such as pipeline, multitasking, and vectors have increased computer speed and CPU utilization.

In a pipeline, instruction decoding, data fetch, and memory operations are done in parallel over multiple queues while the CPU is executing instructions. If the execution of individual tasks does not depend on the results of other tasks, the pipeline will be full. If a process is data-dependent and lacks the most recent values, the pipeline will be blocked. The ’C40 has four levels of pipeline: fetch, decode, read, and execute. In addition, the DMA adds transfer capability and performs as many as three operations per cycle. The ’C40 has a separate multiplier, ALU, address register modifier, loop counter, and delayed branch (which can support eight operations per cycle).

## Parallel Processing Taxonomy

There is a wide diversity in parallel processing terminology. Two classifications [6], one by Flynn and the other by Skillicorn, can be implemented with ’C40s and are discussed briefly. Flynn categorizes computers into three major types:

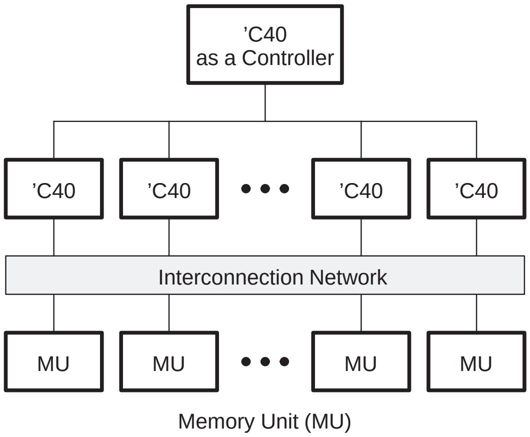
- **SISD (Single-instruction stream/single-data stream)**

  This is a Von Neuman computer. A single ’C40 can serve as an SISD computer. (See Figure 2.)

- **SIMD (Single-instruction stream/multiple-data stream)**

  Traditional SIMD machines have a primitive node with no large local program memory. A controller feeds all the nodes with the same instruction and executes the program synchronously and in lock step. The interconnection network from processor to processor and from processor to memory can be an array, mesh, hypercube, or any network. Figure 4 shows a ’C40-based SIMD architecture. Since the ’C40 has 2K words of on-chip memory (and possibly more local memory on the local bus), the program blocks can be transferred from the host or the controller. The next program block can be transferred via the DMA without CPU intervention; this minimizes communication overhead on the interconnection network. Furthermore, the need for broadcasting from the host to the individual ’C40 is reduced.

**Figure 4. SIMD General Architecture With ’C40s**



- **MIMD (Multiple-instruction stream/multiple-data stream)**

  In multiple-instruction stream/multiple-data stream architecture, as shown in Figure 5, each processor acts autonomously, executing different instructions. There are two categories of MIMD systems:

  – **Shared-memory or tightly coupled**

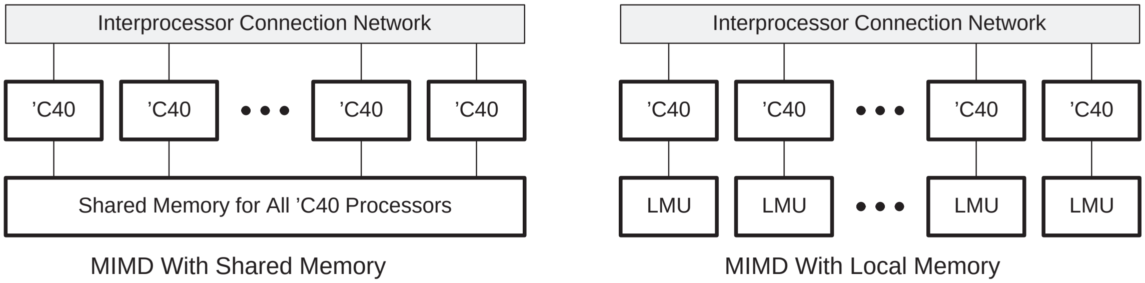
  The processors share the same memory, and there is a rigid protocol for reading, modifying, and writing the data in the shared memory.

  – **Distributed memory or loosely coupled machine**

  Each processor node has its own local memory (program and/or data); hence, there is no memory contention. Processors communicate via message passing. The interconnection network could be hypercube, ring, butterfly, mesh of trees, or another application-specific network.

  The interconnection network is usually bus connected or directly connected. In bus connected systems, parallel memories, network interfaces, and device controllers are all connected to the same bus. In directly connected systems, the interconnection could be crossbar, partially connected graphs, or multistage networks.

**Figure 5. Multiple-Instruction Stream/Multiple-Data Stream Systems**



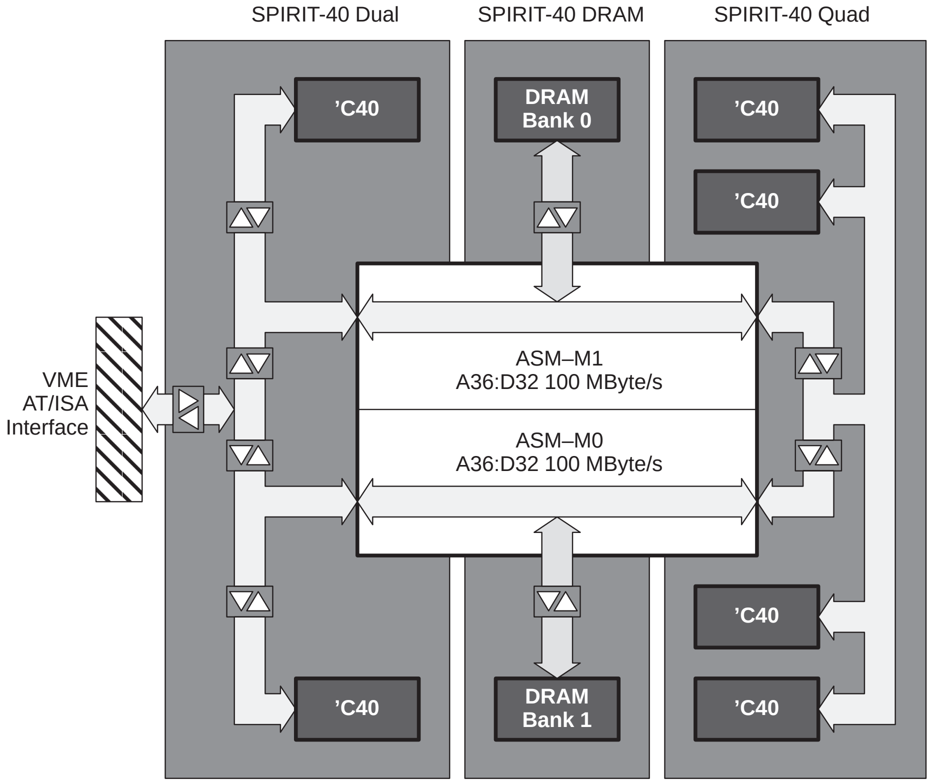
Interprocessor Connection Network
’C40
’C40
’C40
’C40
Shared Memory for All ’C40 Processors
MIMD With Shared Memory
Interprocessor Connection Network
’C40
’C40
’C40
’C40
LMU
LMU
LMU
LMU
MIMD With Local Memory

Flynn’s taxonomy does not fully classify many new architectural configurations. Skillicorn’s taxonomy of computer architecture is based on the functional structure of architecture and the data flow between its component parts, such as IP (instruction processor), DP (data processor), DM (data memory), IM (instruction memory), and SW (switching network). According to Skillicorn’s classification, a total of 28 classes of computer architectural configurations are possible. Each classification is a function of the number of IPs, DPs, DMs, and IMs and their interconnection to each other. The interconnection also determines the method of message passing and/or message sharing and indicates the type of coupling of a particular parallel architecture. For example, Skillicorn’s Computer Taxonomy class 14, known as a loosely coupled parallel processing system, has a total of n IPs and n DPs. Each IP is connected to each DP and to each IM; each DP is connected to each DM and to all other DPs. An n-dimensional hypercube, as shown in Figure 2(c), is an example of Skillicorn’s class 14 parallel processing system.

## TMS320C40-Based AT/ISA and VME Architecture

Sonitech has developed VME and AT/ISA TMS320C40-based boards that are expandable from two nodes (100 MFLOPS) to hundreds of nodes, and memory that is expandable from 8 Mbytes to 768 Mbytes. This report discusses the architecture of these boards for readers who are designing their own target systems, integrating a system, or interfacing to the ’C40 global, local, or communication port bus. Figure 6 shows the SPIRIT-40 Dual with two ’C40s, the Quad-40 with four ’C40s, and the SPIRIT-DRAM with a large shared global memory. An ASM-M (application-specific module–memory) expansion bus brings the host bus from the Spirit-40 Dual to the Quad-40, SPIRIT-DRAM, or other custom boards.

**Figure 6. AT/ISA and VME Architecture With the SPIRIT-40 Dual, SPIRIT-40 Quad, and SPIRIT-40 DRAM**



**Note**: For clarity, SRAM not shown.

This modular system (Figure 6) decouples the main processor card (SPIRIT-40) with a low-cost high-speed coprocessor unit (Quad-40) and with a large DRAM memory (SPIRIT-DRAM). Both the Quad and the DRAM boards feature no host bus (VME or ISA) interface for several reasons:

- The host bus is typically saturated (see Figure 1), and adding host interface will not increase the overall host to ’C40 system I/O,
- The ASM-M brings the host bus to the other two boards and to additional user-customized boards, and
- The configuration:
  - Saves 20 to 30% of board real estate,
  - Encourages the use of communication ports, as opposed to the host bus passing data and messages,
  - Reduces the cost of the Quad card (200 MFLOPS) to only $1000 more than the Dual (100 MFLOPS),
  - Uses minimal power per slot and is therefore more reliable, and
  - Requires no daughter cards.

A more detailed SPIRIT-40 VME architecture is shown in Figure 7, and major features of this architecture are highlighted in Figure 8. The two ’C40s are nearly identical, except that one has an RS232 interface and a boot EPROM. Attached to the ’C40 local bus is up to 4 Mbytes of local SRAM (can be as small as 256 bytes, or not used at all) and DSP control registers. The DSP control registers are specifically designed for configuring and programming massively parallel systems. Each register has a unique processor ID, the size and speed of local and global SRAMs, a processor boot mode (to boot from SRAM, PROM, or communication ports), the capability to reset and interrupt the direction and status of the six communication ports, and ASM-M port access and control. In a data flow architecture, where a large size data block is being routed in search of a processor with a large memory, the memory size register can be quickly read to determine whether that particular node has enough memory.

**Figure 7. AT/ISA and VME Architectures**

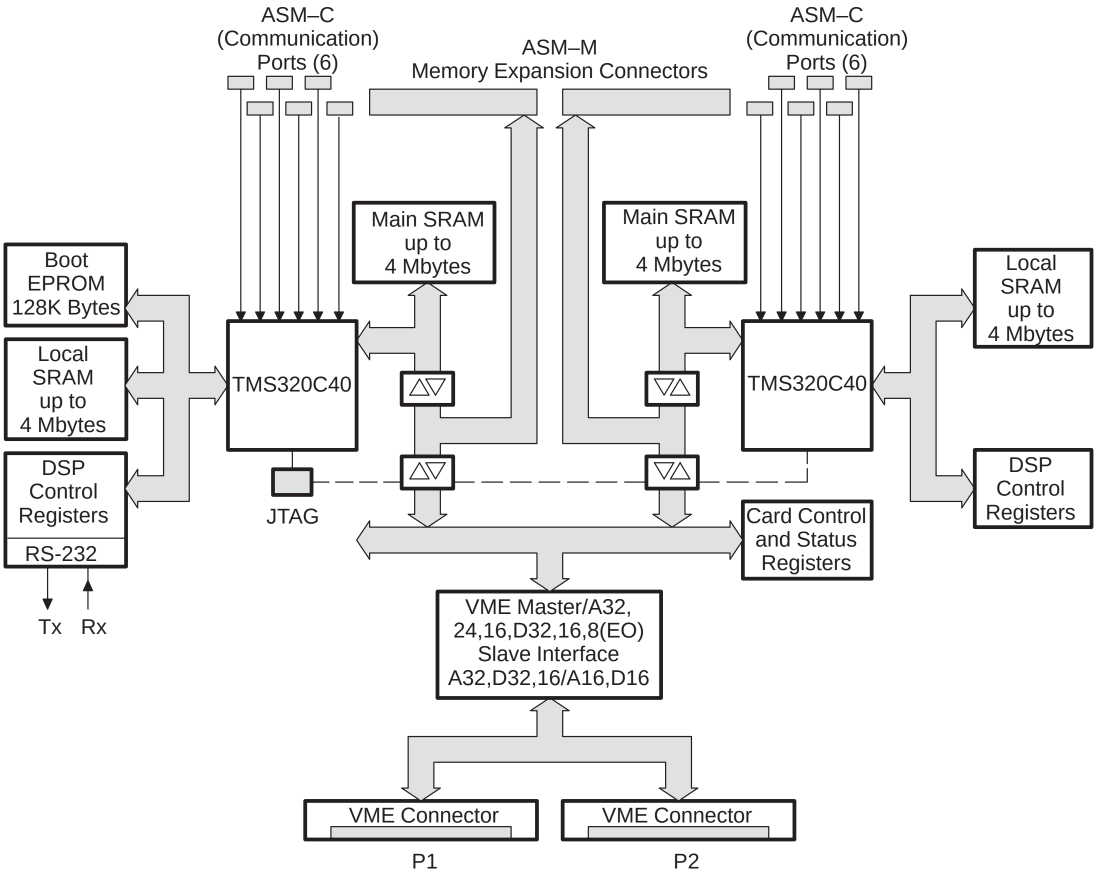
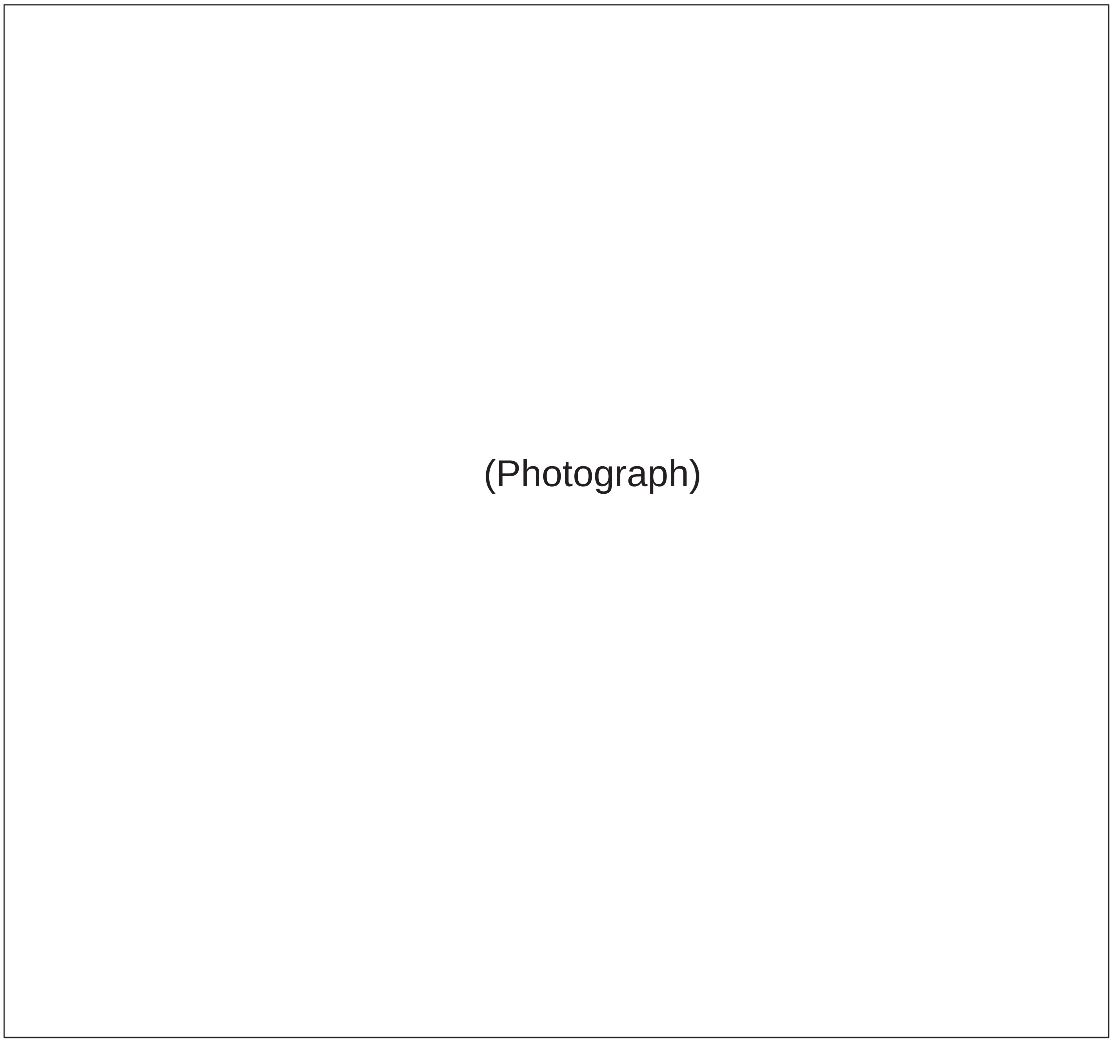


**Figure 8. Dual Processor SPIRIT-40 Major Features**



1) 12 ASM-C Communication Port Connectors. Each 30-pin port connector is capable of 20-Mbyte/s transfers and has its own DMA controller. The 12 ports facilitate large processor arrays, data switches, and external I/O connections.

2) ASM-M Communication Port Connector. This 240-pin connector provides two 50-Mbyte/s buses (32 bits wide, 32-bit address), one for each ’C40. The connectors make 256 Mbytes of DRAM available per ’C40 via a small interconnect board to an adjacent VME slot with Sonitech DRAM cards.

3) Removable Faceplate. This faceplate unplugs to allow access to the ASM-M connector. In large systems (dozens of processors), the faceplate can be replaced by a user-designed panel to eliminate discrete cables.

4) Memory SIMMs (single inline memory modules). These SRAM SIMMs provide zero-wait-state access for the ’C40. Up to 16 Mbytes can be installed on the board.

5) TCLK. The TCLK lines connect to each ’C40’s two timer-counter channels and can be used as inputs or outputs for controlling external equipment.

6) JTAG. The industry-standard JTAG connector provides access to each ’C40’s JTAG path for software development.

7) VME Master/Slave Interface. This card may operate as a VME master, slave, standalone, or Slot 1 controller.

8) Dual 40- or 50-MFLOPS ’C40 DSPs. The ’C40 supports a variety of development tools from Sonitech and other third parties, including a choice of five operating systems, C and FORTRAN compilers, and DSP libraries.

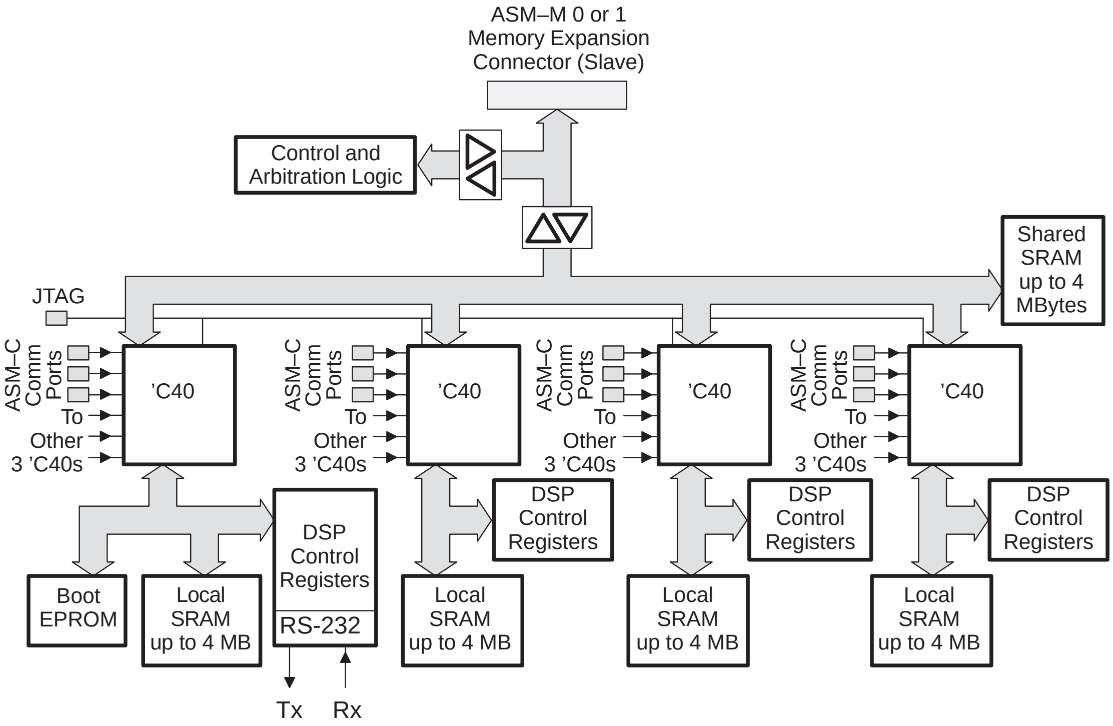
The main bus (or global bus) memory operates on a shared-access basis, giving the VME host and ’C40 access to the entire memory space on a cycle-by-cycle basis. This simplifies software development and speeds movement of large data arrays compared to systems that use a small dual-port RAM. Also on the ’C40 main bus (global bus) is the ASM-M expansion bus, which allows up to three cards to be connected. The VME bus has access to the ASM-M and can transfer data over this bus without interfering with the ’C40’s access to its SRAM (notice the transceivers/buffers). One advantage of this implementation is that the host bus can transfer data to the ASM-M slave without impacting the ’C40 execution. The SPIRIT-40 VME transfers data at 20 to 30 Mbytes/s over the host VME bus and is a slot 1 bus master; that is, it reads and writes to the memory of other cards on the VME bus.

In addition to the local and global memory interfaces to the ’C40, six communication ports are connected to the front panel. To simplify interconnection to external communication port devices (such as data acquisition, video bus, or other custom interfaces), each communication port connector includes an external interrupt signal to the ’C40 and a reset signal that can be configured as an input or output. (See Figure 10.) A secondary connector is provided for access to the timer clock.

The RS232 can interface to any host machines (MAC, Sun, PC, etc.) for code downloading, debugging, and diagnostics. For stand-alone operations, the boot EPROM can boot not only the main ’C40, but also all the other ’C40s.

The coprocessor SPIRIT-Quad is shown in Figure 9. It interfaces with the ASM-M via the shared SRAM, and the global buses of each ’C40 can access this SRAM. The ’C40 processors on the Quad have the same type and size of memory and the same DSP control register as the processor on the Dual card. Such symmetry is highly desirable, as discussed later in this report, for many architectural configurations and algorithms. A total of 12 communication ports are brought out. There are three from each ’C40; the other communication ports are connected among themselves.

**Figure 9. SPIRIT-40 Quad Processor Architecture**



The SPIRIT-DRAM supports applications requiring massive amounts of global memory. Like the SPIRIT-Quad, the SPIRIT-DRAM has no host interface (only power and grounds are connected); the host bus is available via the ASM-M memory expansion connector. A total of 256 Mbytes of DRAM is available on a shared-access basis (one to three wait states) to all the 'C40s on the Dual board.

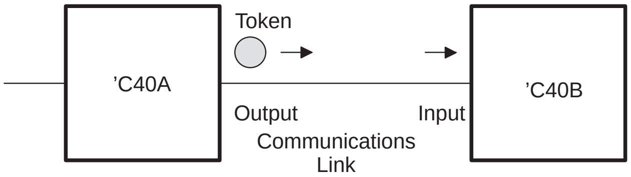
## System-Level Design Issues With the TMS320C40 Communication Ports

The TMS320C40 communication ports transfer data at 20 Mbytes/s. This communication capability makes the 'C40 ideally suited for large interconnect parallel processing networks. You should analyze several design issues, however, before selecting an interconnect topology and peripheral interface design. The 'C40 communication ports consist of 12 lines: eight data, one strobe, one ready for byte-transfer handshaking, one for token transfer, and one for token acknowledge. A communication port can be in either of two I/O states: output state if it has the token, or input state if it does not have the token. During power-up, three of the six communication ports have a token (output ports), and three don't have a token (input port). For bidirectional communication, the connected input and output communication ports pass the tokens back and forth.

### 'C40 Node Reset

Figure 10 shows a configuration with two 'C40 nodes, A and B, each with one communication link. The power-up default is that A-Link has the token and is an output; B-Link does not have the token and is an input. During bidirectional data transfers, A and B exchange tokens. If B-Link has the token, node A is reset; after the reset, both A-Link and B-Link will have the token. Both links then drive the data bus (if for more than 100 ns, the communication ports could be damaged).

**Figure 10. Token State After Reset**



The SPIRIT-40 communication ports have signals that prevent communication link contention and simplify interface to peripheral devices.

In a large topology where multiple ’C40 links are connected to many other ’C40s, a reset to one ’C40 must immediately follow a reset to all ’C40s. Such a scheme has significant software and system-level ramifications. For example, all of the intermediate data and programs, including program and data download, will be lost and must be reinitialized. The problem is further complicated if a peripheral device is attached to a link and has an option to assert a reset to the ’C40.

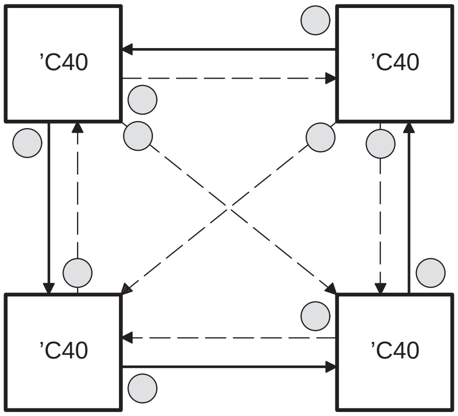
The ASM-C (application-specific module–communication port interface), a superset version of the ’C40 communication port interface, guarantees no contention between the communication links. The bidirectional CTOKRES* (token restore) signal resets all the ’C40 links, is received by all the ’C40’s in the cluster, and is connected to each ’C40’s NMI (nonmaskable interrupt). Once the ’C40 enters the NMI routine, it can select reset or no reset. The CAUXRES* and CINT* I/O signals are provided for non-’C40 reset. These signals will not reset the ’C40 or a communication port token interface. They can reset a remote communication-link-connected peripheral device or one ’C40 interrupting another ’C40. When a remote device or another ’C40 asserts this signal, the ’C40 that receives the signal is interrupted and can take appropriate action in software.

## Design Suggestions for a Large Topology

- **Use redundant links to avoid bidirectional communication port I/O**

  The ’C40 has six communication ports. If a four-processor configuration with single links, as in Figure 11, is required, the remaining links can be used so that only unidirectional transfers are necessary on each link. Such a configuration permits reset of a single ’C40 without cluster reset. For example, a 3-D hypercube (three links per node) can be configured for six unidirectional links (dashed lines).

**Figure 11. Four-Node Configuration With a Single-Link Interface and Unidirectional Transfers**



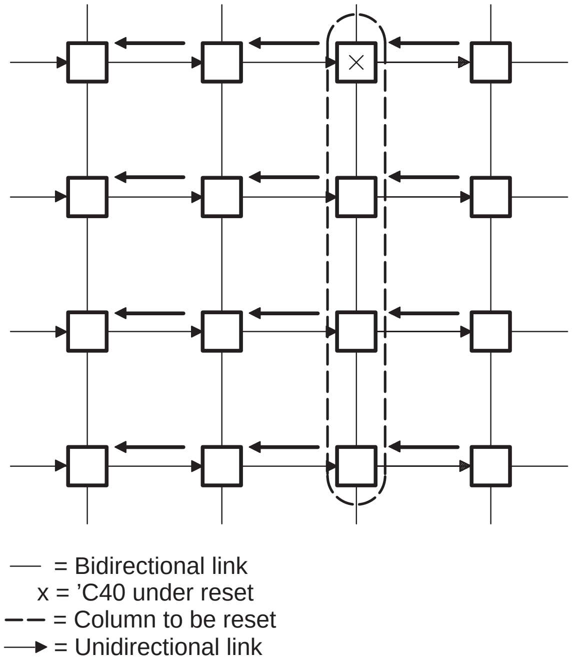
- **Partial cluster reset**

  For applications that cannot tolerate a full system reset, you can configure the hardware and software for partial cluster reset. For example, in the two-dimensional mesh shown in Figure 12, the thick lines represent a second link. This way, each link is maintained unidirectionally in the horizontal direction and so is not affected by a reset. A reset at any one node needs to propagate in the vertical direction only, not in the horizontal direction.

- **Use Buffered Communication Ports**

  Buffering the communication ports will reduce the transfer speeds and, depending on the design, usually allow only unidirectional transfers. The advantage is that the communication port links can be significantly longer than 6–12 inches.

**Figure 12. A Two-Dimensional Mesh Configuration**



## Matrix Multiplication Application of Parallel Processing

Parallel processing has a wide range of applications, including the following:

- embedded real-time
- image processing
- feature extraction
- motion detection
- speed estimation
- thermographic analysis
- speech recognition
- spatial and temporal pattern recognition
- vision
- weather
- seismology
- defense
- radar
- sonar
- scientific
- medical
- industrial
- communications

One commonly used algorithm is matrix multiplication. In general, if A is a matrix of m × n and B is a matrix of n × k dimensions, respectively, then a mesh of size m × k is required to compute matrix-matrix multiplication. In our example, m = n = k = 2.

The example below and Figure 13 show an example of matrix multiplication. Let the nodes of the mesh be labeled like a matrix element (row,column), that is, nodes (1,1), (1,2), (2,1), and (2,2)

$$A \times B = \begin{vmatrix} 2 & 1 \\ 4 & 3 \end{vmatrix} \times \begin{vmatrix} 7 & 8 \\ 5 & 6 \end{vmatrix} = C = \begin{vmatrix} 19 & 22 \\ 43 & 50 \end{vmatrix}$$

Step 1. ’C40(1,1)
d=A(1,1)*B(1,1)=2*7=14

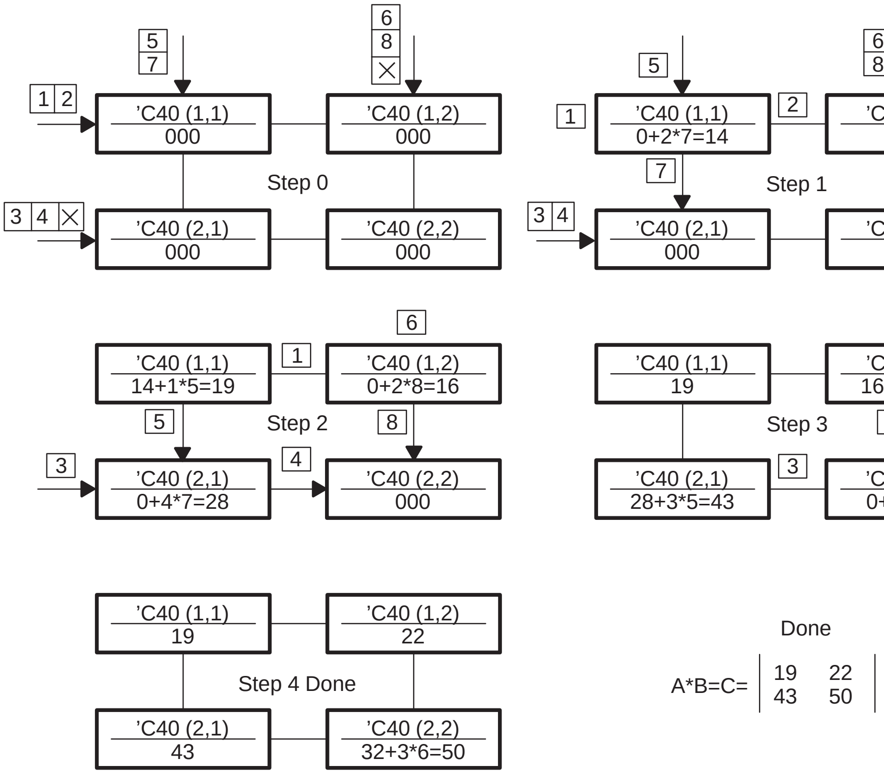
Step 2. ’C40(1,1)
C(1,1)=A(1,2)*B(2,1)+d=1*5+14=19
’C40(1,2)
e=A(1,1)*B(1,2)=2*8=16
’C40(2,1)
f=A(2,1)*B(1,1)=4*7=28

Step 3. ’C40(1,2)
C(1,2)=A(1,2)*B(2,2)+e=1*6+16=22
’C40(2,1)
C(2,1)=A(2,2)*B(2,1)+f=(3*5)+28=43
’C40(2,2)
g=A(2,1)*B(1,2)=4*8=32

Step 4. ’C40(2,2)
C(2,2)=A(2,2)*B(2,2)+g=(3*6)+32=50

At the end of the algorithm, each ’C40 node designated as ’C40(i,j) contains element C(i,j). Reference [3]explains parallel implementation of matrix multiplication using both shared and distributed memory approaches. The features for the ’C40 architecture bring high efficiency, low-communication overhead, and almost perfect speed-up to this implementation.

**Figure 13. Matrix Multiplication**



## Parallel Processing Architecture Topologies

This section discusses the hypercube, pyramid, ring, mesh, and mesh of trees architectures and their properties.

### Hypercube and Its Properties

The hypercube is a configuration of loosely coupled parallel processors based on the binary n-cube network. A significant difference between hypercube and most other parallel architectures is that its modes use message passing instead of shared variables or shared memory to communicate. These are the key advantages of a hypercube architecture:

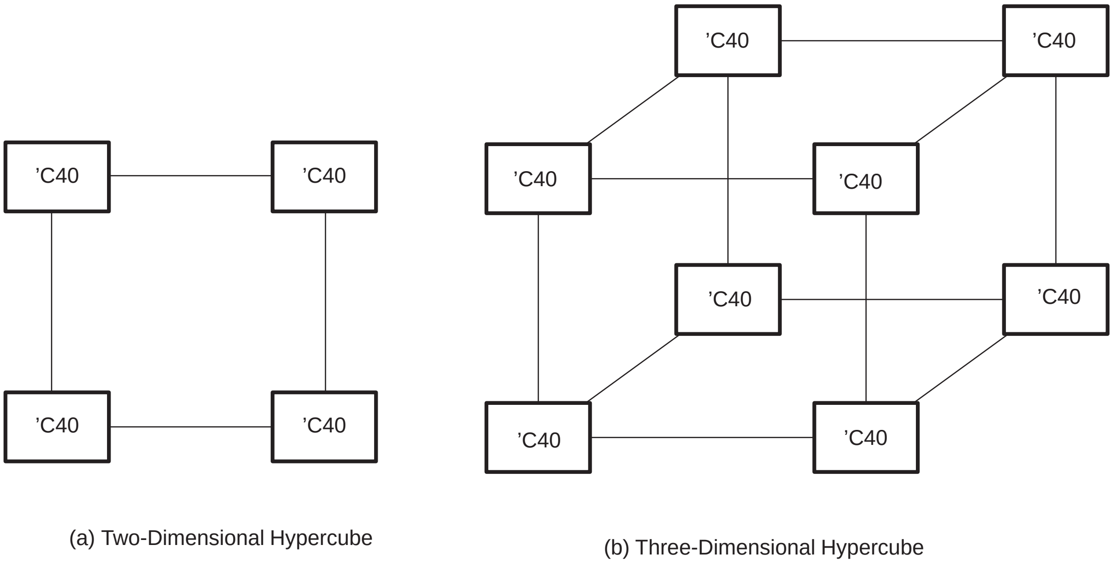
- **Powerful Network**: Easily mapped and efficiently implemented.
- **Small Diameter**: Shortest path separating the farthest two nodes.
- **Alternate Paths**: Alternate paths for all nodes to minimize congestion.
- **Scalability**: Flexibility in expanding and contracting the network without significant hardware and software modification and cost.
- **Fault Tolerance**: Robust when nodes and links malfunction.

The 'C40's six communication ports can be connected at 20 Mbytes/s to form a six-dimensional hypercube (64 nodes). In addition, the 'C40's large amount of addressable local, global, internal, and cache memory, coupled with its CPU performance, DMA, and pipeline, permits a large subprogram to be implemented efficiently on one node. This reduces the node-to-node communication overhead.

An n-dimensional hypercube contains $2^n$ nodes. Nodes are connected directly with each other if their binary addresses differ by a single bit. In an n-dimensional hypercube $H_n$, each processor is directly connected with n neighboring processors with n-bit binary addresses in the interval 0 to $2^n–1$. The processing elements are placed at each vertex of the hypercube. Adjacent nodes are connected directly with each other if and only if their binary addresses differ by a single bit. One commonly noted disadvantage of a p-cube computer (p=number of nodes) is its requirement for large numbers ($\log_2 p$) of input/output ports per node.

If a hypercube has fewer than six dimensions, the extra links from each ’C40 can be used as secondary links for high reliability and full duplex communication and to decrease the node-to-node distance. In special cases where two or more paths of equal distance are to be connected, priority is given to those paths whose planes have complementary MSBs; this yields a system with a minimum diameter. Figure 14 shows two- and three-dimensional hypercubes without extra links.

**Figure 14. Two- and Three-Dimensional Hypercubes**

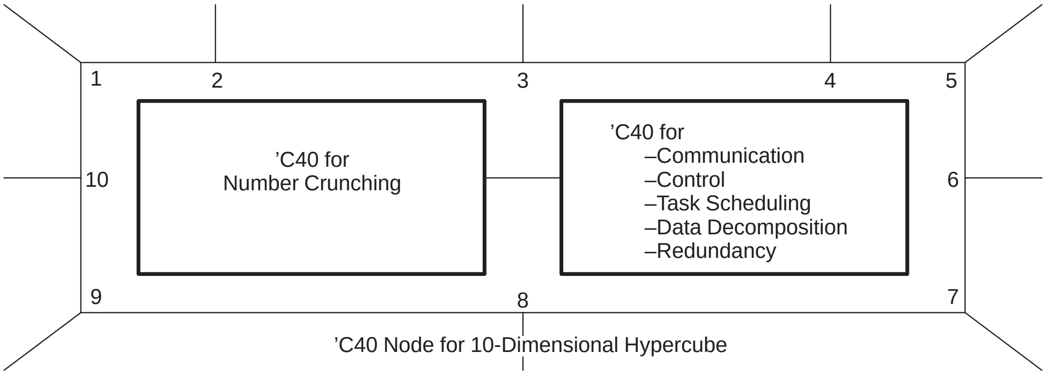


(a) Two-Dimensional Hypercube

(b) Three-Dimensional Hypercube

You can also create a hypercube of up to ten dimensions by combining two ’C40s as one node (see Figure 15). One of the two ’C40s is the number-crunching processor; the other ’C40 is used for communication, control, task scheduling, data decomposition, and, if required, number-crunching support for the first processor. This configuration has a peak performance of 100 GFLOPS. The SPIRIT-40 Dual processor is ideal for this dual-processor/single-node configuration.

**Figure 15. Ten-Dimensional Hypercube Configured With Dual Processor—Single Node**



### *Hypercube Node Assignment*

Hypercube nodes are assigned an address with respect to the Reflexive Gray Code (RGC). RGC is important because it uniquely defines adjacent nodes and node-to-node distances. RGC can be expressed mathematically as:

$$G_{n+1} = \{(0)G_n,\ (1)(G_n^r)\}\ . \tag{1}$$

where $G_n$ is the RGC of order n, $G_n^r$ is the reflection of $G_n$, $0(G_n)$ is the insertion of 0 as a MSB, and $1(G_n)$ is the insertion of 1 as a MSB.

The following are RGCs for n = 2, 3, and 4.

- One-dimensional

$$G_1 = 0;\text{and}\, G_1^r = 1 \tag{2}$$

- Two-dimensional

$$G_1 = 0,\ \ 1,\ \ \text{and}\ \ G_1^r = 1, 0;\text{therefore}, G_2 = \mathbf{0}0, \mathbf{0}1, \mathbf{1}1, \mathbf{1}0 \tag{3}$$

Similarly,

- Three-dimensional

$$G_3 = \mathbf{0}00, \mathbf{0}01, \mathbf{0}11, \mathbf{0}10, \mathbf{1}11, \mathbf{1}10, \mathbf{1}00, \mathbf{1}01 \tag{4}$$

- Four-dimensional

$$\begin{aligned} G_4 = {} & \mathbf{0}000, \mathbf{0}001, \mathbf{0}011, \mathbf{0}010, \mathbf{0}111, \mathbf{0}110, \mathbf{0}100, \mathbf{0}101, \\ & \mathbf{1}111, \mathbf{1}110, \mathbf{11}00, \mathbf{1}101, \mathbf{1}000, \mathbf{1}001, \mathbf{1}011, \mathbf{1}010 \end{aligned} \tag{5}$$

In addition to the node addresses as described above, the link connections must be assigned. The number of links or edges in a hypercube are n × $2^{n-1}$. On the ’C40-based n=6 hypercube, 192 cables will be required. Port connections are assigned via an iterative algorithm.

### *Six Properties of the Hypercube*

Six properties [8, 11, 14] are unique to the hypercube and can be applied to task allocation, data decomposition, communication, and architectural expansion and contraction:

1. “An n-dimensional hypercube can be constructed recursively from a lower dimensional cube.”

   This property makes a hypercube a modular parallel architecture; it allows you to build a higher dimensional hypercube from lower dimensional hypercubes. With ’C40s as nodes, you can always upgrade the hypercube a minimum of six dimensions or higher by combining two ’C40s into one, as shown in Figure 15.

2. “Any n-cube can be tiered in n possible ways into two (n–1) subcubes.”

   Due to this property, hypercube can divide larger data size into smaller data size, assign the tasks to different subcubes in any order, and combine the results.

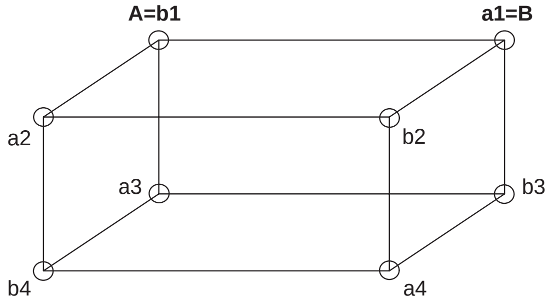
3. “There are n!$2^n$ ways of numbering the $2^n$ nodes of the n-cube.”

   It is not necessary to assign node addresses in any specific order. Since the node addresses are RGC, you can assign the starting addresses to any node. This is especially useful when the data input node changes from one node to another; to implement the same algorithm, just reassign the node address rather than change the program or the physical link connection.

4. “Any two adjacent nodes A and B of an n-cube are such that the nodes adjacent to A and those adjacent to B are connected in a one-to-one fashion.”

   In the three-dimensional cube shown in Figure 16, two adjacent nodes, A and B, are chosen arbitrarily. A has a1, a2, a3 as its adjacent nodes, and B has b1, b2, b3 as its adjacent nodes. Note that the pairs (a1, b1), (a2, b2), and (a3, b3) are also adjacent. If this process is repeated until all the nodes are marked either a or b, then exactly half the nodes are marked a, and the remaining half are marked b. All the nodes are linked in a one-to-one fashion (and are adjacent) like nodes A and B.

**Figure 16. Three-Dimensional Cube**



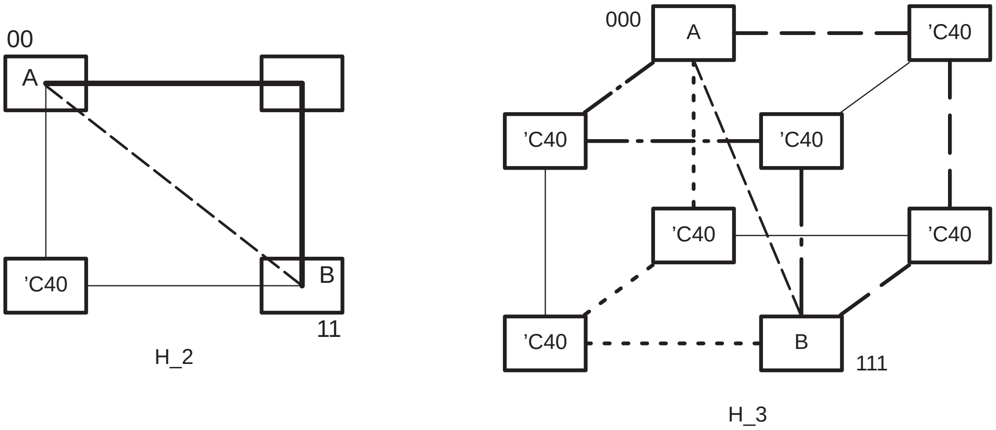
5. “There are no cycles of odd length in an n-cube.”

   This property simplifies message passing, which is the heart of the loosely coupled MIMD hypercubes. For example, a message traveling from node A back to node A will pass through an even number of links. Parity is positive (+tve) if the number of 1s in a binary address of a node is even; if the number of 1s is odd, the parity is negative (–tve).

6. “The n-cube is a connected graph of diameter n.”

   When a message travels from one node to any other node, the shortest distance will not exceed the diameter, which is equal to the hypercube dimension, n. Additionally, there are n different (nonoverlapping) parallel paths (see node A and node B paths in Figure 17) between any two nodes. These parallel paths are equal if A and B are the farthest nodes. The message must pass through n links or less in an n-dimension hypercube. In other words, in an n-dimension hypercube, a message needs to pass through only n–1 (or less) nodes to get to the intended node.

Figure 17.  Hypercube Architecture With Parallel Paths



### *Distinguishing Hypercube From Similar Configurations*

Many architecture configurations such as wraparound mesh or wraparound tori can be mistaken for hypercubes. The following theorem differentiates hypercubes from other similar architecture configurations.

Theorem: A graph G with a total number of V nodes and E edges is a hypercube if and only if:

- V has $2^n$ nodes.
- Every node has degree n.
- G is connected in a diameter of n (see hypercube property six, above).
- Any two adjacent nodes A and B are such that the nodes adjacent to A and those adjacent to B are linked in a one-to-one fashion (see hypercube property four, above).

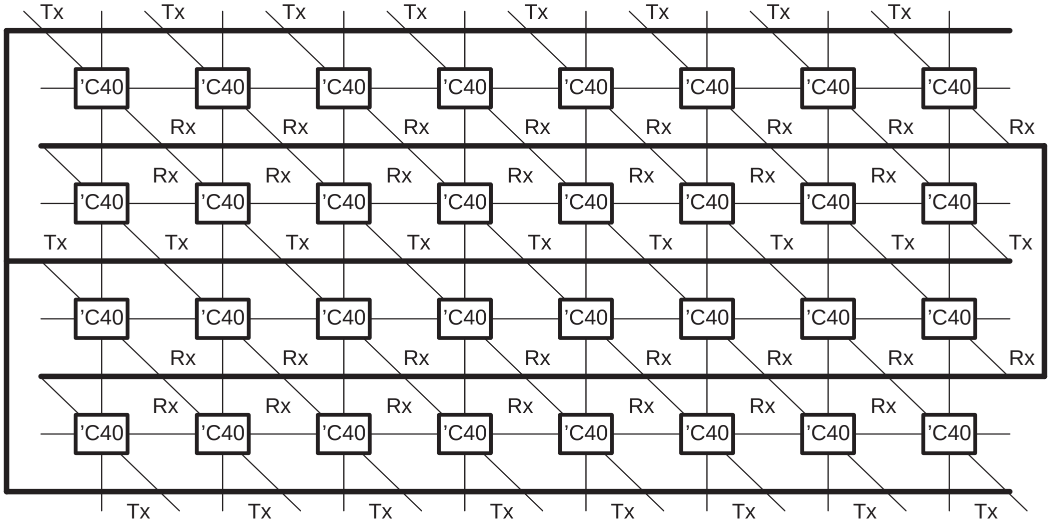
This theorem is proved by induction and discussed elsewhere, but it can be verified easily by applying the theorem to a mesh with eight nodes and a hypercube of three dimensions. In summary, the hypercube is one of the most versatile and efficient networks. It is well suited for both special-purpose and general-purpose tasks and can easily be implemented with the TMS320C40 DSP.

## Mesh Topologies

### *Simple Mesh*

A two-dimensional mesh is the simplest parallel processing topology. Its nodes are located at the intersection of rows and column links. Stacking two-dimensional meshes and linking all the nodes in the vertical plane yields a three-dimensional mesh (see Figure 18). The 'C40 is perfectly suited for a 3-D mesh in which four communication ports are linked to neighbors in the horizontal plane, and the remaining two are linked in the vertical plane. The edge nodes require only four links, and the corner nodes require three links. Extra edge and corner links can be used for data input/output, redundancy, and expandability. The mesh diameter is of the order log n, where n is the total number of mesh nodes.

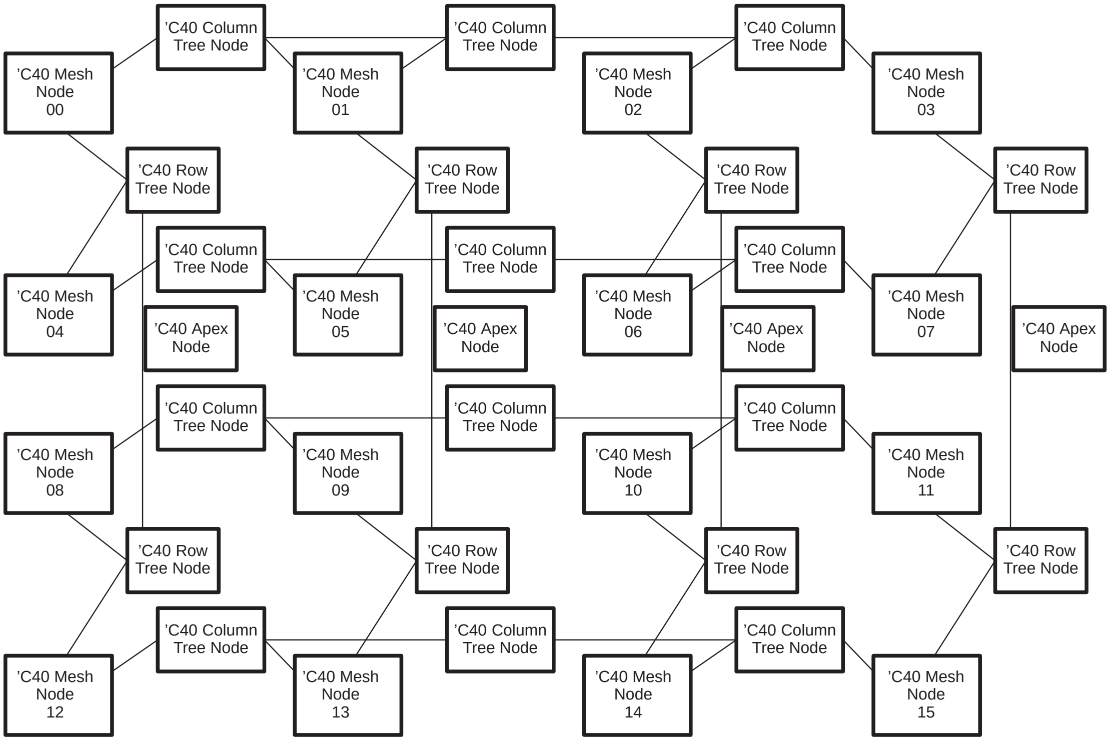
**Figure 18. Three-Dimensional Mesh**



### *Mesh of Trees*

Mesh of trees architecture gives a higher overall performance than an array ORing topology. The n × n mesh of trees [14] is constructed from an n × n mesh by adding processors and links to form a complete binary tree in each row and each column. In Figure 19, the mesh nodes are square and numbered from 00 to 15. A 'C40 row tree node is added to the mesh node 00 and node 04. Similarly, mesh nodes 08 and 12 are linked to another 'C40 tree node, and both of these tree nodes are linked to a 'C40 apex node to complete the binary row tree. After a complete binary row tree is completed for all the mesh nodes, columns are formed in the same way.

**Figure 19. Mesh of Trees Architecture**

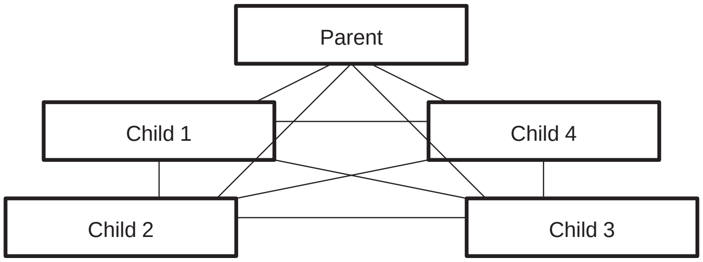


The tree leaves are the original $n^2$ nodes of the mesh. Overall, the network has $3n^2 - 2n$ processors. The leaf and root processors have two links, and all other processors have three links. The diameter of n × n mesh of trees is 4 log n. The mesh of trees also exhibits the recursive decomposition property, which is useful for parallel computations, for designing efficient layouts, and for fabricating a larger dimension mesh of trees from smaller dimension meshes of trees. This property is also helpful in implementing identical computations at low levels using the subnetworks, and for high-level global communication using the root processors. As the size of the network increases, the number of ports for communication increases linearly; this disadvantage is similar to that of hypercubes.

### *Pyramid Architecture and Its Properties*

A pyramid is a repetitive structure whose building block consists of the basic unit shown in Figure 20. Pyramids are widely used in image processing, where large data is segmented and decomposed.

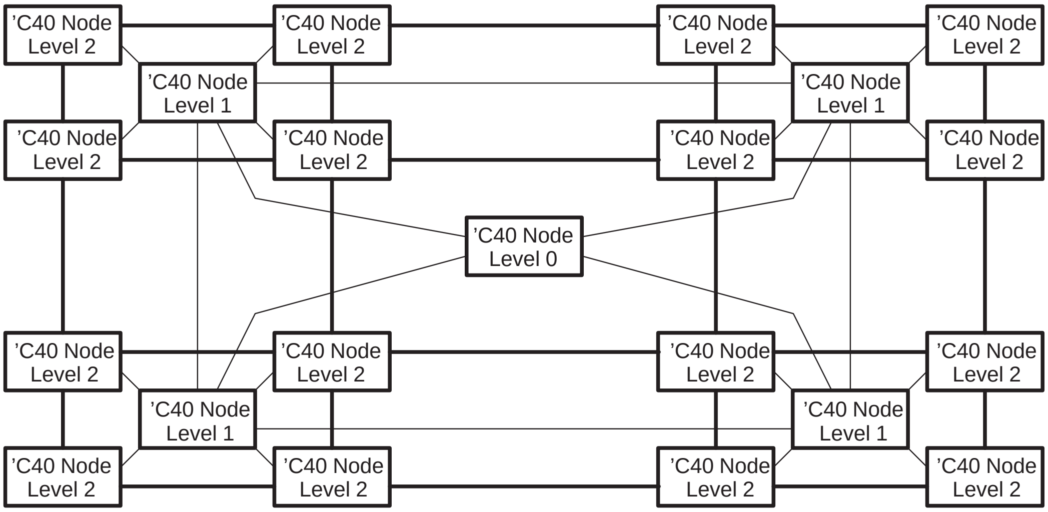
**Figure 20. Pyramid Building Block**



There are five ’C40 nodes in the building block of the pyramid. The ’C40 node on the top is called the parent of the four lower-level child nodes. Each parent can have only four children, and all the children are connected in a mesh. Each level of a pyramid is a mesh, and the bottom level or the level with maximum nodes ($2^n \times 2^n$) is called the *base* level. The ratio of nodes in a lower level to the node(s) in the adjacent upper level is 4:1. The topmost level ( level 0) is called the *apex* of the pyramid, and it has only one node ($2^0 \times 2^0$=1).

A two-level ’C40-based pyramid (see Figure 21) has 21 nodes (one apex node requires four links, four nodes at level 1 require seven links, and 16 base-level nodes require a maximum of five links). Since the ’C40 has a maximum of six communication ports, some nodes at level 1 will not be symmetrical. To compensate, a “dual processor single node,” as shown in Figure 15, is suggested for these nodes. The other option is to include a one-node-to-two-node multiplexer.

**Figure 21. Two-Level Pyramid**



The ’C40 offers several advantages when you build a pyramid. The concurrent DMA and CPU operations over the links reduce communication overhead for large amounts of data. The apex node has two free links that can be used for data I/O. Also, the children can pass data among themselves without slowing down the parent nodes. This is an advantage in image processing, where data decomposition and task scheduling are common.

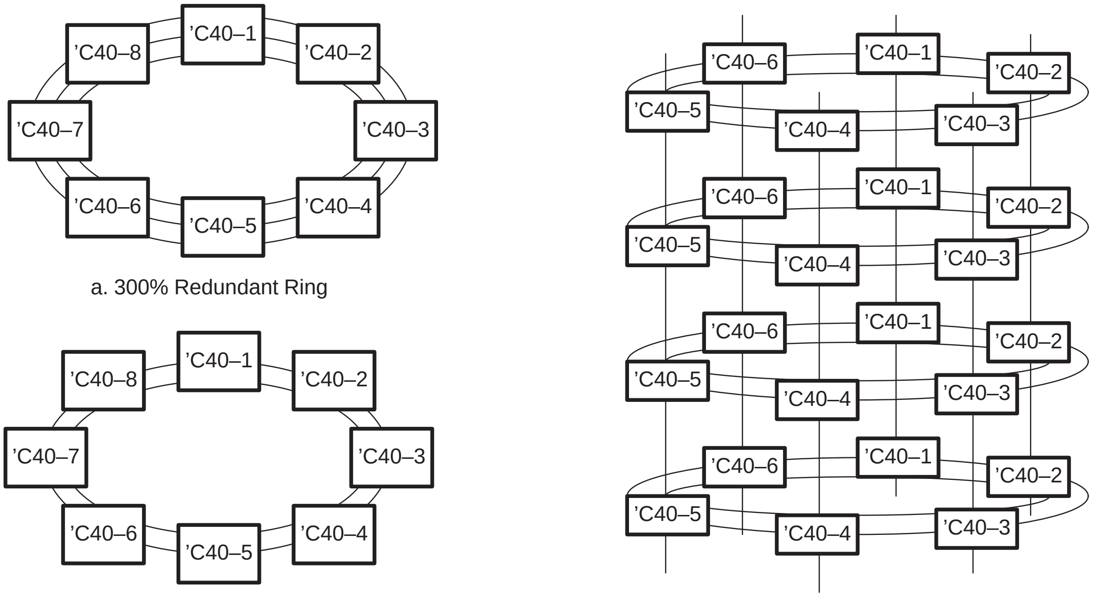
The Quad-40 board is a complete set of four children. To build a two-level pyramid, you need boards for the base level (a total of 16 ’C40s), another board for level 1, and a SPIRIT-40 dual board to complete the pyramid structure.

### *Ring Architecture*

A ring architecture and its variations, as shown in Figure 22, consist of a number of processors connected in an array fashion. When the last processor of an array is connected back to the first processor (wraparound array), a ring is formed. Ring architecture offers several advantages over a simple array in terms of computation and communication times. The communication protocols are well established and simple. When a mode or link malfunctions in ring architecture, however, messages must be rerouted in the reverse direction from the break point. Such rerouting taxes each processor’s CPU and increases the overall bandwidth on the remaining links.

The ’C40 processor node can increase the ring architecture reliability and link bandwidth by 300%, as shown in Figure 22a. The most versatile extension of ring architecture is the vertical stack of rings organized to form a cylindrical architecture. The cylindrical architecture offers two links for increased reliability in the horizontal plane and r number of vertical parallel paths, as shown in Figure 22c, where r is the number of nodes in any ring layer. The cylinder also provides for inputs at the top and outputs at the bottom. Cylindrical structures are highly modular because it is possible to increase the diameter of the ring and the height of the cylinder.

**Figure 22. Ring Architecture**



a. 300% Redundant Ring

b. 200% Redundant Ring With I/O Ports

c. Modular Cylindrical Architecture

## Reconfigurable Massively Parallel Processors

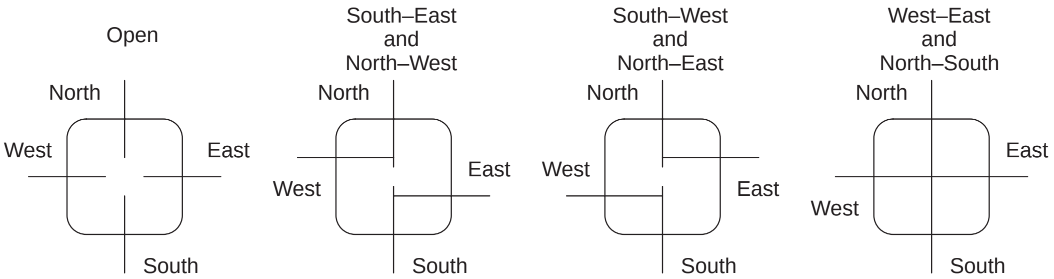
Massively parallel computers require many links, and the link-to-processor ratio differs, depending on the selected architecture. For example, the ’C40 links support hypercubes with a maximum of six dimensions and pyramids with only one level. To overcome the link limitation, massively parallel systems require reconfigurable architecture. Reconfigurable computers, as the name implies, can be reconfigured from one architecture to another without any physical modification in processor-to-processor connections. The computers are reconfigured via a link switch, which multiplexes one link to many or many links to many others. This class of machines offers high fault tolerance and overall lower cost systems if the cost of a link switch is less than adding additional links to the processor.

Switch configuration can be performed before the processing begins (nonreal-time), deterministically, or dynamically as the switch passes the packets to its destination. Hardware switch reconfiguration requires physical switches, and the following factors must be evaluated:

- Total number of paths that enter and exit from the switch (degree)
- Number of possible configurations for the switch (configuration settings)
- Number of lines entering the switch from each direction (data path width)
- Number of different paths that the switch can simultaneously connect (crossover capability)

Figure 23 shows a 2 × 2 switch in which any one link can be connected to any one of the other three links. This switch reconfigurability offers three processor-to-processor connections with only one link per processor. For example, the ’C40’s six ports can be physically increased to 18 links (18 adjacent nodes or an 18-dimension hypercube) via the 2 × 2 switch on each port.

**Figure 23. A Two-by-Two Switch With Four Possible Switch Settings**



The switch connection mechanism can be controlled by circuit switching or packet switching. In circuit switching, the links are pre-established; in packet switching, the packet header carries the destination address. Switches are cost-effective in implementing massively parallel processors.

### Software Approach to Massively Parallel Systems

The software approach for reconfiguration of interconnection networks is based on the graph embedding principle. Graph embedding, also known as mapping, and other techniques based on graph embedding, such as reshaping, direct embedding, embedding by graph decomposition, and many-to-one embedding can be found in related literature.

### Links for Massively Parallel Processing

Massively parallel systems require a large number of links. Large numbers of links can cause problems of electromotive force (emf) generation, signal degradation due to large link distances, cross-talk, ground loops, security, and less bandwidth speed. If a bus architecture is used instead of links, then the shared bus also lowers the system speed and requires extensive control and arbitration logic.

In the future, fiber-optic-based ’C40 communication port links will overcome the problems. The new fiber-optic link uses low power; has a single IC with transmitter, receiver, serial-to-parallel conversion, clock generation, and recovery functions; and is fully compatible with the emerging FDDI and other upcoming standards.

## Benchmarks, Analysis, and Data Decomposition

### Benchmarks

This section summarizes several benchmarks for evaluating multiprocessing architecture performance:

- **MFLOPS**

  This theoretical number represents the maximum number of arithmetic operations or instructions that can be executed by a device and is expressed as millions of floating-point operations per second (MFLOPS). The ’C40 can perform 40–50 MFLOPS. In some implementations such as filtering, it is possible to achieve 99% of theoretical MFLOPS. The factors that decrease throughput are memory contention, input/output bottlenecks, message passing, interrupt latency, and housekeeping. The ’C40 reduces bottlenecks with dual identical buses and a six-channel DMA coprocessor for both internal and I/O operations.

- **LINPACK**

  The LINPACK benchmark, also known as Dongarra’s benchmark, uses the Gaussian elimination process to solve a large matrix. A benchmark [10] for a 64-node hypercube, 1000 × 1000 matrix, three MFLOPS per node, 20-Mbytes/s links, and message overhead of 100 μs gives approximately 45% processor utilization. When this is extrapolated to the ’C40, where each node performance is 50 MFLOPS, 20 Mbytes/s link speed, and message overhead of approximately 100 cycles or 5.0 μs, a 64-node ’C40 hypercube can be expected to have a minimum of 45% or better processor utilization or a minimum LINPACK benchmark of 1.4 GFLOPS.

- **MIPS /MOPS** (Millions of Instructions/Operations per Second)

  These numbers are application-dependent because different instructions require different numbers of operations. For example, the ’C40 can perform as many as eight CPU and three DMA operations per cycle, which is equivalent to 275 MOPS.

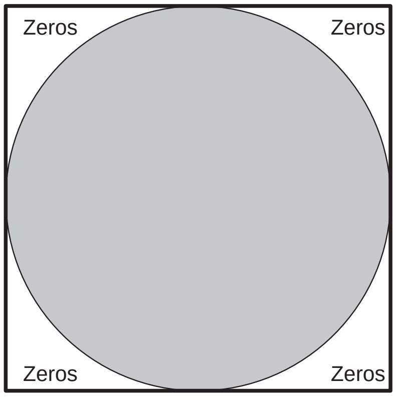
- **Kernel tasks**

  Kernel tasks occur frequently and influence the system throughput. Benchmarking these kernels—for example, the Fast Fourier Transform (FFT) algorithm, FIR, matrix operation, and correlation—shows relative performance for different parallel computers.

- **Calculation Algorithm**

  The dart board algorithm [11], Figure 24, can be used to estimate the value of π.

**Figure 24. Dart Board for π Estimation**



The dart board consists of a circle circumscribed by a square. The radius of the circle is one unit. Darts are thrown at the dart board in such a way that each dart has an equal probability of hitting any part of the square. If a dart lands within the circle, it is counted as 1; outside the circle, as a 0. At the end of the algorithm, the total number of ones are divided by the sum of the 1s and 0s. The estimate of π is obtained as shown below.

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi\ (\text{radius})^2}{(2 \times \text{radius})^2} = \frac{\pi}{4} \tag{6}$$

$$\frac{\#\text{ of 1s}}{(\#\text{ of 1s}) + (\#\text{ of 0s})} = \frac{\pi\ (\text{radius})^2}{(2 \times \text{radius})^2} = \frac{\pi}{4} \tag{7}$$

In the π benchmark algorithm, each ’C40 node runs its own dart board algorithm for π estimation. In a parallel system, the score increments in parallel on all nodes. At the end, the n estimates of π are globally combined. Each ’C40 node generates a nonoverlapping set of random numbers.

True benchmarking can be done only by implementing the actual algorithm on all the systems and architecture configurations of interest. This is a demanding task because it requires programming resources and hardware cost.

## Algorithm Analysis

This section discusses algorithm load distribution and speedup for multiple node systems.

### *Load Distribution*

If all nodes are identical (memory size, communication channel, clock speed) and if they execute a program with equal data size, they will all complete execution at the same instant. In this case, the ’C40 computation time, $T_i$, is the same for all nodes. However, if data size is unequal or if the start of execution is staggered, $T_{max}$ is calculated as follows:

$$n = \text{number of } {}'\text{C40 nodes};\ T_i = \text{computation time at } {}'\text{C40 node i}; \tag{8}$$

$$T_{max} = \max\{T_1,\ T_2,\ \ldots\ldots\ldots\ldots\ T_N\} \tag{9}$$

$$T_{wasted} = \sum_{i=1}^{N}(T_{max} - T_i\ ); \tag{10}$$

$$V = \text{Variation ratio} = \frac{T_{max}}{T_{min}},\ \text{where } T_{min} = \min\ \{T_1,\ T_2,\ \ldots\ldots\ldots\ldots\ T_N\} \tag{11}$$

For an ideal parallel processing system with a perfect load balancing, the variation ratio is:

$$V \cong 1 \tag{12}$$

and

$$T_{wasted} = 0 \tag{13}$$

### *Speedup*

Speedup and cost are two critical parameters for evaluating parallel systems against sequential systems. Speedup is defined as:

$$\text{Speedup} = \frac{\text{worst-case running time of fastest known sequential algorithm for problem}}{\text{worst-case running time of parallel algorithm}} \tag{14}$$

It may appear that speedup is directly proportional to the number of ’C40 nodes. This may not always be true, especially when data dependencies or parallel processing system restrictions prevent dividing the task into equal parallel tasks. Assuming that a program could be subdivided into n equal subprograms for parallel implementation on n ’C40 nodes, the system cost is defined as:

$$\text{Cost} = \text{program execution time} \times \text{number of } {'}\text{C40 nodes used} \qquad \textbf{(15)}$$

$$c(n) = t(n) \times p(n) \qquad \textbf{(16)}$$

If the parallel algorithm is perfectly load balanced, an optimum cost is obtained. If a parallel system is not load balanced, several ’C40 nodes may be idle and contribute to higher cost. A parallel processing system is cost-optimal if the cost of the parallel execution matches the known lower bound on the number of sequential operations. You can reduce the running time of a cost-optimal parallel algorithm by using more ’C40 nodes. You can also maintain the cost-optimal equation and use fewer ’C40 nodes at the price of higher running time. For example, in the *odd-even merging* algorithm, if the number of elements to be merged is equal to n, then the upper bound on a sequential computer is O(n) [order of n]. If the same merging is carried out on a parallel processing machine with n ’C40 nodes, the cost is:

$$\text{Cost} = t(n) \times p(n) \qquad \textbf{(17)}$$

where *t(n)* and *p(n)* are defined above.

$$= [1 + \log n] \times [1 + n \log n] = O(n \log^2 n) \qquad \textbf{(18)}$$

(read as order of n log square n)

Parallel implementation of merging is not cost-optimal, because $O(n) < O(n \log^2 n)$.

Efficiency (η) of a parallel algorithm is evaluated as shown below.

$$\eta = \frac{\text{worst-case running time of fastest known sequential algorithm for problem}}{\text{Cost} = p(n) \times t(n)} \qquad \textbf{(19)}$$

For example, if a program is executed sequentially in 100 ms and the same program is executed parallel over 10 processors in 20 ms, efficiency (η) is 50%.

As the value of n increases, other factors that must be evaluated are the cost of capital investment, maintenance, programming, etc.

## Data Decomposition

Data decomposition is an effective tool in evaluating architecture performance. A brief review of different techniques for data decomposition and task allocation is presented in this section.

### *Uniform Partitioning*

The task execution is independent of input data. Data is completely deterministic. In parallel execution, the data and computation are equally distributed among all ’C40 nodes for best load balancing and maximum efficiency. Uniform partitioning is used in convolution, FFT, and filtering algorithms.

### *Static Partitioning*

In Hough transformation, computation is proportional to the number of edges present and not to the size of the image. For example, if only a chair is present in the right-hand corner of a 512 × 512 image, then the ’C40 nodes with that portion of the image will be busy computing edges, whereas the rest of the ’C40 nodes will be idle because no edges are present. This load is not balanced, and the data is not decomposed. Static load balancing requires a priori information as an input for data decomposition.

### *Weighted Static*

Computation is a function not only of granule size and data density but also of spatial relationship of significant data. For this type of data, a weighted static scheme is useful for load balancing and data decomposition. In Hough transformation, information regarding granular size and the number of edges can be easily obtained. Two granular sizes may have equal amounts of significant data (edges), but in one case, the edges may not be related to each other at all. In the other cases, the edges may be spatially connected to form a certain object, and a large amount of computation may be required to identify an object.

### *Dynamic*

Uniform, static, and weighted static schemes decompose data after the data characteristics are known. However, it is possible to assign new tasks to a particular ’C40 node in advance by predicting that the current task will be completed after a certain period.

Table 2 shows [13] execution time for an eight-node system implementing a motion-detection algorithm. The dynamic method yields near perfect data decomposition (close to 1 variation ratio). This is also true for other algorithms that have data dependency, spatial relationship, and data variation across the inputs.

**Table 2. Comparative Time for Four Methods to Execute an Algorithm**

| Total<br>Eight Nodes | Uniform | Static | Weighted<br>Static | Dynamic |
|---|---|---|---|---|
| T maximum | 3500 | 1100 | 1200 | 1000 |
| T minimum | 100 | 200 | 750 | 950 |
| Variation Ratio V | 35 | 5.5 | 1.6 | 1.1 |

## Conclusion

The TMS320C40 is a nearly perfect node for implementing hypercube, ring, mesh, and pyramid architectural configurations. For integrating a parallel processing system, a 500-MFLOPS system can cost as little as $20,000 to $30,000. Many programming and design tools are now available for the ’C40, and many more will be available soon. For quick time to market, many board-level ’C40-based products that are available now can be integrated. Many low-cost and low-power derivatives of this processor from Texas Instruments are expected. Computationally intensive algorithms will be ported to the ’C40 and will be available as off-the-shelf algorithms. The TMS320C40 is destined to become the most commonly used parallel-processing node.

## Definitions

’C40 node: A node that consists of a processor, execution units, memory modules, communication port, and other necessary hardware.

Dimension of hypercube structure: $n = \log_2 p$ (p = number of processors)

Distance: The number of links used by the message to travel from the source ’C40 node A to the destination ’C40 node B.

Link: A physical connection between any two nodes.

Hamming Distance (HD): The number of bits that are different in two A and B binary addresses. For example, if A is (1010101) and B is (0101010), then the Hamming distance is 7.

Node/Vertex (N): A processing element that is in the network. Also, the point in the graph where one or more edges meet.

Parallel Paths: Paths with equal lengths and nonoverlapping edges.

Computation time ($T_i$) Computational time on a single-processor system.

Maximum computation time ($T_{max}$)

$$T_{max} = \max\{T_1,\ T_2,\ \ldots\ T_N\}$$

Minimum computation time ($T_{min}$)

$$T_{min} = \min\{T_1,\ T_2,\ \ldots\ T_N\}$$

where N = P = # of processors

Variation Ratio (V)

$$V = \frac{T_{max}}{T_{min}}$$

Wasted time ($T_{wasted}$)

$$T_{wasted} = \sum_{i=1}^{N} (T_{max} - T_i\ );$$

’C40 Texas Instruments TMS320C40 floating-point DSP.

Efficiency Ep = Sp/p with values between (0,1), is a measure of processor utilization in terms of cost efficiency.

Speed-up ($S_p$) $S_p = T_s/T_p$, where $T_s$ is the algorithm execution time when the algorithm is completed sequentially; whereas, $T_p$ is the algorithm execution time using p processors.

Parity: The number of 1s in a node’s binary address. Parity of a node is positive if there are an even number of 1s in the node’s binary address; if the number of 1s is odd, parity is negative.

Path: The ordered list of processors visited by the message in going from the source ’C40 node A to the destination ’C40 node B.

# References

[1] Tim Studt. R&D Magazine, July 1992.

[2] TMS320C4x User’s Guide, Dallas; Texas Instruments Incorporated, 1991.

[3] Parallel 2-D FFT Implementation With TMS320C4x DSPs Application Report, Dallas; Texas Instruments Incorporated, 1991.

[4] K. Hwang and D. DeGroot. Parallel Processing for Supercomputers and Artificial Intelligence. New York: McGraw-Hill, 1989.

[5] R.H. Kuhn and D.A. Padua. Tutorial on Parallel Processing. IEEE Computer Society, 1981.

[6] Z. Hussain. Digital Image Processing. Ellis Horwood, 1991.

[7] S.G. Akl. The Design and Analysis of Parallel Algorithms. Englewood Cliffs: Prentice-Hall, Inc., 1989.

[8] Y. Saad and M.H. Schultz. Topological Properties of Hypercubes.

[9] Product Catalog, Sonitech International, Wellesley, Massachusetts 1992.

[10] Hungwen Li and Q. F. Stout. Reconfigurable Massively Parallel Computers. Englewood Cliffs: Prentice-Hall, 1991.

[11] G. Fox & others. Solving Problems on Concurrent Processors, Volume I. Englewood Cliffs: Prentice-Hall, 1988.

[12] M. Ben-Ari. Principles of Concurrent and Distributed Programming. Englewood Cliffs: Prentice-Hall, 1990.

[13] A.N. Choudhary and J.H. Patel. Load Balancing and Task Decomposition Techniques for Parallel Implementation of Integrated Vision Systems Algorithms, Proceedings of Supercomputing 1989, November 13–17, 1989, pp. 266–275. New York: IEEE Computer Society.

[14] F. Thomson Leighton. Introduction to Parallel Algorithms and Architectures. San Mateo: Morgan Kaufmann Publishers, 1992.

[15] Sonitech internal program “Iterative Method for ’C40 Communication Port Interconnection.”

# Prototyping the TI TMS320C40 to the Cypress VIC068/VAC068 Interface

**Peter F. Siy and David L. Merriman**
**The MITRE Corporation**

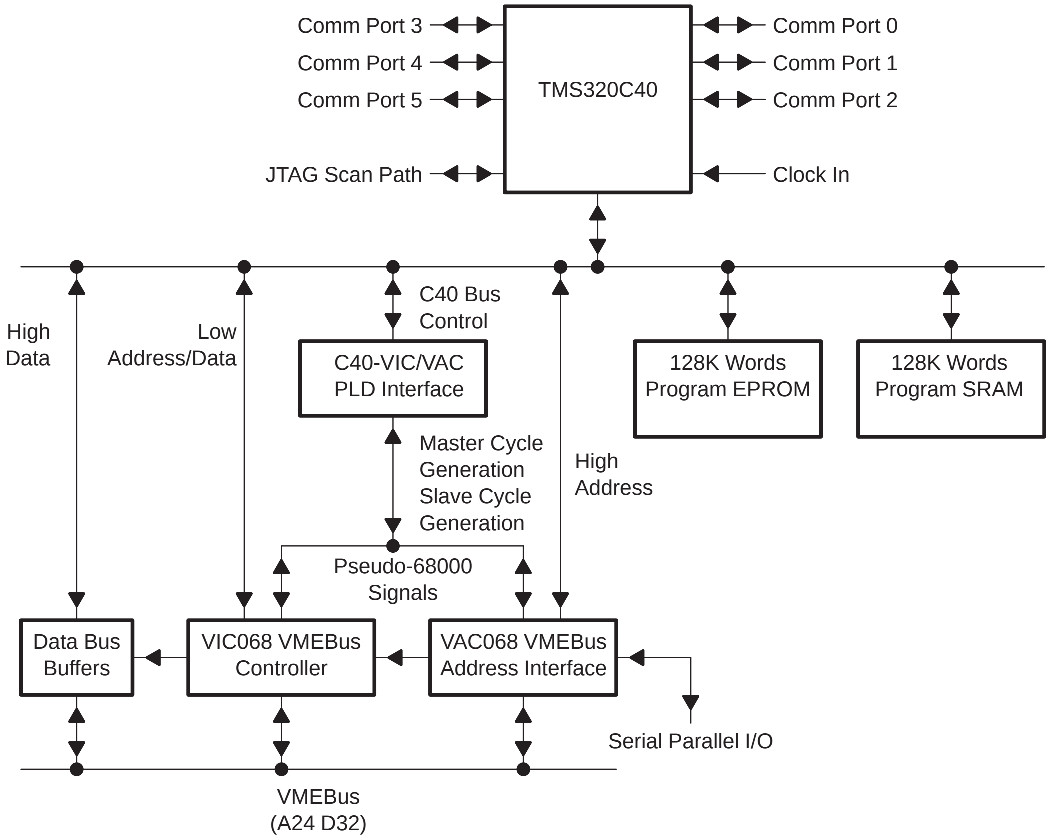
**Timothy V. Blanchard**
**Cypress Semiconductor**

## Introduction

The Texas Instruments TMS320C40 digital signal processor (DSP) represents a state-of-the-art solution to many signal processing problems via its high-speed central processor unit (CPU), unique parallel processing I/O capability, and robust interface to other system components [1]. Likewise, the Cypress Semiconductor VIC068 VMEbus Interface Controller and its companion VAC068 VMEbus Address Controller provide a complete VMEbus interface, including master and slave capability [2,3]. Since these components are effective in a wide variety of applications and since the VIC068/VAC068 is a single- or multiple-TMS320C40 VMEbus card design, we developed a single TMS320C40 VMEbus card for use in a satellite modem application [4].

The VIC068/VAC068 is designed to interface with the Motorola 68000 family of microprocessors, so it was determined that the interfacing to a TMS320C40 required a logic simulation or some form of programmable, configurable prototype. When this design was initiated, only preliminary documentation existed for the VMEbus chip set, and no simulation models were available for either the chip set or the TMS320C40 DSP. Therefore, we first prototyped the interface of these components in a wire-wrap environment before proceeding to a printed circuit board design. This paper provides the high-level as well as low-level details of the prototyping effort so that others may examine our approach and techniques to minimize design time for subsequent efforts. Note that this design has not been optimized for either size or speed. Section 2 outlines the design goals established before design began and gives relevant background regarding the devices involved. The paper focuses on the hardware details, programmable logic source code, and schematics that follow. In addition, the software initialization of the chip set by the ’C40 is described. Throughout this paper, we assume that you are familiar with the ’C40 architecture as well as with the VMEbus and its protocol(s). You can refer to [5,6] for more details on the VMEbus. Figure 1 shows the VIC068/VAC068 prototype block diagram.

### Figure 1.  TMS320 – VIC/VAC Prototype Block Diagram



# Prototype Design

## Design Goals

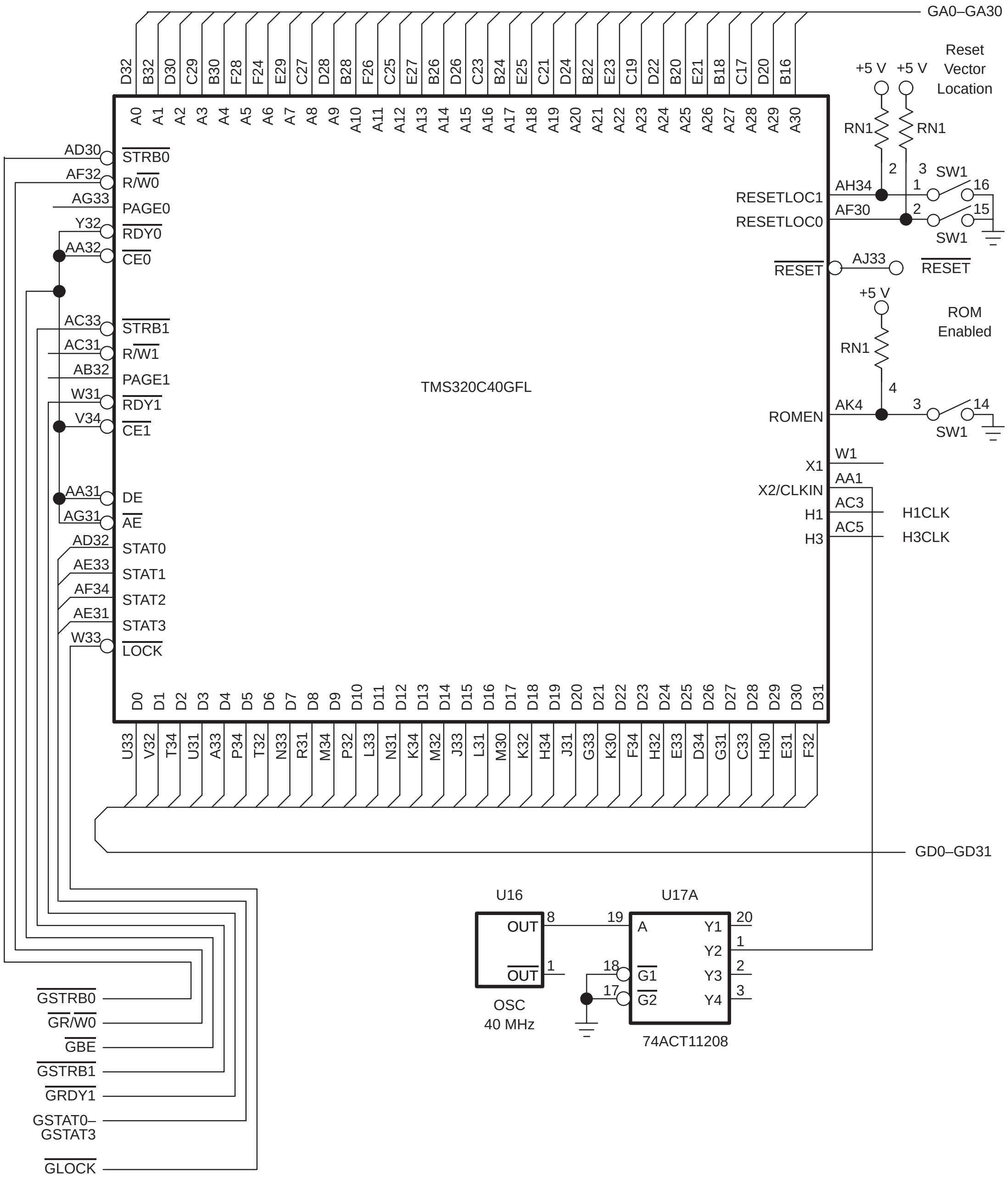
We began by developing a set of design goals for the VME interface that were based on our particular needs. We were interested in a ’C40 card that provided both (VMEbus) master and slave capability for reads, writes, read-modify-writes, write posting, and slave block transfers. We designed the address/data capability according to the most prevalent configuration (for other cards available commercially): 24-bit address and 32-bit data (i.e., A24/D32); however, the design presented here does not preclude 32-bit addressing with minor modifications. Via the VIC068, this design also features system controller capability. We did not incorporate VMEbus interrupt support, because we provided application-specific interrupt inputs directly to the ’C40. We utilized the VAC068 for address control/mapping of the two Universal Asynchronous Receiver/Transmitters (UARTs) that were required for our application, and for general-purpose parallel I/O. The new Cypress CY6C964 Bus Interface Logic Circuit can be used instead of the VAC068. In addition to the VMEbus functionality, we required the interface to be compatible with both the existing ’C40-40 (50-nanosecond cycle time) and the faster ’C40 (40-nanosecond).

## Design Considerations

First, we thoroughly examined the VIC068 and VAC068 and reviewed the 680x0 family bus signals and cycles. In particular, we referred to the 68020 user’s manual [7] extensively. The VIC068 and VAC068 interface directly to the Motorola 680x0 family data, address, and control signals and are driven with the familiar 680x0 address and data strobes ($\overline{\text{PAS}}$, $\overline{\text{DS}}$). An asynchronous transfer protocol is implemented via data transfer and size acknowledgment signals $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$. In addition to these signals, the transfer size signals SIZ0 and SIZ1 are essential elements in the 680x0’s dynamic bus sizing capability and, with the lower address lines, encode the size of the transfer in progress. Also, during the transfer, the function code signals (FC0–FC2) provide information of importance in multiuser environments. Bus arbitration is an integral part of the 680x0 via the bus request ($\overline{\text{BR}}$), bus grant ($\overline{\text{BG}}$), and bus grant acknowledge ($\overline{\text{BGACK}}$) signals and is used directly by the VIC068. Finally, as in many other general-purpose microprocessors, bus cycles for the 680x0 are several clock cycles long.

While the VIC068 and VAC068 are driven by (and can drive) the familiar 680x0 bus signals, the ’C40 bus signals show little similarity to those of the 680x0 family. The ’C40’s bus protocol is common to the TMS320 floating- point DSP product line. An external ready signal allows for wait-state generation and controls the rate of transfer in a synchronous fashion (i.e., cycles can be extended an integer number of clock cycles). As described in [1], the ’C40 has two identical external interfaces: the global memory interface and the local memory interface. Each consists of a 32-bit data bus, a 31-bit address bus, and two sets of control signals. The benchmark of all DSP technology, the ’C40 executes single-cycle instructions (see the ’C4x user’s guide for complete details) and relies on a multistage pipeline for execution speed. Detailed bus status, including type of instruction and type of access, is given for each cycle via the STAT lines. Individual control lines can put the address, data, and control bus(es) in the high-impedance state. There is no read-modify-write signal (as on the 680x0); however, an instruction-driven $\overline{\text{LOCK}}$ signal is available. Each cycle is controlled by a strobe ($\overline{\text{STRBx}}$) signal in conjunction with the corresponding read/write (R/$\overline{\text{Wx}}$) strobe. One of the ’C40’s notable features is its range of configuration options. The ’C40 has evolved from its earlier floating-point counterparts into a truly flexible interface via the local and global bus configuration control registers.

## Figure 2. TMS320C40 – VIC/VAC Prototype ’C40 Global Bus

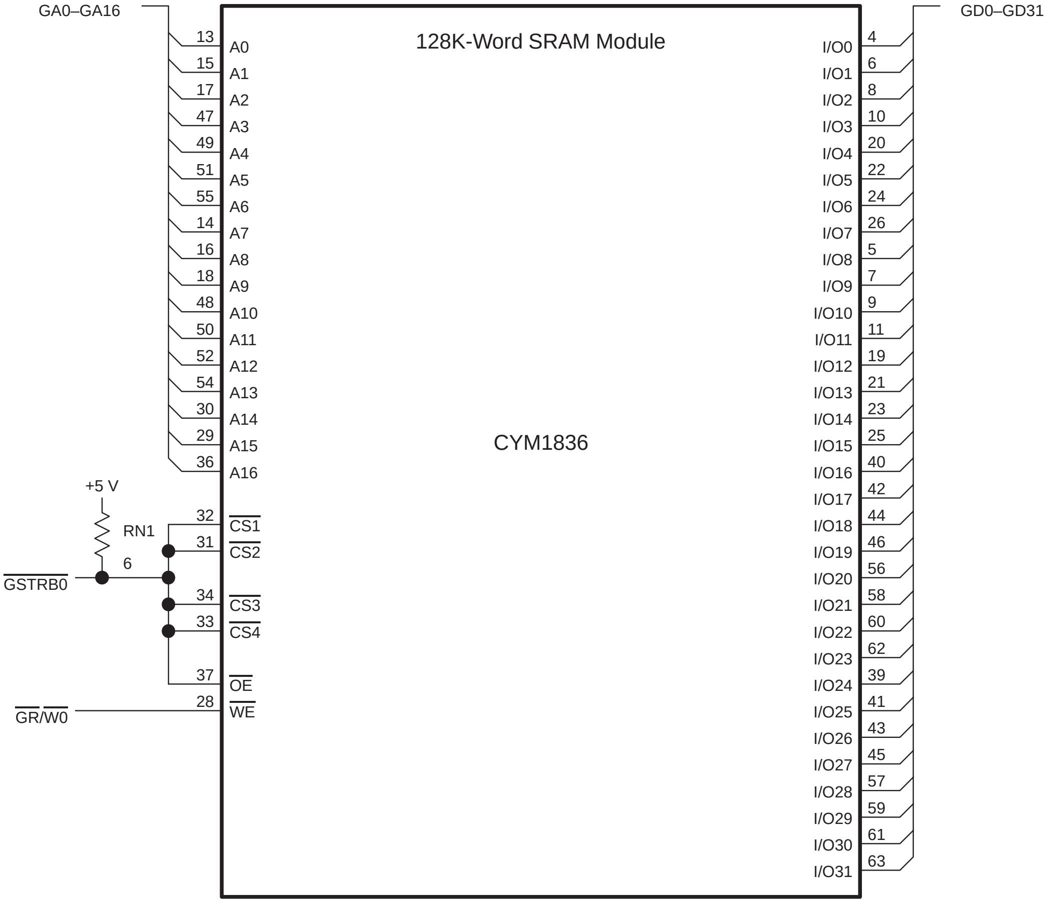

GA0–GA30
D32 B32 D30 C29 B30 F28 F24 E29 C27 D28 B28 F26 C25 E27 B26 D26 C23 B24 E25 C21 D24 B22 E23 C19 D22 B20 E21 B18 C17 D20 B16
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 A17 A18 A19 A20 A21 A22 A23 A24 A25 A26 A27 A28 A29 A30
+5 V
+5 V
Reset
Vector
Location
RN1
RN1
2
3
SW1
1
16
2
15
SW1
AH34
AF30
RESETLOC1
RESETLOC0
RESET
AJ33
RESET
+5 V
ROM
Enabled
RN1
4
ROMEN
AK4
3
14
SW1
X1
W1
X2/CLKIN
AA1
H1
AC3
H1CLK
H3
AC5
H3CLK
TMS320C40GFL
AD30
STRB0
AF32
R/W0
AG33
PAGE0
Y32
RDY0
AA32
CE0
AC33
STRB1
AC31
R/W1
AB32
PAGE1
W31
RDY1
V34
CE1
AA31
DE
AG31
AE
AD32
STAT0
AE33
STAT1
AF34
STAT2
AE31
STAT3
W33
LOCK
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15 D16 D17 D18 D19 D20 D21 D22 D23 D24 D25 D26 D27 D28 D29 D30 D31
U33 V32 T34 U31 A33 P34 T32 N33 R31 M34 P32 L33 N31 K34 M32 J33 L31 M30 K32 H34 J31 G33 K30 F34 H32 E33 D34 G31 C33 H30 E31 F32
GD0–GD31
U16
U17A
OUT
8
19
A
Y1
20
Y2
1
OUT
1
18
G1
Y3
2
17
G2
Y4
3
OSC
40 MHz
74ACT11208
GSTRB0
GR/W0
GBE
GSTRB1
GRDY1
GSTAT0–
GSTAT3
GLOCK

## High-Level Architecture

The high-level architecture for the card places fast 20-ns high-density 4-megabit (128K × 32) Cypress CMOS SRAM modules on both local and global buses of the ’C40 (the size of the memory array does not impact the ’C40-to-VIC068/VAC068 interface design). We designated the global side as program memory and the local side as data memory for our application. In our environment, it is anticipated that the local memory will be fully occupied by DMA coprocessor activity coupled with data fetches during communications-oriented DSP operations. Given this, we chose to place the A24 VME spectrum on the global (program) side, segmenting the local side I/O activity (the critical path for our application) from all VMEbus activity. (However, the interface documented herein can be used on either side because the global and local buses are symmetric.) On the global side, in addition to program SRAM, we also placed two 128K × 16 EPROMs for embedded program store, using the boot load feature of the ’C40.

Because we limited our design to VMEbus A24 addressing, this range fits nicely anywhere in the ’C40 global side address spectrum, from 08000 0000h to 0FFFF FFFFh. Therefore, VMEbus master access is memory mapped into the ’C40 global side address range. When an access occurs in this predefined A24 range, the ’C40 bus signals are transformed into 680x0 bus signals, which drive the VIC068/VAC068 pair and initiate a VMEbus transfer. Global side accesses outside of this range do not generate such signals and occur at full speed (i.e., the speed appropriate for that memory or peripheral). Regarding the “endianess” [8,9] of the interface, we know that the 680x0 family maintains big-endian byte ordering (byte-addressable memory organization) with little-endian bit ordering in each addressable unit. In contrast, the ’C40 is flat in its byte endianess (32-bit word addressable only) and little-endian bit ordered. Therefore, no swapping is done on the interface, because 32-bit word transfers (between processors) maintain D0 as the least significant bit. This forces a tradeoff of transfer speed for a wider range of transfers (byte, word, and three byte) than the ’C40 offers. We chose to limit our transfers to D32. To make transfers of all sizes available, you must preform additional setup and/or decoding before/during the transfer in progress.

Figure 3. TMS320C40 – VIC/VAC Prototype Program SRAM

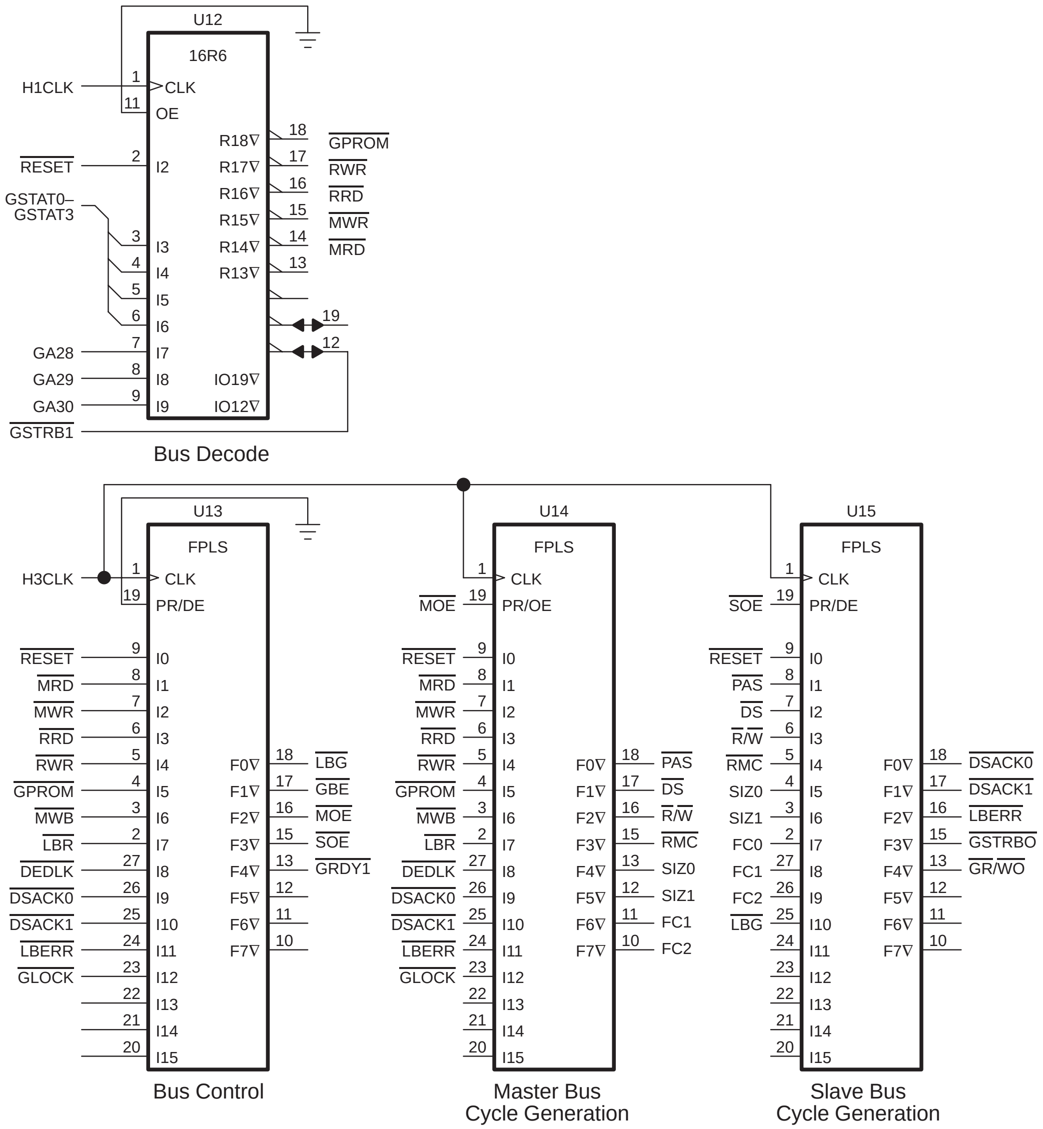


## Hardware Description

After examining the VIC068/VAC068 interface and capabilities and comparing them with the ’C40, we initiated a prototype design. Based on the preceding discussion, the strategy is to map from the given set of ’C40 bus signals to a set of 680x0-like signals, driving their counterparts on the VIC068 and VAC068 for master cycles (the ’C40 is reading from or writing to the VMEbus). Not only can the ’C40 initiate VMEbus cycles as a bus master, but also the card should respond to slave cycles. Most often, slave access is gained through shared memory on the (slave) card. On the ’C40-based VME card, one set of signals is required to respond to bus requests from the VIC068/VAC068, and an additional set is required to “hold off” the ’C40 global side during such transfers.

To accomplish this transformation of signals, programmable logic is applied. We wanted to keep the design time to a minimum while maintaining the most flexible (i.e., programmable) design. Based on this, we used the Texas Instruments TIB82S105BC programmable logic sequencer. This device is a field-programmable Mealy-type state machine with 16 inputs, 8 outputs, 48 product terms, and 14 pairs of sum terms; it operates at a maximum frequency of 50 MHz [10]. Development tools for these sequencers are plentiful and inexpensive — we used Data I/O’s ABEL version 4.0 for all programmable logic device (PLD) development.

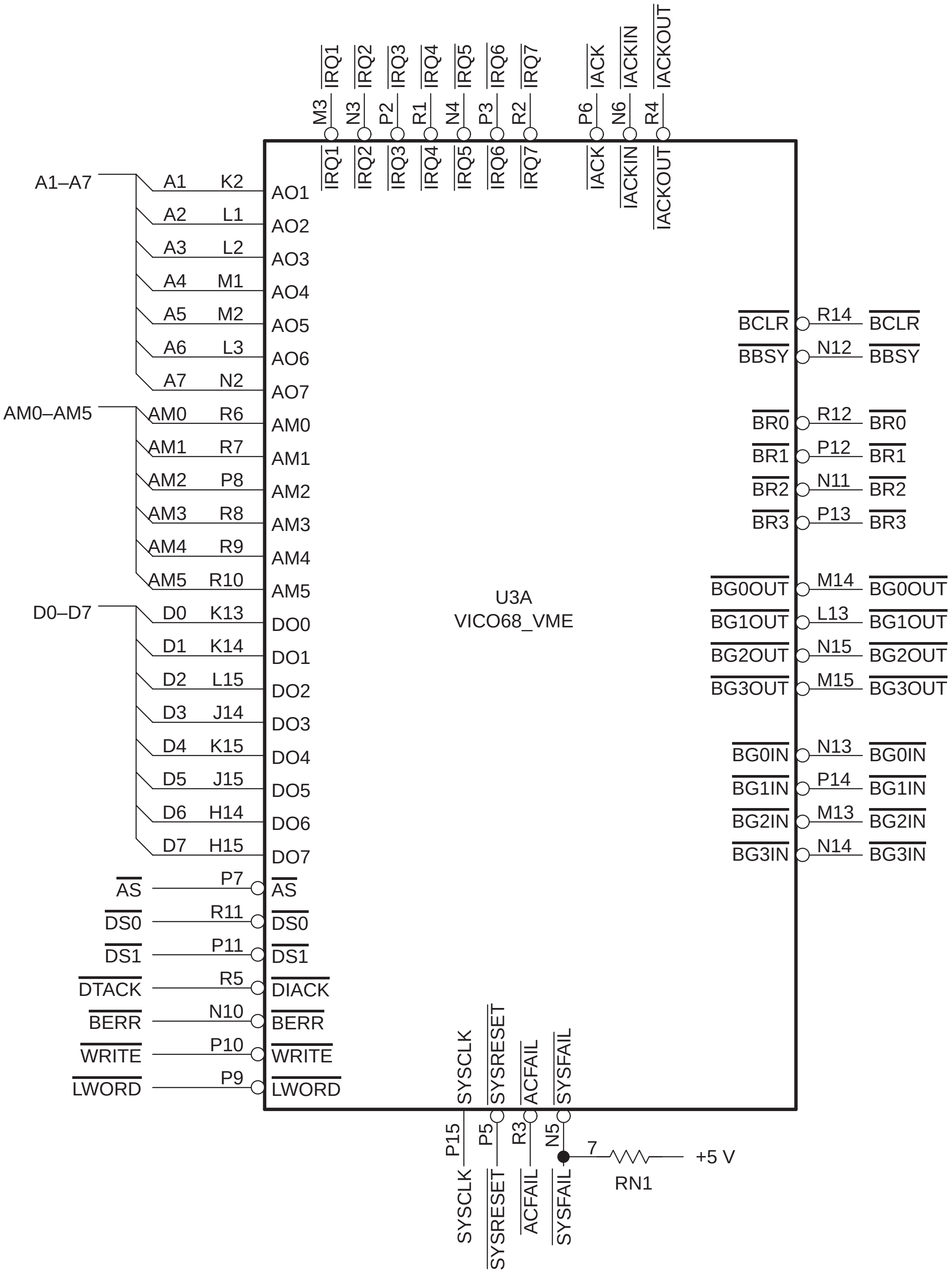
**Figure 4. TMS320C40 – VIC/VAC Prototype Programmable Logic**



## Reset Circuitry

We found that the VIC068/VAC068 required a “global” reset to operate correctly. In particular, the VIC068 $\overline{\text{IRESET}}$ signal should be driven with the incoming reset request. This signal can be buffered as shown to provide the delayed signal to the $\overline{\text{IPL0}}$ input —thereby providing the required stimulus for a VIC068 global reset. Given these inputs, the VIC068, in turn, drives the $\overline{\text{RESET}}$ line (and the $\overline{\text{SYSRESET}}$ line if the VIC068 is configured as a system controller) for 200 milliseconds. We used the $\overline{\text{RESET}}$ output on the VIC068 to reset both the ’C40 and the VAC068, as well as programmable logic on-board.

## Figure 5. TMS320C40 – VIC/VAC Prototype VICO68 VMEbus Interface



## Address Bus Decoding

The VIC068/VAC068 interface (and consequently the VMEbus itself) is mapped into the ’C40 global side at 0D000 0000h. In our application, we divided the global side into two halves via the STRB ACTIVE field in the ’C40 global memory control register. We placed zero-wait-state devices (fast SRAM) in the lower half and placed slower memory (EPROM) and peripherals (the VIC068/VAC068 pair) in the upper half. Therefore, the ’C40 addresses program memory via STRB0 and addresses the VMEbus via STRB1. As shown in the accompanying schematics, U12 is a 16R6 programmable device (in particular, a Texas Instruments TIBAL16R6-5C was used). It decodes each global ’C40 STRB1 bus cycle by using the ’C40 H1 clock. Cycle type decoding is performed fully via the STAT lines (instead of using the R/$\overline{\text{W}}$ strobe) and allows for future expansion/reconfiguration if required. As shown, the STRB1 range is divided into eight distinct segments by using GA28—GA30 (GA31 is implicitly a logic 1). Outputs of the decoding operation are VMEbus master write ($\overline{\text{MWR}}$), master read ($\overline{\text{MRD}}$), VIC068/VAC068 register write ($\overline{\text{RWR}}$), register read ($\overline{\text{RRD}}$) and EPROM read ($\overline{\text{GPROM}}$). The VIC068/VAC068 documentation shows that the VAC068 is hard-wired, starting at address 0FFFD 0000h, and designates VIC068 selection, starting at address 0FFFC 0000h. A memory map for the global side, as decoded by the 16R6 logic device, is shown in Table 1. The ABEL source code is provided in the appendix.

**Table 1. Global Side Memory Map**

| Address | Unit Addressed |
|---|---|
| 08000 0000h | SRAM |
| 0C000 0000h | EPROM |
| 0D000 0000h | VMEbus A24 |
|  | Address 00 0000h |
| 0FFFC 0000h | VIC068 Register Set |
| 0FFFD 0000h | VAC068 Register Set |

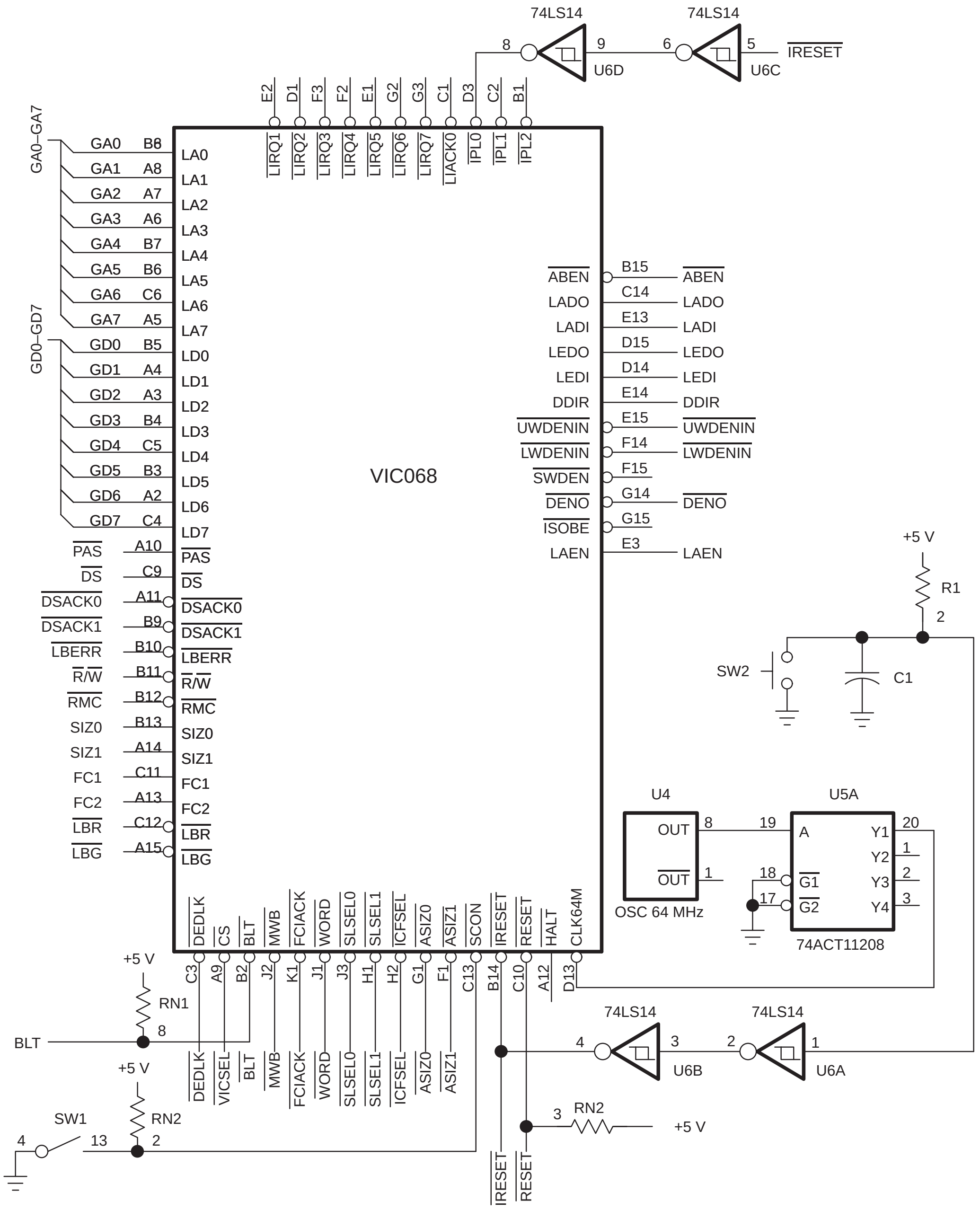
## Bus Control

Once a cycle in the VMEbus address range is detected by the address-decoding programmable logic, the sequencers provide the signals required for both master and slave cycles. U13 is the first of three sequencers and facilitates overall bus control, providing these enable signals: ’C40 global bus ($\overline{\text{GBE}}$), master cycle sequencer output ($\overline{\text{MOE}}$), slave cycle sequencer output ($\overline{\text{SOE}}$), VMEbus slave local bus grant ($\overline{\text{LBG}}$), and a ’C40 ready signal ($\overline{\text{GRDY1}}$). Notice that a full complement of inputs is presented to the bus control sequencer. This was done to accommodate all possible cycles and allow reconfiguration without hardware changes. While the ’C40 H3 clock (20 MHz) was used here, this is not an absolute requirement, because the array of sequencers operates asynchronously, once a master or slave cycle begins. However, using H3 simplifies the sequencer code because the H3 clock serves as a convenient reference to the ’C40 cycle in progress.

A master cycle begins with U12 generating a master read or write signal or a register read or write. This enables the output of the master bus cycle signal generation sequencer U14 (in fact, this signal is asserted during all bus activity other than slave cycles). A master cycle ends with the assertion of the acknowledge signals $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$ and/or the local bus error signal $\overline{\text{LBERR}}$, all generated by the VIC068 in response to acknowledge signals received over the VMEbus. The sequencer responds to these signals by asserting $\overline{\text{GRDY}}$ to supply the ready signal $\overline{\text{RDY1}}$ for this ’C40 STRB1 access. In this design, external ready signals are used exclusively (versus ANDing or ORing with internal ready), and the generation of the ready signal conforms to the second of two methods described in [1]: high between accesses.

Slave cycles are initiated by the assertion of the VIC068 local bus request ($\overline{\text{LBR}}$) input signal. Then, U13 provides bus control by first disabling the ’C40 global bus (deasserting $\overline{\text{GBE}}$) and the master cycle sequencer output ($\overline{\text{MOE}}$) and then by enabling the outputs on the slave cycle sequencer ($\overline{\text{SOE}}$), U15. When the bus has been successfully “seized”, the local bus grant signal ($\overline{\text{LBG}}$) is asserted. Slave cycles are terminated by the deassertion of the local bus request input signal.

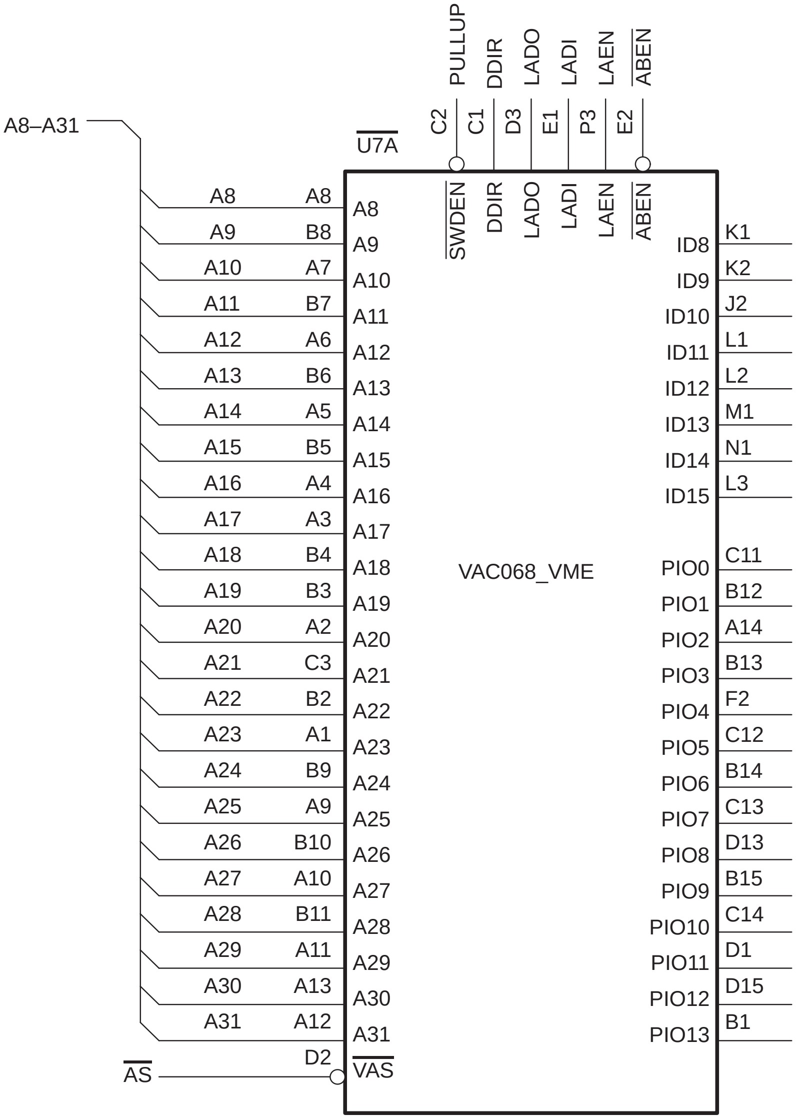
**Figure 6. TMS320C40 – VIC/VAC Prototype VICO68 Local Bus Interface**



## Master Bus Cycle Generation

The master bus cycle generation sequencer U14 runs in tandem with the bus control sequencer U13, and the sequencer code found in U13 and U14 results from one common state diagram. It was necessary to split these functions because of the number of outputs per sequencer. Therefore, the inputs to U14 are identical to those on U13. Master bus cycles proceed according to the appropriate cycle (read or write) definition in [7]. The function code lines are driven to indicate the widest possible audience, supervisory data. Termination of a master cycle ends with the assertion of the acknowledge signals DSACK0 and DSACK1 and/or the local bus error signal LBERR as described above. Note that VIC068/VAC068 register accesses are also master accesses in the ’C40 global side address range. While the sequencer code does not initiate read-modify-write cycles, you could use the ’C40 GLOCK input to do this.

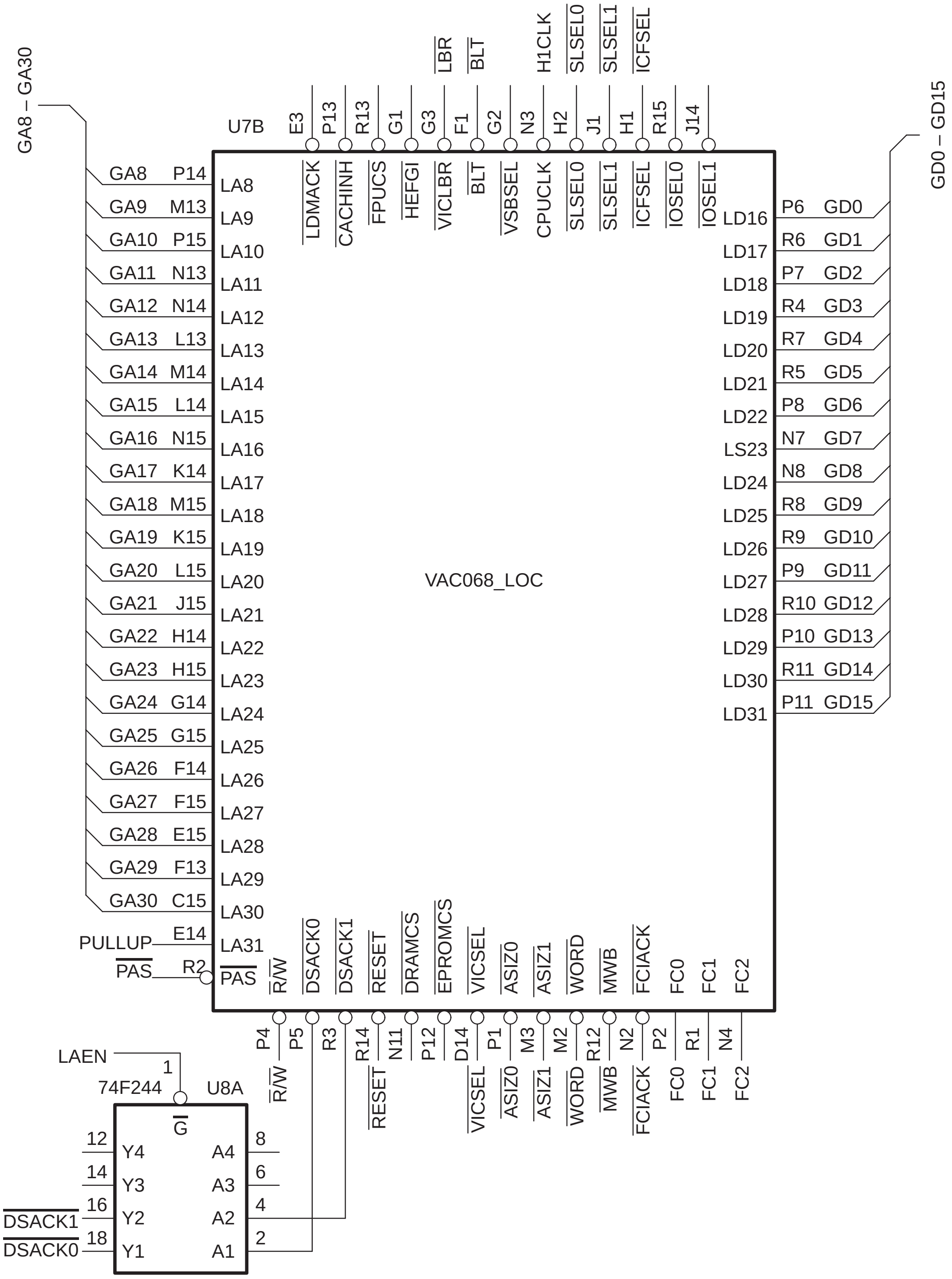
**Figure 7. TMS320C40 – VIC/VAC Prototype VACO68 VME Interface**



### Slave Bus Cycle Generation

Slave cycles are initiated by the VAC068 in response to a request over the VMEbus in the selected range as determined in the appropriate VAC068 register (discussed in *VIC068/VAC068 Software Initialization, page 62*). Inputs to the sequencer are the common 680x0 bus signals driven by the VIC068 for slave cycles (and alternately driven by the master sequencers for master cycles). Assertion of the local bus grant signal $\overline{\text{LBG}}$ by U13 indicates the absence of the ’C40 on the global bus, thereby allowing access of shared global SRAM by the VIC068/VAC068 pair. Assuming the correct transfer size, the memory strobe signals $\overline{\text{GSTRB0}}$ and GR/W0 are driven, providing access to the shared global SRAM. After this, acknowledgement is provided via $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$, ending the slave cycle. Note that while VAC068 documentation states that its DSACK signals can be put in the high-impedance state on the assertion of LAEN, we found this not to be the case with our configuration. Therefore, U8A was required to artificially put those signals in the high-impedance state so that the slave sequencer could control the data acknowledgement.

Figure 8. TMS320C40 – VIC/VAC Prototype VACO68 Local Interface



## VIC068/VAC068 Software Initialization

While the VIC068/VAC068 pair register set is, at first glance, overwhelming, we found that very few registers require attention before the pair can be used for either master or slave operations. The VAC068 should be initialized first because it controls both master and slave address mapping. When initialization is complete, the VIC068 is programmed. You can fine-tune the interface by using the programmable delay registers for the interface after initial capability is verified. As we programmed the VIC068/VAC068 using C, we developed vic.h and vac.h header files, which give base and offset definitions for the complete register set of each device.

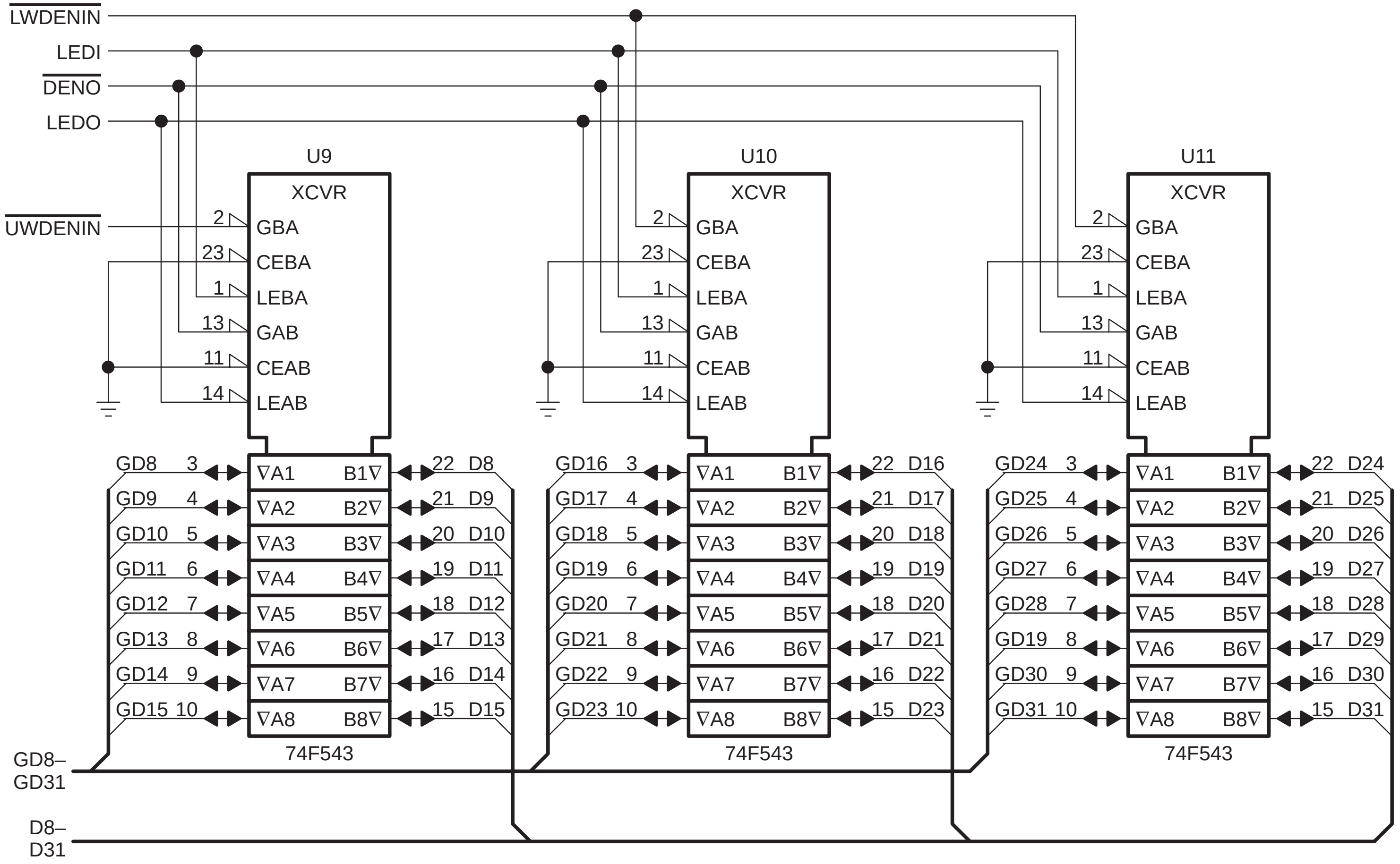
Before programming the VIC068/VAC068 pair, you must bring the VAC068 out of its initial Force EPROM mode (which asserts $\overline{\text{EPROMCS}}$ for all accesses) by reading from the EPROM space beginning at 0FF00 0000h. While the address-decoding programmable logic device U12 does not provide for access to this range, we can initiate a dummy access to this region by manipulating the ’C40 global memory interface control register. We first set the SWW and WTCNT fields so that the register will provide zero-wait-state, internal ready dependent (only) accesses to the appropriate strobe (STRB1 for our case). We then perform a read from address 0FF00 0000h, reset the SWW field to external ready accesses, and perform a second read to the VAC068 — this time at the VAC068 register base 0FFFD 0000h. This second read provides the required access to snap the sequencers back to their default states.

After the Force EPROM mode is exited, we first verify that the VAC068 can be addressed by reading the device ID via the VAC068 ID register. Then, we program the slave SLSEL0 base address register, the SLSEL0 mask register, and the master A24 base address register. To enable the VAC068 decode and compare functions, the last step is to write to the VAC068 ID register. The VIC068 ID register is similarly polled; following the successful read of that register, we set the address modifier source register and the slave select 0 control 0 register. This completes the initial programming of the pair. Now, we can extinguish the SYSFAIL LED (if applicable) by writing to the interprocessor communication 7 register. The initial register settings for our application are provided in Table 2.

**Table 2. VIC068/VAC068 Initial Register Settings**

| Address | Register | Size (Bits) | Setting |
|---|---|---|---|
| 0FFFD 0200h | VAC SLSEL0 Base | 16 | 0010h |
| 0FFFD 0300h | VAC SLSEL0 Mask | 16 | 00F0h |
| 0FFFD 0800h | VAC A24 Base | 13 | 0D10h |
| 0FFFD 2900h | VAC ID | 16 | Write to Enable VAC |
| 0FFFC 00B4h | VIC Address Modifier | 8 | 03Dh |
| 0FFFC 00C0h | VIC Slave Select 0 Control 0 | 8 | 014h |

**Figure 9. TMS320C40 – VIC/VAC Prototype VMEbus Data Bus Interface**



## Conclusion

We have developed a prototype interface between the ’C40 DSP and the Cypress VIC068/VAC068 with a minimum amount of programmable logic in the form of simple PLDs and sequencers. The result is a reconfigurable, programmable interface for A24/D32 VMEbus master/slave capability. While the initial transfer speed is low, you can improve it by increasing the sequencer’s clock rate and eliminating unnecessary states in the prototype sequencer code. You can initiate read-modify-write cycles with the existing hardware by using the ’C40 LOCK instruction group. Ultimately, the knowledge gained from this effort could be used to develop an FPGA interface that improves both speed and size. In the future, simulation models for state-of-the-art devices such as the ’C40 and VIC068/VAC068 should precede the actual hardware release, allowing early proof-of-concept with in-place CAE tools.

## Acknowledgements

The authors would like to thank James E. S. Wilkins of The MITRE Corporation for his essential contributions to the prototype effort and also to David Fuchs, Texas Instruments, Waltham, Massachusetts, for his timely support and encouragement. A special note of thanks go to the staff at Data I/O for their support of the ABEL programming language.

## References

1. *TMS320C4x User’s Guide*, Texas Instruments, 1991.
2. *VIC068 VMEbus Interface Controller Specification*, Cypress Semiconductor, 1991.
3. *VAC068 VMEbus Address Controller Specification*, Cypress Semiconductor, 1991.
4. Siy, P. F., and W. T. Ralston, “Application of the TI ’C40 in Satellite Modem Technology,” presented at the *Third Annual International Conference on Signal Processing Applications and Technology*, Boston, MA, November, 1992.
5. *IEEE Standard for a Versatile Backplane Bus: VMEbus*. New York: Wiley-Interscience, 1987.
6. W.D. Peterson. *The VMEbus Handbook*, VFEA International Trade Association, Scottsdale, AZ, 1990.
7. *MC68020 32-Bit Microprocessor User’s Manual*, Motorola, Inc., 1984.
8. Henessey, J. L., and D. A. Patterson. *Computer Architecture: A Quantitative Approach*.<br>San Mateo: Morgan Kaufmann Publishers, Inc., 1990.
9. Dewar, R. B. K., and M. Smosna. *Microprocessors: A Programmer’s View.*<br>New York: McGraw-Hill, Inc., 1990.
10. *Programmable Logic Data Book*, Texas Instruments, 1990.

## Appendix A: Address Bus Decoder — ABEL Source

| Module | Decode |
|---|---|
| Title | Global Bus Decode |
| Date | 24 March 1992 |
| Revision | 1.0 |
| Part | TIBPAL16R6-5C |
| Abel Version | 4.00 |
| Designer | Peter F. Siy |
| Company | MITRE Corp. |
| Location | Bedford, MA |
| Project | C40 I/O Board |

U12 device ’P16R6’;

```
”Inputs”
clk, reset                    pin 1,2;     ”clock, reset”
gstat0,gstat1,gstat2,gstat3   pin 3,4,5,6; ”C40 status”
ga28,ga29,ga30                pin 7,8,9;   ”C40 address”
gstrb1                        pin 12;      ”C40 strobe 1”
oute                          pin 11;      ”output enable”

”Outputs”
mrd,mwr          pin 13,14;   ”master read & write”
rrd,rwr          pin 15,16;   ”register read & write”
gprom            pin 17;      ”PROM select”

”Misc”
ga31 = 1;                  ”dummy var”

”Sets”
stat = [gstat3,gstat2,gstat1,gstat0];   ”status”
addr = [ga31,ga30,ga29,ga28];           ”ms nibble”
output = [gprom,rwr,rrd,mwr,mrd];       ”output”

H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.c = clk;
output.oe = !oute;

”Master Read”
!mrd := reset & (addr == ^hd) & (stat == [1,0,X,X]) & !gstrb1;

”Master Write”
!mwr := reset & (addr == ^hd) & ((stat == [1,1,0,1]) #
(stat == [1,1,1,0])) & !gstrb1;

”Register Read”
!rrd := reset & (addr == ^hf) & (stat == [1,0,X,X]) & !gstrb1;

”Register Write”
!rwr := reset & (addr == ^hf) & ((stat == [1,1,0,1]) #
(stat == [1,1,1,0])) & !gstrb1;

”PROM Read”
!gprom := reset & (addr == ^hc) & (stat == [1,0,X,X]) & !gstrb1;
```

## Appendix A (Continued)

```
test_vectors

([clk,reset,gstat3,gstat2,gstat1,gstat0,ga30,ga29,ga28,
gstrb1,oute] –> output)

[C,X,X,X,X,X,X,X,X,X,1] –>    Z;  ”1 test for high–z”
[C,0,X,X,X,X,X,X,X,X,0] –>   ^b11111;”2 test for reset”
[C,1,1,0,X,X,1,0,1,0,0] –>   ^b11110;”3 test for master read”
[C,1,1,1,0,1,1,0,1,0,0] –>   ^b11101;”4 test for master write”
[C,1,1,1,1,0,1,0,1,0,0] –>   ^b11101;”5 test for master write”
[C,1,1,0,X,X,1,1,1,0,0] –>   ^b11011;”6 test for register read”
[C,1,1,1,0,1,1,1,1,0,0] –>   ^b10111;”7 test for register write”
[C,1,1,1,1,0,1,1,1,0,0] –>   ^b10111;”8 test for register write”
[C,1,1,0,X,X,1,0,0,0,0] –>   ^b01111;”9 test for PROM read”
[C,1,1,0,X,X,0,0,0,0,0] –>   ^b11111;”10 test bad address”
[C,1,0,0,0,0,1,1,1,0,0] –>   ^b11111;”11 test bad status”
```

end decode

## Appendix B: Bus Control Sequencer — ABEL Source

```
module    bus_control
title        ’C40 Bus Control
Date         30 March 1992
Revision     1.0
Part         TIB82S105BC
Abel Version 4.00
Designer     Peter F. Siy
Company      MITRE Corp.
Location     Bedford, MA
Project      ’C40 I/O Card ’

U13 device ’F105’;

”Inputs”
clk, reset             pin 1,9;         ”clock, reset”
mrd,mwr,rrd,rwr,gprom  pin 8,7,6,5,4;   ”decoded cycle”
mwb,lbr                pin 3,2;         ”master/slave requests”
dedlk                  pin 27;          ”m/s deadlock”
dsack0,dsack1,lberr    pin 26,25,24;    ”cycle responses”
glock                  pin 23;          ”C40 lock”
oe                     pin 19;          ”output enable”

”Outputs”
lbg       pin 18       istype ’buffer,reg_RS’;          ”slave grant”
gbe       pin 17       istype ’buffer,reg_RS’;          ”C40 g bus enable”
soe,moe   pin 15,16    istype ’buffer,reg_RS’;          ”pls oe(s)”
grdy1     pin 13       istype ’buffer,reg_RS’;          ”C40 ready 1”

”Sets”
cycle = [gprom,rwr,rrd,mwr,mrd];     ”cycle request”
ack = [dsack1,dsack0];               ”acknowledge”
output = [grdy1,soe,moe,gbe,lbg];    ”output”

”State Description”
P4,P3,P2,P1,P0node 41,40,39,38,37 istype ’reg_RS’;
sreg = [P4,P3,P2,P1,P0];
```

## Appendix B (Continued)

```
 S0 = [0,0,0,0,0];
 S1 = [0,0,0,0,1];
 S2 = [0,0,0,1,0];
 S3 = [0,0,0,1,1];
 S4 = [0,0,1,0,0];
 S5 = [0,0,1,0,1];
 S6 = [0,0,1,1,0];
 S7 = [0,0,1,1,1];
 S8 = [0,1,0,0,0];
 S9 = [0,1,0,0,1];
 S10 = [0,1,0,1,0];
 S11 = [0,1,0,1,1];
 S12 = [0,1,1,0,0];
 S13 = [0,1,1,0,1];
 S14 = [0,1,1,1,0];
 S15 = [0,1,1,1,1];
 S16 = [1,0,0,0,0];
 S17 = [1,0,0,0,1];
 S18 = [1,0,0,1,0];
 S19 = [1,0,0,1,1];
 S20 = [1,0,1,0,0];
 S31 = [1,1,1,1,1];

”Misc”
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe;    ”set output enable”
output.CLK = clk;   ”clock the output regs”
sreg.CLK = clk;     ”and state regs”

@page
state_diagram sreg
state S0:

if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”slave disable”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else if (!mrd # !mwr & lbr) then S1;    ”master read/write”
else if (!rrd # !rwr & lbr) then S4;    ”reg read/write”
else if (!gprom & lbr) then S8;         ”EPROM read”
else if (!lbr # !dedlk) then S16 WITH   ”slave request”
 gbe.S = 1;                             ”disable global side”
 moe.S = 1;                             ”and master pls”
 ENDWITH;

else S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”slave disable”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

@page
”Master Read/Write”
```

## Appendix B (Continued)

```
state S1:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else if !dedlk & ((!mwb) # (mwb)) then S16 WITH
 moe.S = 1;
 gbe.S = 1;
 ENDWITH;

else if !mwb then S2;”wait for !mwb”

else S1;

state S2:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else if !dedlk & ((!mwb) # (mwb)) then S16 WITH;
 moe.S = 1;
 gbe.S = 1;
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S3 WITH
 grdy1.R = 1;
 ENDWITH;

else S2;

state S3:
goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
”Register Read/Write”

state S4:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else if !dsack1 then S5 WITH
 grdy1.R = 1;
 ENDWITH;

else S4;

state S5:
goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
”EPROM Read, 150ns EPROMs”
```

```
state S8:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S9;

state S9:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S10;

state S10:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;d   ”not ready”
 ENDWITH;

else goto S11;

state S11:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S12 WITH
 grdy1.R = 1;
 ENDWITH;

state S12:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;d   ”not ready”
 ENDWITH;

else goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
”Local Bus Request”
```

## Appendix B (Continued)

```
state S16:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S17 WITH
 soe.R = 1;      ”enable slave PLS”
 ENDWITH;

state S17:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S18 WITH
 lbg.R = 1;      ”finally allow slave access”
 ENDWITH;

state S18:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else if lbr then goto S19 WITH
 lbg.S = 1;      ”slave disable”
 ENDWITH;

else S18;

state S19:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S20 WITH
 soe.S =1;       ”disable slave pls”
 ENDWITH;

state S20:
if !reset then S0 WITH
 lbg.S = 1;      ”slave disable”
 gbe.R = 1;      ”enable C40 global side”
 soe.S = 1;      ”disable slave pls”
 moe.R = 1;      ”enable master pls”
 grdy1.S = 1;    ”not ready”
 ENDWITH;

else goto S0 WITH
 moe.R = 1;
 gbe.R = 1;
 ENDWITH;
```

## Appendix B (Continued)

```
@page
”Power–Up”

 state S31:
goto S0 WITH
 lbg.S = 1;      ”slave disable”
 lbg.R = 0;      ”dummy err 6099”
 gbe.R = 1;      ”enable C40 global side”
 gbe.S = 0;      ”dummy err 6099”
 soe.S = 1;      ”disable slave PLS”
 soe.R = 0;      ”dummy err 6099”
 moe.R = 1;      ”enable master pls”
 moe.S = 0;      ”dummy err 6099”
 grdy1.S = 1;    ”not ready”
 grdy1.R = 0;    ”dummy err 6099”
 ENDWITH;

@page
test_vectors

([clk,reset,gprom,rwr,rrd,mwr,mrd,lbr,mwb,
 dsack1,dsack0,dedlk,lberr,glock,oe] –>
 [sreg,grdy1,soe,moe,gbe,lbg])

[1,X,X,X,X,X,X,X,X,X,X,X,X,X,0] –> [S31,X,X,X,X,X];  ”1 power up”
[0,X,X,X,X,X,X,X,X,X,X,X,X,X,0] –> [S31,X,X,X,X,X];  ”2 power up”
[C,0,X,X,X,X,X,X,X,X,X,X,X,X,0] –> [S0,1,1,0,0,1];   ”3 reset state”
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,0] –> [S1,1,1,0,0,1];   ”4 master read”
[C,1,1,1,1,1,0,1,0,1,1,1,1,1,0] –> [S2,1,1,0,0,1];   ”5 mwb asserted”
[C,1,1,1,1,1,0,1,0,0,0,1,1,1,0] –> [S3,0,1,0,0,1];   ”6 data acked”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0,1,1,0,0,1];   ”7 ready for nxt”
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] –> [S1,1,1,0,0,1];   ”8 master write”
[C,1,1,1,1,1,0,1,0,1,1,1,1,1,0] –> [S2,1,1,0,0,1];   ”9 mwb asserted”
[C,1,1,1,1,1,0,1,0,0,0,1,1,1,0] –> [S3,0,1,0,0,1];   ”10 data acked”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0,1,1,0,0,1];   ”11 ready for nxt”
[C,1,1,1,0,1,1,1,1,1,1,1,1,1,0] –> [S4,1,1,0,0,1];   ”12 reg read”
[C,1,1,1,0,1,1,1,1,0,1,1,1,1,0] –> [S5,0,1,0,0,1];   ”13 data ackd”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0,1,1,0,0,1];   ”14 ready for nxt”
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,0] –> [S8,1,1,0,0,1];   ”15 prom read”
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,0] –> [S9,1,1,0,0,1];   ”16 prom read”
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,0] –> [S10,1,1,0,0,1];  ”17 wait”
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,0] –> [S11,1,1,0,0,1];  ”18 wait”
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,0] –> [S12,0,1,0,0,1];  ”19 wait”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 1,1,0,0,1];  ”20 ready for nxt”
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S16,1,1,1,1,1];  ”21 slave request’
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S17,1,0,1,1,1];  ”22 en slve pls”
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S18,1,0,1,1,0];  ”23 slave grant”
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S18,1,0,1,1,0];  ”24 slave aces”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S19,1,0,1,1,1];  ”25 rescend grant”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S20,1,1,1,1,1];  ”26 disable sl pls”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 1,1,0,0,1];  ”27 end sl access”
[C,1,1,1,1,1,1,1,1,1,1,0,1,1,0] –> [S16,1,1,1,1,1];  ”29 deadlock”
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S17,1,0,1,1,1];  ”30 en slve pls”
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S18,1,0,1,1,0];  ”31 slave grant”
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,0] –> [S18,1,0,1,1,0];  ”32 slave aces”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S19,1,0,1,1,1];  ”33 rescend grant”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S20,1,1,1,1,1];  ”34 disable sl pls”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 1,1,0,0,1];  ”35 end sl access”
```

end bus_control

# Appendix C: Master Cycle Generation Sequencer — ABEL Source

```
module       master
title        ’C40 Bus Control
Date         31 March 1992
Revision     1.0
Part         TIB82S105BC
Abel Version 4.00
Designer     Peter F. Siy
Company      MITRE Corp.
Location     Bedford, MA
Project      ’C40 I/O Card ’

U14 device ’F105’;

”Inputs”
clk, reset             pin 1,9;         ”clock, reset”
mrd,mwr,rrd,rwr,gprom  pin 8,7,6,5,4;   ”decoded cycle”
mwb,lbr                pin 3,2;         ”master/slave requests”
dedlk                  pin 27;          ”m/s deadlock”
dsack0,dsack1,lberr    pin 26,25,24;    ”cycle responses”
glock                  pin 23;          ”C40 lock”
oe                     pin 19;          ”output enable”

”Outputs”
pas   pin 18 istype ’buffer,reg_RS’;    ”68K address strobe”
ds    pin 17 istype ’buffer,reg_RS’;    ”68K data strobe”
rw    pin 16 istype ’buffer,reg_RS’;    ”68K read/write bar”
rmc   pin 15 istype ’buffer,reg_RS’;    ”68K read–mod–write”
siz0  pin 13 istype ’buffer,reg_RS’;    ”68K size 0”
siz1  pin 12 istype ’buffer,reg_RS’;    ”68K size 1”
fc1   pin 11 istype ’buffer,reg_RS’;    ”68K function 1”
fc2   pin 10 istype ’buffer,reg_RS’;    ”68K function 0”

”Sets”
cycle = [gprom,rwr,rrd,mwr,mrd];           ”cycle request”
ack = [dsack1,dsack0];                     ”acknowledge”
output = [pas,ds,rw,rmc,siz0,siz1,fc1,fc2];”68K ouputs”

”State Description”
P4,P3,P2,P1,P0node 41,40,39,38,37 istype ’reg_RS’;
sreg = [P4,P3,P2,P1,P0];
 S0 = [0,0,0,0,0];
 S1 = [0,0,0,0,1];
 S2 = [0,0,0,1,0];
 S3 = [0,0,0,1,1];
 S4 = [0,0,1,0,0];
 S5 = [0,0,1,0,1];
 S6 = [0,0,1,1,0];
 S7 = [0,0,1,1,1];
 S8 = [0,1,0,0,0];
 S9 = [0,1,0,0,1];
 S10 = [0,1,0,1,0];
 S11 = [0,1,0,1,1];
 S12 = [0,1,1,0,0];
 S13 = [0,1,1,0,1];
 S14 = [0,1,1,1,0];
 S15 = [0,1,1,1,1];
 S16 = [1,0,0,0,0];
 S17 = [1,0,0,0,1];
 S18 = [1,0,0,1,0];
 S19 = [1,0,0,1,1];
```

## Appendix C (Continued)

```
 S20 = [1,0,1,0,0];
 S21 = [1,0,1,0,1];
 S22 = [1,0,1,1,0];
 S23 = [1,0,1,1,1];
 S24 = [1,1,0,0,0];
 S25 = [1,1,0,0,1];
 S26 = [1,1,0,1,0];
 S27 = [1,1,0,1,1];
 S28 = [1,1,1,0,0];
 S29 = [1,1,1,0,1];
 S30 = [1,1,1,1,0];
 S31 = [1,1,1,1,1];

”Misc”
rwmemnode 42 istype ’reg_RS’; ”r/w flag”
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe;              ”set output enable”
output.CLK = clk;             ”clock the output regs”
sreg.CLK = clk;               ”and state regs”
rwmem.CLK = clk;              ”and r/w store”

@page

state_diagram sreg
state S0:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;                   ”no strobe”
 ds.S= 1;                     ”no strobe”
 rw.S = 1;                    ”read”
 rwmem.S = 1;                 ”flag for mem”
 rmc.S = 1;                   ”no rmc
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for supervisory”
 fc2.S = 1;  ”data access”
 ENDWITH;

else if (!mrd & !rwmem & lbr) then S1 WITH    ”master read”
 rw.S = 1;                                    ”assert read/write”
 rwmem.S =1;
 ENDWITH;

else if (!mrd & rwmem & lbr) then S2 WITH     ”master read”
 pas.R = 1;      ”assert pas”
 ds.R = 1;       ”and ds”
 ENDWITH;

else if (!mwr & rwmem & lbr) then S8 WITH     ”master write”
 rw.R = 1;       ”assert r/w”
 rwmem.R = 1;
 ENDWITH;

else if (!mwr & !rwmem & lbr) then S9 WITH    ”master write”
 pas.R = 1;      ”assert pas only”
 ENDWITH;

else if (!rrd & !rwmem & lbr) then S16 WITH   ”reg read”
 rw.S = 1;       ”assert r/w”
 rwmem.S = 1;
 ENDWITH;

else if (!rrd & rwmem & lbr) then S17 WITH ”reg read”
 pas.R = 1;      ”assert pas”
 ds.R = 1;       ”and ds”
 ENDWITH;
```

```
else if (!rwr & rwmem & lbr) then S24 WITH ”reg write”
 rw.R = 1;
 rwmem.R = 1;
 ENDWITH;

else if (!rwr & !rwmem & lbr) then S25 WITH
 pas.R = 1;      ”assert pas only”
 ENDWITH;

else S0 WITH
 pas.S = 1;      ”no strobe”
 ds.S= 1;        ”no strobe”
 rw.S = 1;       ”read”
 rwmem.S = 1;    ”flag for mem”
 rmc.S = 1;      ”no rmc”
 siz0.R = 1;     ”set for”
 siz1.R = 1;     ”32–bit xfers”
 fc1.R = 1;      ”set for supervisory”
 fc2.S = 1;      ”data access”
 ENDWITH;

@page
”Master Read”

state S1:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      ”no strobe”
 ds.S= 1;        ”no strobe”
 rw.S = 1;       ”read”
 rwmem.S = 1;    ”flag for mem”
 rmc.S = 1;      ”no rmc”
 siz0.R = 1;     ”set for”
 siz1.R = 1;     ”32–bit xfers”
 fc1.R = 1;      ”set for super”
 fc2.S = 1;      ”data access”
 ENDWITH;

else S2 WITH
 pas.R = 1;
 ds.R = 1;
 ENDWITH;

state S2:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      ”no strobe”
 ds.S= 1;        ”no strobe”
 rw.S = 1;       ”read”
 rwmem.S = 1;    ”flag for mem”
 rmc.S = 1;      ”no rmc”
 siz0.R = 1;     ”set for”
 siz1.R = 1;     ”32–bit xfers”
 fc1.R = 1;      ”set for super”
 fc2.S = 1;      ”data access”
 ENDWITH;

else if !mwb then S3;      ”wait for !mwb”

else S2;

state S3:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      ”no strobe”
 ds.S= 1;        ”no strobe”
 rw.S = 1;       ”read”
 rwmem.S = 1;    ”flag for mem”
```

## Appendix C (Continued)

```
rmc.S = 1;       ”no rmc”
 siz0.R = 1;     ”set for”
 siz1.R = 1;     ”32–bit xfers”
 fc1.R = 1;      ”set for supervisory”
 fc2.S = 1;      ”data access”
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S4 WITH
 grdy1.R = 1
 ENDWITH;

else S3;

state S4:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
”Master Write”

state S8:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      ”no strobe”
 ds.S= 1;        ”no strobe”
 rw.S = 1;       ”read”
 rwmem.S = 1;
 rmc.S = 1;      ”no rmc”
 siz0.R = 1;     ”set for”
 siz1.R = 1;     ”32–bit xfers”
 fc1.R = 1;      ”set for supervisory”
 fc2.S = 1;      ”data access”
 ENDWITH;

else S9 WITH
 pas.R = 1;
 ENDWITH;

state S9:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      ”no strobe”
 ds.S= 1;        ”no strobe”
 rw.S = 1;       ”read”
 rwmem.S = 1;
 rmc.S = 1;      ”no rmc”
 siz0.R = 1;     ”set for”
 siz1.R = 1;     ”32–bit xfers”
 fc1.R = 1;      ”set for supervisory”
 fc2.S = 1;      ”data access”
 ENDWITH;

else S10 WITH
 ds.r = 1;
 ENDWITH;
```

```
state S10:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     ”no strobe”
 ds.S= 1;       ”no strobe”
 rw.S = 1;      ”read”
 rwmem.S = 1;
 rmc.S = 1;     ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for super”
 fc2.S = 1;  ”data access”
 ENDWITH;

else if !mwb then S11;

else S10;

state S11:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;  ”no strobe”
 ds.S= 1;    ”no strobe”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rmc.S = 1;  ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for supervisory”
 fc2.S = 1;  ”data access”
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S12;

else S11;

state S12:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
”Register Read”

state S16:
if !reset then S0 WITH
 pas.S = 1;  ”no strobe”
 ds.S= 1;    ”no strobe”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rmc.S = 1;  ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for super”
 fc2.S = 1;  ”data access”
 ENDWITH;

else S17 WITH
 pas.R = 1;
 ds.R =1;
 ENDWITH;
```

## Appendix C (Continued)

```
state S17:
if !reset then S0 WITH
 pas.S = 1;  ”no strobe”
 ds.S= 1;    ”no strobe”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rmc.S = 1;  ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for super”
 fc2.S = 1;  ”data access”
 ENDWITH;

else if !dsack1 then S18 WITH
 grdy1.R = 1
 ENDWITH;

else S17;

state S18:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
”Register Write”

state S24:
if !reset then S0 WITH
 pas.S = 1;  ”no strobe”
 ds.S= 1;    ”no strobe”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rmc.S = 1;  ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for supervisory”
 fc2.S = 1;  ”data access”
ENDWITH;

else S25 WITH
 pas.R = 1;
 ENDWITH;

state S25:
if !reset then S0 WITH
 pas.S = 1;  ”no strobe”
 ds.S= 1;    ”no strobe”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rmc.S = 1;  ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for supervisory”
 fc2.S = 1;  ”data access”
 ENDWITH;

else S26 WITH
 ds.r = 1;
 ENDWITH;
```

## Appendix C (Continued)

```
state S26:
if !reset then S0 WITH
 pas.S = 1;  ”no strobe”
 ds.S= 1;    ”no strobe”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rmc.S = 1;  ”no rmc”
 siz0.R = 1; ”set for”
 siz1.R = 1; ”32–bit xfers”
 fc1.R = 1;  ”set for supervisory”
 fc2.S = 1;  ”data access”
 ENDWITH;
else if !dsack1 then S27;

else S26;
state S27:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;
@page
”Power–Up”
state S31:
goto S0 WITH
 pas.S = 1;  ”no strobe”
 pas.R = 0;  ”error 6099 fix”
 ds.S= 1;    ”no strobe”
 ds.R= 0;    ”error 6099 fix”
 rw.S = 1;   ”read”
 rwmem.S = 1;
 rw.R = 0;   ”error 6099 fix”
 rmc.S = 1;  ”no rmc”
 rmc.R = 0;  ”error 6099 fix”
 siz0.R = 1; ”set for”
 siz0.S = 0; ”error 6099 fix”
 siz1.R = 1; ”32–bit xfers”
 siz1.S = 0; ”error 6099 fix”
 fc1.R = 1;  ”set for supervisory”
 fc1.S = 0;  ”error 6099 fix”
 fc2.S = 1;  ”data access”
 fc2.R = 0;  ”error 6099 fix”
ENDWITH;
@page
test_vectors
([clk,reset,gprom,rwr,rrd,mwr,mrd,lbr,mwb,
 dsack1,dsack0,dedlk,lberr,glock,oe] –>
[sreg,rwmem,fc2,fc1,siz1,siz0,rmc,rw,ds,pas])
[1,X,X,X,X,X,X,X,X,X,X,X,X,X,0] –> [S31,X,X,X,X,X,X,X,X,X]; ”1 power up”
[0,X,X,X,X,X,X,X,X,X,X,X,X,X,0] –> [S31,X,X,X,X,X,X,X,X,X]; ”2 power up”
[C,0,X,X,X,X,X,X,X,X,X,X,X,X,0] –> [S0, 1,1,0,0,0,1,1,1,1]; ”3 reset state”
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,0] –> [S2, 1,1,0,0,0,1,1,0,0]; ”4 master read”
[C,1,1,1,1,1,0,1,0,1,1,1,1,1,0] –> [S3, 1,1,0,0,0,1,1,0,0]; ”5 mwb
asserted”
[C,1,1,1,1,1,0,1,0,0,0,1,1,1,0] –> [S4, 1,1,0,0,0,1,1,0,0]; ”6 data acked”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 1,1,0,0,0,1,1,1,1]; ”7 ready for
nxt”
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] –> [S8, 0,1,0,0,0,1,0,1,1]; ”8 master
```

## Appendix C (Continued)

```
write”
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] –> [S9, 0,1,0,0,0,1,0,1,0]; ”9 assert pas”
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] –> [S10,0,1,0,0,0,1,0,0,0]; ”10 assert ds”
[C,1,1,1,1,0,1,1,0,1,1,1,1,1,0] –> [S11,0,1,0,0,0,1,0,0,0]; ”11 mwb”
[C,1,1,1,1,0,1,1,0,0,0,1,1,1,0] –> [S12,0,1,0,0,0,1,0,0,0]; ”12 data ackd”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 0,1,0,0,0,1,0,1,1]; ”13 ready for
next”
[C,1,1,1,0,1,1,1,1,1,1,1,1,1,0] –> [S16,1,1,0,0,0,1,1,1,1]; ”14 reg read”
[C,1,1,1,0,1,1,1,1,1,1,1,1,1,0] –> [S17,1,1,0,0,0,1,1,0,0]; ”15 assert
strobes”
[C,1,1,1,0,1,1,1,1,0,1,1,1,1,0] –> [S18,1,1,0,0,0,1,1,0,0]; ”16 data ackd”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 1,1,0,0,0,1,1,1,1]; ”17 ready for
nxt”
[C,1,1,0,1,1,1,1,1,1,1,1,1,1,0] –> [S24,0,1,0,0,0,1,0,1,1]; ”18 reg write”
[C,1,1,0,1,1,1,1,1,1,1,1,1,1,0] –> [S25,0,1,0,0,0,1,0,1,0]; ”19 assert pas”
[C,1,1,0,1,1,1,1,1,1,1,1,1,1,0] –> [S26,0,1,0,0,0,1,0,0,0]; ”20 assert ds”
[C,1,1,0,1,1,1,1,1,0,1,1,1,1,0] –> [S27,0,1,0,0,0,1,0,0,0]; ”21 data ackd”
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] –> [S0, 0,1,0,0,0,1,0,1,1]; ”22 ready for
next”
end master
```

APPENDIX D

SLAVE CYCLE GENERATION SEQUENCER – ABEL SOURCE

```
module    slave
title     ’C40 Bus Control
Date         2 April 1992
Revision     1.0
Part         TIB82S105BC
Abel Version 4.00
Designer     Peter F. Siy
Company      MITRE Corp.
Location     Bedford, MA
Project      ’C40 I/O Card ’

U15 device ’F105’;

”Inputs”
clk, reset   pin 1,9;     ”clock, reset”
pas,ds       pin 8,7;     ”address,data strobe”
rw,rmc       pin 6,5;     ”read/write strobes”
siz0,siz1    pin 4,3;     ”bus sizing”
fc0,fc1,fc2  pin 2,27,26; ”function codes”
lbg          pin 25;      ”local bus grant”
oe           pin 19;      ”output enable”

”Outputs”
dsack0    pin 18 istype ’buffer,reg_RS’;   ”data ack 0”
dsack1    pin 17 istype ’buffer,reg_RS’;   ”data ack 1”
lberr     pin 16 istype ’buffer,reg_RS’;   ”bus error”
gstrb0    pin 15 istype ’buffer,reg_RS’;   ”C40 mem strobe”
grw0      pin 13 istype ’buffer,reg_RS’;   ”C40 read/write”

”Sets”
size = [siz1,siz0];    ”size”
func = [fc2,fc1,fc0];  ”function”
output = [grw0,gstrb0,lberr,dsack1,dsack0];

”State Description”
P3,P2,P1,P0  node 40,39,38,37    istype ’reg_RS’;
sreg = [P3,P2,P1,P0];
 S0 = [0,0,0,0];
 S1 = [0,0,0,1];
 S2 = [0,0,1,0];
 S3 = [0,0,1,1];
 S4 = [0,1,0,0];
 S5 = [0,1,0,1];
 S6 = [0,1,1,0];
 S7 = [0,1,1,1];
 S8 = [1,0,0,0];
 S9 = [1,0,0,1];
 S10 = [1,0,1,0];
 S11 = [1,0,1,1];
 S12 = [1,1,0,0];
 S13 = [1,1,0,1];
 S14 = [1,1,1,0];
 S15 = [1,1,1,1];
”Misc”
rwmemnode 42 istype ’reg_RS’;     ”r/w flag”
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;
equations
output.OE = !oe;                  ”set output enable”
output.CLK = clk;                 ”clock the output regs”
sreg.CLK = clk;                   ”and state regs”
rwmem.CLK = clk;                  ”and r/w store”
@page
state_diagram sreg
state S0:
if (!reset) then S0 WITH
 dsack0.S = 1;          ”deassert”
 dsack1.S = 1;   ”all”
 lberr.S = 1;    ”strobes”
 gstrb0.S = 1;   ”deassert C40”
 grw0.R = 1;     ”strobe, read”
 rwmem.S = 1;    ”set to read”
 ENDWITH;
else if (!lbg) then S1;
else S0 WITH
 dsack0.S = 1;   ”deassert”
 dsack1.S = 1;   ”all”
 lberr.S = 1;    ”strobes”
 gstrb0.S = 1;   ”deassert C40”
 grw0.R = 1;     ”strobe, read”
 rwmem.S = 1;    ”set to read”
 ENDWITH;
@page
”Sort Slave Request”
state S1:
”Reset”
if (!reset) then S0 WITH
 dsack0.S = 1;   ”deassert”
 dsack1.S = 1;   ”all”
 lberr.S = 1;    ”strobes”
 gstrb0.S = 1;   ”deassert C40”
 grw0.R = 1;     ”strobe, read”
 rwmem.S = 1;    ”set to read”
 ENDWITH;
”32–Bit Read”
else if (!pas & !ds & rw & !rwmem & !siz0 & !siz1) then S2 WITH
 grw0.S = 1;
 rwmem.S = 1;
 ENDWITH;
```

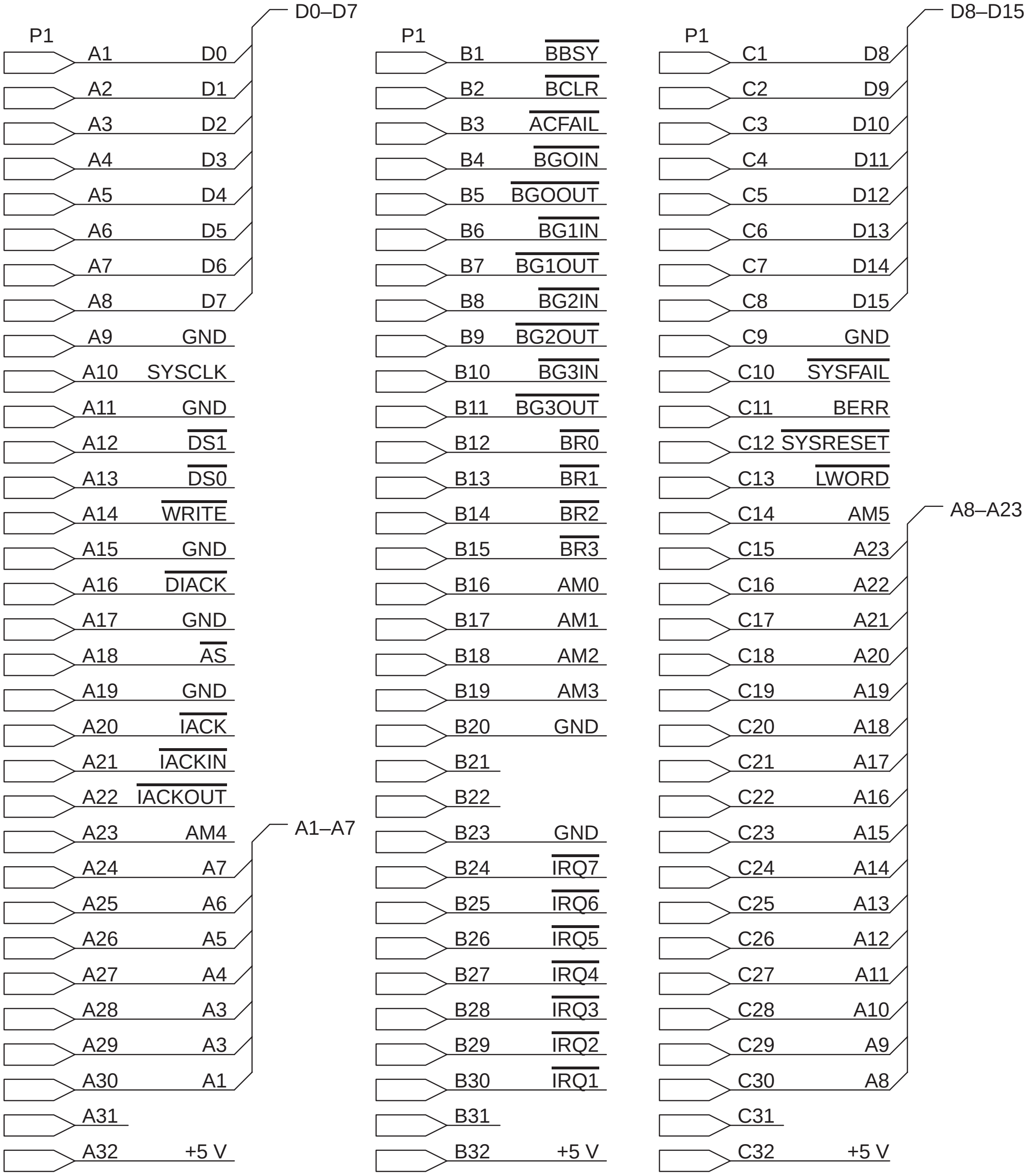
## Appendix C (Continued)

```
else if (!pas & !ds & rw & rwmem & !siz0 & !siz1) then S3 WITH
 gstrb0.R = 1;
 ENDWITH;

”32–Bit Write”
else if (!pas & !ds & !rw & rwmem & !siz0 & !siz1) then S2 WITH
 grw0.R = 1;
 rwmem.R = 1;
 ENDWITH;

else if (!pas & !ds & !rw & !rwmem & !siz0 & !siz1) then S3 WITH
 gstrb0.R = 1;
 ENDWITH;

”Illegal Access (non–32 bit access)”
 else if (!pas & !ds & (rw # !rw) & (siz0 # siz1)) then S9 WITH
 lberr.R = 1;
 ENDWITH;

else S1;

@page
”32–Bit Read/Write”

state S2:
goto S3 WITH
 gstrb0.R = 1;
 ENDWITH;

state S3:
goto S4 WITH
 dsack0.R = 1;
 dsack1.R = 1;
 ENDWITH;

state S4:
if pas then S0 WITH
 dsack0.S = 1;
 dsack1.S = 1;
 gstrb0.S = 1;
 ENDWITH;

else S4;

@page
”Illegal Access”

state S9:
if pas then S0 WITH
 lberr.S = 1;
 ENDWITH

@page
”Power–Up”

state S15:
goto S0 WITH
 dsack0.S = 1;   ”no ack”
 dsack0.R = 0;   ”error 6099 fix”
 dsack1.S = 1;   ”no ack”
 dsack1.R = 0;   ”error 6099 fix”
 rwmem.S = 1;    ”r/w mem”
 rwmem.R = 0;    ”error 6099 fix”
 lberr.S = 1;    ”no bus error”
 lberr.R = 0;    ”error 6099 fix”
 gstrb0.S = 1;   ”no strobe”
 grw0.S = 1;     ”read”
 ENDWITH;
```

```
@page
test_vectors

([clk,reset,pas,ds,rw,rmc,siz0,siz1,fc0,fc1,fc2,lbg,oe] –>
[sreg,rwmem,dsack0,dsack1,lberr,gstrb0,grw0])

[1,X,X,X,X,X,X,X,X,X,X,X,0] –> [S15,X,X,X,X,X,X]; ”1 power up”
[0,X,X,X,X,X,X,X,X,X,X,X,0] –> [S15,X,X,X,X,X,X]; ”2 power up”
[C,0,X,X,X,X,X,X,X,X,X,X,0] –> [S0, 1,1,1,1,1,1]; ”3 reset state”
[C,1,1,1,1,1,1,1,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”4 slave read,lbg”
[C,1,0,1,1,1,0,0,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”5 pas asserted”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S3, 1,1,1,1,0,1]; ”6 and ds, strobe”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S4, 1,0,0,1,0,1]; ”7 ack”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S4, 1,0,0,1,0,1]; ”8 wait for pas release”
[C,1,1,1,1,1,1,1,X,X,X,1,0] –> [S0, 1,1,1,1,1,1]; ”9 done, release gstrb”
[C,1,1,1,0,1,1,1,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”10 slave write,lbg”
[C,1,0,1,0,1,0,0,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”11 pas asserted”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S2, 0,1,1,1,1,0]; ”12 and ds”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S3, 0,1,1,1,0,0]; ”13 assert strobe”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S4, 0,0,0,1,0,0]; ”14 ack”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S4, 0,0,0,1,0,0]; ”15 wait for pas release”
[C,1,1,1,1,1,1,1,X,X,X,1,0] –> [S0, 0,1,1,1,1,0]; ”16 done,relase gstrb”
[C,1,1,1,1,1,1,1,X,X,X,0,0] –> [S1, 0,1,1,1,1,0]; ”17 slave read,lbg”
[C,1,0,1,1,1,0,0,X,X,X,0,0] –> [S1, 0,1,1,1,1,0]; ”18 pas asserted”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S2, 1,1,1,1,1,1]; ”19 and ds, r/w asserted”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S3, 1,1,1,1,0,1]; ”20 and strobe”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S4, 1,0,0,1,0,1]; ”21 ack”
[C,1,0,0,1,1,0,0,X,X,X,0,0] –> [S4, 1,0,0,1,0,1]; ”22 wait for pas release”
[C,1,1,1,1,1,1,1,X,X,X,1,0] –> [S0, 1,1,1,1,1,1]; ”23 done,relase gstrb”
[C,1,1,1,1,1,1,1,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”24 bad access,lbg”
[C,1,0,1,1,1,0,1,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”25 pas asserted”
[C,1,0,0,1,1,0,1,X,X,X,0,0] –> [S9, 1,1,1,0,1,1]; ”26 and ds, error”
[C,1,0,0,1,1,0,1,X,X,X,0,0] –> [S9, 1,1,1,0,1,1]; ”27 wait for pas release”
[C,1,1,1,1,1,1,1,X,X,X,1,0] –> [S0, 1,1,1,1,1,1]; ”28 done, release lberr”
[C,1,1,1,0,1,1,1,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”29 slave write,lbg”
[C,1,0,1,0,1,0,0,X,X,X,0,0] –> [S1, 1,1,1,1,1,1]; ”30 pas asserted”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S2, 0,1,1,1,1,0]; ”31 and ds”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S3, 0,1,1,1,0,0]; ”32 assert strobe”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S4, 0,0,0,1,0,0]; ”33 ack”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S4, 0,0,0,1,0,0]; ”34 wait for pas release”
[C,1,1,1,1,1,1,1,X,X,X,1,0] –> [S0, 0,1,1,1,1,0]; ”35 done,relase gstrb”
[C,1,1,1,0,1,1,1,X,X,X,0,0] –> [S1, 0,1,1,1,1,0]; ”36 slave write,lbg”
[C,1,0,1,0,1,0,0,X,X,X,0,0] –> [S1, 0,1,1,1,1,0]; ”37 pas asserted”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S3, 0,1,1,1,0,0]; ”38 and ds,assert strobe”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S4, 0,0,0,1,0,0]; ”39 ack”
[C,1,0,0,0,1,0,0,X,X,X,0,0] –> [S4, 0,0,0,1,0,0]; ”40 wait for pas release”
[C,1,1,1,1,1,1,1,X,X,X,1,0] –> [S0, 0,1,1,1,1,0]; ”41 done,relase gstrb”

end slave
```

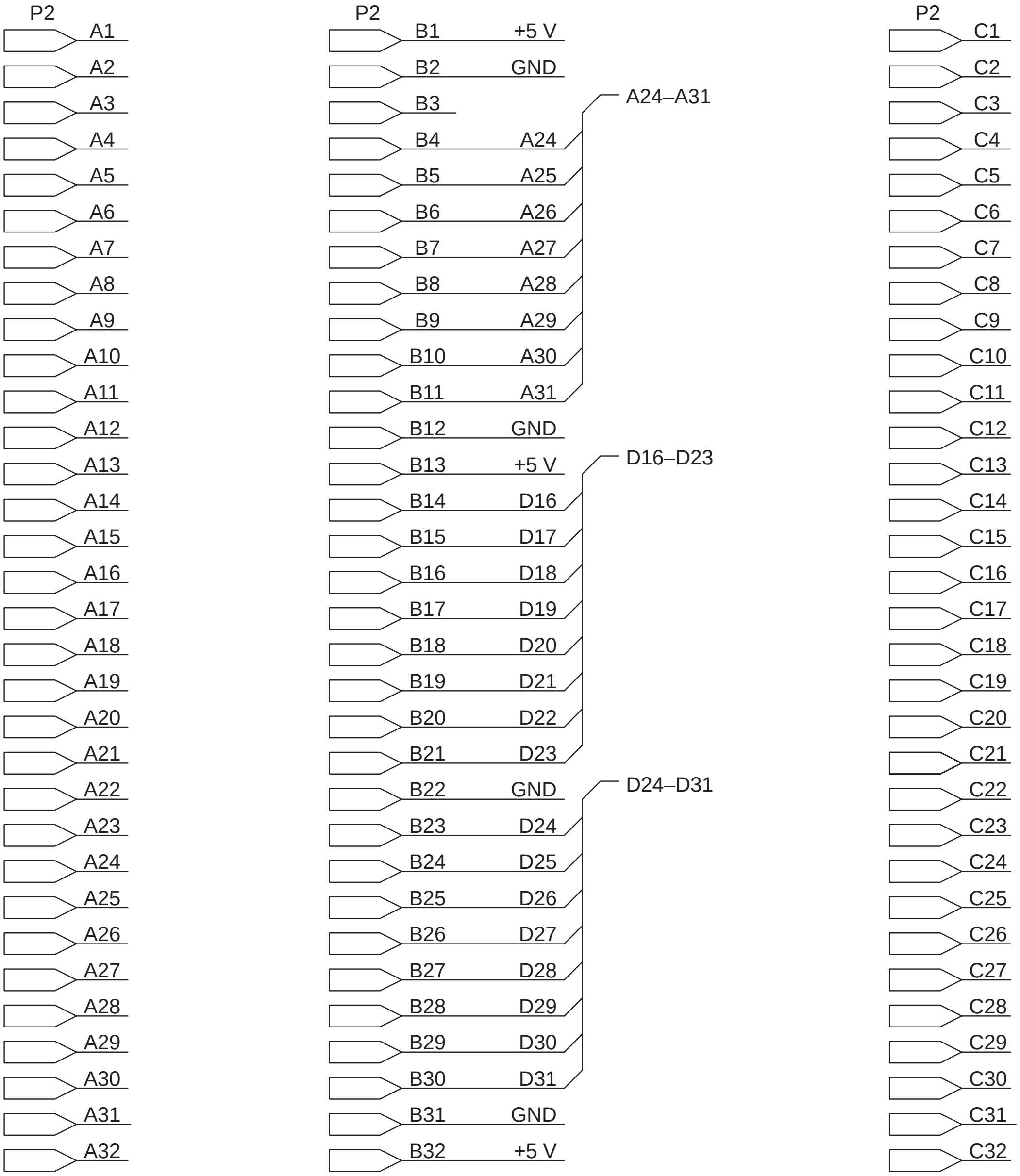
# Appendix D: Schematics

Figure 10. TMS320C40 – VIC/VAC Prototype VMEbus P1 Connector



## Appendix D: Schematics (Continued)

### Figure 11.  TMS320C40 – VIC/VAC Protototype VMEbus P2 Connector



# AMELIA — An A/D-D/A Interface to the TMS320C40 Global Bus

Steve R. France
Loughborough Sound Images Ltd.

## Introduction

AMELIA is Loughborough Sound Images' (LSI) Analog ModulE Link Interface Adapter. It is used on a number of LSI's development boards, including those that use the TMS320C40 DSP from Texas Instruments. It allows you to build a modular interface system that can be upgraded as technology progresses. In addition, AMELIA uses none of the 'C40's parallel communication links, so the processing system maintains its flexibility.

This application note describes the broad functionality of AMELIA, how it integrates analog and digital operations, and how it enhances the interface options open to the system builder.

## Analog Conversion — A Brief Overview

The large number of 'C40 systems that connect to the outside world via an application-specific analog interface require a range of solutions. In these solutions, it is crucial that data be presented to the processor accurately to maintain its value.

A number of analog interface devices have been available for some time and the comparison has been made primarily in terms of conversion bandwidth, or how fast the converter can operate. The tradeoff has been in the conversion performance, or accuracy of the device. Generally, the wider the bandwidth, either the resolution of the converter (number of bits) or the signal-to-noise ratio is reduced. Thus, it is common to find converters with 8-bit resolutions that operate in the megahertz sampling range, but 16-bit devices are limited to hundreds of kilohertz.

Today, there is a much greater concern for accuracy of conversion. Ultimately, the application determines the exact performance requirements: from 8-bit servo control and 10- to 14-bit requirements of telecomms and radar, through the growing 16-bit arena, to the digital audio applications requiring 24 bits of resolution. Consequently, users are becoming more selective, and a single general-purpose device cannot fulfill all requirements.

## Modular Interface Design Techniques

A modular approach to interface design makes possible a range of interface solutions to meet the requirements of all the varied applications. A modular design also protects earlier investments when you update your system.

Analog conversion techniques are progressing at a fast pace in the semiconductor industry; this means that traditional methods of design in which the converter is mounted on the same PCB as the DSP are somewhat limited. If your application would benefit from the improved performance that a new device can provide but your system has a traditional design, the whole system must be replaced, including DSP technology that may still be current. With the modular approach, the new module simply replaces the old, maintaining your investment in the 'C40 processing system.

LSI's modular technique attaches a separate PCB directly to LSI's 'C40 boards on both the PC and VME chassis; the PCB remains within the single height constraints of those systems. This approach realizes other benefits in the performance of the system. A single-board system has inherent flaws in the way that digital noise easily transmits into sensitive analog components and imposes an upper performance limit that is less than optimal. When the analog section is removed from the digital PCB, system performance figures approaching those of the converters are achieved. This is attributed to better isolation between the two sets of components.

Analog component layout also accounts for performance differences. With a single-board system, cost constraints on total system design prevent repeated changes on the circuit board. Consequently, a number of these systems rely on analog interfaces that exhibit suboptimal performance. Having a separate board for the analog interface affords two advantages: the layout problem is eased, and a new analog design can proceed independently of the digital system, allowing a more rapid time to market with a new design.

A modular interface design technique is now in place at LSI. To illustrate the performance benefits with the ’C40, typical measured figures are 90 dB signal to noise and distortion for a delta-sigma converter, and 84 dB for a 200-kHz successive approximation device. These figures were measured with the complete assembled system inserted into the host platform.

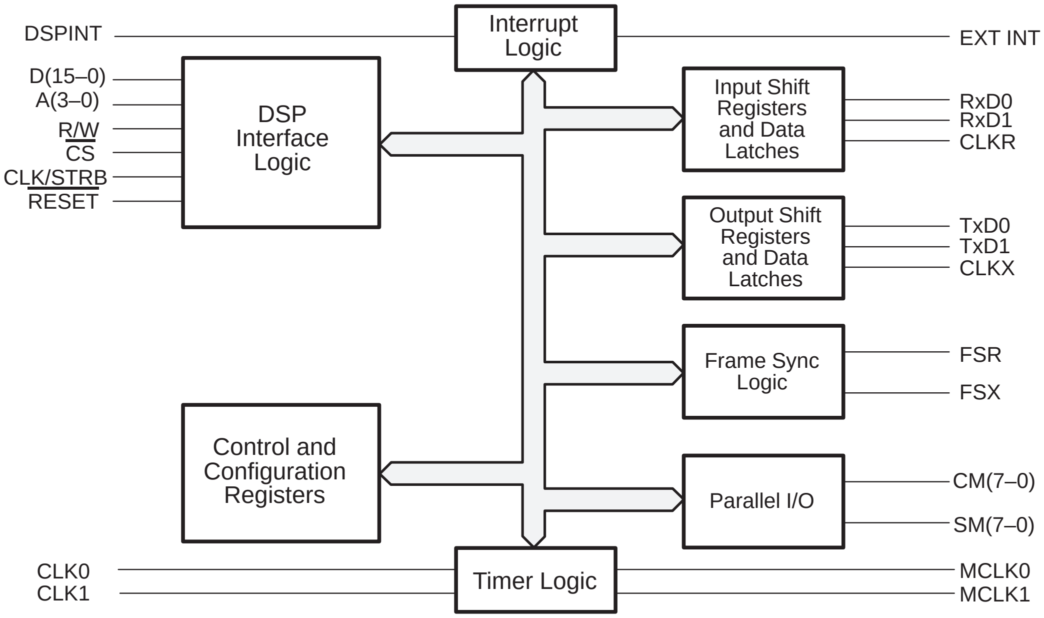
## AMELIA

To develop a modular system, it was first necessary to find a common way in which to interface a wide range of conversion modules to an equally wide range of DSPs. The paramount consideration for LSI was flexibility so that all functionality of the DSP could be available; this precluded the use of serial ports on ’C3x and ’C5x devices. Additional design objectives were to avoid consuming a ’C40 communication link and to consider the differing connection methods and protocol requirements. The best solution was to memory map the analog interface.

Memory mapping solves the problems of interfacing to different TI DSP families and does not remove any device functionality. AMELIA serves as the link between the DSP and the interface device, absorbing any interconnection differences.

The basic functional blocks of AMELIA are shown in Figure 1. The device is produced in a 2000-gate FPGA and is packaged in a 68-pin PLCC. The pin assignment is shown in Table 1. AMELIA provides two synchronous serial ports for the ’C40. Sixteen-bit data words are transmitted and received under control of the sophisticated frame sync logic. The flexibility of that logic is ideally suited for connection to a wide range of A/D, D/A, and other serial communication devices. The ’C40 and serial sides of AMELIA operate asynchronously to each other. They can handle very slow peripherals without slowing the ’C40; the DSP is interrupted only when the data is ready for processing.

**Figure 1. Block Diagram**



## Table 1. Pin Assignments

| Pin Name | Pin Number | State | Description |
|---|---|---|---|
| **Analog Control/Status** | | | |
| CM0<br>CM1<br>CM2<br>CM3<br>CM4<br>CM5<br>CM6<br>CM7 | 34<br>33<br>31<br>30<br>29<br>28<br>27<br>26 | O | Control port to the analog daughter module. |
| SM0<br>SM1<br>SM2<br>SM3 | 39<br>37<br>36<br>35 | I | Status port from the analog daughter module. |
| **Analog Serial Bus Interface** | | | |
| CLKR | 52 | I | Serial receive clock. This is the serial shift clock for both receive channels. |
| CLKX | 47 | I | Serial transmit clock. This is the serial shift clock for both transmit channels. |
| FSR | 53 | I | Frame synchronization pulse for the serial receive channels. |
| FSX | 48 | I/O | Frame synchronization pulse for the serial transmit channels. |
| RXD0<br>RXD1 | 50<br>51 | I | Data receive for channel 0/1. Serial data for channel 0/1 is received on this pin. |
| TXD0<br>TXD1 | 45<br>46 | O | Data transmit for channel 0/1. Serial data for channel 0/1 is transmitted on this pin. |
| **DSP Parallel Bus Interface** | | | |
| A0<br>A1<br>A2<br>A3 | 11<br>12<br>13<br>16 | I | Four-bit address port. |
| D0<br>D1<br>D2<br>D3<br>D4<br>D5<br>D6<br>D7<br>D8<br>D9<br>D10<br>D11<br>D12<br>D13<br>D14<br>D15 | 61<br>62<br>63<br>64<br>65<br>67<br>68<br>1<br>2<br>3<br>5<br>6<br>7<br>8<br>9<br>10 | I/O/Z | Sixteen-bit data port lines. |
| $\overline{\text{CS}}$ | 18 | I | Chip select. When an access is performed, this signal must be low. |
| CLK/<br>STRB | 19 | I | Clock or strobe signal. This signal generates early write strobes for write cycles if the data hold time is insufficient for the device. |

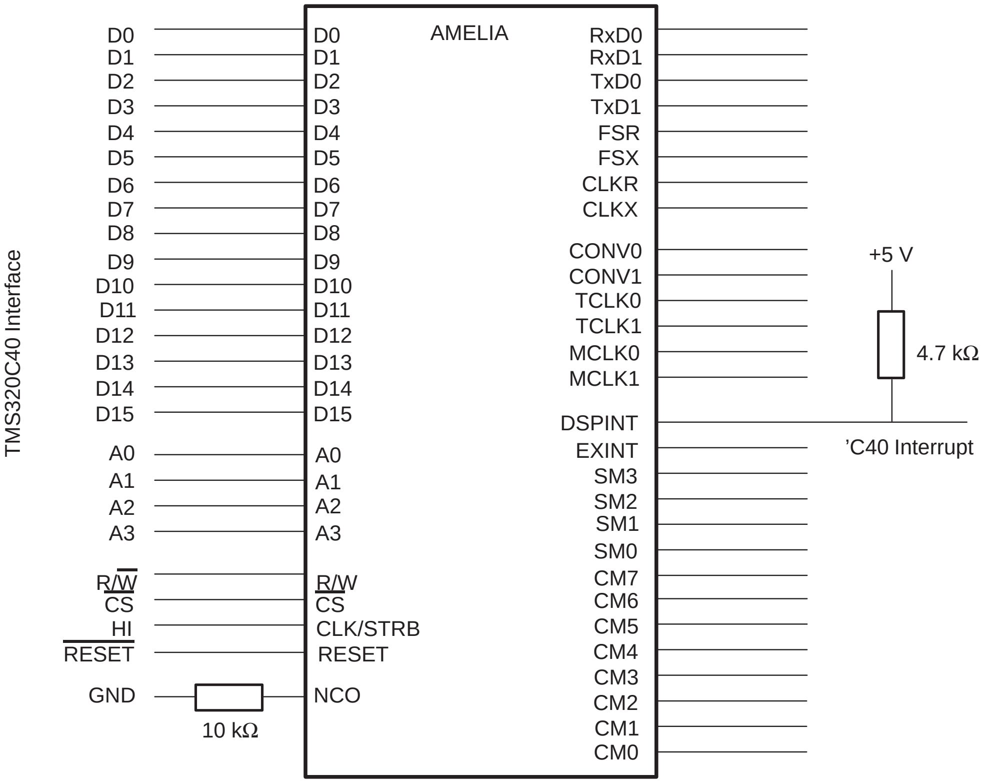
### Table 1. Pin Assignments (Continued)

| Pin Name | Pin Number | State | Description |
|---|---|---|---|
| $\overline{\text{RESET}}$ | 20 | I | Reset. When this pin is low, the device is placed in the reset condition. |
| R/$\overline{\text{W}}$ | 17 | I | Read/write signal When a read is performed, this signal must be held high. When a write is performed, this signal is low. |
| **Interrupt** | | | |
| DSPINT | 22 | O | Open-collector interrupt on the DSP. Interrupts generated by the serial interface or by an external interrupt source can be output to the DSP on this pin. |
| EXINT | 44 | I | External interrupt. Interrupts generated on this pin can be passed through to the DSP. |
| **Miscellaneous** | | | |
| NC0 | 54 | Z | Unused pin. This pin must be terminated to ground via a 10-kΩ resistor. |
| NC1<br>NC2<br>NC3<br>NC4<br>NC5<br>NC6 | 59<br>56<br>57<br>58<br>60<br>40 | Z | Unused pins. These pins must be left unconnected. |
| **Power Supply** | | | |
| GND1<br>GND2<br>GND3<br>GND4<br>GND5 | 14<br>15<br>32<br>49<br>66 | GND | Ground pins. |
| $V_{CC1}$<br>$V_{CC2}$<br>$V_{CC3}$<br>$V_{CC4}$<br>$V_{CC5}$ | 4<br>21<br>25<br>38<br>55 | $V_{CC}$ | +5-Volt supply pins. |
| **Timer** | | | |
| CONV1 | 41 | O/Z | Conversion pulse generator 1 output. This pin outputs pulses generated by timer 1 as programmed from the DSP interface. |
| MCLK0<br>MCLK1 | 42<br>43 | I/O | Master clock. When configured as an input, this clock can drive either of the two internal timers. As an output, this pin is driven by the timer clock TCLK0/1. |
| TCLK0<br>TCLK1 | 23<br>24 | I | Timer clock. The clock on this pin can drive either of the two internal timers; its source is on the motherboard. |

AMELIA has three main components: a DSP parallel interface section, a synchronous serial interface, and a sample-rate generation section.

The parallel section is connected onto the ’C40 global (or local) bus with no signal modifications, as shown in Figure 2. AMELIA uses 16 bits of data and just four address lines; its signals include read/write, chip select, a clock input, and chip reset. The functional block includes data buffers and multiplexers, internal address decoders, and an interrupt generator. Interrupts can be generated from three different sources: receive register full, transmit register empty, and external interrupt signals received from the daughter module. This provides a flexible method of data transfer control.

**Figure 2. AMELIA Circuit Connections**



The timer section comprises a full 16-bit reloadable time-out counter and a divide-by-1,-2,-4 or -8 prescaler. The timer can be independently clocked from one of four sources, any of which can be used to provide the sample rate clock. The LSI ’C40-based board has a socketed oscillator that can be used for this purpose. The clock source is a user-selectable feature because sample rate sources are already provided on the analog interface modules.

The control and configuration block is a software programmable section that gives AMELIA its wide operating characteristic. These controls make it possible to accommodate a wide range of serial standards found on converter devices. Most converters use a form of synchronous serial interface (SSI), but this varies across manufacturers. The main differences are the phasing and the type of frame sync signals that control the transfer. AMELIA has sophisticated frame sync shaping circuitry to accommodate as wide a range of protocols as possible. As a result, AMELIA can input FSR and FSX signals in bit, word, or I2S formats with either normal or inverted polarity and can also generate FSX in any of these protocols.

The parallel I/O block provides eight digital outputs (CM7–0) and four digital inputs (SM3–0). These are general-purpose signals that can be used to provide autoconfiguration, control, or status information about the interface module being used. Analog interfaces use this feature to configure converter-specific signals on the modules.

The present version of AMELIA is a two-channel design incorporating two input and two output channels. As such, AMELIA integrates two parallel-to-serial and two serial-to-parallel converters. For analog data input, data is read from the analog module under control of the frame sync logic and transferred to the data latches, which are then read by the ’C40. Data sampling is synchronous because both data channels are driven from the same clock source.

From performance measurements of AMELIA, the maximum clock rate that can be accommodated at the serial interface is 12 MHz. This gives adequate bandwidth for a wide range of devices to be interfaced to the ’C40.

## Programming Interface

AMELIA can accommodate a wide range of fundamentally different conversion techniques. To implement this, the ASIC incorporates a high level of programmability. All control signals generated for the converters are configurable in terms of polarity. The duty cycle of the sample rate trigger is selectable as either a pulse train or as a square wave, allowing both delta-sigma and successive approximation parts to be accommodated. Because of the fundamental differences in the operation of these conversion standards, the delta-sigma converter requires a sample rate clock, whereas successive approximation devices need a stream of conversion pulses to operate.

This level of flexibility requires a block of 16 address locations from the ’C40. These locations include both channels’ input and output data registers, timer programming register, and the control and configuration registers. Also, a number of locations are reserved for future enhancements. The register map is shown in Table 2.

**Table 2. Register Map**

| Location (Hex) | Read | Write |
|---|---|---|
| 9000 0000 | NU | NU |
| 9000 0001 | NU | NU |
| 9000 0002 | Ch0 Input Data | Ch0 Output Data |
| 9000 0003 | NU | NU |
| 9000 0004 | NU | NU |
| 9000 0005 | NU | Timer1 |
| 9000 0006 | Ch1 Input Data | Ch1 Output Data |
| 9000 0007 | NU | NU |
| 9000 0008 | NU | User control |
| 9000 0009 | NU | NU |
| 9000 000A | Analog Status | Analog Control |
| 9000 000B | Interrupt Status | Interrupt Mask |
| 9000 000C | NU | NU |
| 9000 000D | NU | NU |
| 9000 000E | NU | NU |
| 9000 000F | NU | Configuration |

NOTE: NU = Not Used.

Once the configuration for the analog module is complete, the details of the control registers may be ignored. The programming task is reduced to writing five prepared control words. Example 1 shows a two-channel echo program that simply reads data from the ADC and immediately writes that data back out to the DAC.

### Example 1.  Two-Channel Echo Program

```
********************************************************************
*                                                                  *
*   ECHO example program.                                          *
*                                                                  *
*   ECHO initializes the AM/D16DS on the DPC/C40 and echoes the    *
*   input on each channel.                                         *
*                                                                  *
********************************************************************

              .data

STACK         .word   002ffc00h       ;Define stack space
IACKLOC       .word   80000000h       ;Interrupt acknowledge location
IVECTAB       .word   002ff800h       ;Interrupt vector table
CHANA         .word   90000002h       ;Channel A address
CHANB         .word   90000006h       ;Channel B address
UCR           .word   90000008h       ;User control register address
ACR           .word   9000000ah       ;Analog control register address
IMR           .word   9000000bh       ;Analog interrupt mask register address
CONFIG        .word   9000000fh       ;Configuration register address

              .text

                                      ;Set up interrupt vector table
               BR     START
              .word   0h              ;Unused interrupts: NMI
              .word   0h              ;TINT0
              .word   0h              ;IIOF0
              .word   ISR             ;Amelia Interrupts on IIOF1

                                      ;Start of program...

START:          LDP     @STACK,DP     ;Initialize the stack
                LDI     @STACK,SP

                LDI     @IVECTAB,R0   ;Set up the interrupt vector table
                LDPE    R0,IVTP

;;;;            Write to the registers within AMELIA...


                LDI     @UCR,AR0      ;User control register
                LDI     0a000h,R5     ;
                STI     R5,*AR0       ;ADMCLK0 to be used
                LDI     @ACR,AR0             ;Analog control register
                LDI     0a0H,R5              ;
                STI     R5,*AR0              ;48 KHz sample rate
                                             ;AMELIA into reset

                LDI     @ACR,AR0             ;Analog control register
                LDI     0e0H,R5              ;
                STI     R5,*AR0              ;AMELIA released from reset,
                                             ;calibrating

                LDI     @CONFIG,AR0          ;Analog configuration register
                LDI     0b390H,R5            ;loaded with Key value
                STI     R5,*AR0              ;

                LDI     @IMR,AR0             ;Analog interrupt mask register
                LDI     01H,R5               ;Int when RX register full
                STI     R5,*AR0              ;

;;              AMELIA configuration complete. Initialize ’C40

                LDI     0,R3                 ;channel a output
                LDI     0,R4                 ;channel b output

                LDI     @IMR,AR0             ;AR0 = Interrupt mask register
                LDI     @CHANA,AR1           ;AR1 = channel a input
                LDI     @IACKLOC,AR2         ;AR2 = Interrupt ack. location
                LDI     @CHANB,AR3           ;AR3 = channel b input

                OR     90h, IIE              ;Enable IIOF1 interrupt
                OR     02000h, ST            ;CPU global interrupt

LOOP:           BR     LOOP                  ;IDLE until interrupted

ISR:
                LDI     *AR0,R5              ;read interrupt to clear

                LDI     *AR1,R3              ;Load channel a input –> R3
                STI     R3,*AR1              ;Save R3 –> channel a output

                LDI     *AR3,R4              ;Load channel b input –> R4
                STI     R4,*AR3              ;Save R4 –> channel b output

                RETI

               .end
```

The program begins by initializing the ’C40 stack and interrupt vector table location. AMELIA resides at addresses above 9000 0000h in the ’C40 memory map, placing it on the global expansion bus. The analog module used in the example was a delta-sigma converter, the AM/D16DS daughter module from LSI. All the specific programming details for the module are included with that product.

To begin analog interface programming, you configure the user control register (UCR). This register defines the clock source to be used on the module and also any prescaler values that may be required. The on-module oscillator ADMCLK0 is selected. The analog control register then selects the sample rate of ADMCLK0 to be 48 kHz. The UCR also controls of the module calibration. The first figure written to the register resets the analog module, and the second write causes the register to enter an offset calibration cycle.

The configuration register defines the correct communication protocol between the ’C40 and the analog module, which has already been defined at LSI, so a simple write to the register is sufficient. In practice, the configuration register sets a number of controls to define clock polarities, a valid clock edge on which to read data, and the frame sync controls.

The interrupt mask register is the last initialization task; the rest of the program is application-specific. The final task to start the system is to enable the IIOF1 interrupt on the ’C40, the signal to which AMELIA is connected.

The data is read and written in the interrupt service routine, which also illustrates the simple software interface to AMELIA.

## Conclusion

This article has outlined a significant advance in the techniques required to provide a modular analog interface to the ’C40. The single-chip interface adapter offers a number of advantages: all the functionality of the ’C40 is preserved, converters can easily be exchanged if the requirements change, and the interface modules have a higher performance specification. The combination adds up to a powerful interface solution that can be used to apply generic ’C40 signal processing technology to a range of specific applications.

ADC and DAC conversion is the basis of this paper, but other communication protocols could easily be transferred onto a common interface platform, including RS-232 and PSTN extension line connections. Digital interfaces can also be added, including digital audio, telecomm, and control applications. Adding functionality to the standard platform is a straightforward task.

# A Parallel Approach for Matrix Multiplication on the TMS320C4x DSP

**Rose Marie Piedra**
**Digital Signal Processing — Semiconductor Group**
**Texas Instruments Incorporated**

## Introduction

Matrix operations, like matrix multiplication, are commonly used in almost all areas of scientific research. Matrix multiplication has significant application in the areas of graph theory, numerical algorithms, signal processing, and digital control.

With today’s applications requiring ever higher computational throughputs, parallel processing is an effective solution for real-time applications. The TMS320C40 is designed for these kinds of applications.

This application note shows how to achieve higher computational throughput via parallel processing with the TMS320C40. Although the focus is on parallel solutions for matrix multiplication, the concepts stated here are relevant to many other applications employing parallel processing.

The algorithms that are presented were implemented on the Parallel Processing Development System (PPDS), which has four TMS320C40s and both shared- and distributed-memory support. The algorithms make use of parallel-runtime-support library (PRTS) functions available with the ’C40 C compiler for easy message passing.

This report is structured in the following way:

***Matrix Multiplication*** Gives a brief review of matrix multiplication and some common application areas.

***Fundamentals of Parallel Processing*** Presents some basic concepts of parallel processing. Partitioning, memory configuration, interconnection topologies, and performance measurements are some of the issues discussed.

***Parallel Matrix Multiplication*** Focuses on parallel implementations of matrix multiplication. Shared- and distributed-memory implementations are considered, as well as TMS320C40 suitability for each.

***Results of Matrix Multiplication on a TMS320C40-Based Parallel System*** Presents the results of shared- and distributed-memory imple mentations of parallel matrix multiplication on the ’C40 PPDS. Includes analysis of speed-up, efficiency, and load balance.

***Conclusion*** States conclusions.

***Appendices*** List the code for parallel matrix multiplication. The programs have been written in C. For faster execution, a C-callable assembly language routine is also supplied.

## Matrix Multiplication

Let A and B be matrices of size $n \times m$ and $m \times l$, respectively. The product matrix C = A * B is an $n \times l$ matrix, for which elements are defined as follows [7]:

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik}\, b_{kj} \qquad \text{where } 0 \le i < n,\ 0 \le j < l$$

The matrix multiplication requires O(*nml*) arithmetic operations, with each arithmetic operation requiring a cumulative multiply-add operation. When $l$=1, a matrix-vector multiplication exists. Assuming that $n$=$m$=$l$, matrix multiplication is an O($n^3$) operation.

Matrix-multiplication applications range from systems-of-equations solutions to graph representation. Also, matrix-vector multiplication can be applied to compute linear convolution. Refer to [2] and [7] for further information on these techniques.

## Fundamentals of Parallel Processing

When applications require throughput rates that are not easily obtained with today's sequential machines, parallel processing offers a solution.

Generally stated, parallel processing is based on several processors working together to accomplish a task. The basic idea is to break down, or partition, the computation into smaller units that are distributed among the processors. In this way, computation time is reduced by a maximum factor of $p$, where $p$ is the number of processors present in the multiprocessor system.

Most parallel algorithms incur two basic cost components[7]:

- computation delay—under which we subsume all related arithmetic/logic operations, and
- communication delay—which includes data movement.

In a realistic analysis, both factors should be considered.

This application report presents some basic concepts of parallel processing. Refer to [2], [4], [5], [6], and [7] for more detailed information.

### Partitioning Schemes

From the software point of view, two basic approaches are used to create a parallel application:

- **Functional Partitioning:** In this case, the task is a single function that has been subdivided between the processors. Each processor performs its subfunction on the data as it moves from one processor to the next in an assembly line or pipeline fashion.
- **Data Partitioning:** In this case, the task is partitioned so that each processor performs exactly the same function, but on different subblocks of the data. This approach requires algorithms with strong intrinsic parallelism. The parallel matrix multiplication implemented with the TMS320C40 PPDS applies this data-partitioning approach.

### Architectural Aspects

From the hardware point of view, two important issues should be considered:

- **Memory configuration (shared- versus distributed-memory):** In a distributed-memory system, each processor has only local memory, and information is exchanged as messages between processors. In contrast, the processors in a shared-memory system share a common memory. Although data is easily accessible to any processor, memory conflict constitutes the bottleneck of a shared-memory configuration. Because the PPDS has both shared and distributed memory, it is an excellent tool for implementing and evaluating different parallel configurations.
- **Connectivity network:** This issue relates to the way the processors are interconnected with each other. Fully connected networks (in which all the processors are directly connected to each other) are the ideal networks from an "ease of use" point of view. However, they are impractical in large

multiprocessor systems because of the associated hardware overhead. Linear arrays, meshes, hypercubes, trees, and fully-connected networks are among the topologies most commonly used. Hypercube topologies are widely popular in commercially available multiprocessor systems because they provide higher connectivity and excellent mapping capabilities. In fact, it is possible to embed almost any other topology in a hypercube network [4]. Mesh topologies are also commonly used to make systems modular and easily expandable. When distributed-memory systems are used, interconnectivity issues play an important role in the message-passing mechanism.

### Performance Measurements

Two measurements apply to the performance of a parallel algorithm—speed-up and efficiency.

- **Speed-up** of a parallel algorithm is defined as $S_p = T_s / T_p,$ where $T_s$ is the algorithm execution time when the algorithm is completed sequentially, and $T_p$ is the algorithm execution time using *p* processors. Theoretically, the maximum speed-up that can be achieved by a parallel computer with *p* identical processors working concurrently on a single problem is *p*. However, other important factors (such as the natural concurrence in the problem to be computed, conflicts over memory access, and communication delay) must be considered. These factors can reduce the speed-up.
- **Efficiency**, defined as $E_p = S_p / p$ with values between (0,1), is a measure of processor utilization in terms of cost efficiency. An efficiency close to 1 reveals an efficient algorithm. If the efficiency is lower than 0.5, it is often better to use fewer processors because using more processors offers no advantage.

Generally, the communication cost should be minimized by using wiser partitioning schemes and by overlapping CPU and I/O operations. DMA channels help to alleviate the communication burden.

## Parallel Matrix Multiplication

In parallel matrix multiplication, successive vector inner products are computed independently.

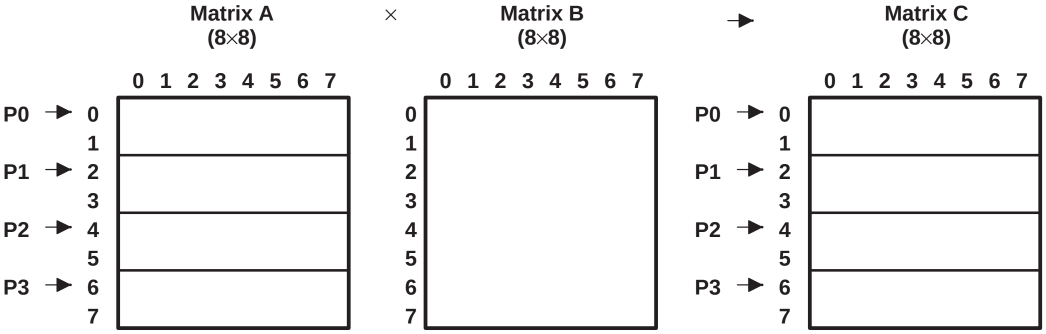
Because this application report focuses on multiple instruction multiple data (MIMD) implementations (shared- and distributed-memory approaches), systolic implementations are not discussed. However, single instruction multiple data (SIMD) implementations are also feasible with the TMS320C40.

### Shared-Memory Implementation

Let $n = qp$, where $n$ is the number of rows of matrix A, $p$ is the number of processors, and $q \geq 1$ is an integer.

Matrices A and B are stored in global memory so that each processor can have access to all the rows/columns. The basic idea is to allocate a different working set of rows/columns to each processor. Processor $i$ computes row vectors $qi$, $qi$+1, ... , $qi$+$q$–1 of product matrix C, where $i$ = 0,1,..., $p$–1. This is illustrated in Figure 1 for $p$ = 4 and $n$ = $m$ = $l$ = 8.

Figure 1.  Shared-Memory Implementation



**Note:** All processors have full access to the entire matrix B.

Two different approaches can be followed:

1. **Execute operations totally in shared memory (full memory conflict):** This implementation does not require any initial data transfer, but a conflict among memory accesses results (see code in Appendix A).

2. **Transfer data for execution in on-chip memory of each processor (reduced memory conflict):** This approach reduces the delay caused by memory conflicts, but it requires extra data transfer. This moving of data can be executed by the CPU or DMA via double-buffering techniques.

   Using double-buffering techniques can minimize the data-transfer delay. For matrix-vector multiplication, vector B is initially transferred to on-chip RAM. While the CPU is working on row $A_{(i)}$, the DMA is bringing row $A_{(i+1)}$ to on-chip RAM. If the two buffers are allocated in different on-chip RAM blocks, no DMA/CPU conflict will be present. If the DMA transfer time is less than or equal to the CPU computation time, the communication delay will be fully absorbed. The TMS320C40 has two 4K-byte on-chip RAM blocks that enable it to support double buffering of up to 1K words. Although this approach is not implemented in this application report, page 104 shows what kind of performance can be expected.

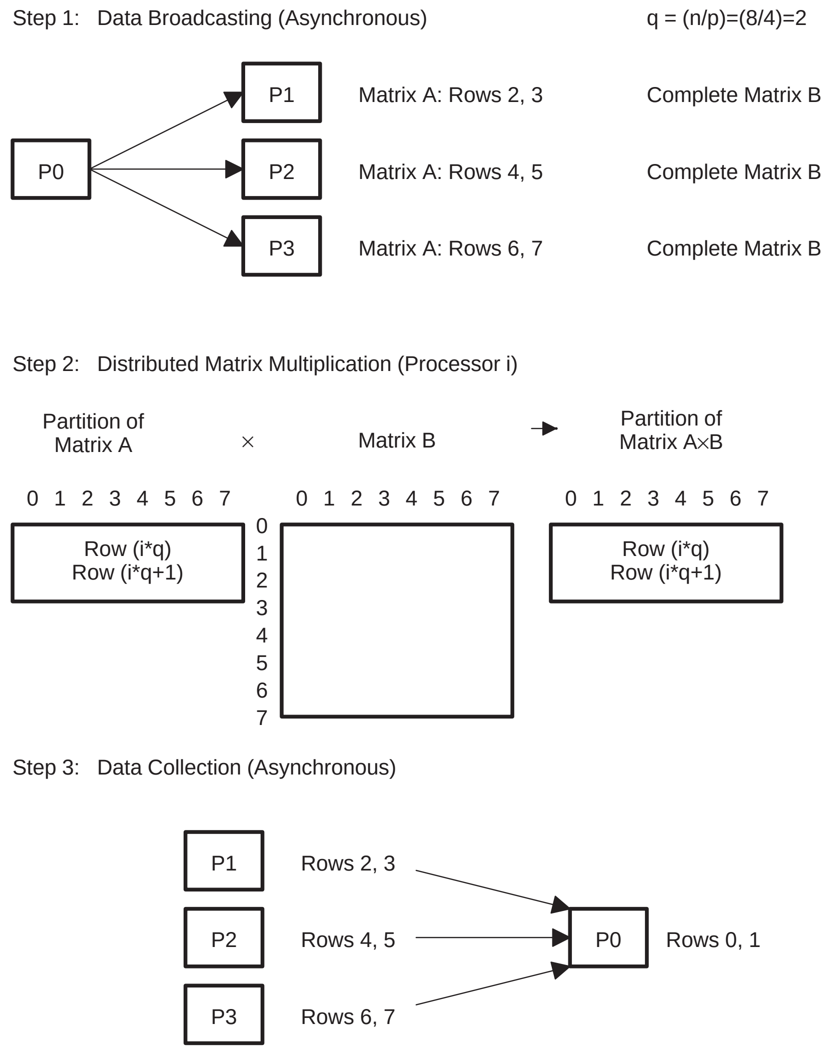
## Distributed-Memory Implementation

Let $n = qp$, where $n$ is the number of rows of matrix A, $p$ is the number of processors, and $q \geq 1$ is an integer.

Matrix A has been partitioned into $p$ regions with each region containing $q$ rows and being assigned to the local-memory (LM) of each processor. Matrix B is made available to all the processors. The data-partitioning scheme is similar to the shared-memory approach. The differences are the extra time required for data distribution/collection via message passing and the fact that all computations are done in the LM of each processor with no memory-access conflict involved. With the use of double-buffering, this communication delay can be reduced.

In this implementation, it is assumed that only processor 0 has access to matrix A and B. Processor 0 acts as a host processor responsible for broadcasting the needed data to each of the other processors and waiting for the vector results from the other processors. This is illustrated in Figure 2. Data distribution/collection is system-specific and may not be needed for certain applications.

**Figure 2. Distributed-Memory Implementation**



**Note:** Asynchronous = using DMA channels.

## TMS320C40 Implementation

The TMS320C40 is the first parallel-processing DSP. In addition to a powerful CPU that can execute up to 11 operations per cycle with a 40- or 50-ns cycle time, it contains 6 communication ports and a multichannel DMA [3]. 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. Also, special interlocked instructions provide support for shared-memory arbitration. These features make the TMS320C40 suitable for both distributed- and shared-memory computing systems.

## Results of Matrix Multiplication on a TMS320C40-Based Parallel System

Parallel matrix multiplication was implemented in the TMS320C40 PPDS. The PPDS is a stand-alone development board with four fully interconnected TMS320C40s. Each ’C40 has 256K bytes of local memory (LM) and shares a 512K-byte global memory (GM)[1].

Features of implementing parallel matrix multiplication in the TMS320C40 PPDS:

- The programs are generic. You can run the programs for different numbers of processors in the system just by changing the value of P (if you set P=1, you will have a serial program).
- Data input is provided in a separate file to preserve the generality of the programs.
- A node ID must be allocated to each processor. In this way, each processor will select automatically the row/column working set allocated to it. In this implementation, a different node ID is allocated to each processor by using the ’C40 debugger commands to initialize that variable. It is also possible to allocate a node ID by using the *my_id* function in the parallel-runtime-support library (PRTS), which reads a predetermined set node ID value from a user-specified memory location.
- For benchmarking of shared-memory programs, a global start of all the processors is absolutely necessary; otherwise, the real-memory-access conflict will not be observed. To help with this process, a C-callable assembly routine is provided in Appendix C (*syncount.asm*) for debugging systems without global start capability. Rotating priority for shared memory access should be selected by setting the PPDS LCSR register to 0x40. On this basis, the total execution time of the parallel algorithm can be defined as $T = \max(T_i)$ , where $T_i$ is the execution time taken by processor i (see Appendix A, *shared.c.*: $T_i$ = time between labels $t_2$ and $t_1$).
- For benchmarking of distributed-memory programs, I/O-execution time is optional. Data I/O is system-specific and normally is not considered. In this application report, speed-up/efficiency figures are given for both cases—including and not including I/O—in order to show the effect of the communication delay in a real application. In this program (see Appendix B, *distrib.c*), when processor 0 is acting as a host, then

  Execution time (I/O included) = time between labels t1 and t4 in processor 0.

  Execution time (I/O not included) = time between labels t2 and t3 in the processor with more load, or in any processor in the case of load balancing.
- If a debugger with benchmarking options (*runb*) is not available, the ’C40 analysis module or the ’C40 timer can be used. In this application report, the ’C40 timer and the timer routines provided in the PRTS library have been used for speed-up efficiency measures. Serial program timings for the speed-up figures were taken with the shared-memory program with P = 1.
- When the number of rows of matrix A is not a multiple of the number of processors in the system, load imbalance occurs. This case has been considered for the shared-memory (full-memory-conflict) implementation. (See Appendix A, *shared.c*.)

In parallel processing, speed-up/efficiency figures are more important than cycle counting because speed-up/efficiency figures show how much performance improves if you make an application parallel. You can apply the speed-up factors to any known sequential benchmarks to get a rough idea of the parallel-execution time (assuming that the same memory allocation is used). Appendix D includes a C-callable assembly-language function that executes matrix multiplication in approximately nrowsa*(5+ncolsb*(6+ncolsa)) cycles in single-processor execution. This assumes use of program and data in on-chip RAM. It also shows how you can use that function for parallel-processing execution.

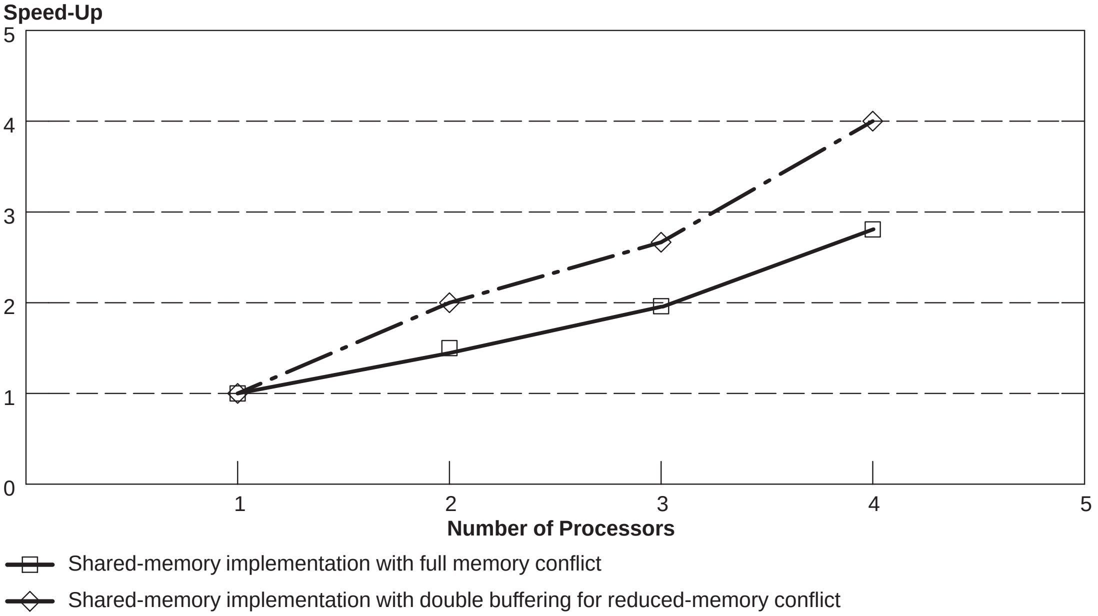
### Analysis of the Results

The performance of a parallel algorithm depends on the problem size (matrix size in our case) and on the number of processors in the system. Speed-up and efficiency figures covering those issues can be observed from Figure 3 to Figure 8 for the parallel algorithms presented. As you can see:

- Shared-memory (full-memory conflict) has the lowest speed-up and efficiency. However, the initial transfer of data to on-chip memory increases the speed-up, and if double-buffering techniques are used, shared-memory implementation becomes as ideal as the distributed-memory approach.
- Speed-up is proportional to the number of processors. In the shared-memory implementation (reduced-memory conflict) or in the distributed case (computation only), an optimal speed-up of *p* can be reached. See Figure 3 and Figure 5. This result occurs because matrix multiplication does not require any intermediate communication steps. In Figure 4, when $p = 3$, there is a decline in efficiency due to load imbalance.
- In general, efficiency is a better measure to analyze a parallel algorithm because it is more meaningful in processor utilization. For example, compare Figure 3 and Figure 4—the efficiency figure shows more clearly how increasing the number of processors negatively affects the performance of the shared-memory (full-memory conflict) implementation.
- Speed-up/efficiency increases for larger matrices in all cases, except for the shared-memory (full-memory conflict) case. In the distributed-memory case (with I/O), speed-up/efficiency increases because the communication delay ($O(n^2)$ operation) becomes negligible against the computation delay ($O(n^3)$ operation) for large *n*. See Figure 6 and Figure 7.
- In the case of load imbalance, efficiency decreases because computation is not evenly distributed among the processors. This is plotted in Figure 8. As you can see, if P = 4, the worst case occurs when matrix size = 5, because while processor 0 is calculating the last row (row 4), all the other processors are idle. The results shown here were taken for the shared-memory implementation but are applicable for the distributed case.
- The shared-memory implementation requires $n*m+m*l+n*l$ words of shared memory (for matrices A, B, and C, respectively). When this amount of memory is not available in the system, intermediate-file downloading can be used. Another option is in-place computation ( $A * B \rightarrow A$ ) using one intermediate buffer of size *n* per processor. For the distributed-memory case, the performance depends on the way you implement your initial data distribution. In the application, processor 0 requires $n*m+m*l+n*l$ words of local memory. The other processors require $q*m+m*l+q*l$ of local memory, where $q = \lfloor n/p \rfloor$.

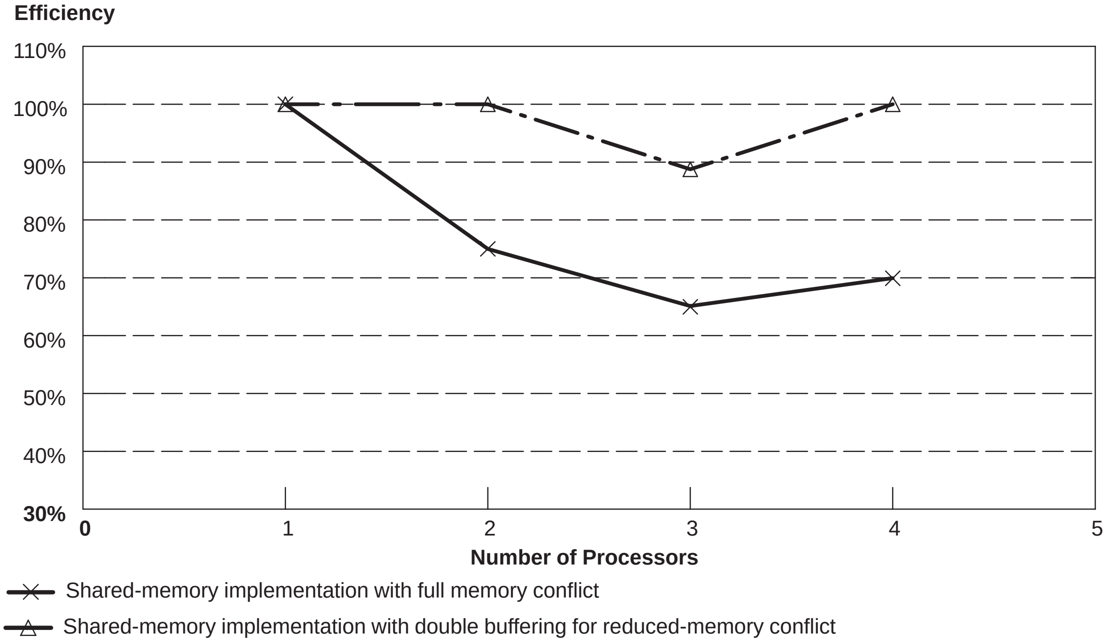
The programs in Appendices A and B have been used to calculate the speed-up/efficiency figures. For the assembly-language case (Appendix D), the speed-up figures for the computation timing are still valid. For the total timing (I/O included) using the assembly-language routine, the C implementation of the PRTS routines lowers the speed-up, but for larger matrices, this is minimized.

**Figure 3. Speed-Up Vs. Number of Processors (Shared Memory)**



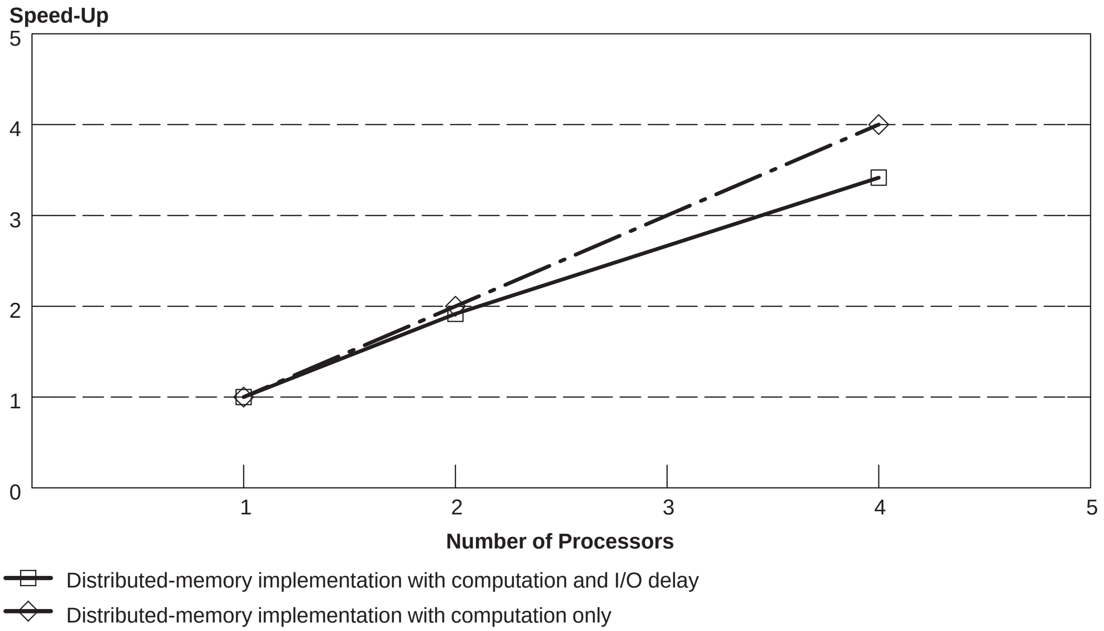
**Note:** Matrix size = 16×16

**Figure 4. Efficiency Vs. Number of Processors (Shared Memory)**



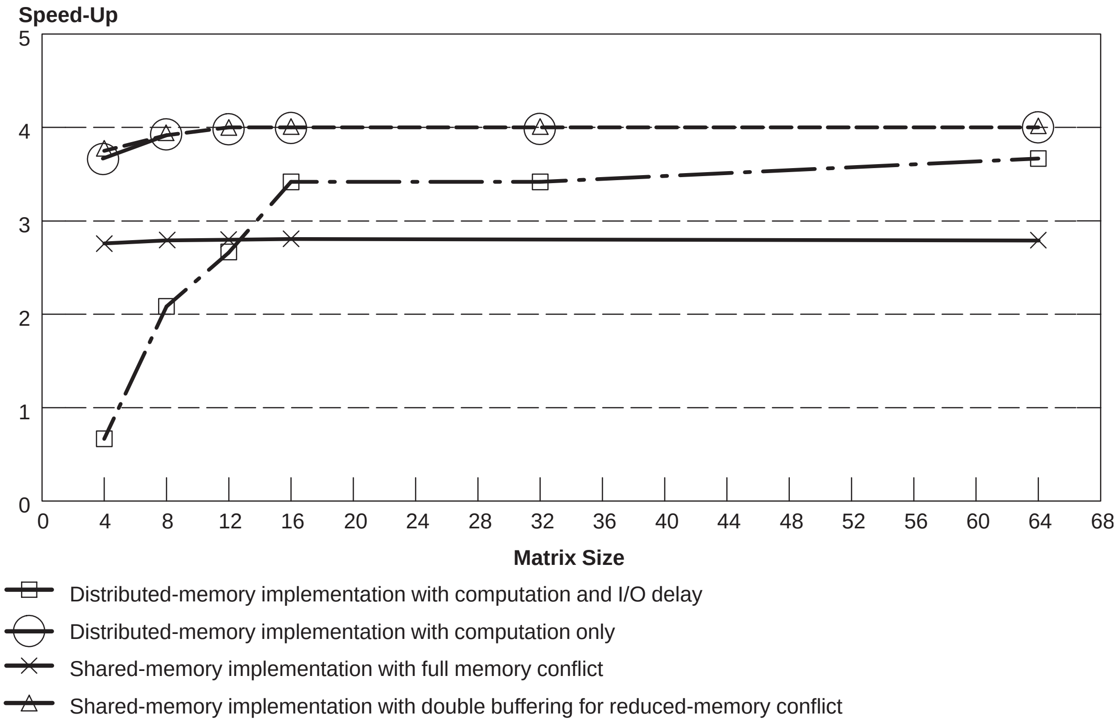
**Note:** Matrix size = 16×16

Figure 5. Speed-Up Vs. Number of Processors (Distributed Memory)



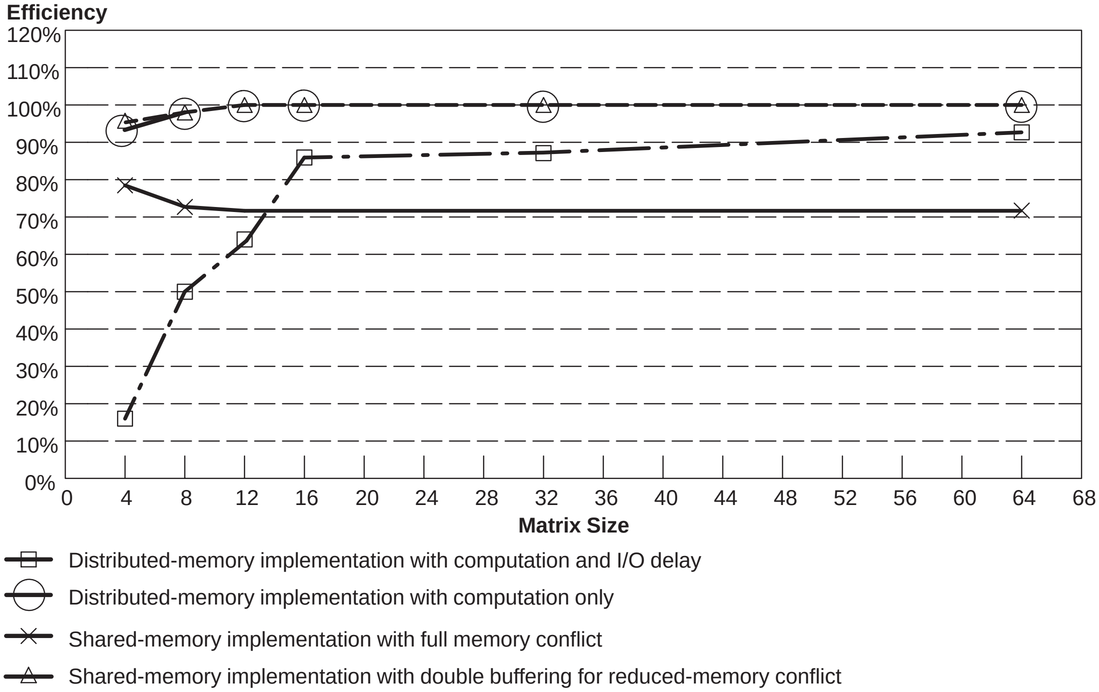
**Note:** Matrix size = 16×16

Figure 6. Speed-Up Vs. Matrix Size



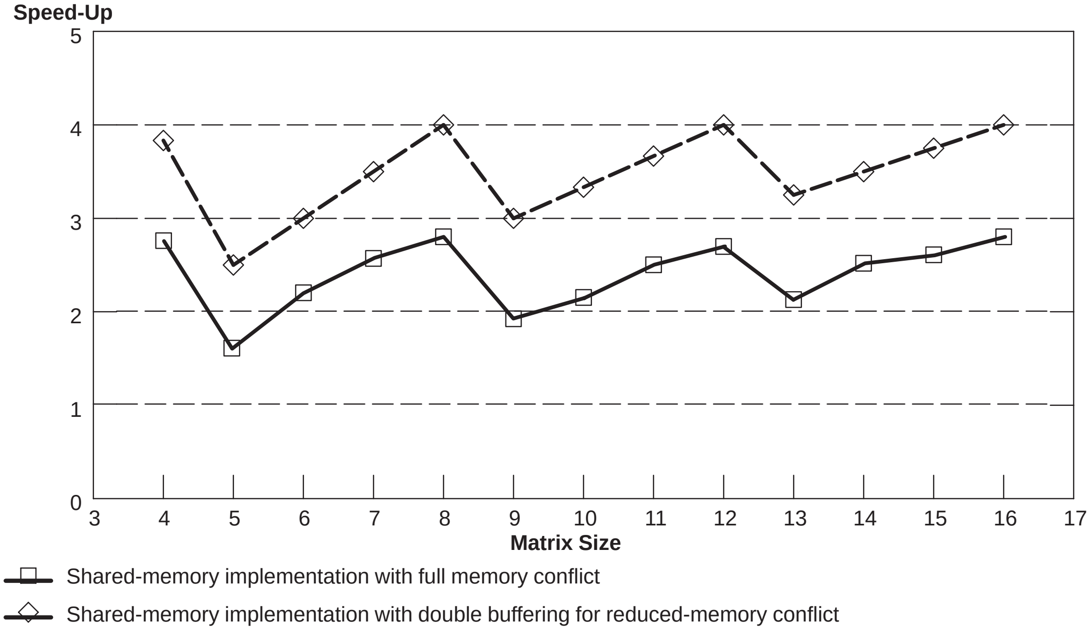
**Note:** Number of Processors = 4

## Figure 7.  Efficiency Vs. Matrix Size



**Note:** Number of Processors = 4

## Figure 8.  Speed-Up Vs. Matrix Size (Load Imbalance for Shared-Memory Program)



**Note:** Number of Processors = 4

# Conclusion

This report has presented parallel implementations of matrix multiplication using both shared- and distributed-memory approaches. Matrix multiplication is an excellent algorithm for parallel processing, as the speed-up/efficiency figures have shown. To avoid memory conflict when using the shared-memory approach, it is important to transfer the data to on-chip/local memory for execution. Because interprocessor communication is required only initially, it does not have a strong effect on the performance of the distributed-memory approach; but with double-buffering techniques, this can be minimized even more. Load balancing must also be considered.

## References

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

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

[3] *TMS320C4x User’s Guide*, Texas Instruments, Incorporated, 1991.

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

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

[6] U. Schendel. *Introduction to Numerical Methods for Parallel Computers*. England: John Wiley & Sons, 1984.

[7] J. J. Modi. *Parallel Algorithms and Matrix Computation*. New York: Oxford University Press, 1988.

# Appendix A: Shared-Memory Implementation

## INPUT0.ASM

```
*****************************************************************
*
*    INPUT0.ASM: Contains matrix A and B input values.
*
*****************************************************************

        .global _MAT_A
        .global _MAT_B
        .global _MAT_AxB
        .global _synch        ; counter for synchronization (global start)

        .data

_synch  .int    0

_MAT_A           ; stored by rows
        .float   1.0, 2.0, 3.0, 4.0
        .float   5.0, 6.0, 7.0, 8.0
        .float   9.0, 10.0, 11.0, 12.0
        .float   13.0, 14.0, 15.0, 16.0

_MAT_B           ; stored by rows
        .float   1.0, 2.0, 3.0, 4.0
        .float   1.0, 2.0, 3.0, 4.0
        .float   1.0, 2.0, 3.0, 4.0
        .float   1.0, 2.0, 3.0, 4.0

_MAT_AxB .space  16       ; must produce (by rows):
                                        ; 10,20,30,40
                                        ; 26,52,78,104
                                        ; 42,84,126,168
                                        ; 58,116,174,232

        .end

/***********************************************************************
```

## SHARED.C

```
/************************************************************************
SHARED.C : Parallel matrix multiplication (Shared memory version: full memory
conflict)
              – All the matrices (A,B,C) are stored by rows.

To run:
     cl30 –v40 –g –o2 –as –mr shared.c
     asm30 –v40 –s input0.asm
     asm30 –v40 –s syncount.asm
     lnk30 shared.obj input0.obj shared.cmd
************************************************************************/
#define   NROWSA    4               /* number of rows in mat A    */
#define   NCOLSA    4               /* number of columns in mat A */
#define   NCOLSB    4               /* number of columns in mat B */
#define   P         4               /* number of processors       */

extern    float MAT_A[NROWSA][NCOLSA];
extern    float MAT_B[NCOLSA][NCOLSB];
extern    float MAT_AxB[NROWSA][NCOLSB];

extern    int    synch;              /* synchronization for global start  */
extern    void  syncount();

float     *A[NROWSA],*B[NCOLSA],*AxB[NROWSA],temp;

int       *synch_p     = &synch,
   q  = NROWSA/P,
   l1 = 0,
             my_node, i, j, k,tcomp;
/***********************************************************************/

main()
{
asm(” OR 1800h,st”);                 /* cache enable  */

/* accesing matrices declared in an external assembly file  */
for (i=0;i<NROWSA;i++)  A[i] = MAT_A[i];
for (i=0;i<NCOLSA;i++)  B[i] = MAT_B[i];
for (i=0;i<NROWSA;i++)  AxB[i] = MAT_AxB[i];

syncount(synch_p,P);                 /* global start:loop until counter=P */

if (((i = NROWSA %P) >0))  {         /* load imbalancing:optional         */
  if (my_node<i) ++q; else  l1 =i;
  }
l1 += q*my_node;                     /* select beginning of row working set
                                        for processor ”my_node”           */

t1: time_start(0);                   /* benchmarking with C40 timer       */

for (i=l1;i<(l1+q);i++)              /* matrix multiplication             */
for (j=0;j<NCOLSB;j++)
{
temp = 0;
for (k=0;k<NCOLSB;k++)   temp += A[i][k] * B[k][j] ;
AxB[i][j] = temp;
}

t2 : tcomp = time_read(0);           /* shared–memory benchmark           */
syncount(synch_p,2*P);               /* optional: if you want all processors
                                        finish at the same time           */
} /*main*/
```

## SHARED.CMD

```
/************************************************************************
SHARED.CMD: Linker Command File for Shared-Memory Program
************************************************************************/

syncount.obj
–c                                   /* link using C conventions   */
–stack 0x0100
–lrts40r.lib                         /* get run–time support       */
–lprts40r.lib
–m a.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   ROM:   org = 0x00          len = 0x1000
   RAM0:  org = 0x0002ff800   len = 0x0400     /* RAM block0    */
   RAM1:  org = 0x0002ffc00   len = 0x0400     /* RAM block1    */
   LM: org = 0x040000000      len = 0x10000    /* local memory  */
   GM: org = 0x080000000      len = 0x20000    /* global memory */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   .text:    {} > RAM0    /* code                          */
   .cinit:   {} > RAM1    /* initialization tables         */
   .stack:   {} > RAM0    /* system stack                  */
   .bss :    {} > RAM1    /* global & static vars          */
   .data:    {} > GM      /* for input matrix              */
}
```

# Appendix B: Distributed-Memory Implementation

## INPUT.ASM

```
******************************************************************
*
*   INPUT.ASM :  Input file for processors 1 to (P–1)
*
******************************************************************

          .global   _MAT_A
          .global   _MAT_B
          .global   _MAT_AxB

          .data

_MAT_A    .space  16
_MAT_B    .space  16
_MAT_AxB  .space  16

          .end
```

## DISTRIB.C

```
/**********************************************************************

DISTRIB.C : Parallel matrix multiplication (distributed–memory
implementation)
            (no load imbalancing has been considered)

cl30 –v40 –g –mr –as –o2  distrib.c
asm30 –v40 –s input0.asm      (see Input0.asm on page 111 )
asm30 –v40 –s input.asm
lnk30 distrib.obj input0.obj distrib.cmd –o a0.out (For processor 0)
lnk30 distrib.obj input.obj distrib.cmd –o a.out (For processors 1 to (P–1))

**********************************************************************/
#define   NROWSA        4            /* number of rows in mat A    */
#define   NCOLSA        4            /* number of columns in mat A */
#define   NCOLSB        4            /* number of columns in mat B */
#define   P             4            /* number of processors       */

extern    float MAT_A[NROWSA][NCOLSA];
extern    float MAT_B[NCOLSA][NCOLSB];
extern    float MAT_AxB[NROWSA][NCOLSB];

float     *A[NROWSA], *B[NCOLSA], *AxB[NROWSA], temp;

int       my_node ,
          q = NROWSA/P,
          tcomp, ttotal,
          i,j,k,l1;

int port[4][4] = {  0,0,4,3,
                    3,0,0,4,
                    1,3,0,0,
                    0,1,3,0 };       /*  connectivity matrix: processor i is
                                     connected to processor j thru port[i][j]:
                                     system specific PPDS          */

/************************************************************************/
```

```
main()
{

asm(” OR 1800h,st”);
/* accesing assembly variables */
for (i=0;i<NROWSA;i++)  A[i] = MAT_A[i];
for (i=0;i<NCOLSA;i++)  B[i] = MAT_B[i];
for (i=0;i<NROWSA;i++)  AxB[i] = MAT_AxB[i];

t1: time_start(0);
/* Processor 0 distributes data. Other processors receive it */

if (my_node==0)
   for(i=1;i<P;++i){   /* asynchronous sending (DMA)    */
      send_msg(port[0][i],&A[i*q][0],(q*NCOLSA),1);
      send_msg(port[0][i],&B[0][0],(NCOLSA*NCOLSB),1);
/* autoinitialization can also be used  */
       }
else {              /* synchronous receiving (CPU)       */
   k = in_msg(port[my_node][0],&A[0][0],1);
   k = in_msg(port[my_node][0],&B[0][0],1);
   }

t2: tcomp = time_read(0);

for (i=0;i<q;i++)   /* Matrix multiplication             */
for (j=0;j<NCOLSB;j++)
{
temp = 0;
for (k=0;k<NCOLSB;k++)   temp += A[i][k] * B[k][j];
AxB[i][j] = temp;
}
t3: tcomp = time_read(0) – tcomp;

/* Processors 1–(P–1) send result to proc. 0. Processor 0:ready to receive it */

if (my_node==0)
   for(i=1;i<P;++i)receive_msg(port[0][i],&AxB[i*q][0],1);  /* asynchronous*/
   else send_msg(port[my_node][0],&AxB[0][0],(q*NCOLSB),1);

if (my_node==0)     /* Wait for interprocessor communication to finish   */
   for (i=1;i<P;++i)  while (chk_dma(port[0][i]) );
else while (chk_dma(port[my_node][0])) ;

t4: ttotal = time_read(0);
/* this is including: comp + input + output + 2 timer_reads              */

} /*main*/
```

## DISTRIB.CMD

```
/************************************************************************
DISTRIB.CMD: Linker Command File for Distributed-Memory Program
************************************************************************/

–c                  /* link using C conventions   */
–stack 0x0100
–lrts40r.lib        /* get run–time support    */
–lprts40r.lib
–m a.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   ROM:   org =  0x0          len = 0x1000
   RAM0:  org =  0x0002ff800  len = 0x0400     /* RAM block0    */
   RAM1:  org =  0x0002ffc00  len = 0x0400     /* RAM block1    */
   LM:    org =  0x040000000  len = 0x10000    /* local memory  */
   GM:    org =  0x080000000  len = 0x20000    /* global memory */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY       */

SECTIONS
{
   .text:    {} > RAM0    /* code                    */
   .cinit:   {} > RAM1    /* initialization tables   */
   .stack:   {} > RAM0    /* system stack            */
   .bss :    {} > RAM1    /* global & static vars    */
   .data:    {} > LM      /* for input matrix        */
}
```

# Appendix C: Synchronization Routine for Shared-Memory Implementation

```
******************************************************************
*
*  syncount.asm :  assembly language synchronization routine to provide a
*  global start for all the processors. Initially, a counter in shared
*  memory is set to zero. Each processor increments the counter by 1. When
*  the counter equals value, the processors exit this routine. Rotating
*  priority for shared-memory access should be selected. The processors
*  start with a maximum cycle difference of 3 instruction cycles, which for
*  practical  purposes is acceptable. This routine is C–callable and uses
*  registers for parameter passing.
*
*  Calling conventions:
*      void syncount((int *)counter,int value)          ar2 , r2
*
*  where  counter   = synchronization counter in shared memory
*         value     = counter value to be reached.
*
******************************************************************

   .global _syncount
   .text
_syncount:
       LDII   *AR2,R1
       ADDI   1,R1
       CMPI   R1,R2
       STII   R1,*AR2
       BZ     L1

AGAIN  LDI    *AR2,R1
       CMPI   R1,R2
       BNZ    AGAIN
       L1     RETS
       .end
```

## Appendix D: C-Callable Assembly Language Routine for Matrix Multiplication

### INPUT0_A.ASM

```
*****************************************************************
*
* INPUT0_A.ASM:  Contains matrix A and B input values. Matrix B is
* stored by columns.
*
*****************************************************************

       .global _MAT_A
       .global _MAT_B
       .global _MAT_AxB
       .global _synch                   ; counter for synchronization

          .data

_synch    .int    0

_MAT_A                                  ; stored by rows
          .float   1.0, 2.0, 3.0, 4.0
          .float   5.0, 6.0, 7.0, 8.0
          .float   9.0, 10.0, 11.0, 12.0
          .float   13.0, 14.0, 15.0, 16.0

_MAT_B                                  ; stored by columns!!!
          .float   1.0, 1.0, 1.0, 1.0
          .float   2.0, 2.0, 2.0, 2.0
          .float   3.0, 3.0, 3.0, 3.0
          .float   4.0, 4.0, 4.0, 4.0

_MAT_AxB  .space 16                     ; must produce (stored by rows)
                                        ; 10,20,30,40
                                        ; 26,52,78,104
                                        ; 42,84,126,168
                                        ; 58,116,174,232

          .end
```

## MMULT.ASM

```
****************************************************************************
*
*    MMULT.ASM: Matrix multiplication (assembly language C–callable program)
*
*  mmult(&C,  &A, &B, nrowsa, ncolsa, ncolsb)
*      ar2, r2, r3, rc, rs, re
*
*   – Matrix A (nrowsa×ncolsa): is stored by rows (row–major order)
*   – Matrix B (ncolsa×ncolsb): is stored by columns (column–major order)
*   – Matrix C (nrowsa×ncolsb): is stored by rows (row–major order)
*   – ”ncolsb” must be greater or equal to 2
*   – This routine uses register to pass the parameters (refer to C compiler
*     users’guide for more information)
*
*   5/1/90 : Subra Ganesan
*   10/1/90: Rosemarie Piedra
*
****************************************************************************

       .global   _mmult
       .text

_mmult
       LDI    R2,AR0                 ; AR0: address of A[0][0]
       LDI    R3,AR1                 ; AR1: address of B[0][0]
                                     ; AR2: address of C[0][0]
       LDI    RS,IR0                 ; IR0: NCOLSA
       LDI    RE,R10                 ; R10: NCOLSB
       PUSH   AR5                    ; preserve registers
       PUSH   AR3
       PUSH   R5

       SUBI   1,RC,AR3               ; AR3: NROWSA–1
       SUBI   2,RS,R1                ; R1: NCOLSA–2
       SUBI   1,RE,R9                ; R9: NCOLSB–1

ROWSA
       LDI    R9,AR5

COLSB                                ; initialize R2
       MPYF3  *AR0++(1),*AR1++(1),R0 ; perform one multiplication
||     SUBF3  R2,R2,R2               ; A(I,1)*B(1,I) –> R0
       RPTS   R1                     ; repeat the instruction NCOLSA–1 times
       MPYF3  *AR0++(1),*AR1++(1),R0
||     ADDF3  R0,R2,R2
                                     ; M(I,J) * V(J) –>R0
                                     ; M(I,J–1) * V(J–1) + R2 –> R2
       DBD    AR5,COLSB              ; loop for NCOLSB times
       ADDF   R0,R2                  ; last accumulate
       STF    R2,*AR2++(1)           ; result –> C[I][J]
       SUBI   IR0,AR0                ; set AR0 to point A[0][0]
       DBD    AR3,ROWSA              ; repeat NROWSA times
       ADDI   IR0,AR0                ; set AR0 to point A[1][0]
       MPYI3  IR0,R10,R5             ; R5 : NCOLSB*NROWSB(IR0)
       SUBI   R5,AR1                 ; set AR1 to point B[0][0]
       POP    R5
       POP    AR3
       POP    AR5                    ; recover register values
       RETS
```

## SHAREDA.C

```
/*********************************************************************

SHAREDA.C : Parallel Matrix Multiplication (shared memory version: full
memory conflict)
   –This program uses an assembly language C–callable routine for matrix
   multiplication for faster program execution.
   –Matrix A and C are stored by rows. Matrix B is stored by columns to
   take better advantage of the assembly language implementation.
To run:
     cl30 –v40 –g –o2 –as –mr shareda.c
     asm30 –v40 –s input0_a.asm
     asm30 –v40 –s syncount.asm
     asm30 –v40 –s mmult.asm
     lnk30 mmult.obj shareda.obj input0_a.obj shared.cmd

*********************************************************************/
#define   NROWSA 4               /* number of rows in mat A       */
#define   NCOLSA 4               /* number of columns in mat A    */
#define   NCOLSB 4               /* number of columns in mat B    */
#define   P      4               /* number of processors          */

extern    float  MAT_A[NROWSA][NCOLSA];     /* stored by rows     */
extern    float  MAT_B[NCOLSB][NCOLSA];     /* stored by columns  */
extern    float  MAT_AxB[NROWSA][NCOLSB];   /* stored by rows     */
extern    void   mmult(float*C, float*A, float*B, int nrowsa, int ncolsa,
                        int ncolsb);

extern    int    synch;          /* synchronization for benchmarking  */
extern    void   syncount();

float     *A[NROWSA], *B[NCOLSB], *AxB[NROWSA], temp;
int       *synch_p  = &synch,
   q  = NROWSA/P,
   l1 = 0,
   my_node, i, j, k, tcomp;

/*********************************************************************/

main()
{
asm(” OR 1800h,st”);          /* cache enable                   */

/* accesing matrices declared in an external assembly file     */
for (i=0;i<NROWSA;i++)  A[i] = MAT_A[i];
for (i=0;i<NCOLSB;i++)  B[i] = MAT_B[i];
for (i=0;i<NROWSA;i++)  AxB[i] = MAT_AxB[i];

syncount(synch_p,P);          /* global start                   */

if (((i = NROWSA %P) >0))  {  /* load imbalancing: optional     */
  if (my_node<i) ++q; else  l1 =i;
  }
l1 += q*my_node;              /* select beginning of row working set
                                     for processor ”my_node”    */

t1: time_start(0);            /* benchmarking with C40 timer    */

mmult  (&AxB[l1][0],&A[l1][0],&B[0][0],q,NCOLSA,NCOLSB);/* matrix mult.*/

t2 : tcomp = time_read(0);    /*  shared–memory benchmark       */
syncount(synch_p,2*P);        /*  optional: if you want all processors
                                  finish at the same time       */
} /*main*/
```

# Parallel 1-D FFT Implementation With TMS320C4x DSPs

Rose Marie Piedra
Digital Signal Processing — Semiconductor Group
Texas Instruments Incorporated

## 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**

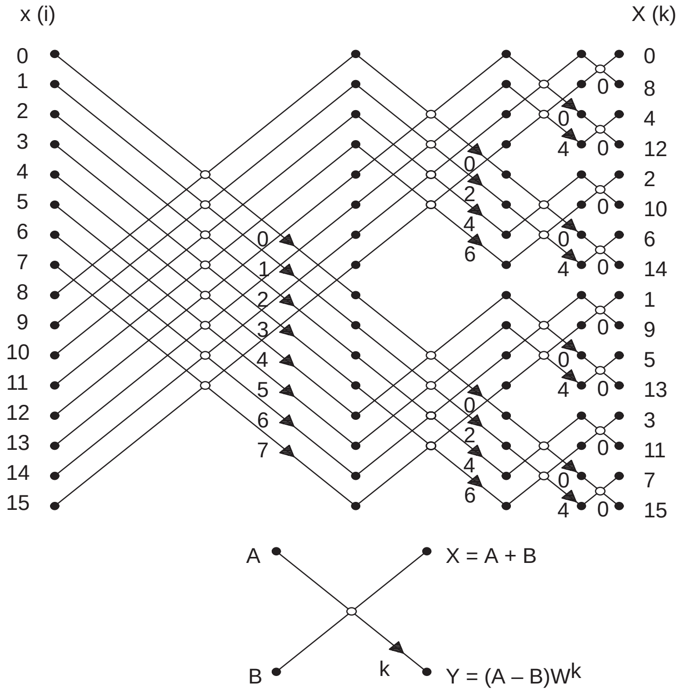
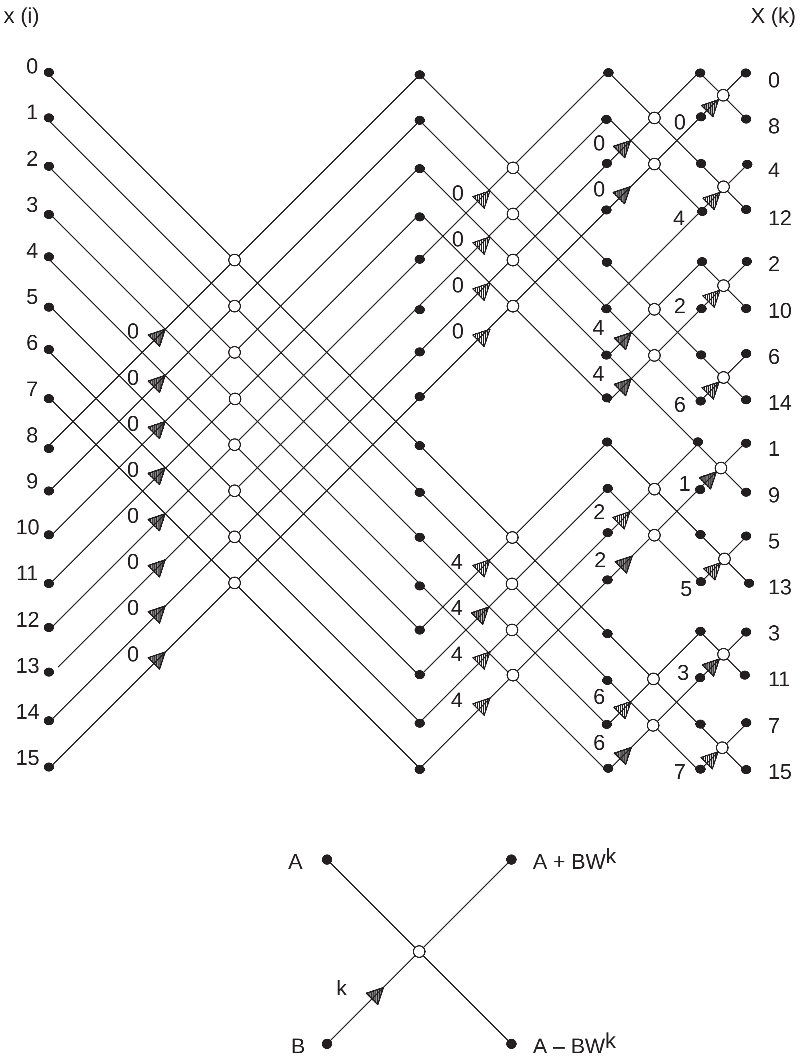


**Figure 2. Flow Chart of a 16-Point DIT FFT**



## 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

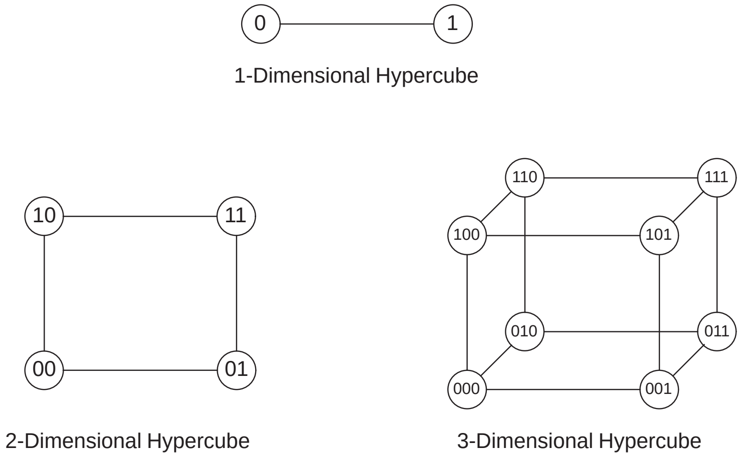
Decimation-in-Frequency (DIF) and Decimation-in-Time (DIT) decomposition schemes are investigated for parallel implementation. Parallel FFT theory is covered in [1], [11], [12], and [13].

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

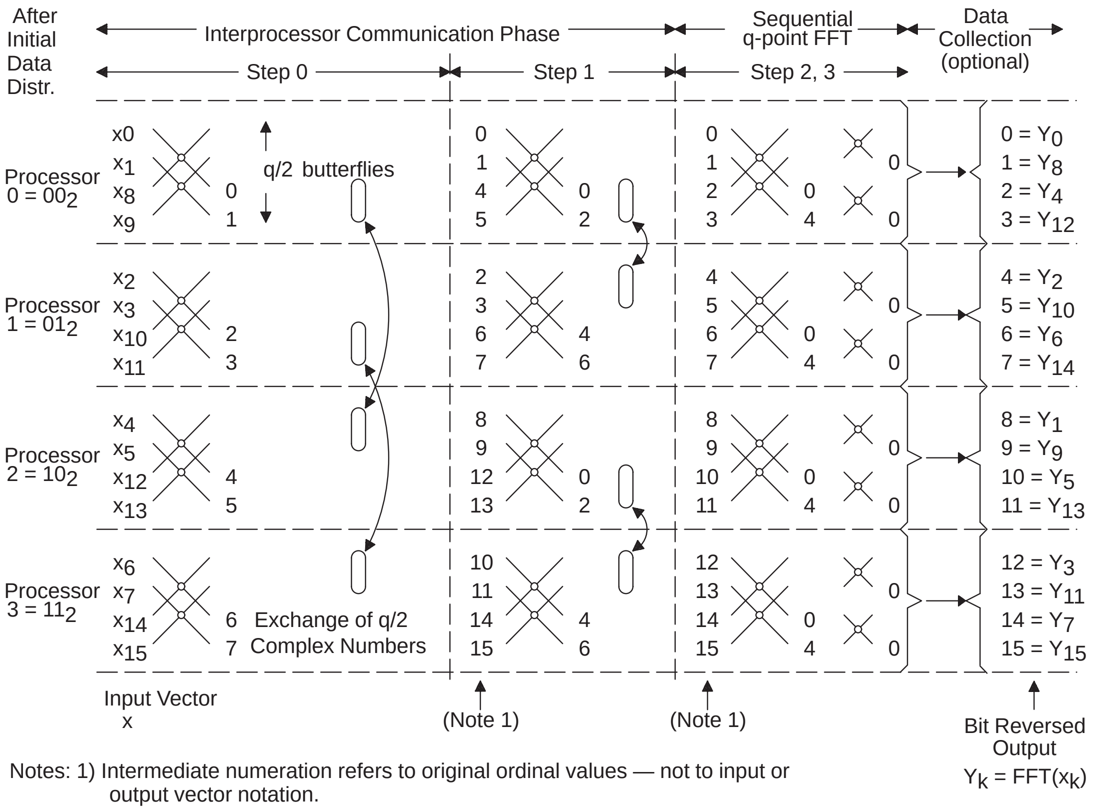
2-Dimensional Hypercube

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



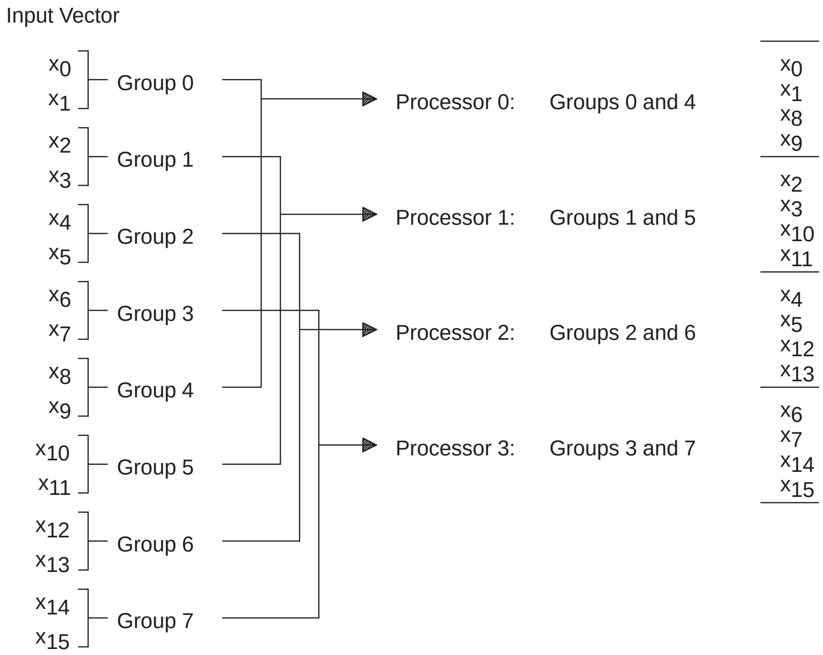
Notes: 1) Intermediate numeration refers to original ordinal values — not to input or output vector notation.

2) $\times_k = \times_{W_{16}^k}$

### 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 ($0 \leq j < q$ and $0 \leq i < p$). 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 $q \geq 1$ is an integer. FFT input is in normal order, and FFT output is bit-reversed after data collection.

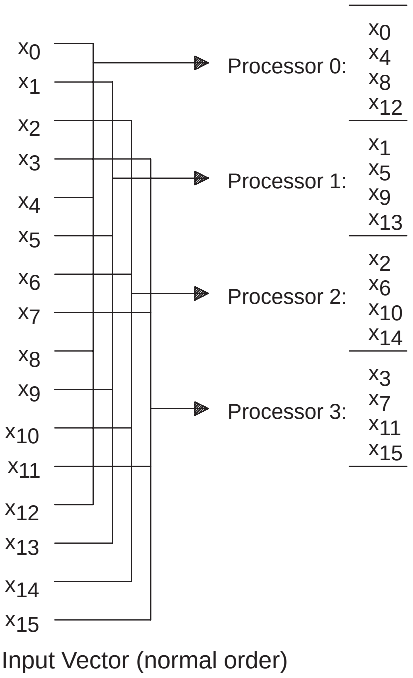
Parallel DIT requires a parallelization scheme different than the one presented for parallel DIF FFT. Sequential *q*-point FFT execution is required in the first ($\log_2 n - \log_2 p$) steps, and interprocessor communication is required in the last $\log_2 p$ steps. This is exactly opposite to the DIF 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 $0 \leq j < (n/p)$ and $0 \leq i < p$.

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)**



$x_0$
$x_1$
$x_2$
$x_3$
$x_4$
$x_5$
$x_6$
$x_7$
$x_8$
$x_9$
$x_{10}$
$x_{11}$
$x_{12}$
$x_{13}$
$x_{14}$
$x_{15}$
Input Vector (normal order)
Processor 0: $x_0$ $x_4$ $x_8$ $x_{12}$
Processor 1: $x_1$ $x_5$ $x_9$ $x_{13}$
Processor 2: $x_2$ $x_6$ $x_{10}$ $x_{14}$
Processor 3: $x_3$ $x_7$ $x_{11}$ $x_{15}$

**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.

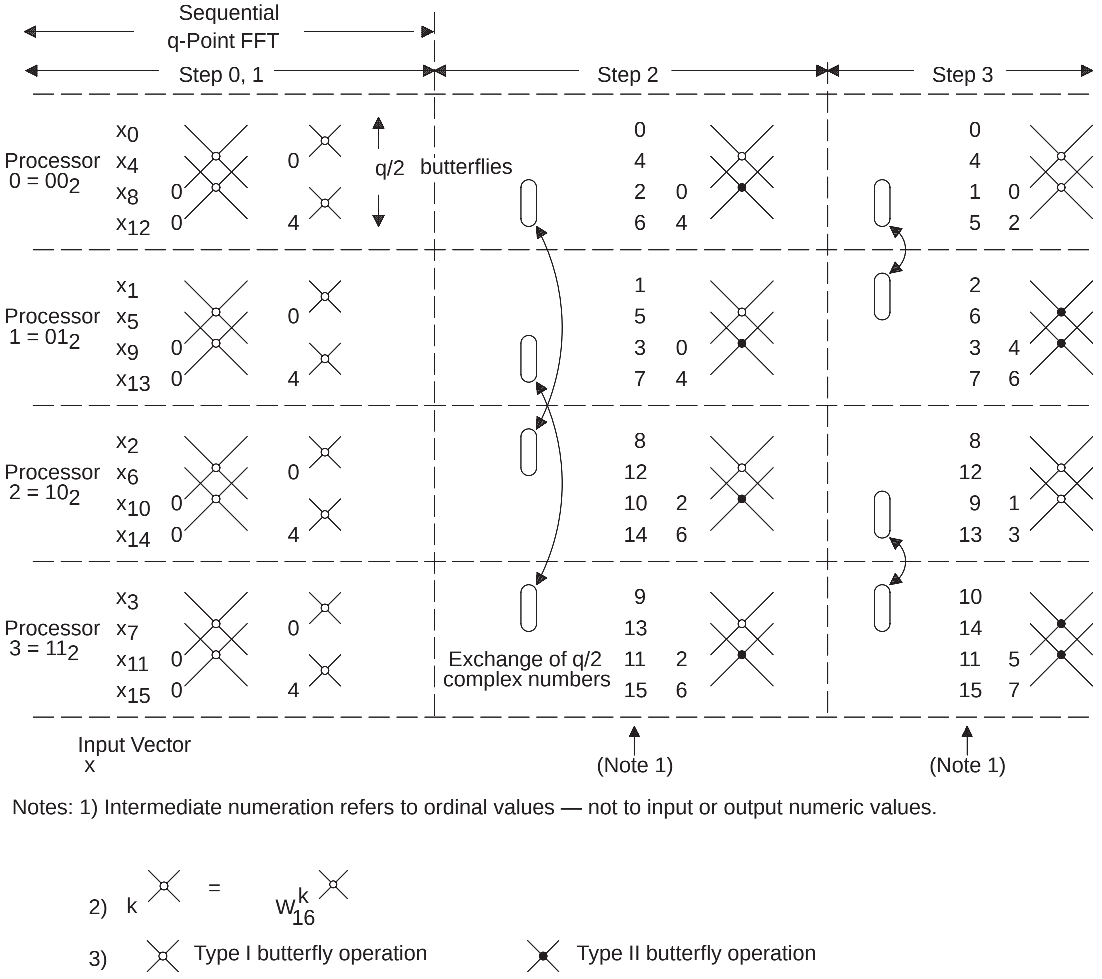
If bit $j$ of its node ID is 0,

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 7 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)

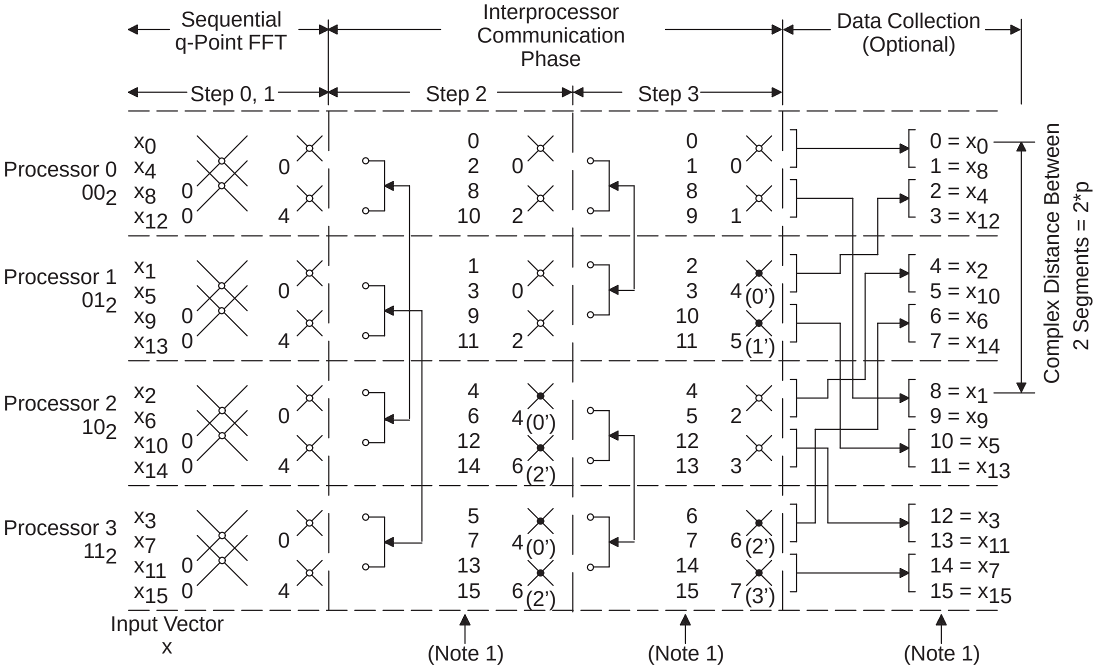
Figure 8 shows this new interprocessor communication scheme. Nonconsecutive complex numbers are exchanged between the processors. Notice that:

- 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 ($0\ j \leq q$ *and* $0 \leq i < p$). 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.

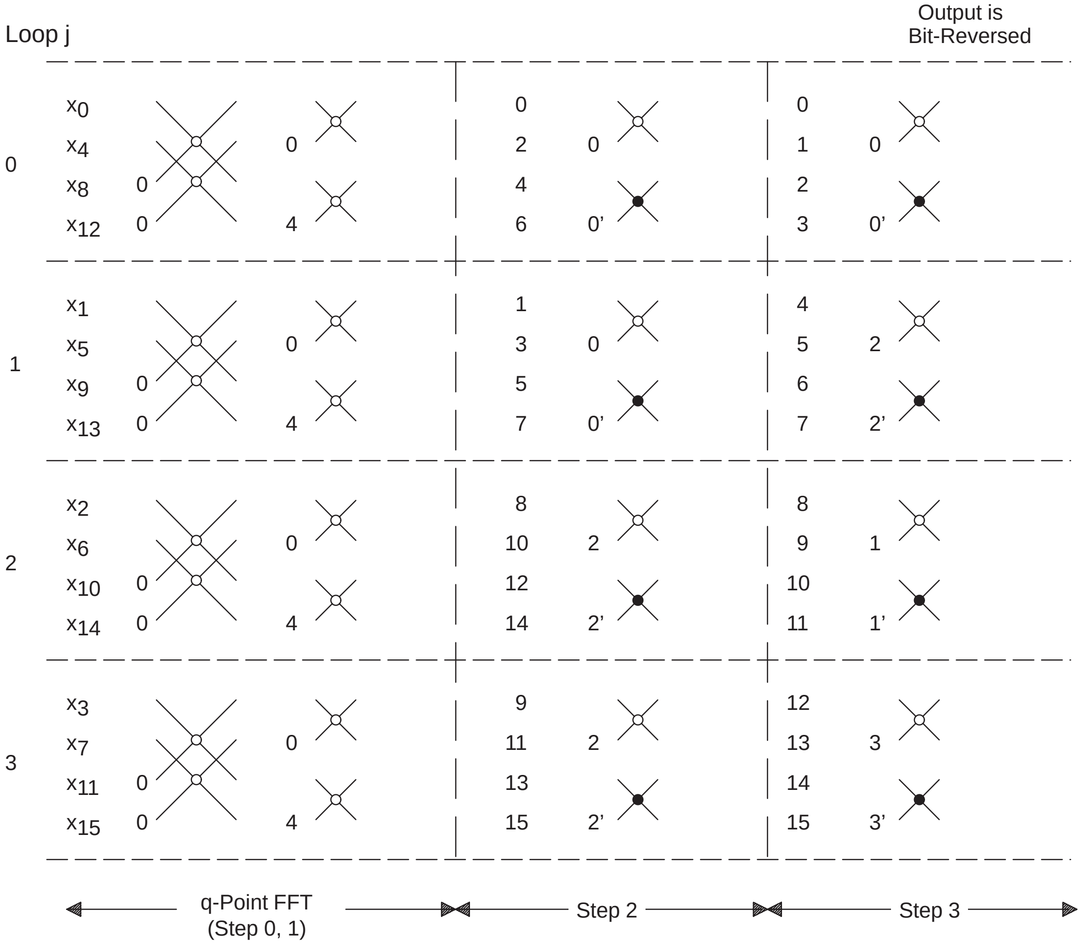
**Phase 1.** Execution of $p$ $q$-point FFTs: elements $i + j*p$ where $0 \leq j < (n/p)$ are transferred to on-chip memory for a $q$-point FFT execution. The process repeats for each loop $i$ ($0 \leq i < p$).

**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 ○ = $W_{16}^{k}$ ○

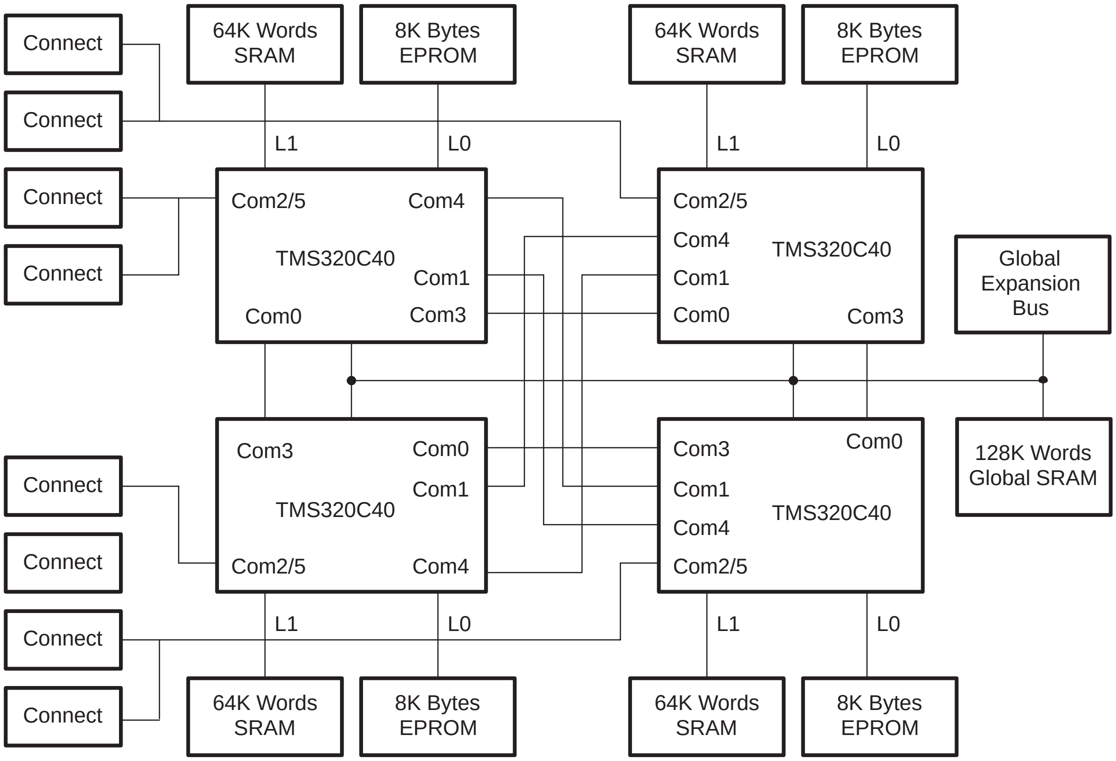
3) ○ Type I butterfly operation ● Type II butterfly operation

## 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*

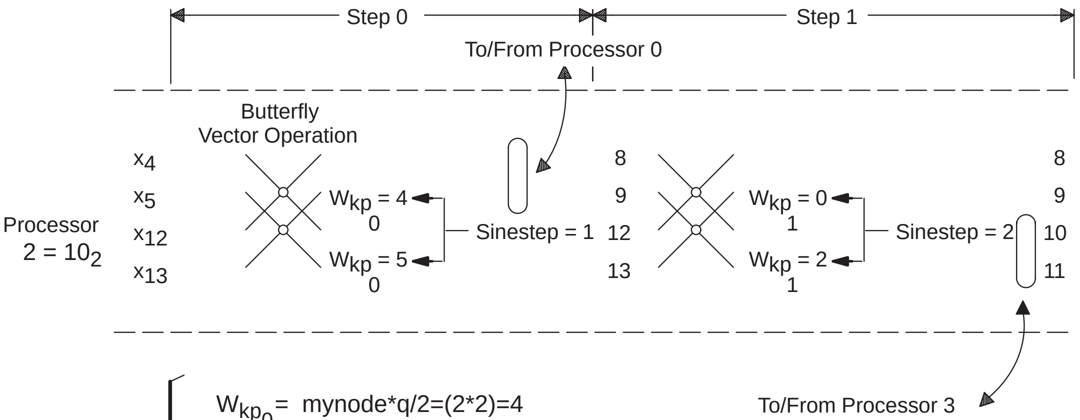
This implementation uses the CPU for interprocessor communication, even though the DMA could also be used. Because the CPU must wait for the interprocessor communication to finish before executing the next butterfly vector step, no real advantage is observed in using the DMA for data transfer. Aykanat 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)**



Step 0

$W_{kp_0}$= mynode*q/2=(2*2)=4
$\text{Sinestep}_0$=1
Destination node= (mynode=$10_2$) XOR (msbit =$10_2$)= $00_2$ = $0_{10}$

Step 1

$W_{kp_1}$= ($W_{kp_0}$*2) modulo (n/2) =(4*2) modulo 8 = 0
$\text{Sinestep}_1$=$\text{sinestep}_0$*2=1*2= 2

**Note:** $W_{kp_i}$= *Wkpointer* value for step i;
$\text{Sinestep}_i$ = *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.

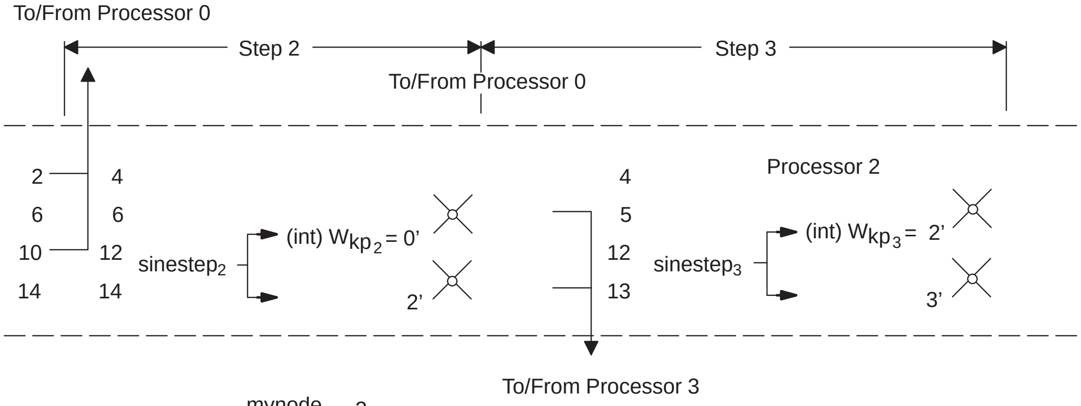
Either approach can be selected by changing the version number in DIS_DIF.C (Appendix C). The second approach (VERSION =1) is faster but requires the modification of the FFT core function. If you plan to use your own FFT routine but don't want to enter into the specifics of the parallel modification,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

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

(int) $W_{kp_2} = 0$

sinestep$_2$ = 2 (complex distance =1)

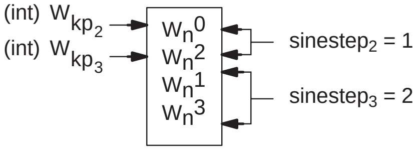
Step 3

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

(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 × 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 ($n > 1$K), 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

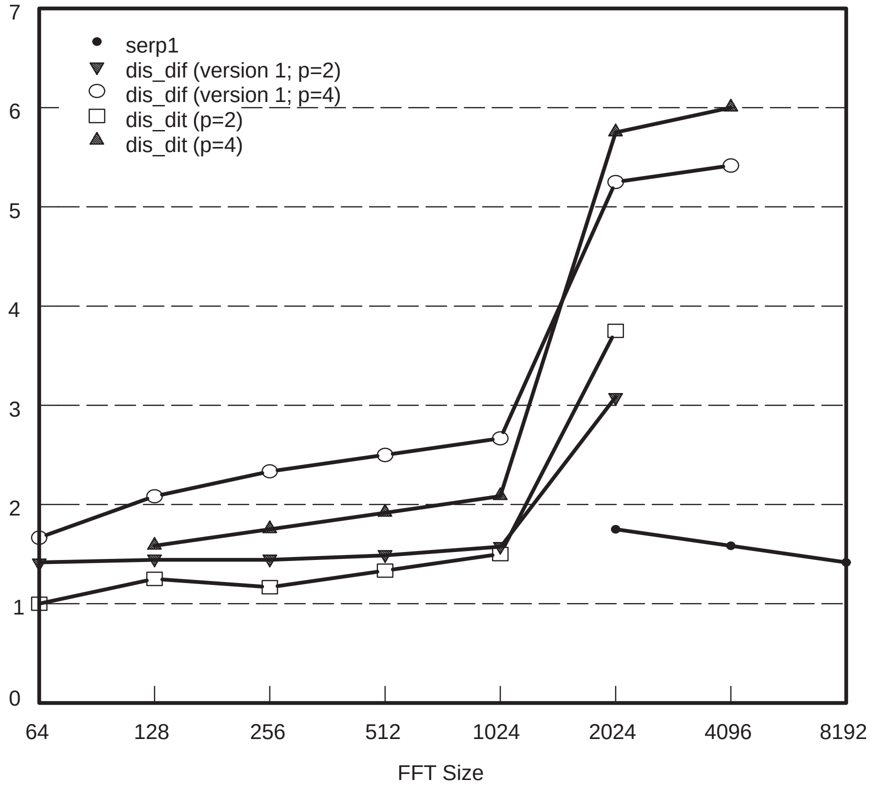
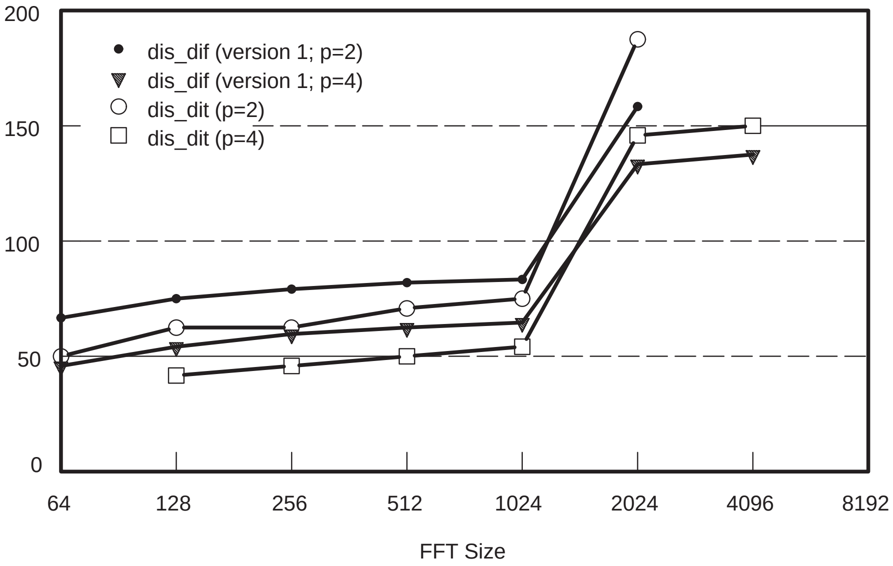


Figure 14. FFT Efficiency Vs. FFT Size



## 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
        LDPSINTAB
        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:  .cffttext : 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]
```

# Parallel 2-D FFT Implementation With TMS320C4x DSPs

**Rose Marie Piedra**
**Digital Signal Processing — Semiconductor Group**
**Texas Instruments Incorporated**

## Introduction

Fourier transform techniques are of fundamental importance in digital signal processing (DSP) applications. Among the most commonly used algorithms in image processing is the Fast Fourier Transform (FFT). FFT is used for computation of the Discrete Fourier Transform (DFT).

FFT computations can be used to solve image correlations and convolutions. Two-dimensional convolutions and correlations are used for feature extraction in image processing. For example, applications on fluid dynamics (2-D turbulences) can lead to calculation of velocity vectors and gradients. One important advantage of using frequency domain tools over direct methods is faster execution. The FFT algorithm significantly reduces the computation time of the DFT.

This application note compares serial and parallel implementations of 2-D complex FFTs with the TMS320C40 processor. Special attention is given to parallel implementation of 2-D FFTs. The increasing demands for speed and performance in some real-time DSP applications make sequential systems inadequate. Parallel systems provide higher throughput rates.

The algorithms were implemented 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:

***2-D FFT Algorithm*** Gives a brief review of the FFT algorithm and its extension to the 2-D case. Describes applications of FFTs in the calculation of correlation and convolution algorithms.

***Parallel 2-D FFT*** Focuses on parallel implementations of 2-D FFTs. Shared- and distributed-memory implementations are considered, as well as the TMS320C40's suitability for each.

***TMS320C40 Implementation*** Presents the results of shared- and distributed-memory implementations of parallel 2-D FFT realized on the PPDS. Gives analyses of speed-up and efficiency.

***Conclusion*** States conclusions.

***Appendixes*** Lists the code for performing serial and parallel 2-D FFTs in C and 'C40 assembly language code.

## The 2-D FFT Algorithm

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 $0 \le k \le n - 1$ and $W_n = e^{-j\,2\frac{\pi}{n}}$.

Direct DFT computation requires $O\,(n^2)$ arithmetic operations. A faster method of computing the DFT is the Fast Fourier Transform (FFT) algorithm. 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 $\frac{n}{2}\log_2 n \approx O\ (n \log_2 n)$ arithmetic operations (which is $\frac{n}{\log_2\ n}$ times faster than direct DFT computation). See [3], [6], and [4] for a more detailed analysis of the 1-D FFT case.

Two-dimensional DFT can be defined in a manner similar to the 1-D case [10]. The 2-D DFT is given by:

$$X\ (k_{1,}k_{2)}\ =\ \sum_{i_1=0}^{n-1}\sum_{i_2=0}^{n-1} x\ (\ i_{1,}i_2\ )\ W_n{}^{(\,i_1k_1\ +\ i_2k_2\,)}$$

where $0\ \leq\ k_1, k_2\ \leq\ n\ -\ 1$ and $W_n\ =\ e^{-j\,2\frac{\pi}{n}}$.

A standard approach to computing the 2-D FFT of a matrix A is to perform a 1-D FFT on the rows of A, giving an intermediate matrix A', then performing a 1-D FFT on the columns of A'[10]. This is the approach followed in this application report.

### Timing Analysis

A 2-D FFT of a complex matrix of size ($n \times n$) requires the execution of a 1-D FFT on $n$ rows, followed by a 1-D FFT on $n$ columns. The number of arithmetic operations required will therefore be as follows:

$$Time\ =\ n\ *\ O\ (n\ \log_2\ n)\ +\ n\ *\ O\ (n\ \log_2\ n)\ \approx\ O\ (n^2\ \log_2\ n)$$

(FFT on $n$ rows)        (FFT on $n$ columns)

### Application of FFT on Correlation/Convolution Algorithms

Relationships between image and transform domains can be described by convolution and correlation. Convolution is used for linear interpolation or filtering. Correlation plays an important role in feature extraction in image processing. These image operations are computationally intensive; Fourier transforms can be used to enhance speed.

The correlation of two sequences *x(i)* and *y(i)* of length *n* is defined as[10]:

$$w(i)\ =\ \sum_{k=0}^{n-1} x\ (k)\ y\ (i\ +\ k)$$

For a 1-D correlation, the common direct approach (in time domain) based on shift-and-multiply operations requires O ($n^2$) arithmetic operations.

Based on the convolution property of the Fourier transform [3], an efficient way to compute correlation is by using FFT and inverse FFT (IFFT), as illustrated below:

1. Compute FFT{*x(i)*} and FFT{*y(i)*}.
2. Multiply FFT{*x(i)*} by the complex conjugate of FFT{*y(i)*}.
3. Compute the IFFT of this result.

Similarly, a convolution operation reduces to a simple multiplication in the Fourier domain. FFT correlation/convolution becomes computationally faster than spatial convolution for large images. Speed-up is approximately $\frac{n^2}{n \log_2 n} = \frac{n}{\log_2 n}$, which is significant when dealing with very large images.

## Parallel 2-D FFT

A 2-D FFT is an intrinsically parallel operation; a 1-D FFT is applied separately to each row and column of a matrix.

### The Parallel Algorithm

Let $n = qp$, where $n$ is the order of the squared matrix A, $p$ is the number of processors, and $q \geq 1$ is an integer. The basic idea is to allocate a unique working set of rows/columns to each processor. The algorithm consists of three basic steps:

**Step 1. FFT on rows:** Processor $i$ executes 1-D sequential FFT on rows $qi$, $qi$+1, ... , $qi$+$q$-1, with $i$ = 0,1,...,$p$-1. Because each processor executes a 1-D FFT on $q$ different rows, this step requires $q * O\ (n \log_2 n) \approx O\ (\frac{n^2}{p} \log_2 n)$ arithmetic operations.

**Step 2. Matrix Transposition:** Because column elements are not stored in contiguous memory location, row/column transposition of matrix A is necessary prior to executing FFT on columns. Matrix transposition requires $(n - 1) + (n - 2) + ... + 1 = \frac{n\ (n - 1)}{2}$ exchange steps, where $n$ is the order of the matrix [7]. The computation delay involved in this operation is therefore $O\ (n^2)$ for serial execution or $O\left(\frac{n^2}{p}\right)$ for parallel execution.

**Step 3. FFT on columns:** Same as in Step 1, but by column.

### Speed-Up Analysis

The speed-up factor can be calculated as follows*:*

$$Speed\text{-}up \approx \frac{n^2 \log_2 n}{\frac{n^2}{p} \log_2 n} \approx O(p)$$

The parallelism in the 2D-FFT is suitable for implementation on distributed-memory or shared-memory multiprocessors. Let’s consider those cases.

### Shared-Memory Implementation

Matrix A is stored in global memory, so each processor has easy access to all the rows/columns. Even when all processors share the same physical data memory, each processor points to a different row/column working set.

Shared-memory systems require careful consideration of the memory-contention problem. Matrix transposition (Step 2) is simpler, but row/column access can create a major bottleneck.

Shared-memory implementation requires at least $2 * n * n = 2\, n^2$ words of shared memory. If this amount of RAM is unavailable in the system, consider either intermediate downloading of files or distributed-memory implementation as an alternative.

Figure 1 illustrates the shared-memory 2-D FFT implementation for $p = 4$ and $n = 8$.

### Distributed-Memory Implementation

Matrix A is partitioned into $p$ regions. Each region contains $q$ rows and is assigned to each processor's local memory. Processors communicate via message passing.

- Steps 1 and 3 of the parallel 2-D FFT algorithm are executed in the local memory of each processor. No interprocessor communication is necessary.
- Step 2, matrix transposition, is more complex because matrix A is distributed between processors by row. You must perform message passing of row segments before you execute matrix transposition. This procedure can be described as follows:
  - ***Total-exchange* step**: Processor $i$ sends to processor $j$ columns $qj, qj + 1, \dots , qj + q - 1$ of each of the rows allocated to it, where $0 \le j < p$ and $i \neq j$. In such DMA-supported devices as the TMS320C40, this step can be executed simultaneously with Step 1, after computation of each row FFT. Better DMA-CPU parallelism can thus be achieved. This is the approach followed in the parallel 2-D FFT (distributed-memory implementation) presented in this application report.
  - ***Transposition of submatrices***: After the *total exchange* step, each processor contains all the column elements needed to perform a row/column transposition. Transposition is executed on $p$ squared submatrices of size $(q \times q)$. Submatrix $G_k$ contains elements $(kq + i, j)$, where $(0 \le i,\ j < q)$ and $(0 \le \mathrm{k} < \mathrm{p})$. The computation delay involved in this operation is $p\ O\ (q^2) = O\left(\frac{n^2}{p}\right)$.

Figure 2 illustrates Step 2 of the distributed-memory implementation, with $p = 4$ and $n = 8$.

- Memory requirements: Each processor requires at least $2n^2 / p$ words of local memory to store the $( n/p )$ rows allocated to it.
- Required topology: This implementation requires a fully connected multiprocessor configuration. Other configurations with rerouting capabilities are also feasible. Refer to [9] for information on *total exchange* techniques for configurations for cube, mesh, and linear arrays.
- Output result: Matrix results are stored by column, with column elements stored in successive memory locations in the local memory of each processor. Processor $i$ contains columns $qi, qi + 1, \dots , qi + q - 1$, where $i = 0,1,\dots,p - 1$.

### Figure 1.  2-D FFT Shared-Memory Implementation

**FFT on Rows:**

**Matrix A (Shared Memory)**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Processor 0 → | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| Processor 1 → | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| Processor 2 → | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| Processor 3 → | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

**FFT on Columns:**

**Matrix A (Shared Memory)**

| Processor 0<br>↓ | | Processor 1<br>↓ | | Processor 2<br>↓ | | Processor 3<br>↓ | |
|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

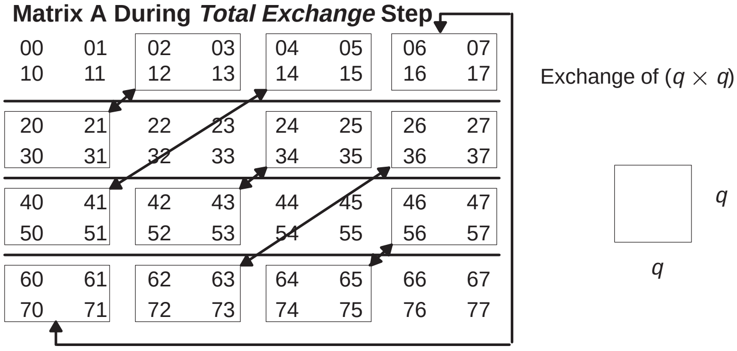
**Note:** Matrix element A[*i*][*j*] = *ij*
Matrix size = *n* = 8
Number of processors = *p* = 4

## Figure 2. 2-D FFT Distributed-Memory Implementation
## (Step 2: Transposition of Submatrices)

### Matrix A After Processors Have Completed FFTs on All the Rows

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | Processor 0 (rows 0,1) |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | Processor 1 (rows 2,3) |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | Processor 2 (rows 4,5) |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | Processor 3 (rows 6,7) |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | |

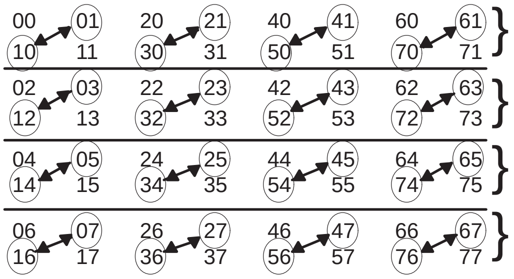
### Matrix A During *Total Exchange* Step



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

Exchange of ($q \times q$) submatrices

### Matrix A After *Total Exchange* Step and During Submatrix Transposition



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 20 | 21 | 40 | 41 | 60 | 61 | Processor 0 |
| 10 | 11 | 30 | 31 | 50 | 51 | 70 | 71 | |
| 02 | 03 | 22 | 23 | 42 | 43 | 62 | 63 | Processor 1 |
| 12 | 13 | 32 | 33 | 52 | 53 | 72 | 73 | |
| 04 | 05 | 24 | 25 | 44 | 45 | 64 | 65 | Processor 2 |
| 14 | 15 | 34 | 35 | 54 | 55 | 74 | 75 | |
| 06 | 07 | 26 | 27 | 46 | 47 | 66 | 67 | Processor 3 |
| 16 | 17 | 36 | 37 | 56 | 57 | 76 | 77 | |

### Matrix A After Transpositions of Submatrices (ready for FFT on columns)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 00 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | Processor 0 (columns 0,1) |
| 01 | 11 | 21 | 31 | 41 | 51 | 61 | 71 | |
| 02 | 12 | 22 | 32 | 42 | 52 | 62 | 72 | Processor 1 (columns 2,3) |
| 03 | 13 | 23 | 33 | 43 | 53 | 63 | 73 | |
| 04 | 14 | 24 | 34 | 44 | 54 | 64 | 74 | Processor 2 (columns 4,5) |
| 05 | 15 | 25 | 35 | 45 | 55 | 65 | 75 | |
| 06 | 16 | 26 | 36 | 46 | 56 | 66 | 76 | Processor 3 (columns 6,7) |
| 07 | 17 | 27 | 37 | 47 | 57 | 67 | 77 | |

## TMS320C40 Implementation

### The TMS320C40

The TMS320C40 is the world’s first parallel-processing DSP. In addition to a powerful CPU that can execute up to 11 operations per cycle with a 40- or 50-ns cycle time, the TMS320C40 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 TMS320C40 suitable for both distributed- and shared-memory computing systems [2].

The 2-D FFT algorithm was implemented on the TMS320C40 Parallel Processing Development System (PPDS). The PPDS includes four TMS320C40s, fully interconnected via the on-chip communication ports. Each ’C40 has 256KB of local memory SRAM, and all share a 512KB global memory [5].

General features of the programs:

- All the programs have been written to be independent of the FFT size and the number of processors in the system. Further optimization is possible for a fixed number of processors.
- Real and imaginary parts of complex numbers are stored in successive memory locations.
- Both C and assembly language versions of the programs are in the appendices. The programs can be downloaded from the TMS320 bulletin board at (713) 274-2323. Set your modem to 8 data bits,1 stop bit, no parity.
- For the C programs, there are core functions, such as the 1-D FFT and CPU complex moves, and routines to set DMA register values in assembly code to enhanced optimization. For the assembly programs, most of the functions are in-lined to avoid the delay associated with calling routines. But, in order to keep the program flexible, the 1D-FFT has been retained as a subroutine. The new ’C40 LAJ and BUD R11 instructions permit routine calls with just one delay cycle. To make the programs more generic, the (5/4)-cycle sine/cosine table [4] and the input matrix are provided in separate files.
- The radix-2 1-D FFT routine presented in Appendix D is used as the core for the 2-D FFT implementation. But you can use any FFT routine that complies with the calling conventions. Thus, as faster 1-D FFT algorithms are developed, they can be used to implement faster 2-D FFT algorithms.
- The ’C40 timer 0 and the timer routines in the parallel runtime support (PRTS) library, available with the ’C40 C compiler, 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 $T = max\,(T_i)$, where $T_i$ is the execution time taken by processor *i*. Note that in the programs, $T_i$ is the time between labels *t2* and *t0*.
- The load-imbalancing case was not considered. Refer to [2] for an example of this case.
- The compiler/assembler tools were run under OS/2 to avoid memory-limitation problems with the optimizer.

### Serial Implementation

Serial implementations of the 2-D FFT provide accurate speed-up measures for the parallel programs. Appendix A illustrates single- and double-buffered 2-D FFT serial implementations in C and ’C40 assembly code.

Observations:

- Although the 1-D FFT can be executed directly in off-chip RAM, the preferred method is to transfer the row/column to on-chip RAM first. This fully exploits the dual-access single-cycle characteristic of the ’C40 on-chip RAM for some parallel instructions. This transfer delay can be minimized with double-buffering techniques.
- Double-buffering technique:
  - CPU operations are confined to computing 1-D FFT (1 row at a time). The DMA processor performs the data transfer between on-chip RAM and external RAM, providing the CPU with a new set of data. This requires a continuous CPU-DMA synchronization.
  - While the CPU is computing FFT on row/column *i*, the DMA processor transfers the vector result of row/column (*i*-1) bit-reversed and the next row/column (*i*+1) from external RAM to on-chip RAM to have it ready for the next FFT computation. This technique is called double buffering.
  - The 2K × 32-bit-word on-chip RAM constantly holds 2 buffers. Each buffer must be located in a different on-chip RAM block. Because each on-chip RAM block has an independent bus path, CPU/DMA access conflict is minimized. You can compute up to 1K-point real FFT or 512-point complex FFT in on-chip memory. If the double-buffering technique is not used, the system can compute up to 2K-point real FFT or 1K-point complex FFT .
- For DMA bit-reversed complex transfers, you can use autoinitialization to transfer the real part of the FFT vector result first and the imaginary part later. You must set the “read bit-rev” bit (control register bit 12) to 1 and the source address index to the FFT size (*n*). Given the ’C40 architecture, no extra delay occurs with CPU/DMA bit-reversed addressing.
- For DMA column transfers, autoinitialization is also used to transfer the real and imaginary parts of each complex vector.
- Given the offset addressing capabilities of the ’C40, the transposition step requires no extra cycles when moving columns from off-chip to on-chip RAM.

### Shared-Memory Parallel Implementation

Appendix B contains single- and double-buffered versions of the 2-D FFT (shared-memory version) in C and ’C40 assembly code.

Observations:

- A node ID (my_node) is allocated by software to each processor. In this way, each processor automatically selects its associated row/column working set.
- Each row/column is initially transferred to on-chip memory to minimize memory access conflict among the processors. Using the DMA for double-buffering minimizes not only this access delay but also the effect of a nonzero wait-state global memory similar to that of the PPDS.
- Interprocessor synchronization is required before you execute FFT on columns. Synchronization is implemented via a counter flag in global memory. Every processor increments the counter by 1 after completing the execution on the rows allocated to it. In this way, the processors begin executing FFT on columns only after the counter equals *p* (number of processors).

- The transposition step that is necessary prior to executing FFT on columns is implemented simultaneously with the transfer of columns to on-chip memory, with no delay penalty.
- The global memory of the PPDS can contain a complex matrix with a maximum of 256 × 256 elements. Because of the need for an extra location for the synchronization counter, the program has been tested with a maximum of 128 × 128 elements.
- For benchmarking of shared-memory programs, a global start of all the processors is absolutely necessary; otherwise, the real-memory-access conflict resolution will not occur. To facilitate this process, a C-callable routine (*syncount.asm*) is provided in Appendix D for debugging systems lacking global start capability capability. Rotating priority for shared memory access should be selected by setting the PPDS LCSR register to 0x40.

### Distributed-Memory Parallel Implementation

Two distinct single-buffered 2-D FFT implementations were used:

- Use of DMA only for interprocessor communication (See *dis1.c* in Appendix C).
- Use of DMA for interprocessor communication and matrix transposition (See *dis2.c* and *dis2.asm* in Appendix C).

Observations:

- The six-channel DMA coprocessor is used for interprocessor communication during the *total-exchange* step as follows:
  - As soon as 1-D FFT is completed on a row, the DMA coprocessor will be in charge of transferring (*n/p*) complex points of this result already stored in local memory, from one processor to the other in *total exchange* fashion.
  - Each processor will transmit a total of $(n/p)(p–1) \approx O(n)$ complex numbers per row. In the PPDS, a memory-to-memory interprocessor transfer operation of an integer number requires seven clock cycles—four to transmit the 4-byte word, two to write it to/from memory, and one to set up the communication channel. The communication delay will therefore be approximately 7 * 2 * *n* = 14*n* clock cycles.
- Careful consideration of the communication delay involved is necessary to achieve true CPU-DMA parallelism. If the *total exchange* step requires more time than the 1-D FFT computation, the application will slow down.
- DMA channels are set in split mode [8] with source and destination synchronization using the ICRDY and OCRDY port signals, respectively. In this way, DMA will be interrupted when there is new data to read in the input FIFO. A value will be written to the output FIFO if the output FIFO is not full. Transferring is done in a linear fashion (not bit-reversed).
- Because the interprocessor communication occurs in an *exchange* fashion, no extra memory is needed for temporary buffers. Source address and destination addresses are set to the same address values. Destination node IDs help to determine the location of the data to be exchanged.

  Data will never overlap, because the communication port FIFOs act as data buffers, **as long as the communicating processors start executing the exchange.asm routine at approximately the same time**. This can be achieved by using a common system reset or the parallel debugger manager (PDM), which is part of the ’C4x emulator.

  For systems without common reset, use the *exch2.asm* routine instead of *exchange.asm*. The *exch2.asm* routine can be downloaded from the TMS320 bulletin board at (713) 274-2323. Set your modem to 8 data bits,1 stop bit, no parity.

- The destination node is selected in such a way that each pair of processors are synchronized to *talk* to each other at approximately the same moment, thus facilitating communication scheduling and avoiding a system lock that could occur if a processor sent data with no processor ready to receive it. You select the destination node by using a XOR operation: (*my_node*) XOR ($i$), $0 < i < p$. In the case of a 4-processor system, the following situation exists during the first step ($i$ = 1):

  Processor 0 : *(my_node* = 0*) XOR* 1 = 1

  Processor 1 : *(my_node* = 1*) XOR* 1 = 0

  Processor 2 : *(my_node* = 2*) XOR* 1 = 3

  Processor 3 : *(my_node* = 3*) XOR* 1 = 2

  Processors 0,1 and 2,3 select each other for the first data exchange. Similar analysis can be done for the other steps.

- Matrix *port* is the connectivity matrix and shows the connectivity between the processors. Processor $i$ is connected to processor $j$ through *port*[$i$][$j$]. This matrix is the only system-specific part of the program.

- To attain as even a transmission as possible between processors, shifting priority among DMA channels has been selected.

- To synchronize DMA/CPU operation, each processor must know whether transferring is complete on a DMA channel before initializing the DMA with a new set of values. In unified mode, you can check for completion either by determining whether the start bits (DMA control register) are set to 10 or by checking the IIF register (if TCC was previously set to 1 in the DMA control register). In split mode, even when the corresponding bit in the IIF register is set, there is no guarantee that transfer is complete on both the primary and secondary channels. For this reason, the preferred method is to determine whether the start/aux_start bits (DMA control register) are both set to 10. In the programs, both methods have been used.

- DMA can be used for matrix transposition also. While the CPU is performing a 1-D column FFT (in on-chip RAM), the DMA is doing the next row/column transposition in off-chip memory.

- Double-buffered distributed-memory implementations were not described, but the approach is similar for shared memory.

### Implementation Results

The following 2-D FFT programs were implemented and tested on the ’C40 PPDS (see Appendices for source code):

- SER.C/SER.ASM: Single-buffered serial implementations (C/assembly language code versions)

- SERB.C/SERB.ASM: Double-buffered serial implementations (C/assembly language code versions)

- SH.C/SH.ASM: Single-buffered shared-memory implementations (C/assembly language code conversions)

- SHB.C/SHB.ASM: Double-buffered shared-memory implementations (C/assembly language code versions)

- DIS1.C: Distributed-memory implementation, with DMA being used only for interprocessor communication. (C code version)
- DIS2.C/DIS2.ASM: Distributed-memory implementation, with DMA being used for interprocessor communication and matrix transposition (C/assembly code versions).

Speed-up of a parallel algorithm is defined as $S_p = T_s/T_p$ , where $T_s$ is the serial time ($p$ = 1) and $T_p$ is the time of the algorithm executed using $p$ processors. Efficiency is defined as $E_p = S_p / p$ , where $0 < E_p < 1$ [2]. An efficiency below 50% reflects poor parallel implementation performance. Figure 4 through Figure 13 show speed-up and efficiency figures obtained for the shared- and distributed-memory programs implemented on the PPDS. The figures are based on the TMS320C40 2-D FFT timing benchmarks shown in Table 1. Execution time for program $i$ is denoted as $T_i$.

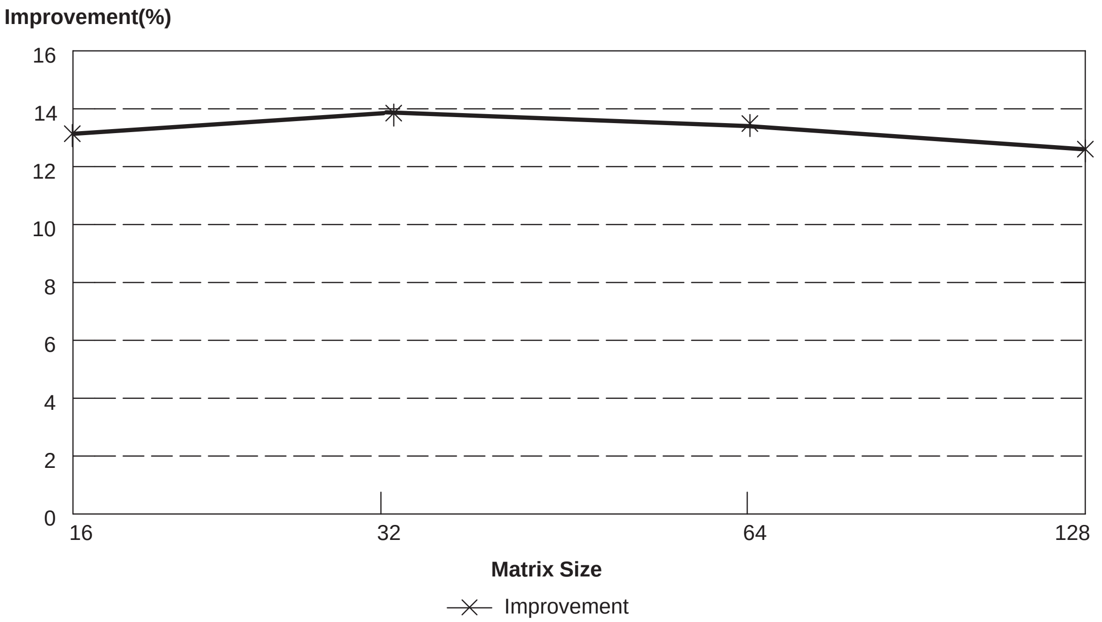
The following analysis shows the effect of the matrix size and the number of processors in the system.

- Serial implementations:
  - There was a 13% improvement using double-buffering in the serial program. (See Figure 3)
  - Using registers to pass function parameters had a positive effect on the performance of the C implementations (using assembly language core functions). As seen in Table 1, the timing difference between C and assembly code is minimal for large matrices.
- Shared-memory implementations:
  - The double-buffered shared-memory implementation displays a better performance than the single-buffered shared-memory version for $p$ = 2 (see Figure 7). The DMA helps to reduce the data transfer delay. For $p$ = 4, however, the performance declines because of the increase in shared-memory arbitration. In this case, the single-buffered shared-memory implementation is more beneficial (see Figure 11).
  - Shared-memory programs are strongly affected by the design of the shared-memory arbitration unit. For example, in the case of the PPDS, a processor will not release access to shared-memory during back-to-back reads. The speed of the single-buffered shared-memory implementation is thus increased because of the reduction in the delay penalty for continuous switching.
  - Efficiency decreases with more processors because the memory conflict delay increases. This effect can be seen in Figure 13, where efficiency figures are plotted against the number of processors in the system.
- Distributed-memory implementations:

  The distributed-memory implementations show an excellent performance. Speed-up/efficiency for large matrices is high, and the decline in the efficiency when the number of processors increases is very slight (Figure 13). Performance also improves when DMA is used to help with the combined task of matrix transposition and interprocessor communication. See Figure 4 and Figure 5.

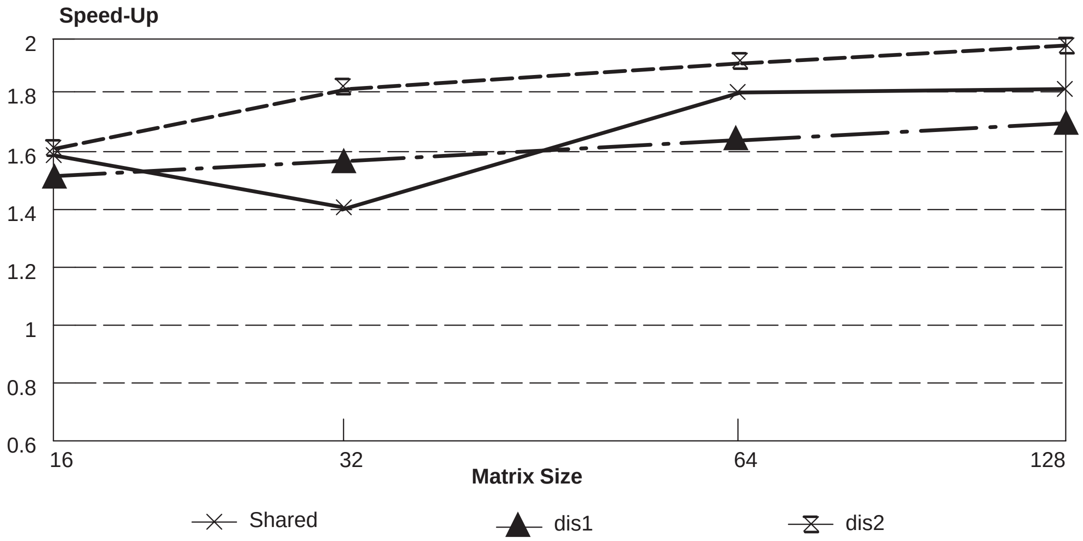
  Table 1 and Table 2 show the TMS320C40 2-D FFT timing benchmarks. Data I/O is not considered, because it is host-computer dependent.

## Figure 3. Double-Buffering Performance Analysis (Serial Program)



$$\text{Percentage of Improvement} = \frac{\text{Tser.asm} - \text{Tserb.asm}}{\text{Tser.asm}} \times 100\%$$

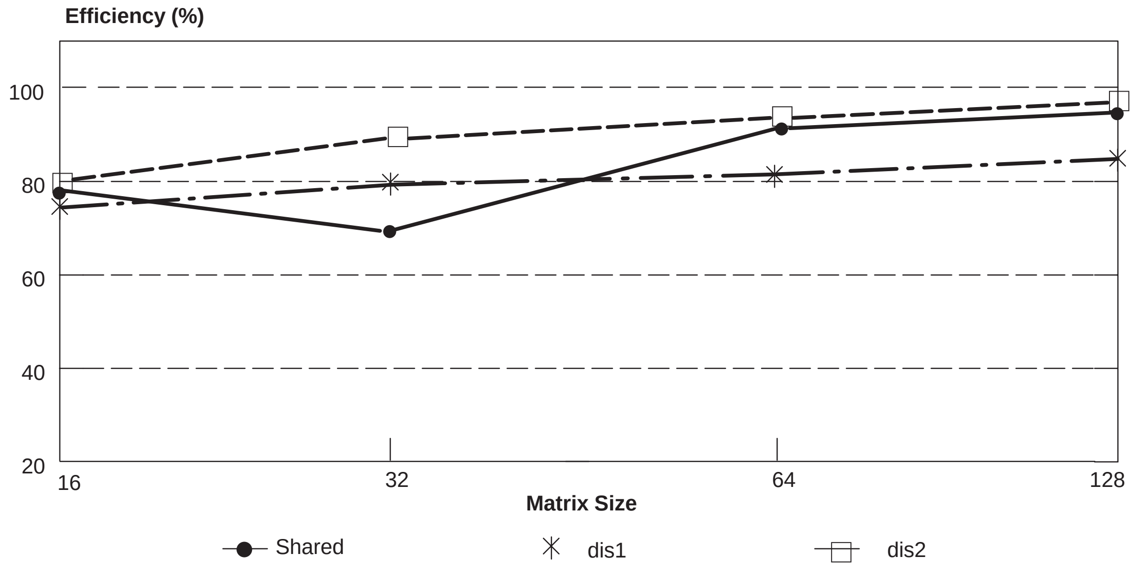
## Figure 4. Speed-Up Vs. Matrix Size (p = 2) Over Single-Buffered Serial Program



$$Sp = \frac{\text{Tser.asm}}{\text{Tsh.asm}} \qquad Sp = \frac{\text{Tser.c}}{\text{Tdis1.c}} \qquad Sp = \frac{\text{Tser.asm}}{\text{Tdis2.asm}}$$

**Note**: Number of processors = p = 2

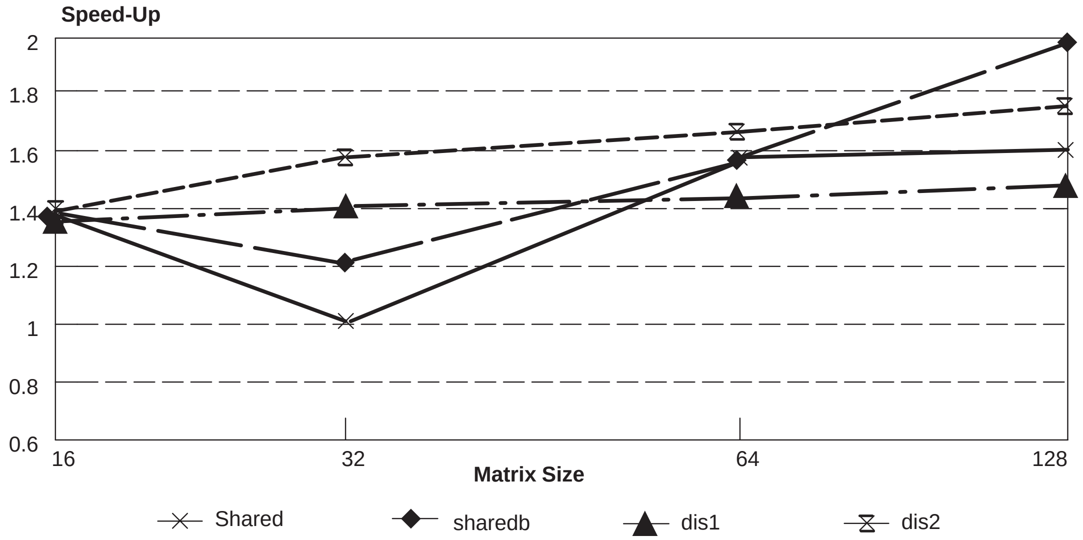
Figure 5. Efficiency Vs. Matrix Size (p = 2) Over Single-Buffered Serial Program



$$Eff = \frac{Tser.asm}{Tsh.asm * p} \qquad Eff = \frac{Tser.c}{Tdis1.c * p} \qquad Eff = \frac{Tser.asm}{Tdis2.asm * p}$$

**Note**: Number of processors = p = 2

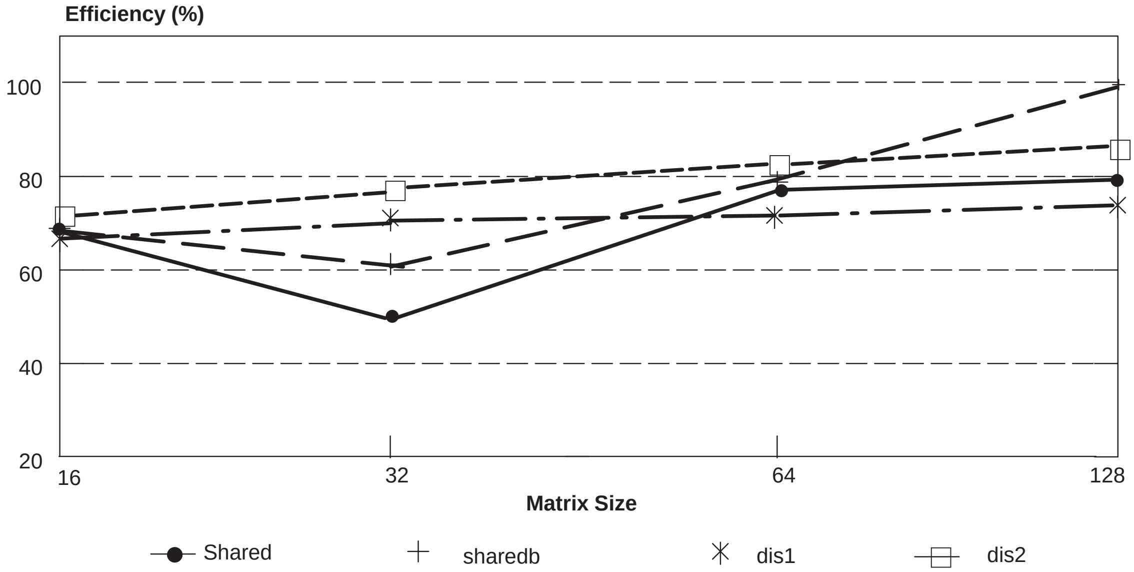
Figure 6. Speed-Up Vs. Matrix Size (p = 2) Over Double-Buffered Serial Program



$$Sp = \frac{Tserb.asm}{Tsh.asm} \qquad Sp = \frac{Tserb.asm}{Tshb.asm} \qquad Sp = \frac{Tserb.c}{Tdis1.c} \qquad Sp = \frac{Tserb.asm}{Tdis2.asm}$$

**Note**: Number of processors = p = 2

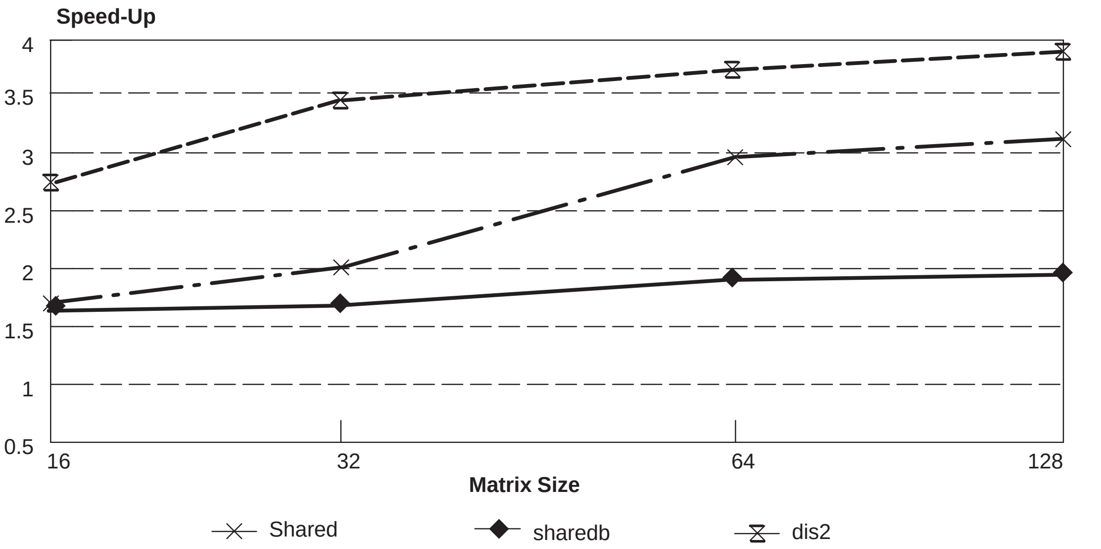
**Figure 7. Efficiency Vs. Matrix Size (p = 2) Over Double-Buffered Serial Program**



$$Eff = \frac{Tserb.asm}{Tsh.asm * p} \qquad Eff = \frac{Tserb.asm}{Tshb.asm * p} \qquad Eff = \frac{Tserb.c}{Tdis1.c * p} \qquad Eff = \frac{Tserb.asm}{Tdis2.asm * p}$$

**Note**: Number of processors = p = 2

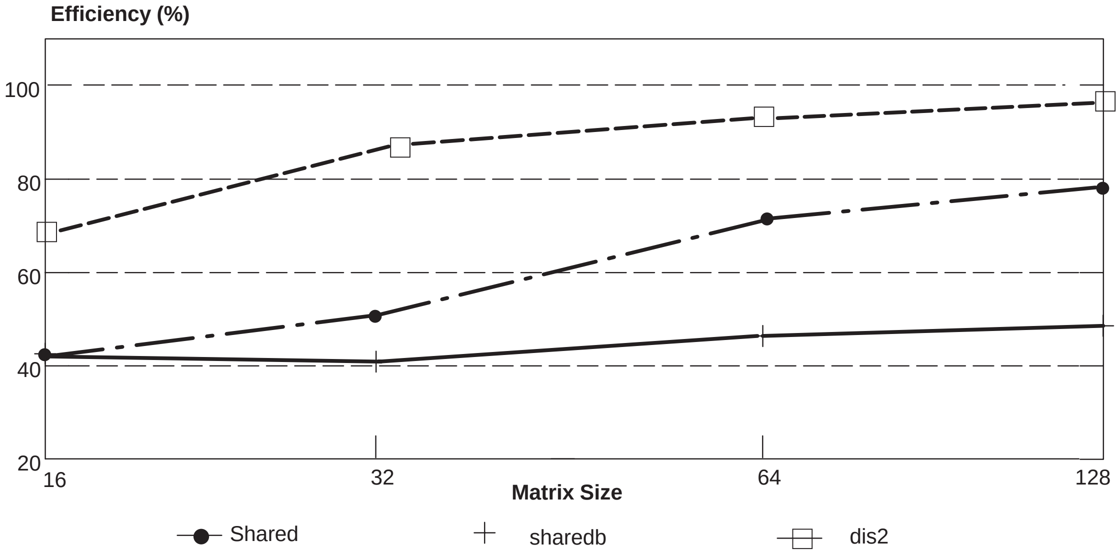
**Figure 8. Speed-Up Vs. Matrix Size (p = 4) Over Single-Buffered Serial Program**



$$Sp = \frac{Tser.asm}{Tsh.asm} \qquad Sp = \frac{Tser.asm}{Tshb.asm} \qquad Sp = \frac{Tser.c}{Tdis2.c}$$

**Note**: Number of processors = p = 4

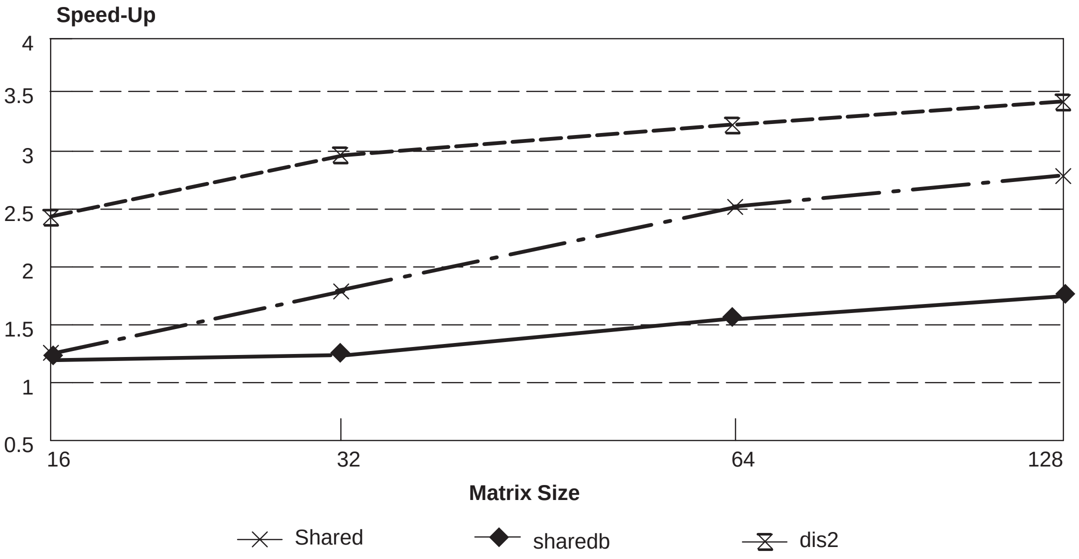
**Figure 9. Efficiency Vs. Matrix Size (p = 4) Over Single-Buffered Serial Program**



$$\text{Eff} = \frac{\text{Tser.asm}}{\text{Tsh.asm} * p} \qquad \text{Eff} = \frac{\text{Tser.asm}}{\text{Tshb.asm} * p} \qquad \text{Eff} = \frac{\text{Tser.asm}}{\text{Tdis2.asm} * p}$$

**Note**:Number of processors = p = 4

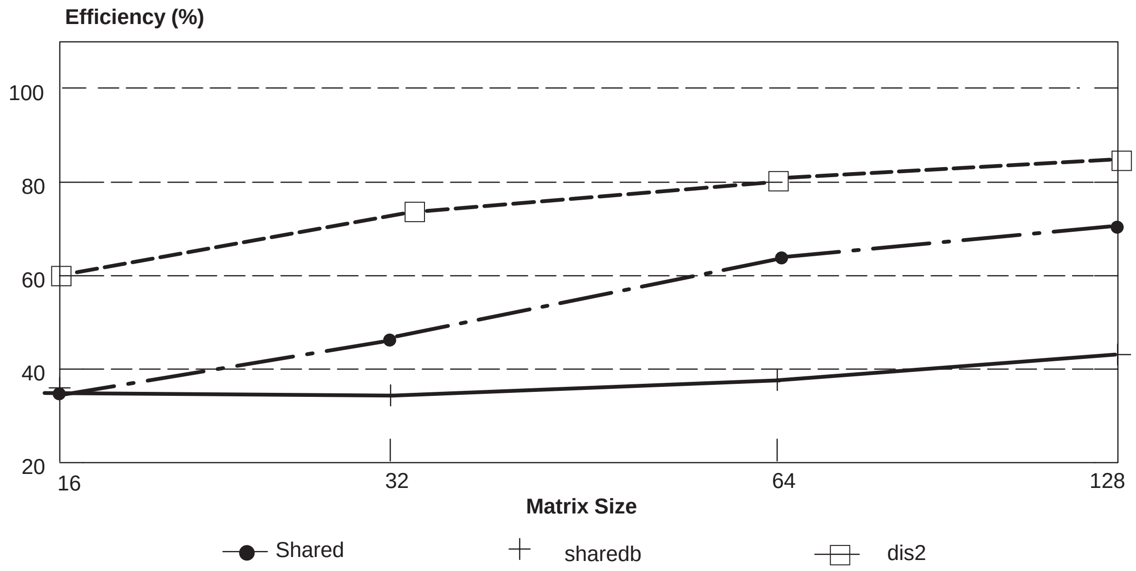
**Figure 10. Speed-Up Vs. Matrix Size (p = 4) Over Double-Buffered Serial Program**



$$\text{Sp} = \frac{\text{Tserb.asm}}{\text{Tsh.asm}} \qquad \text{Sp} = \frac{\text{Tserb.asm}}{\text{Tshb.asm}} \qquad \text{Sp} = \frac{\text{Tserb.asm}}{\text{Tdis2.asm}}$$

**Note**: Number of processors = p = 4

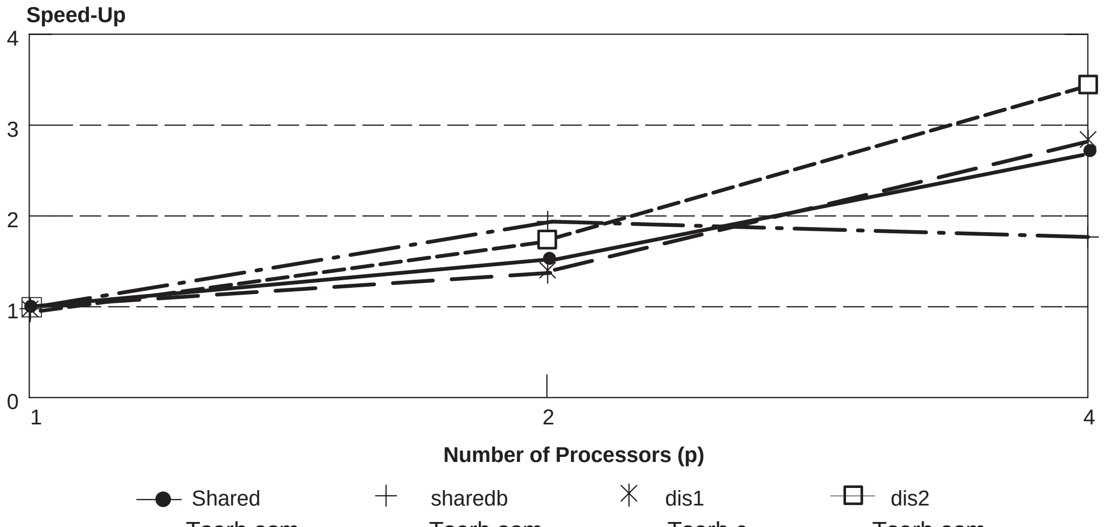
## Figure 11. Efficiency Vs. Matrix Size (p = 4) Over Double-Buffered Serial Program



$$Eff = \frac{Tserb.asm}{Tsh.asm * p} \qquad Eff = \frac{Tserb.asm}{Tshb.asm * p} \qquad Eff = \frac{Tserb.asm}{Tdis2.asm * p}$$

**Note**: Number of processors = p = 4

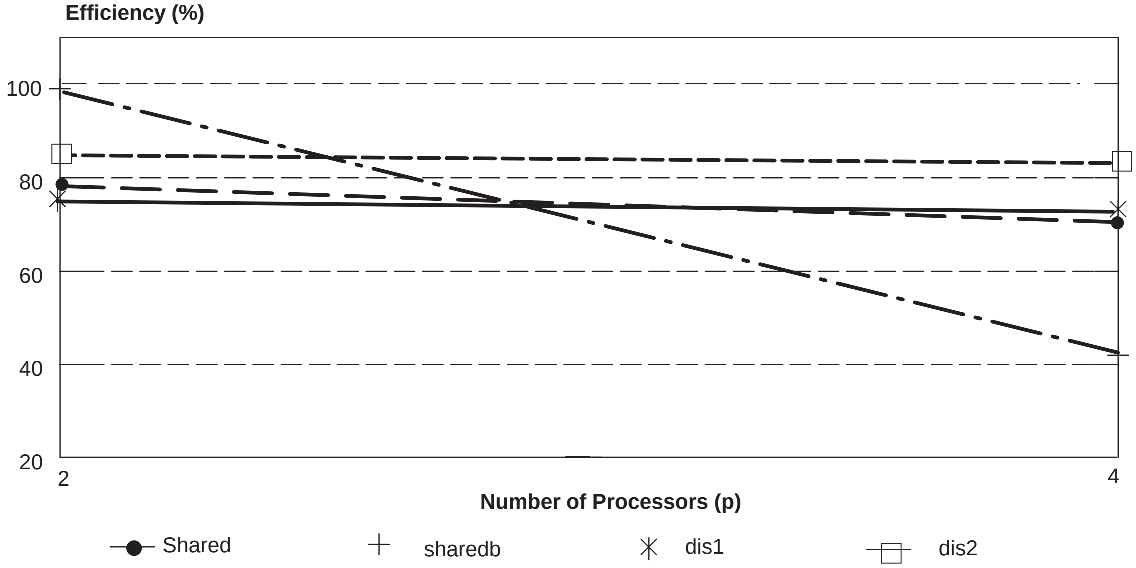
## Figure 12. Speed-Up Vs. Number of Processors Over Double-Buffered Serial Program



$$Sp = \frac{Tserb.asm}{Tsh.asm} \qquad Sp = \frac{Tserb.asm}{Tshb.asm} \qquad Sp = \frac{Tserb.c}{Tdis1.c} \qquad Sp = \frac{Tserb.asm}{Tdis2.asm}$$

**Note**: Matrix Size = 128

## Figure 13. Efficiency Vs. Number of Processors Over Double-Buffered Serial Program



$$\text{Eff} = \frac{\text{Tserb.asm}}{\text{Tsh.asm} * p} \qquad \text{Eff} = \frac{\text{Tserb.asm}}{\text{Tshb.asm} * p} \qquad \text{Eff} = \frac{\text{Tserb.c}}{\text{Tdis1.c} * p} \qquad \text{Eff} = \frac{\text{Tserb.asm}}{\text{Tdis2.asm} * p}$$

**Note**: Matrix Size = 128

## Table 1. TMS320C40 2-D FFT Timing Benchmarks (in Milliseconds)

| Program | Number of Processors | Matrix Size in Complex Numbers | | | |
|---|---|---|---|---|---|
| | | 16 × 16 | 32 × 32 | 64 × 64 | 128 × 128 |
| SER.ASM | (*p* = 1) | 0.706 | 3.115 | 13.812 | 61.138 |
| SERB.ASM | (*p* = 1) | 0.614 | 2.682 | 11.959 | 53.498 |
| SER.C | (*p* = 1) | 0.773 | 3.248 | 14.078 | 61.671 |
| SERB.C | (*p* = 1) | 0.671 | 2.794 | 12.183 | 53.944 |
| SH.ASM | (*p* = 2) | 0.442 | 2.217 | 7.737 | 33.564 |
| SHB.ASM | (*p* = 2) | 0.441 | 1.850 | 7.550 | 26.800 |
| DIS2.ASM | (*p* = 2) | 0.438 | 1.707 | 7.200 | 31.150 |
| DIS1.C | (*p* = 2) | 0.504 | 2.020 | 8.467 | 36.231 |
| DIS2.C | (*p* = 2) | 0.448 | 1.744 | 7.266 | 31.270 |
| SH.ASM | (*p* = 4) | 0.424 | 1.496 | 4.757 | 19.196 |
| SHB.ASM | (*p* = 4) | 0.421 | 1.880 | 7.692 | 31.104 |
| DIS2.ASM | (*p* = 4) | 0.255 | 0.902 | 3.693 | 15.750 |

**Note**: The data in this table was obtained with the complex FFT routine in Appendix D.

- ’C40 instruction cycle, Tcycle = 40 ns
- C compiler optimization level : o2

**Table 2. TMS320C40 2-D FFT Timing Benchmarks (in Milliseconds)**

| Program | Number of Processors | Matrix Size in Complex Numbers | | |
|---|---|---|---|---|
| | | 32 × 32 | 64 × 64 | 128 × 128 |
| SER.ASM | (*p* = 1) | 2.080 | 9.418 | 40.390 |
| SERB.ASM | (*p* = 1) | 1.791 | 8.146 | 35.340 |
| SH.ASM | (*p* = 2) | 1.478 | 5.274 | 22.172 |
| SHB.ASM | (*p* = 2) | 1.234 | 5.147 | 17.825 |
| DIS2.ASM | (*p* = 2) | 1.140 | 4.910 | 20.586 |
| DIS2.ASM | (*p* = 4) | 0.602 | 2.518 | 10.410 |

**Note**: This table gives expected values using the faster version complex Radix-2 DIT FFT routine in Example 12-44 of the *TMS320C4x User's Guide* (1993). Tcycle = 40 ns.

## Conclusion

This report has presented shared- and distributed-memory 2-D FFT parallel implementations. High speed-up/efficiency has been attained. Parallelization of the 2-D FFT is important when dealing with large matrices. For small matrices, a serial implementation is more convenient.

A virtually unlimited number of parallel algorithms can be implemented in ’C40-based systems. Parallel implementations of 1-D FFT can be found in [1]. These require cube/mesh mapping techniques that can also be implemented in a parallel system, such as the PPDS.

# References

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

[2] R. M. Piedra. *A Parallel Approach for Solving Matrix-Multiplication on the TMS320C4x DSP*. Dallas, Texas: Texas Instruments, Incorporated, 1991.

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

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

[5] D.C. Chen and R. H. Price. "A Real-Time TMS320C40 Based Parallel System for High Rate Digital Signal Processing." *Proceedings of ICASSP 91*, USA, Volume 2, page 1573, May 1991.

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

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

[8] *TMS320C4x User's Guide*. Dallas, Texas: Texas Instruments, Incorporated, 1991.

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

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

[11] H. Kunieda and K. Itoh. "Parallel 2D-FFT Algorithm on Practical Multiprocessor Systems." *Proceedings of the 3rd Transputer/Occam International Conference*, May 1990.

# Appendices

**Appendix A:  Serial Implementations of 2-D FFT**

- A.1  SER.C: Single-Buffered Implementation (C Program)
- A.2  SER.ASM: Single-Buffered Implementation (’C40 Assembly Program)
- A.3  SERB.C: Double-Buffered Implementation (C Program)
- A.4  SERB.ASM: Double-Buffered Implementation (’C40 Assembly Program)

**Appendix B:  Parallel 2-D FFT (Shared-Memory Version)**

- B.1  SH.C: Single-Buffered Implementation (C Program)
- B.2  SH.ASM: Single-Buffered Implementation (’C40 Assembly Program)
- B.3  SHB.C: Double-Buffered Implementation (C Program)
- B.4  SHB.ASM: Double-Buffered Implementation (’C40 Assembly Program)

**Appendix C:  Parallel 2-D FFT  (Distributed-Memory Version)**

- C.1  DIS1.C: Distributed-Memory Implementation (C Program) — DMA Used Only for Interprocessor Communication
- C.2  DIS2.C: Distributed-Memory Implementation (C Program) — DMA Used for Interprocessor Communication and Matrix Transposition
- C.3  DIS2.ASM: Distributed-Memory (’C40 Assembly Program) — DMA Used for Interprocessor Communication and Matrix Transposition

**Appendix H:  Mylib.lib Routines**

- D.1  CFFT.ASM: Assembly Language FFT Routine
- D.2  CFFTC.ASM: Assembly Language FFT Routine (C-Callable)
- D.3  CMOVE.ASM: Complex-Vector Move Routine
- D.4  CMOVEB.ASM: Complex-Vector Bit-Reversed Move Routine
- D.5  SET_DMA.ASM: Routine to Set DMA Register Values
- D.6  EXCHANGE.ASM: Routine for Interprocessor Communication
- D.7  SYNCOUNT.ASM: Interprocessor Synchronization Routine

# Appendix A: Serial Implementations of 2-D FFT

## A.1. SER.C: Single-Buffered Implementation (C Program)

### *SER.C*

```
/*************************************************************************************

SER.C :  Serial 2–dimensional complex FFT (Single–buffered version)

To run:

cl30 –v40 –gs –mr –o2 ser.c
asm30 –v40 –s sintab.asm
asm30 –v40 –s input.asm
lnk30 serc.cmd

*************************************************************************************/
#define  SIZE    4                           /* FFT size (n)                        */
#define  LOGSIZE 2                           /* log(FFT size)                       */
#define  BLOCK0  0x002ff800                  /* on–chip RAM buffer                  */

extern   void    cfftc(),                    /* C–callable complex FFT              */
         cmove(),                            /* CPU complex move                    */
         cmoveb();                           /* CPU bit–reversed move               */

extern   float MATRIX[SIZE][SIZE*2];         /* Input matrix                        */

float    *block0      = (float *)BLOCK0,
         *MM[SIZE];

int      size2        = 2*SIZE,
         i;
int      tcomp;                              /* for benchmarking                    */
/***************************************************************************/
main()
{
asm(” or 1800h,st”);                         /* cache enable                        */
for (i=0;i<SIZE;i++) MM[i]=MATRIX[i];        /* accessing assembly variables        */

t0: time_start(0);                            /* start timer 0 for benchmark        */
/************************** Step 1: FFT by rows  ***************************/
for (i=0;i<SIZE;i++) {
    cmove (&MM[i][0],block0,2,2,SIZE);        /* move row(i) to on–chip mem.        */
    cfftc(block0,SIZE,LOGSIZE);               /* FFT on row (i)                     */
    cmoveb(block0,&MM[i][0],SIZE,2,SIZE);     /* move FFT{row(i)} to off–chip mem.  */
    }

/********************** Step 2: FFT by columns **************************************/
t1:
for (i=0;i<size2;i+=2) {
    cmove (&MM[0][i],block0,size2,2,SIZE);    /* move column (i) to on–chip mem.    */
    cfftc(block0,SIZE,LOGSIZE);               /* FFT on column (i)                  */
    cmoveb (block0,&MM[0][i],SIZE,size2,SIZE);/* move FFT {col. (i) to off–chip mem */
    }
tcomp= time_read(0);
t2: ;
} /*main*/
```

### SERC.CMD

```
–c                                                     /* LINK USING C CONVENTIONS  */
ser.obj
sintab.obj
input.obj
–stack 0x0040
–lrts40.lib                                            /* GET RUN–TIME SUPPORT      */
–lprts40r.lib
–lmylib.lib
–m serc.map
–o serc.out

MEMORY
{
    ROM:    org = 0x00          len = 0x1000
    RAM0:   org = 0x002ff800    len = 0x0400
    RAM1:   org = 0x002ffc00    len = 0x0400
    LM:     org = 0x40000000    len = 0x10000          /* LOCAL MEMORY              */
    GM:     org = 0x80000000    len = 0x20000          /* GLOBAL MEMORY             */
}

SECTIONS
{
    .text:  () > RAM1                                  /* CODE                      */
    .cinit: {) > RAM1                                  /* INITIALIZATION TABLES     */
    .stack: {} > RAM1                                  /* SYSTEM STACK              */
    .bss:   {} > RAM1                                  /* GLOBAL & STATIC VARS      */
    .data:  {} > RAM1                                  /* Sine tables               */
    INPUT:  {} > LM                                    /* Input matrix              */
}
```

## INPUT.ASM

```
***********************************************************
*
* INPUT.ASM : input matrix 4 x 4 for serial/shared program
*
***********************************************************

        .global   _MATRIX
        .sect     ”INPUT”

_MATRIX

        .float 130.0,90.0    ;[0][0] :   output 2264.0  ,   2288.0
        .float 66.0,230.0    ;[0][1] :   output –56.0   ,   –532.0
        .float 205.0,136.0   ;[0][2] :   output –50.0   ,   378.0
        .float 15.0,187.0    ;[0][3] :   output –182.0  ,   –22.0

        .float 150.0,164.0   ;[1][0] :   output –401.0  ,   227.0
        .float 222.0,44.0    ;[1][1] :   output –353.0  ,   1.0
        .float 95.0,243.0    ;[1][2] :   output 423.0   ,   –373.0
        .float 80.0,60.0     ;[1][3] :   output 167.0   ,   229.0

        .float 97.0,36.0     ;[2][0] :   output –68.0   ,   –26.0
        .float 215.0,191.0   ;[2][1] :   output 106.0   ,   –176.0
        .float 209.0,239.0   ;[2][2] :   output 418.0   ,   –636.0
        .float 161.0,22.0    ;[2][3] :   output –616.0  ,   –266.0

        .float 117.0,238.0   ;[3][0] :   output –131.0  ,   83.0
        .float 203.0,44.0    ;[3][1] :   output 175.0   ,   319.0
        .float 104.0,187.0   ;[3][2] :   output 225.0   ,   –133.0
        .float 195.0,177.0   ;[3][3] :   output 159.0   ,   79.0
        .end
```

## SINTAB.ASM

```
*****************************************************************
*
*   SINTAB.ASM : Table with twiddle factors for a 4–point CFFT
*                and data input. File to be linked with the
*                source code for a 4–point, radix–2 FFT.
*
*****************************************************************

         .global   SINE
         .global   N
         .global   M

N        .set      4
M        .set      2

         .data

SINE     .float    0.000000
COSINE   .float    1.000000
         .float    –0.000000
         .float    –1.000000
         .float    0.000000

         .end
```

## A.2. SER.ASM: Single-Buffered Implementation (’C40 Assembly Program)

### *SER.ASM*

```
*********************************************************
*
*    SER.ASM : TMS320C40 complex 2D–FFT serial program
*               (Single–buffered version)
*
*    Routines used:
*                  cfft.asm (complex FFT)
*
*    Requirements: Matrix size = N  > 0
*
*    To run:
*
*         asm30 –v40 –gs ser.asm
*         asm30 –v40 –gs sintab.asm
*         asm30 –v40 –gs input.asm
*         lnk30 ser.cmd
*
*********************************************************

         .global   N              ; FFT size
         .global   _MATRIX        ; Matrix address
         .global   CFFT           ; Complex 1D–FFT subroutine
         .global   C2DFFT         ; Entry point for execution

_STACK   .usect    ”STACK”,10h    ; Stack definition

         .text
FFTSIZE  .word     N
MATR     .word     _MATRIX
STACK    .word     _STACK         ; Stack address
BLOCK0   .word     002FF800H      ; On–chip buffer (RAM block 0)
TIMER    .word     0100020H       ; Timer 0 address

C2DFFT
         LDP       FFTSIZE        ; Data page pointer initialization
         LDI       @STACK,SP      ; Stack pointer initialization

t0:      LDI       @TIMER,AR2     ; Optional: benchmarking (time_start)
         STIK      –1,*+AR2(8)
         LDI       961,R0
         STI       R0,*AR2

         OR        1800h,ST       ; Enabling cache
         LDI       @FFTSIZE,AR3   ; AR3 = N = FFT size
         SUBI      1,AR3,AR5      ; AR5 = row counter

         LDI       @MATR,AR7      ; AR7 = matrix pointer
         LDI       @BLOCK0,AR6    ; AR6 = on–chip buffer pointer

*********************************************************
*                     FFT ON ROWS
*********************************************************

LOOPR
```

```
*************************
* Move row X            *
* to on–chip memory     *
*************************

         SUBI3     2,AR3,RC              ; RC = N–2
         LDI       AR7,AR0               ; Source address
         RPTBD     LOOP1
         LDI       AR6,AR1               ; Destination address
         LDI       2,IR0
         LDF       *+AR0(1),R0           ; R0 = X(I) Im

         LDF       *AR0++(IR0),R1        ; X(I) Re & points to X(I+1)
||       STF       R0, *+AR1(1)          ; Store X(I) Im
LOOP1    LDF       *+AR0(1),R0           ; R0 = X(I+1) Im
||       STF       R1,*AR1++(IR0)        ; Store X(I) Re

****************
* FFT on row X *
****************

         LAJ       CFFT                  ; Call 1D–FFT (complex)
         LDF       *AR0,R1               ; Load X(N–1) Re
         NOP
         STF       R0,*+AR1(1)           ; Store X(N–1) Im
||       STF       R1,*AR1               ; Store X(N–1) Re

*************************************
* Move row X (bit–reversed) from    *
* on–chip memory to external memory *
*************************************

         SUBI3     2,AR3,RC
         LDI       AR6,AR0               ; Source address
         LDI       AR7,AR1               ; Destination Address
         RPTBD     LOOP2
         LDI       AR3,IR0               ; Source offset for bit–reverse = N
         LDI       2,IR1                 ; Destination offset
         LDF       *+AR0(1),R0

         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
LOOP2    LDF       *+AR0(1),R0
||       STF       R1,*AR1++(IR1)
         LDF       *AR0++(IR0)B,R1
||       STF     R0,*+AR1(1)

         DBUD      AR5,LOOPR
         STF       R1,*AR1++(IR1)
         LSH3      1,AR3,R0
         ADDI      R0,AR7

*********************************************************
*                    FFT ON COLUMNS
*********************************************************

t1:      SUBI      1,AR3,AR5  ; AR5  = column counter
         LDI       @MATR,AR7  ; AR7 = Matrix pointer
```

```
LOOPC

*************************
* Move column X (X=AR7) *
* to on–chip memory     *
*************************

         SUBI3     2,AR3,RC           ; RC=N–2
         LDI       AR7,AR0            ; Source address
         LDI       AR6,AR1            ; Destination address
         RPTBD     LOOP3
         LSH3      1,AR3,IR1          ; Source offset = 2*N
         LDI       2,IR0              ; Destination offset
         LDF       *+AR0(1),R0        ; R0= X(I) Im

         LDF       *AR0++(IR1),R1     ; X(I) Re & points to X(I+1)
||       STF       R0, *+AR1(1)       ; Store X(I) Im
LOOP3    LDF       *+AR0(1),R0        ; R0=X(I+1) Im
||       STF       R1,*AR1++(IR0)     ; Store X(I) Re

*******************
* FFT on column X *
*******************

         LAJ       CFFT
         LDF       *AR0,R1            ; Load X(N–1) Re
         NOP
         STF       R0,*+AR1(1)        ; Store X(N–1) Im
||       STF       R1,*AR1            ; Store X(N–1) Re

*************************************
* Move column X (bit–reversed) from *
* on–chip memory to external memory *
*************************************

         SUBI3     2,AR3,RC           ; RC=N–2
         LDI       AR6,AR0            ; Source address
         LDI       AR7,AR1            ; Destination address
         RPTBD     LOOP4
         LDI       AR3,IR0            ; Source offset = IR0 = N  (bit–reverse)
         LSH       1,AR3,IR1          ; Destination offset (columns) = IR1 = 2N
         LDF       *+AR0(1),R0

         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
LOOP4    LDF       *+AR0(1),R0
||       STF       R1,*AR1++(IR1)

         DBUD      AR5,LOOPC
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
         STF       R1,*AR1++(IR1)
         ADDI      2,AR7

         LDI       @TIMER,AR2         ; Optional: benchmarking
         LDI       *+AR2(4),R0        ; tcomp = R0

t2       B         t2
         .end
```

## SER.CMD

```
input.obj
ser.obj
sintab.obj
–lmylib.lib
–m ser.map
–o ser.out

MEMORY
{
        ROM:      o = 0x00000000 l = 0x1000
        RAM0:     o = 0x002ff800 l = 0x400
        RAM1:     o = 0x002ffc00 l = 0x400
        LM:       o = 0x40000000 l = 0x10000
        GM:       o = 0x80000000 l = 0x20000
}

SECTIONS
{
        .text     :{}  > RAM1
        .data     :{}  > RAM1    /* SINE TABLE      */
        STACK     :{}  > RAM1    /* STACK    */
        INPUT     :{}  > LM      /* INPUT MATRIX    */
}
```

## A.3. SERB.C: Double-Buffered Implementation (C Program)

### *SERB.C*

```
/*************************************************************************************

SERB.C :  Serial 2–dimensional complex FFT (Double–buffered version)
To run:
cl30 –v40 –g –s –mr –o2 serb.c
asm30 –v40 –s sintab.asm
asm30 –v40 –s input.asm
lnk30 serbc.cmd
Requirement: SIZE ≥ 4

*************************************************************************************/
#define  SIZE           4                /* FFT size                */
#define  LOGSIZE        2                /* log(FFT size)           */

#define  BLOCK0         0x002ff800       /* on–chip buffer 1        */
#define  BLOCK1         0x002ffc00       /* on–chip buffer 2        */
#define  DMA0           0x001000a0       /* DMA0 address            */
#define  SWAP(x,y)      temp = x; x = y; y = temp;
#define  WAIT_DMA(x) while ((0x00c00000 & *x) != 0x00800000);

extern   void      cfftc(),              /* C–callable complex FFT      */
                   cmove(),              /* CPU complex move            */
                   cmoveb(),             /* CPU bit–reversed move       */
                   set_dma();            /* Set–up DMA registers        */

extern   float     MATRIX[SIZE][SIZE*2]; /* Input complex matrix    */

/* DMA control register values */

int      ctrl0= 0x00c41004;              /* no autoinit.,dmaint,bit_rev    */
int      ctrl1= 0x00c01008;              /* autoinit.,no dmaint,bitrev     */
int      ctrl2= 0x00c00008;              /* autoinit, no dmaint            */
int      ctrl3= 0x00c40004;              /* no autoinit.,dmain             */
int      dma01[7], dma02[7], dma03[7], dma04[7];

float    *CPUbuffer =(float *)BLOCK0,    /* block for CPU FFT operations   */
         *DMAbuffer =(float *)BLOCK1,    /* block for DMA operations       */
         *MM[SIZE], *temp;

volatile int       *dma0 = (int *)DMA0;
int      size2           = (SIZE*2),i,j;
int      tcomp;
/*************************************************************************************/
main()
{
asm(” or 1800h,st”);
for (i=0;i<SIZE;i++) MM[i]=MATRIX[i];

t0: time_start(0);
```

```
/********************** FFT on rows *************************************************/

/***************************************************************
1. DMA:      – moves row 1 to on–chip RAM buffer 1              *
2. CPU:      – moves row 0 to on–chip RAM buffer 0              *
             – FFT on row 0                                     *
****************************************************************/
set_dma(dma0,ctrl3,&MM[1][0],1,size2,DMAbuffer,1,1);
cmove(&MM[0][0],CPUbuffer,2,2,SIZE);
cfftc(CPUbuffer,SIZE,LOGSIZE);

/***************************************************************
1. DMA:      – moves Re FFT(row 0) to off–chip RAM              *
             – moves Im FFT(row 0) to off–chip RAM              *
             – moves row 2 to on–chip RAM                       *
2. CPU:      FFT on row 1                                       *
****************************************************************/
WAIT_DMA (dma0);
set_dma(dma01,ctrl1,CPUbuffer,SIZE,SIZE,&MM[0][0],2,dma02);
set_dma(dma02,ctrl1,(CPUbuffer+1),SIZE,SIZE,&MM[0][1],2,dma03);
set_dma(dma03,ctrl3,&MM[2][0],1,size2,CPUbuffer,1,1);
*(dma0+3) = 0;           *(dma0+6) = (int) dma01; *dma0 =ctrl2; /* start DMA   */
cfftc(DMAbuffer,SIZE,LOGSIZE);

/***************************************************************
1. DMA:  – moves Re FFT(row i) to off–chip RAM                  *
         – moves Im FFT(row i) to off–chip RAM                  *
         – moves row (i+2)to on–chip RAM                        *
2. CPU:  FFT on row (i+1)                                       *
****************************************************************/
for (i=1;i<SIZE–2;i+=1)   {
WAIT_DMA (dma0);

*(dma03+1) = (int)&MM[i+2][0];    *(dma03+4) = (int)DMAbuffer;
*(dma01+1) = (int)DMAbuffer;      *(dma01+4) = (int)&MM[i][0];
*(dma02+1) = (int)DMAbuffer+1;    *(dma02+4) = (int)&MM[i][1];
*(dma0+3) = 0;     *(dma0+6) = (int) dma01; *dma0 = ctrl2;      /* start DMA   */

cfftc(CPUbuffer,SIZE,LOGSIZE);       /* work in current row    */
SWAP (CPUbuffer,DMAbuffer);          /* switch buffers         */
}

/***********************************************************************
1. DMA:  – moves Re FFT(row(size–2)) to off–chip RAM                   *
         – moves Im FFT(row(size–2)) to off–chip RAM                   *
         – moves Re column 0 to on–chip RAM (except last row element)  *
         – moves Im column 0 to on–chip RAM                            *
2. CPU:  – FFT on last row: row (size–1)                               *
         – transfer element [size–1][0] to corresponding position in   *
           on–chip buffer 0                                            *
************************************************************************/

WAIT_DMA (dma0);
*(dma01+1) = (int)DMAbuffer;      *(dma01+4) = (int)&MM[i][0];
*(dma02+1) = (int)DMAbuffer+1;    *(dma02+4) = (int)&MM[i][1];

set_dma(dma03,ctrl2,&MM[0][0],size2,(SIZE–1),DMAbuffer,2,dma04);
set_dma(dma04,ctrl3,&MM[0][1],size2,(SIZE–1),(DMAbuffer+1),2,2);
```

```
*(dma0+3) = 0;     *(dma0+6) = (int) dma01;  *dma0 = ctrl2;/* start DMA    */

cfftc(CPUbuffer,SIZE,LOGSIZE);

WAIT_DMA (dma0);
*(DMAbuffer+size2–2) = *(CPUbuffer);
*(DMAbuffer+size2–1) = *(CPUbuffer+1);

/************************* FFT on columns *******************************************/

/***********************************************************************
1. DMA:  – moves Re FFT(row(size–1)) to off–chip RAM                   *
         – moves Im FFT(row(size–1)) to off–chip RAM                   *
         – moves Re column 1 to on–chip RAM (except last row element)  *
         – moves Im column 1 to on–chip RAM                            *
2. CPU:  – FFT on column 0                                             *
************************************************************************/

CPUbuffer= (float *) BLOCK0;      /*initialize buffer pointer   */
DMAbuffer= (float *) BLOCK1;

*(dma01+1) = (int)DMAbuffer;      *(dma01+4) = (int)&MM[SIZE–1][0];
*(dma02+1) = (int)DMAbuffer+1;    *(dma02+4) = (int)&MM[SIZE–1][1];
*(dma03+1) = (int)&MM[0][2];      *(dma03+4) = (int)DMAbuffer;
*(dma04+1) = (int)&MM[0][3];      *(dma04+4) = (int)DMAbuffer+1;
*(dma03+3) =                      *(dma04+3) = SIZE;

*(dma0+3) = 0; *(dma0+6) = (int) dma01; *dma0 = ctrl2;  /* start DMA   */

cfftc(CPUbuffer,SIZE,LOGSIZE);           /* work in column 0           */

/***********************************************************************
1. DMA:  – moves Re FFT(column 0) to off–chip RAM                      *
         – moves Im FFT(column 0) to off–chip RAM                      *
         – moves Re column 2 to on–chip RAM                            *
         – moves Im column 2 to on–chip RAM                            *
2. CPU:  – FFT on column 1                                             *
************************************************************************/
WAIT_DMA (dma0);
t1:

*(dma01+1) = (int)CPUbuffer;      *(dma01+4) = (int)&MM[0][0];
*(dma02+1) = (int)CPUbuffer+1;    *(dma02+4) = (int)&MM[0][1];
*(dma01+5) = *(dma02+5) = size2;
*(dma03+1) = (int)&MM[0][4];      *(dma03+4) = (int)CPUbuffer;
*(dma04+1) = (int)&MM[0][5];      *(dma04+4) = (int)CPUbuffer+1;

*(dma0+3) = 0;     *(dma0+6) = (int) dma01;*dma0 = ctrl2; /* start DMA */

cfftc(DMAbuffer,SIZE,LOGSIZE);
```

```
/***********************************************************************
1. DMA:  – moves Re FFT(column i) to off–chip RAM                      *
         – moves Im FFT(column i) to off–chip RAM                      *
         – moves Re column (i+2) to on–chip RAM                        *
         – moves Im column (i+2) to on–chip RAM                        *
2. CPU:  – FFT on column (i+1)                                         *
************************************************************************/

for (i=2;i<size2–4;i+=2)   {
WAIT_DMA (dma0);

*(dma01+1) = (int)DMAbuffer;         *(dma01+4) = (int)&MM[0][i];
*(dma02+1) = (int)DMAbuffer+1;       *(dma02+4) = (int)&MM[0][i+1];
*(dma03+1) = (int)&MM[0][i+4];       *(dma03+4) = (int)DMAbuffer;
*(dma04+1) = (int)&MM[0][i+5];       *(dma04+4) = (int)DMAbuffer+1;

*(dma0+3) = 0; *(dma0+6) = (int) dma01; *dma0 = ctrl2;/* start DMA */

cfftc(CPUbuffer,SIZE,LOGSIZE);       /* work in current column     */
SWAP (CPUbuffer,DMAbuffer);
}

/***********************************************************************
1. DMA:     – moves Re FFT(column (size–2)) to off–chip RAM            *
            – moves Im FFT(column (size–2)) to off–chip RAM            *
2. CPU:     – FFT on last column (size–1)                              *
      –      moves FFT(last column) to off–chip RAM                    *
************************************************************************/

WAIT_DMA (dma0);

*(dma01+1) = (int)DMAbuffer;         *(dma01+4) = (int)&MM[0][i];
*(dma02+1) = (int)DMAbuffer+1;       *(dma02+4) = (int)&MM[0][i+1];
*(dma02) = ctrl0;
*(dma0+3) = 0; *(dma0+6) = (int) dma01; *dma0 = ctrl2;  /* start DMA   */

cfftc (CPUbuffer,SIZE,LOGSIZE);      /* fft on last column             */
cmoveb (CPUbuffer,&MM[0][size2–2],SIZE,size2,SIZE);

WAIT_DMA (dma0);                     /* wait for DMA to finish         */
tcomp= time_read(0);
t2:;
} /*main*/
```

## SERBC.CMD

```
–c                            /* LINK USING C CONVENTIONS          */
serb.obj
sintab.obj
input.obj
–stack 0x0040
–lrts40.lib                   /* GET RUN–TIME SUPPORT              */
–lprts40r.lib                 /* PARALLEL RUN-TIME SUPPORT LIBRARY */
–lmylib.lib
–m serbc.map
–o serbc.out

MEMORY
{
         ROM:      org = 0x00         len = 0x1000
         BUF0:     org = 0x002ff800   len = 0x0200
         RAM0:     org = 0x002ffa00   len = 0x0200
         BUF1:     org = 0x002ffc00   len = 0x0200
         RAM1:     org = 0x002ffe00   len = 0x0200
         LM:       org = 0x40000000   len = 0x10000
         GM:       org = 0x80000000   len = 0x20000
}

SECTIONS
{
         INPUT:    {} > LM            /* INPUT MATRIX           */
         .text:    {} > LM            /* CODE                   */
         .cinit:   {} > RAM1          /* INITIALIZATION TABLES  */
         .stack:   {} > RAM1          /* SYSTEM STACK           */
         .bss :    {} > RAM1          /* GLOBAL & STATIC VARS   */
         .data:    {} > RAM1          /* SINE TABLES            */
}
```

## A.4. SERB.ASM: Double-Buffered Implementation (’C40 Assembly Program)

### *SERB.ASM*

```
**************************************************************************************
*
*    SERB.ASM :     TMS320C40 complex 2D–FFT serial program
*                   (Double–buffered version)
*
*    Routines used: cfft.asm (complex FFT)
*
*    Requirements: matrix size = N >= 4
*
*    To run:
*        asm30 –v40 –s –g serb.asm
*        asm30 –v40 –s –g sintab.asm
*        asm30 –v40 –s –g input.asm
*        lnk30 serb.cmd
*
**************************************************************************************

         .global   N              ; FFT SIZE
         .global   _MATRIX        ; MATRIX ADDRESS
         .global   CFFT           ; 1D–FFT SUBROUTINE
         .global   C2DFFT         ; ENTRY POINT FOR EXECUTION

_STACK   .usect    ”STACK”, 10h   ; Stack definition

* DMA AUTOINITIALIZATION VALUES

         .sect     ”DMA_AUTOINT’ ’    ; DMA autoinitialization values
DMA01    .space    6
         .word     DMA02
DMA02    .space    6
         .word     DMA03
DMA03    .space    6
         .word     DMA04
DMA04    .space    6

         .text
FFTSIZE  .word     N
MATR     .word     _MATRIX
BLOCK0   .word     002FF800H      ; RAM BLOCK 0
BLOCK1   .word     002FFC00H      ; RAM BLOCK 1
STACK_A  .word     _STACK         ; STACK ADDRESS
DMA0     .word     001000A0H      ; ADDRESS OF DMA0
CTRL0    .word     00C41004H      ; NO AUTOINITIALIZATION, DMA INT., BITREV
CTRL1    .word     00C01008H      ; AUTOINITIALIZATION, NO DMA INT., BITREV
CTRL2    .word     00C00008H      ; AUTOINITIALIZATION, NO DMA INT.
CTRL3    .word     00C40004H      ; NO AUTOINITIALIZATION, DMA INT.
P04      .word     DMA04          ; POINTER TO REGISTER VALUES (DMA04)
P03      .word     DMA03          ; POINTER TO REGISTER VALUES (DMA03)
P02      .word     DMA02          ; POINTER TO REGISTER VALUES (DMA02)
P01      .word     DMA01          ; POINTER TO REGISTER VALUES (DMA01)
MASK     .word     02000000H      ; 1 IN DMAINT0
TIMER    .word     0100020H       ; TIMER 0 address

C2DFFT   LDP       FFTSIZE        ; LOAD DATA PAGE POINTER
         LDI       @STACK_A,SP    ; INITIALIZE THE STACK POINTER
```

```
t0:     LDI       @TIMER,AR2     ; OPTIONAL: BENCHMARKING (TIME_START)
        STIK      –1,*+AR2(8)
        LDI       961,R0
        STI       R0,*AR2
        OR        1800h,ST       ; ENABLE CACHE
        LDI       @FFTSIZE,AR3   ; AR3=N
        LDI       @MATR,AR7      ; POINTER TO MATRIX
        LDI       @BLOCK1,R7     ; POINTER TO DMA BUFFER
        LDI       @BLOCK0,AR6    ; POINTER TO FFT BUFFER

******************************************************
*                 CPU MOVES ROW 0                    *
******************************************************
        SUBI3     2,AR3,RC       ; RC=N–2
        LDI       AR7,AR0        ; SOURCE
        RPTBD     LOOP1
        LDI       AR6,AR1        ; DESTINATION
        LDI       2,IR1
        LDF       *+AR0(1),R0    ; R0= X0(I) IM


* LOOP
        LDF       *AR0++(IR1),R1 ; X0(I) RE & POINTS TO X0(I+1)
||      STF       R0, *+AR1(1)   ; STORE X0(I) IM
LOOP1   LDF       *+AR0(1),R0    ; R0=X0(I+1) IM
||      STF       R1,*AR1++(IR1) ; STORE X0(I) RE


* STORE LAST VALUE
        LDI       @P02,AR2       ; POINTS DMA REGISTER
        LDF       *AR0++(IR1)    ,R1 ; LOAD X0(N–1) RE
||      STF       R0,*+AR1(1)    ; STORE X0(N–1) IM
        STF       R1,*AR1        ; STORE X0(N–1) RE

**********************************************************************
*
* SET PARAMETERS FOR DMA02, DMA03 THAT WILL ALWAYS BE FIXED          *
*
**********************************************************************


        LDI       @P03,AR1       ; POINTS DMA REGISTER
        LDI       @P01,AR4
        LDI       @CTRL1,R0
        STI       R0,*AR2
        STI       AR3,*+AR2(2)   ; SOURCE INDEX
        STI       R0,*AR4
        STI       AR3,*+AR4(2)   ; SOURCE INDEX
        STI       AR3,*+AR2(3)   ; COUNTER
        STI       AR3,*+AR4(3)   ; COUNTER
        LSH3      1,AR3,R0       ; R0=2*N
        STIK      2H,*+AR2(5)    ; DESTINATION INDEX
        STIK      2H,*+AR4(5)    ; DESTINATION INDEX
        LDI       @CTRL3,R2
        STI       R2,*AR1
        LDI       @DMA0,AR2      ; POINTS DMA REGISTER
        STIK      1H,*+AR1(2)    ; SOURCE INDEX
        SUBI      3,AR3,AR5      ; AR5=N–3 : (N–2) DMA TRANSFERS
        STI       R0,*+AR1(3)    ; COUNTER
        STI       AR0,*+AR2(1)   ; SOURCE
        STIK      1H,*+AR1(5)    ; DESTINATION INDEX
```

```
**********************************************************
*              CPU : FFT ON ROW 0                        *
*              DMA : BEGINS TO TRANSFER ROW1             *
**********************************************************
         STIK      1H,*+AR2(2)       ; SOURCE INDEX
         STI       R0,*+AR2(3)       ; COUNTER=2N
         LAJ       CFFT              ; FFT ON ROW 0
         STI       R7,*+AR2(4)       ; DESTINATION
         STIK      1H,*+AR2(5)       ; DESTINATION INDEX
         STI       R2,*AR2           ; CONTROL3

**********************************************************
*                 FFT ON ROWS                            *
**********************************************************
         LDI       @P02,AR1
         LDI       @P01,AR0
         LSH3      1,AR3,R0
LOOP2    TSTB      @MASK,IIF
         BZAT      LOOP2
* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         ADDI      1H,AR6,R1
         STI       R1,*+AR1(1)       ; SOURCE
         ADDI      1H,AR7,R1
         STI       R1,*+AR1(4)       ; DST

* DMA01: BIT–REVERSED TRANSFER OF LAST RESULT (Re)
         STI       AR6,*+AR0(1)      ; SOURCE
         STI       AR7,*+AR0(4)      ; DST

* DMA03: TRANSFER NEXT ROW
         LDI       @P03,AR1          ; DMA0
         ADDI      R0,AR7
         ADDI      AR7,R0
         STI       R0,*+AR1(1)       ; SOURCE: NEXT ROW
         AND       0H,IIF            ; CLEAR FLAG
         STI       AR6,*+AR1(4)      ; DESTINATION:
         LDI       @DMA0,AR1
         LDI       R7,R2             ; EXCHANGE BUFFER POINTERS
         LDI       AR6,R7            ; R7: POINTER FOR NEXT DMA
         STIK      0,*+AR1(3)        ; TEMPORAL FIX
         STI       AR0,*+AR1(6)

* FFT ON CURRENT ROW
         LAJ       CFFT
         LDI       R2,AR6            ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR1           ; START (DMA)

         DBUD      AR5,LOOP2
         LDI       @P02,AR1          ; DMA0
         LDI       @P01,AR0
         LSH3      1,AR3,R0
```

```
*********************************************************************
* DMA:– TRANSFER BACK RESULT (ROW N–2). BIT–REVERSED                *
*     – TRANSFER FIRST COLUMN (EXCEPT LAST LOCATION)                *
*                                                                   *
* CPU:FFT ON LAST ROW (ROW N–1)                                     *
*********************************************************************

         LDI       @P01,AR2       ; DMA0
         LDI       @P02,AR1       ; DMA02
         LDI       @P03,AR0       ; AR0 POINTS TO DMA03
         LDI       @P04,AR4       ; AR1 POINTS TO DMA04
B2       TSTB      @MASK,IIF
         BZAT      B2
* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         ADDI      1H,AR6,R0
         STI       R0,*+AR1(1)    ; SOURCE
         ADDI      1H,AR7,R0
         STI       R0,*+AR1(4)    ; DST

* DMA01: BIT–REVERSED TRANSFER OF LAST RESULT (ROW N–2: Re)
         STI       AR6,*+AR2(1)   ; SOURCE
         STI       AR6,*+AR0(4)   ; DESTINATION: BLOCK0(RE)
         STI       AR7,*+AR2(4)   ; DST
         STIK      2H,*+AR0(5)    ; DESTINATION INDEX=2 (RE)
         STIK      2H,*+AR4(5)    ; DESTINATION INDEX=2 (IM)
* DMA03: TRANSFER COLUMN 0 (Re) EXCEPT LAST LOCATION
* DMA04: TRANSFER COLUMN 0 (Im) EXCEPT LAST LOCATION
         LDI       @CTRL2,R1
         STI       R1,*AR0
         LDI       @CTRL3,R0
         STI       R0,*AR4
         LDI       @MATR,R0       ; R0:ADDRESS OF FIRST COLUMN
         STI       R0,*+AR0(1)    ; SOURCE: (RE)

         ADDI      1,R0 ; POINTS TO IMAGINARY PART
         STI       R0,*+AR4(1)    ; SOURCE: (IM)
         ADDI      1,AR6,R1       ;
         STI       R1,*+AR4(4)    ; DESTINATION: BLOCK0(IM)
         LSH3      1,AR3,R1
         STI       R1,*+AR0(2)    ; SOURCE INDEX=2*N
         SUBI      1,AR3,R0       ; R0=N–1
         STI       R1,*+AR4(2)    ;
         AND       0H,IIF         ; CLEAR FLAG
         STI       R0,*+AR0(3     ; COUNTER=N–1
         ADDI      R1,AR7         ; R1=2N
         STI       R0,*+AR4(3)

         LDI       @DMA0,AR0      ; GIVE THE START
         LDI       R7,R2          ; EXCHANGE BUFFER POINTERS
         LDI       AR6,R7         ; R7: POINTER FOR NEXT DMA
         STIK      0,*+AR0(3)
         STI       AR2,*+AR0(6)

* FFT ON LAST ROW
         LAJ       CFFT
         LDI       R2,AR6         ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR0        ; START (DMA)
```

```
************************************************************
*        DMA: –TRANSFER BACK RESULT (LAST ROW)
*
*        –TRANSFER SECOND COLUMN (COLUMN 1)
*
*        CPU: FFT ON FIRST COLUMN
*
************************************************************


         LDI       @P02,AR1       ; DMA02
         LDI       @P01,AR0       ; P01
         LDI       @P03,AR4       ; AR0 POINTS TO DMA03
         LDI       @P04,AR5       ; AR1 POINTS TO DMA04


B3       TSTB      @MASK,IIF
         BZAT      B3
* CPU MOVES LAST VALUE (1ST COLUMN)FROM AR6: BLOCK1 TO R7:
         LSH3      1,AR3,R2
         ADDI      R2,R7,AR2
         SUBI      2,AR2          ; AR2= BLOCK0+SIZE2–2
         ADDI      1H,AR6,R2
         LDF       *AR6,R0        ; RE
||       LDF       *+AR6(1),R1    ; IM
         STI       R2,*+AR1(1)    ; SOURCE
         STF       R0,*AR2
||       STF       R1,*+AR2(1)


* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         ADDI      1H,AR7,R0
         STI       R0,*+AR1(4)    ; DST
* DMA01: BIT–REVERSED TRANSFER OF LAST RESULT (Re)
         STI       AR6,*+AR0(1)   ; SOURCE
         STI       AR7,*+AR0(4)   ; DST
* DMA03: TRANSFER COLUMN 1 (Re)
* DMA04: TRANSFER COLUMN 1 (Im)
         LDI       @MATR,AR7
         ADDI      2,AR7,R0       ; R0: POINTS TO COLUMN 1
         STI       R0,*+AR4(1)    ; SOURCE: (RE)
         ADDI      1,R0           ; POINTS TO IMAGINARY PART
         STI       R0,*+AR5(1)    ; SOURCE: (IM)
         AND       0H,IIF         ; CLEAR FLAG
         STI       AR6,*+AR4(4)   ; DESTINATION: BLOCK1(RE)
         ADDI      1,AR6,R1
         STI       R1,*+AR5(4)    ; DESTINATION: BLOCK0(IM)
         LDI       R7,R2
         STI       AR3,*+AR4(3)   ; COUNTER=N
         LDI       @DMA0,AR1      ; GIVE THE START
         LDI       AR6,R7         ; R7: BLOCK1
         STI       AR3,*+AR5(3)
         STIK      0,*+AR1(3)
         STI       AR0,*+AR1(6)


* FFT ON FIRST COLUMN


         LAJ       CFFT
         LDI       R2,AR6         ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR1        ; START (DMA)
```

```
*********************************************************
*                 FFT ON COLUMNS                        *
*********************************************************

t1:      LDI       @P02,AR1
         LDI       @P01,AR0       ; P01
         SUBI      3,AR3,AR5      ; AR5=N–3: (N–2) DMA TRANSFERS
         LSH3      1,AR3,R1
         STI       R1,*+AR0(5)    ; DST INDEX
         STI       R1,*+AR1(5)    ; DST INDEX
         ADDI      1H,AR6,R0

B4       TSTB      @MASK,IIF
         BZAT      B4

* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         STI       R0,*+AR1(1)    ; SOURCE
         ADDI      1H,AR7,R0
         STI       R0,*+AR1(4)    ; DST
* DMA01: BIT–REVERSED TRANSFER OF LAST RESULT (Re)
         STI       AR6,*+AR0(1)   ; SOURCE
         STI       AR7,*+AR0(4)   ; DST
* DMA03: TRANSFER NEXT COLUMN (Re)
* DMA04: TRANSFER NEXT COLUMN (Im)

         LDI       @P03,AR4       ; AR0 POINTS TO DMA03
         LDI       @P04,AR2       ; AR1 POINTS TO DMA04
         ADDI      2,AR7          ; R0: POINTS TO NEXT COLUMN
         ADDI      2,AR7,R0
         STI       R0,*+AR4(1)    ; SOURCE: (RE)
         AND       0H,IIF         ; CLEAR FLAG
         STI       AR6,*+AR4(4)   ; DESTINATION: BLOCK1(RE)
         ADDI      1,R0 ; POINTS TO IMAGINARY PART
         STI       R0,*+AR2(1)    ; SOURCE: (IM)
         ADDI      1,AR6,R1
         STI       R1,*+AR2(4)    ; DESTINATION: BLOCK0(IM)
         LDI       @DMA0,AR1      ; GIVE THE START
         LDI       R7,R2
         LDI       AR6,R7         ; R7: BLOCK1
         STIK      0,*+AR1(3)
         STI       AR0,*+AR1(6)

* FFT ON CURRENT COLUMN
         LAJ       CFFT
         LDI       R2,AR6         ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR1        ; START (DMA)
         DBUD      AR5,B4
         LDI       @P02,AR1
         LDI       @P01,AR0
         ADDI      1H,AR6,R0

***********************************************************
*   DMA: TRANSFER LAST FFT RESULT
*   CPU:   FFT ON LAST COLUMN
***********************************************************

         LDI       @P02,AR1       ; DMA0
B5       TSTB      @MASK,IIF
         BZAT      B5
```

```
* DMA02/DMA01: BIT–REVERSED TRANSFER OF LAST RESULT
         ADDI      1H,AR6,R0
         STI       R0,*+AR1(1)        ; SOURCE
         ADDI      1H,AR7,R0
         STI       R0,*+AR1(4)        ; DST
         LDI       @P01,AR0           ; P01
         LDI       @CTRL0,R0
         STI       R0,*AR1
         STI       AR6,*+AR0(1)       ; SOURCE
         STI       AR7,*+AR0(4)       ; DST
         LDI       @DMA0,AR1          ; GIVE THE START
         ADDI      2,AR7
         AND       0,IIF
         STIK      0,*+AR1(3)
         STI       AR0,*+AR1(6)
         LDI       @CTRL2,R0
         STI       R0,*AR1            ; START (DMA)

* FFT ON CURRENT ROW
         LAJ       CFFT
         LDI       R7,R2
         LDI       AR6,R7
         LDI       R2,AR6
         SUBI3     2,AR3,RC           ; RC=N–2
         LDI       AR6,AR0            ; SOURCE
         LDI       AR7,AR1            ; DESTINATION
         RPTBD     B6
         LDI       AR3,IR0
         LSH3      1,AR3,IR1
         LDF       *+AR0(1),R0; R0= X(I) IM

* LOOP
         LDF       *AR0++(IR0)B,R1    ; X(I) RE & POINTS TO X(I+1)
||       STF       R0, *+AR1(1)       ; STORE X(I) IM
B6       LDF       *+AR0(1),R0        ; R0=X(I+1) IM
||       STF       R1,*AR1++(IR1)     ; STORE X(I) RE

* STORE LAST VALUE
B7       TSTB      @MASK,IIF
         BZ        B7
         LDF       *AR0++(IR0)B,R1    ; LOAD X(N–1) RE
||       STF       R0,*+AR1(1)        ; STORE X(N–1) IM
         STF       R1,*AR1            ; STORE X(N–1) RE
         LDI       @TIMER,AR2         ; OPTIONAL: BENCHMARKING (TIME_READ)
         LDI       *+AR2(4),R0        ; TCOMP = R0

t2       B         t2
         .end
```

## SERB.CMD

```
input.obj
serb.obj
sintab.obj
–lmylib.lib
–m serb.map
–o serb.out

MEMORY
{
         ROM:      o = 0x00000000 l = 0x1000
         BUF0:     o = 0x002ff800 l = 0x200
         RAM0:     o = 0x002ffa00 l = 0x200
         BUF1:     o = 0x002ffc00 l = 0x200
         RAM1:     o = 0x002ffe00 l = 0x200
         LM:       o = 0x40000000 l = 0x10000
         GM:       o = 0x80000000 l = 0x20000
}

SECTIONS
{
         INPUT          :{}   > LM
         .text          :{}   > LM
         .data          :{}   > RAM1            /* SINE TABLE         */
         STACK          :{}   > RAM1
         DMA_AUTOINI    :{}   > RAM1            /* AUTOINIT. VALUES   */
}
```

## Appendix B: Parallel 2-D FFT (Shared-Memory Version)

### B.1. SH.C: Single-Buffered Implementation (C Program)

### *SH.C*

```
/*************************************************************************************

SH.C :   Parallel 2–dimensional complex FFT
         (shared–memory single–buffered version)

Routines used:    cfftc.asm       (C–callable complex–fft)
                  cmove.asm       (CPU complex move)
                  cmoveb.asm      (CPU bit–reversed complex move)
                  syncount.asm    (synchronization routine via counter in-shared memory)
To run:
         cl30 –v40 –g –as –mr –o2 sh.c
         asm30 –v40 –s input.asm
         asm30 –v40 –s synch.asm
         asm30 –v40 –s sintab.asm
         lnk30 shc.cmd

Note: Before running, initialize my_node variable with the corresponding value, using
the ’C40 emulator or an assembly file.

*************************************************************************************/
#define  SIZE      16                 /* FFT size (n)               */
#define  LOGSIZE   4                  /* log (FFT size)             */
#define  P         2                  /* number of processors       */
#define  Q         SIZE/P             /* rows/col. per processor    */
#define  BLOCK0    0x002ff800         /* on–chip RAM buffer  */

extern   void cfftc(), cmove(), cmoveb(), syncount();
extern   int colsynch;                /* column/row synchronization */
extern   float MATRIX[SIZE][SIZE*2];  /* Input matrix               */

float    *block0 = (float *)BLOCK0,
         *MM[SIZE];

int      my_node ,                    /* node–id                    */
         *colsynch_p    =& colsynch,  /* row/column synchronization */
         size2 = 2*SIZE,
         q2 = 2*Q,
         i,l1,l2;
int      tcomp;

/************************************************************************************/

main()
{
asm(” OR 1800h,st”);                  /* cache enable                   */

for (i=0;i<SIZE;i++) MM[i]=MATRIX[i];/* accessing assembly variables   */

t0: syncount(colsynch_p,P);           /* Optional: Common start         */
         time_start(0);               /* Optional: Benchmarking         */
/************************** FFT on rows *********************************************/

l1= Q*my_node;      l2 = l1+Q;        /* select row working set             */

for (i=l1; i<l2;i++) {
        cmove (&MM[i][0],block0,2,2,SIZE);
        cfftc(block0,SIZE,LOGSIZE);
        cmoveb (block0,&MM[i][0],SIZE,2,SIZE);
        }

t1: syncount(colsynch_p,2*P);        /* row/column synchronization        */

/************************** FFT by columns ******************************************/

l1 = l1<<1;        l2 = l2<<1;        /* select column working set: multiply by 2  */

for (i=l1;i<l2;i+=2) {
        cmove (&MM[0][i],block0,size2,2,SIZE);
        cfftc(block0,SIZE,LOGSIZE);
        cmoveb (block0,&MM[0][i],SIZE,size2,SIZE);
        }

tcomp = time_read(0);                /* Optional: Benchmarking (timer)    */

t2: ;
} /*main*/
```

### SHC.CMD

```
sh.obj
input.obj
sintab.obj
synch.obj
–c
–stack 0x0100
–lrts40.lib
–lprts40r.lib
–lmylib.lib
–m shc.map
–o shc.out

/* SPECIFY THE SYSTEM MEMORY MAP         */

MEMORY
{
          ROM:      org = 0x0        len = 0x1000
          RAM0:     org = 0x002ff800 len = 0x0400   /* on–chip RAM block 0    */
          RAM1:     org = 0x002ffc00 len = 0x0400   /* on–chip RAM block 1    */
          LM:       org = 0x40000000 len = 0x10000  /* LOCAL MEMORY           */
          GM:       org = 0x80000000 len = 0x20000  /* GLOBAL MEMORY          */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
          .text:    {}  > RAM1
          .cinit:   {}  > RAM1       /* INITIALIZATION TABLES  */
          .stack:   {}  > RAM1       /* SYSTEM STACK           */
          .bss:     {}  > RAM1
          .data:    {}  > RAM1       /* Sine table             */
          INPUT:    {}  > GM         /* Input matrix           */
          SYNCH:    {}  > GM         /* Synchronization        */
}
```

### SYNCH.ASM

```
***********************************************************
*
* SYNCH.ASM : File containing shared–memory location for
* interprocessor synchronization
*
***********************************************************

          .global _colsynch
          .sect ”SYNCH”

_colsynch .int 0

          .end
```

## B.2. SH.ASM: Single-Buffered Implementation (’C40 Assembly Program)

### *SH.ASM*

```
*****************************************************************
*
*         SH.ASM :            TMS320C40 complex 2D–FFT serial program
*                             (Single–buffered version)
*
*         Routines used:      cfft.asm (radix–2 complex FFT)
*
*         Requirements:       Number of processors = P > 0
*                             rows/columns per processor = Q > 0
*
*         To run:
*         asm30 –v40 –g –s sh.asm
*         asm30 –v40 –g –s spinput.asm
*         asm30 –v40 –g –s input.asm
*         asm30 –v40 –g –s ssintab.asm
*         asm30 –v40 –g –s synch.asm
*         asm30 –v40 –g –s 0.asm
*         asm30 –v40 –g –s 1.asm
*         lnk30 sh.cmd 0.obj –o a0.out (program for processor 0)
*         lnk30 sh.cmd 1.obj –o a1.out (program for processor 1)
*
*****************************************************************

          .global   N                ; FFT size
          .global   P                ; Number of processors
          .global   Q                ; Rows per processor
          .global   MYNODE
          .global   _MATRIX          ; Matrix address
          .global   _colsynch        ; Synchronization counter
          .global   CFFT             ; Complex 1D–FFT subroutine
          .global   C2DFFT           ; Entry point for execution
          .global   _syncount
_STACK    .usect    ”STACK”,10h      ; Stack definition
          .text
FFTSIZE   .word     N
PROC      .word     P
NROWS     .word     Q
MYID      .word     MYNODE
MATR      .word     _MATRIX
SYNCH     .word     _colsynch
STACK     .word     _STACK           ; Stack address
BLOCK0    .word     002FF800H        ; On–chip buffer (RAM block 0)
TIMER     .word     0100020H         ; Timer 0 address (Benchmarking)

C2DFFT
          LDP       FFTSIZE          ; Data page pointer initialization
          LDI       @STACK,SP        ; Stack pointer initialization
          LDI       @SYNCH,AR2       ; Optional: Common start (benchmarking)
          LDI       @PROC,R2         ; wait until counter = P
```

```
t0:       CALL      _syncount
          LDI       @TIMER,AR2       ; Optional: benchmarking (time_start)
          STIK      –1,*+AR2(8)
          LDI       961,R0
          STI       R0,*AR2
          OR        1800h,ST         ; Enabling cache
          LDI       @FFTSIZE,AR3     ; AR3 = N = FFT size
          LDI       @MYID,R0
          LDI       @NROWS,AR5       ; AR5 = row counter = Q
          MPYI      AR5,R0           ; Q*MY_NODE
          MPYI      AR3,R0           ; R0  = N*Q*MYNODE
          LSH       1,R0             ; R0  = 2*N*Q*MYNODE
          LDI       @MATR,AR7
          ADDI      R0,AR7           ; AR7 = matrix pointer = &MATRIX[Q*MYNODE][0]
          SUBI      1,AR5            ; AR5 = Q–1
          LDI       @BLOCK0,AR6      ; AR6 = on–chip buffer pointer

*********************************************************
*                     FFT ON ROWS
*********************************************************

LOOPR
*************************
* Move row X (X = AR7)  *
* to on–chip memory     *
*************************

          SUBI3     2,AR3,RC         ; RC = N–2
          LDI       AR7,AR0          ; Source address
          RPTBD     LOOP1
          LDI       AR6,AR1          ; Destination address
          LDI       2,IR0            ; Destination offset
          LDF       *+AR0(1),R0      ; R0 = X(I) Im
          LDF       *AR0++(IR0),R1   ; X(I) Re & points to X(I+1)
||        STF       R0, *+AR1(1)     ; Store X(I) Im
LOOP1     LDF       *+AR0(1),R0      ; R0 = X(I+1) Im
||        STF       R1,*AR1++(IR0)   ; Store X(I) Re

****************
* FFT on row X *
****************

          LAJ       CFFT             ; Call 1D–FFT (complex)
          LDF       *AR0,R1          ; Load X(N–1) Re
          NOP
          STF       R0,*+AR1(1)      ; Store X(N–1) Im
||        STF       R1,*AR1          ; Store X(N–1) Re

*************************************
* Move row X (bit–reversed) from    *
* on–chip memory to external memory *
*************************************

          SUBI3     2,AR3,RC
          LDI       AR6,AR0          ; Source address
          LDI       AR7,AR1          ; Destination Address
          RPTBD     LOOP2
          LDI       AR3,IR0          ; Source offset for bit–reverse = N
          LDI       2,IR1            ; Destination offset
          LDF       *+AR0(1),R0
          LDF       *AR0++(IR0)B,R1
||        STF       R0,*+AR1(1)
LOOP2     LDF       *+AR0(1),R0
||        STF       R1,*AR1++(IR1)
          LDF       *AR0++(IR0)B,R1
||        STF       R0,*+AR1(1)
          DBUD      AR5,LOOPR
          STF       R1,*AR1++(IR1)
          LSH3      1,AR3,R0
          ADDI      R0,AR7

*********************************************************
*                    FFT ON COLUMNS
*********************************************************

          LDI       @MYID,R0
          LDI       @NROWS,AR5           ; AR5 = column counter = Q
          MPYI      AR5,R0               ; Q*MY_NODE
          LSH       1,R0                 ; 2*Q*MY_NODE (Complex numbers)

          LDI       @MATR,AR7
          ADDI      R0,AR7               ; AR7 = &MATRIX[0][2*Q*MYNODE]

          SUBI      1,AR5                ; AR5 = Q–1

          LDI       @SYNCH,AR2           ; Row/column synchronization
          LDI       @PROC,R2
          LSH       1,R2                 ; Optional: not needed if no common start
                                         ; is required
t1:       CALL      _syncount

LOOPC
*************************
* Move column X (X=AR7) *
* to on–chip memory     *
*************************

          SUBI3     2,AR3,RC             ; RC=N–2
          LDI       AR7,AR0              ; Source address
          LDI       AR6,AR1              ; Destination address
          RPTBD     LOOP3
          LSH3      1,AR3,IR1            ; Source offset = 2*N
          LDI       2,IR0                ; Destination offset
          LDF       *+AR0(1),R0          ; R0= X(I) Im

          LDF        *AR0++(IR1),R1      ; X(I) Re & points to X(I+1)
||        STF       R0, *+AR1(1)         ; Store X(I) Im
LOOP3     LDF       *+AR0(1),R0          ; R0=X(I+1) Im
||        STF       R1,*AR1++(IR0)       ; Store X(I) Re

*******************
* FFT on column X *
*******************

          LAJ       CFFT
          LDF       *AR0,R1              ; Load X(N–1) Re
          NOP
          STF       R0,*+AR1(1)          ; Store X(N–1) Im
||        STF       R1,*AR1              ; Store X(N–1) Re
```

```
*************************************
* Move column X (bit–reversed) from *
* on–chip memory to external memory *
*************************************

          SUBI3     2,AR3,RC          ; RC=N–2
          LDI       AR6,AR0           ; Source address
          LDI       AR7,AR1           ; Destination address
          RPTBD     LOOP4
          LDI       AR3,IR0           ; Source offset = IR0 = N  (bit–reverse)
          LSH       1,AR3,IR1         ; Destination offset (columns) = IR1 = 2N
          LDF       *+AR0(1),R0
          LDF       *AR0++(IR0)B,R1
||        STF       R0,*+AR1(1)
LOOP4     LDF       *+AR0(1),R0
||        STF       R1,*AR1++(IR1)
          DBUD      AR5,LOOPC
          LDF       *AR0++(IR0)B,R1
||        STF       R0,*+AR1(1)
          STF       R1,*AR1++(IR1)
          ADDI      2,AR7
          LDI       @TIMER,AR2        ; Optional: benchmarking (time_read)
          LDI       *+AR2(4),R0       ; tcomp = R0

t2        B         t2
          .end
```

### *SH.CMD*

```
input.obj
sh.obj
spinput.obj
ssintab.obj
synch.obj
–m sh.map
–lmylib.lib
–osh.out

MEMORY
{
          ROM:      org = 0x00        len = 0x1000
          RAM0:     org = 0x002ff800  len = 0x0400   /* On–chip RAM  block 0   */
          RAM1:     org = 0x002ffc00  len = 0x0400   /* On–chip RAM block 1    */
          LM:       org = 0x40000000  len = 0x10000  /* LOCAL MEMORY           */
          GM:       org = 0x80000000  len = 0x20000  /* GLOBAL MEMORY          */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
          .text:        {}    >RAM1          /* CODE                 */
          .data:        {}    >RAM1          /* Sine tables          */
          INPUT:        {}    >GM            /* Input matrix         */
          STACK:        {}    >RAM1          /* SYSTEM STACK         */
          SYNCH:        {}    >GM            /* synchronization      */
}
```

### SPINPUT.ASM

```
*********************************************************************************
*
* SPINPUT.ASM : input file for shared–memory program (Data on parallel system)
*
*********************************************************************************

          .global   N   ; FFT size
          .global   M   ; LOG2 FFT
          .global   P   ; Number of processors
          .global   Q   ; Rows per processor
N         .set      16
M         .set      4
P         .set      2
Q         .set      N/P

     .end
```

### 0.ASM

```
******************************************************
*   0.ASM : File containing node–id for processor 0
******************************************************
          .global MYNODE
MYNODE    .set  0
```

### 1.ASM

```
******************************************************
*   1.ASM : File containing node–id for processor 1
******************************************************
          .global MYNODE
MYNODE    .set  1
```

## B.3. SHB.C: Double-Buffered Implementation (C Program)

### *SHB.C*

```
/*************************************************************************************

SHB.C :   Parallel 2–dimensional complex FFT
          (shared–memory double–buffered version)

Routines used:      cfftc.asm     (C–callable radix–2 complex–fft)
                    cmove.asm     (CPU complex move)
                    cmoveb.asm    (CPU bit–reversed complex move)
                    set_dma.asm   (Routine to set DMA register values)
                    syncount.asm  (synchronization routine)

To run:
          cl30 –v40 –g –as –mr –o2 shb.c
          asm30 –v40 –s input.asm
          asm30 –v40 –s synch.asm
          asm30 –v40 –s sintab.asm
          lnk30 shbc.cmd
Requirement: Q ≥ 4

Note: Before running initialize the my_node variable to the corresponding value using
      the ’C40 emulator or an assembly file.

************************************************************************************/
#define   SIZE      16                   /* FFT size                   */
#define   LOGSIZE   4                    /* log (FFT size)             */
#define   P         2                    /* number of processors       */
#define   Q         SIZE/P               /* row/col. per processor     */
#define   BLOCK0    0x002ff800           /* on–chip buffer 0           */
#define   BLOCK1    0x002ffc00           /* on–chip buffer 1           */
#define   DMA0      0x001000a0           /* DMA0 address               */
#define   SWAP(x,y) temp = x; x = y; y = temp;
#define   WAIT_DMA(x)   while ((0x00c00000 & *x) != 0x00800000);

extern    void cfftc(), set_dma(), cmove(), cmoveb(), syncount();
extern    int colsynch;       /* counter in GM */
extern    float MATRIX[SIZE][SIZE*2];    /* input matrix                   */
int       ctrl0= 0x00c41004,             /* no autoinit.,dmaint,bit_rev    */
          ctrl1= 0x00c01008,             /* autoinit.,no dmaint,bit–rev    */
          ctrl2= 0x00c00008,             /* autoinit., no dmaint           */
          ctrl3= 0x00c40004;             /* no autoinit.,dmaint            */
float     *CPUbuffer    =(float *)BLOCK0,/* For CPU FFT operations         */
          *DMAbuffer    =(float *)BLOCK1,/* For DMA operations             */
          *temp,
          *MM[SIZE];
volatile  int       *dma0 = (int *)DMA0;
int       dma01[7], dma02[7], dma03[7], dma04[7];   /* DMA autoinit.values */
int       my_node,
          base,
          *colsynch_p = &colsynch,
          size2 = (SIZE*2),
          q = Q,
          q2 = 2*Q,
          ii,i,j;
int       tcomp;
```

```
/************************************************************************************/
main()
{
asm(” OR 1800h,ST”);
for (i=0;i<SIZE;i++) MM[i]=MATRIX[i];    /* accessing assembly variables          */

t0:

syncount(colsynch_p,P);       /* Optional: Common start                           */
time_start(0);                /* Optional: Benchmarking (timer)                   */
base = q*my_node;             /* point to 1st row allocated to each processor     */

/************************** FFT on rows *********************************************/

/********************************************************
1.DMA:    moves row (base+1) to on–chip RAM buffer 0    *
2.CPU:    – moves row (base+0) to on–chip RAM buffer 1  *
          – FFT on row (base+0) in buffer 1             *
*********************************************************/
ii =base+1;
set_dma(dma0,ctrl3,&MM[ii][0],1,size2,DMAbuffer,1,1);
cmove(&MM[base][0],CPUbuffer,2,2,SIZE);
cfftc(CPUbuffer,SIZE,LOGSIZE);

/*******************************************************************
1.DMA:    – moves Re FFT (row(base+0)) to off–chip RAM             *
          – moves Im FFT (row(base+0)) to off–chip RAM             *
          – moves row (base+2) to on–chip RAM                      *
2.        CPU:      FFT on row (base+1)                            *
********************************************************************/

WAIT_DMA(dma0);
set_dma(dma01,ctrl1,CPUbuffer,SIZE,SIZE,&MM[base][0],2,dma02);
set_dma(dma02,ctrl1,(CPUbuffer+1),SIZE,SIZE,&MM[base][1],2,dma03);
set_dma(dma03,ctrl3,&MM[base+2][0],1,size2,CPUbuffer,1,1);
*(dma0+3) = 0;  *(dma0+6) = (int) dma01;     *dma0 =ctrl2;/* start DMA */

cfftc(DMAbuffer,SIZE,LOGSIZE);

/************************************************************
1.        DMA: – moves Re FFT row ii to off–chip RAM       *
                    – moves Im FFT row ii to off–chip RAM   *
                    – moves row(ii+2) to on–chip RAM        *
2.        CPU:      FFT on row(ii+1)                        *
*************************************************************/

for (i=1;i<q–2;i++,ii++)   {
WAIT_DMA(dma0);

*(dma03+1) = (int)&MM[ii+2][0];   *(dma03+4) = (int)DMAbuffer;
*(dma01+1) = (int)DMAbuffer;      *(dma01+4) = (int)&MM[ii][0];
*(dma02+1) = (int)DMAbuffer+1;    *(dma02+4) = (int)&MM[ii][1];
*(dma0+3) = 0; *(dma0+6) = (int) dma01; *dma0 = ctrl2;  /* start DMA   */

cfftc(CPUbuffer,SIZE,LOGSIZE);               /* work in current row    */
SWAP (CPUbuffer,DMAbuffer);                  /* switch buffers         */
}
```

```
/*******************************************************************
1.    DMA:  – moves Re FFT(row(base+q–2)) to off–chip RAM          *
            – moves Im FFT(row(base+q–2)) to off–chip RAM          *
2.    CPU:  – FFT on last row : row(base+q–1)                      *
            – moves FFT last row to off–chip RAM                   *
*******************************************************************/

WAIT_DMA(dma0);
*(dma01+1) = (int)DMAbuffer      ;   *(dma01+4) = (int)&MM[ii][0];
*(dma02+1) = (int)DMAbuffer+1    ;   *(dma02+4) = (int)&MM[ii][1];
*dma02     = (int)ctrl0;
*(dma0+3)  = 0         ;      *(dma0+6) = (int) dma01; *dma0 = ctrl2; /* start DMA*/

cfftc(CPUbuffer,SIZE,LOGSIZE);    /* fft on last row */
cmoveb(CPUbuffer,&MM[ii+1][0],SIZE,2,SIZE);

/*********************** FFT on columns ****************************/

CPUbuffer= (float *) BLOCK0;
DMAbuffer= (float *) BLOCK1;
WAIT_DMA(dma0);
syncount(col       synch_p,2*P);    /* row/column synchronization    */

/*******************************************************************
1.    DMA:  – moves Re column (base+1) to on–chip RAM              *
            – moves Im column (base+1) to on–chip RAM              *
2.    CPU:  – moves column (base+0) to on–chip RAM                 *
            – FFT on column (base+0)                               *
*******************************************************************/

ii = 2*base;
set_dma(dma03,ctrl2,&MM[0][ii+2],size2,SIZE,DMAbuffer,2,dma04);
set_dma(dma04,ctrl3,&MM[0][ii+3],size2,SIZE,(DMAbuffer+1),2,2);
*(dma0+3) = 0;      *(dma0+6) = (int) dma03; *dma0  = ctrl2;
cmove(&MM[0][ii],CPUbuffer,size2,2,SIZE);
cfftc(CPUbuffer,SIZE,LOGSIZE);

/*******************************************************************
1.    DMA:  – moves Re FFT column (ii) to off–chip RAM             *
            – moves Im FFT column (ii) to off–chip RAM             *
            – moves Re column (ii+2) to on–chip RAM                *
            – moves Im column (ii+2) to on–chip RAM                *
2.    CPU:  – FFT on column (ii+1)                                 *
*******************************************************************/

*(dma02+5) = *(dma01+5) = (int)size2;/* offset */
*dma02     = (int)ctrl1;

for (i=0;i<q2–4;i+=2,ii+=2)   {

SWAP (CPUbuffer,DMAbuffer);
WAIT_DMA(dma0);

*(dma01+1) = (int)DMAbuffer;      *(dma01+4) = (int)&MM[0][ii];
*(dma02+1) = (int)DMAbuffer+1;    *(dma02+4) = (int)&MM[0][ii+1];
*(dma03+1) = (int)&MM[0][ii+4];   *(dma03+4) = (int)DMAbuffer;
*(dma04+1) = (int)&MM[0][ii+5];   *(dma04+4) = (int)DMAbuffer+1;
*(dma0+3) = 0; *(dma0+6) = (int) dma01; *dma0 = ctrl2;

cfftc(CPUbuffer,SIZE,LOGSIZE);       /* work in current column */
}

/************************************************************************************
1.        DMA:      – moves Re FFT column (base+q–2) to off–chip RAM                 *
                    – moves Im FFT column (base+q–2) to off–chip RAM                 *
2.        CPU:      – FFT on last column (base+q–1)                                  *
                    – moves FFT (last column) to off–chip RAM                        *
*************************************************************************************/
WAIT_DMA(dma0);

*(dma01+1) = (int)CPUbuffer;      *(dma01+4) = (int)&MM[0][ii];
*(dma02+1) = (int)CPUbuffer+1;    *(dma02+4) = (int)&MM[0][ii+1];
*(dma02)   = ctrl0;

*(dma0+3) = 0;      *(dma0+6) = (int) dma01; *dma0  = ctrl2;

cfftc(DMAbuffer,SIZE,LOGSIZE);    /* fft on last column */
cmoveb(DMAbuffer,&MM[0][ii+2],SIZE,size2,SIZE);

WAIT_DMA(dma0);

tcomp= time_read(0);          /* Optional: Benchmarking (timer) */
t2: ;
} /*main*/
```

### *SHBC.CMD*

```
shb.obj
input.obj
sintab.obj
synch.obj
–c
–stack 0x0100
–lrts40.lib
–lprts40r.lib
–lmylib.lib
–m shbc.map
–o shbc.out

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
     ROM:    org = 0x0        len = 0x1000
     BUF0:   org = 0x002ff800 len = 0x0200   /* buffer in onchip RAM block0     */
     BUF1:   org = 0x002ffc00 len = 0x0200   /* buffer in onchip RMA block1     */
     RAM0:   org = 0x002ffa00 len = 0x0200   /* on–chip RAM block 0             */
     RAM1:   org = 0x002ffe00 len = 0x0200   /* on–chip RAM block 1             */
     LM:     org = 0x40000000 len = 0x10000  /* local memory                    */
     GM:     org = 0x80000000 len = 0x20000  /* global memory                   */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
```

```
SECTIONS
{
          .text:    {}  >     LM
          .cinit:   {}  >     RAM1   /* initialization tables      */
          .stack:   {}  >     RAM1   /* system stack               */
          .bss :    {}  >     RAM1
          .data:    {}  >     RAM1   /* Sine table                 */
          INPUT:    {}  >     GM     /* Input matrix               */
          SYNCH:    {}  >     GM     /* Synchronization counter    */
}
```

## B.4. SHB.ASM: Double-Buffered Implementation (’C40 Assembly Program)

### *SHB.ASM*

```
*************************************************************************
*
*        SHB.ASM :      TMS320C40 complex 2D–FFT shared–memory program
*                       (Double–buffered version)
*
*        Routines used: cfft.asm (radix–2 complex FFT)
*
*        Requirements:  Number of processors = P > 0
*                       Rows/columns per processor = Q >= 4
*
*
*        To run:
*        asm30 –v40 –g –s shb.asm
*        asm30 –v40 –g –s spinput.asm
*        asm30 –v40 –g –s input.asm
*        asm30 –v40 –g –s sintab.asm
*        asm30 –v40 –g –s synch.asm
*        asm30 –v40 –g –s 0.asm
*        asm30 –v40 –g –s 1.asm
*        lnk30 shb.cmd 0.obj –o a0.out (program for processor 0)
*        lnk30 shb.cmd 1.obj –o a1.out (program for processor 1)
*
*************************************************************************

         .global   N                 ; FFT SIZE
         .global   P                 ; NUMBER OF PROCESSORS
         .global   Q                 ; ROWS/COLUMNS PER PROCESSOR
         .global   MYNODE            ; PROCESSOR ID
         .global   _MATRIX           ; MATRIX ADDRESS
         .global   _colsynch         ; SYNCHRONIZATION COUNTER
         .global   CFFT              ; COMPLEX 1D–FFT SUBROUTINE
         .global   C2DFFT            ; ENTRY POINT FOR EXECUTION
         .global   _syncount         ; SYNCHRONIZATION ROUTINE

_STACK   .usect    ”STACK”, 10h      ; STACK DEFINITION

* DMA AUTOINITIALIZATION VALUES

         .sect     ”DMA_AUTOINI”     ; DMA AUTOINITIALIZATION VALUES
DMA01    .space    6
         .word     DMA02
DMA02    .space    6
         .word     DMA03
DMA03    .space    6
         .word     DMA04
DMA04    .space    6    .text
FFTSIZE  .word     N
PROC     .word     P
NROWS    .word     Q
MYID     .word     MYNODE
MATR     .word     _MATRIX
SYNCH    .word     _colsynch
STACK    .word     _STACK            ; STACK ADDRESS
BLOCK0   .word     002FF800H         ; RAM BLOCK 0
BLOCK1   .word     002FFC00H         ; RAM BLOCK 1
```

```
DMA0      .word     001000A0H     ; ADDRESS OF DMA0
CTRL0     .word     00C41004H     ; NO AUTOINITIALIZATION, DMA INT., BITREV
CTRL1     .word     00C01008H     ; AUTOINITIALIZATION, NO DMA INT., BITREV
CTRL2     .word     00C00008H     ; AUTOINITIALIZATION, NO DMA INT.
CTRL3     .word     00C40004H     ; NO AUTOINITIALIZATION, DMA INT.
P04       .word     DMA04         ; POINTER TO REGISTER VALUES (DMA04)
P03       .word     DMA03         ; POINTER TO REGISTER VALUES (DMA03)
P02       .word     DMA02         ; POINTER TO REGISTER VALUES (DMA02)
P01       .word     DMA01         ; POINTER TO REGISTER VALUES (DMA01)
MASK      .word     02000000H     ; 1 IN DMAINT0
TIMER     .word     0100020H      ; TIMER 0 ADDRESS (BENCHMARKING)

C2DFFT    LDP       FFTSIZE       ; LOAD DATA PAGE POINTER
          LDI       @STACK,SP     ; INITIALIZE THE STACK POINTER
          LDI       @SYNCH,AR2    ; OPTIONAL: COMMON START (BENCHMARKING)
          LDI       @PROC,R2      ; WAIT UNTIL COUNTER = P

t0:       CALL      _syncount
          LDI       @TIMER,AR2    ; OPTIONAL: BENCHMARKING (TIME_START)
          STIK      –1,*+AR2(8)
          LDI       961,R0
          STI       R0,*AR2
          OR        1800h,ST      ; ENABLE CACHE
          LDI       @FFTSIZE,AR3  ; AR3 = N = FFT SIZE
          LDI       @MYID,R0
          LDI       @NROWS,AR5    ; AR5 = Q = ROW COUNTER
          MPYI      AR5,R0        ; R0 = Q*MYNODE
          MPYI      AR3,R0        ; R0 = N*Q*MYNODE
          LSH       1,R0          ; R0 = 2*N*Q*MYNODE
          LDI       @MATR,AR7
          ADDI      R0,AR7        ; MATRIX POINTER = &MATRIX[2*Q*MYNODE][0]
          LDI       @BLOCK1,R7    ; POINTER TO DMA BUFFER
          LDI       @BLOCK0,AR6   ; POINTER TO FFT BUFFER

*********************************************************
*                 CPU MOVES ROW 0                      *
*********************************************************

          SUBI3     2,AR3,RC      ; RC=N–2
          LDI       AR7,AR0       ; SOURCE
          RPTBD     LOOP1
          LDI       AR6,AR1       ; DESTINATION
          LDI       2,IR1
          LDF       *+AR0(1),R0   ; R0= X0(I) IM

* LOOP
          LDF       *AR0++(IR1),R1    ; X0(I) RE & POINTS TO X0(I+1)
||        STF       R0, *+AR1(1)      ; STORE X0(I) IM
LOOP1     LDF       *+AR0(1),R0       ; R0=X0(I+1) IM
||        STF       R1,*AR1++(IR1)    ; STORE X0(I) RE

* STORE LAST VALUE

          LDI       @P02,AR2          ; POINTS DMA REGISTER
          LDF       *AR0++(IR1),R1    ; LOAD X0(N–1) RE
||        STF       R0,*+AR1(1)       ; STORE X0(N–1) IM
          STF       R1,*AR1           ; STORE X0(N–1) RE
```

```
******************************************************************************
*
* SET PARAMETERS FOR DMA AUTOINITIALIZATION VALUES THAT ARE FIXED
*
******************************************************************************
         LDI       @P03,AR1       ; POINTS DMA REGISTER
         LDI       @P01,AR4
         LDI       @CTRL1,R0
         STI       R0,*AR2
         STI       AR3,*+AR2(2)   ; SOURCE INDEX
         STI       R0,*AR4
         STI       AR3,*+AR4(2)   ; SOURCE INDEX
         STI       AR3,*+AR2(3)   ; COUNTER
         STI       AR3,*+AR4(3)   ; COUNTER
         LSH3      1,AR3,R0       ; R0=2*N
         STIK      2H,*+AR2(5)    ; DESTINATION INDEX
         STIK      2H,*+AR4(5)    ; DESTINATION INDEX
         LDI       @CTRL3,R2
         STI       R2,*AR1
         LDI       @DMA0,AR2      ; POINTS DMA REGISTER
         STIK      1H,*+AR1(2)    ; SOURCE INDEX
         SUBI      3,AR5          ; AR5=Q–3 : (Q–2) DMA TRANSFERS
         STI       R0,*+AR1(3)    ; COUNTER
         STI       AR0,*+AR2(1)   ; SOURCE
         STIK      1H,*+AR1(5)    ; DESTINATION INDEX

**********************************************************
*              CPU : FFT ON ROW 0                        *
*              DMA : BEGINS TO TRANSFER ROW1             *
**********************************************************


         STIK      1H,*+AR2(2)    ; SOURCE INDEX
         STI       R0,*+AR2(3)    ; COUNTER=2N
         LAJ       CFFT           ; FFT ON ROW 0
         STI       R7,*+AR2(4)    ; DESTINATION
         STIK      1H,*+AR2(5)    ; DESTINATION INDEX
         STI       R2,*AR2

**********************************************************
*                 FFT ON ROWS                            *
**********************************************************


         LDI       @P02,AR1
         LDI       @P01,AR0
         LSH3      1,AR3,R0
LOOP2    TSTB      @MASK,IIF
         BZAT      LOOP2


* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         ADDI      1H,AR6,R1
         STI       R1,*+AR1(1)    ; SOURCE
         ADDI      1H,AR7,R1
         STI       R1,*+AR1(4)    ; DST


* DMA01: BIT–REVERSED TRANSFER OF LAST RESULT (Re)
         STI       AR6,*+AR0(1)   ; SOURCE
         STI       AR7,*+AR0(4)   ; DST
```

```
* DMA03: TRANSFER NEXT ROW
         LDI       @P03,AR1       ; DMA0
         ADDI      R0,AR7
         ADDI      AR7,R0
         STI       R0,*+AR1(1)    ; SOURCE: NEXT ROW
         AND       0H,IIF         ; CLEAR FLAG
         STI       AR6,*+AR1(4)   ; DESTINATION:
         LDI       @DMA0,AR1
         LDI       R7,R2          ; EXCHANGE BUFFER POINTERS
         LDI       AR6,R7         ; R7: POINTER FOR NEXT DMA
         STIK      0,*+AR1(3)     ; TEMPORAL FIX
         STI       AR0,*+AR1(6)

* FFT ON CURRENT ROW
         LAJ       CFFT
         LDI       R2,AR6         ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR1        ; START (DMA)
         DBUD      AR5,LOOP2
         LDI       @P02,AR1       ; DMA0
         LDI       @P01,AR0
         LSH3      1,AR3,R0

**************************************************************************************
* DMA: – TRANSFERS BACK RESULT (ROW N–2). BIT–REVERSED
*
* CPU: – FFT ON LAST ROW (ROW N–1)
*      – TRANSFERS BACK FFT OF LAST ROW (ROW N–1)
**************************************************************************************

         LDI       @P01,AR2       ; DMA01
         LDI       @P02,AR1       ; DMA02
B2       TSTB      @MASK,IIF
         BZAT      B2

* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         ADDI      1H,AR6,R0
         STI       R0,*+AR1(1)    ; SOURCE
         ADDI      1H,AR7,R0
         STI       R0,*+AR1(4)    ; DST
         LDI       @CTRL0,R0
         STI       R0,*AR1        ; CONTROL REGISTER

* DMA01: BIT–REVERSED TRANSFER OF LAST RESULT (ROW N–2: Re)
         STI       AR6,*+AR2(1)   ; SOURCE
         STI       AR7,*+AR2(4)   ; DST
         AND       0H,IIF         ; CLEAR FLAG
         LSH       1,AR3,R0
         ADDI      R0,AR7         ; AR7 POINTS TO MATRIX (LAST ROW)
         LDI       @DMA0,AR0      ; GIVE THE START
         LDI       R7,R2          ; EXCHANGE BUFFER POINTERS
         LDI       AR6,R7         ; R7: POINTER FOR NEXT DMA
         STIK      0,*+AR0(3)     ; COUNTER = 0
         STI       AR2,*+AR0(6)   ; LINK POINTER = DMA01

* FFT ON LAST ROW
         LAJ       CFFT
         LDI       R2,AR6         ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR0        ; START (DMA)
```

```
* CPU TRANSFERS FFT OF LAST ROW TO OFF–CHIP RAM

         SUBI3     2,AR3,RC          ; RC=N–2
         LDI       AR6,AR0           ; SOURCE
         LDI       AR7,AR1           ; DESTINATION
         RPTBD      B66
         LDI       AR3,IR0
         LDI       2,IR1
         LDF       *+AR0(1),R0       ; R0= X(I) IM

* LOOP
         LDF       *AR0++(IR0)B,R1   ; X(I) RE & POINTS TO X(I+1)
||       STF       R0, *+AR1(1)      ; STORE X(I) IM
B66      LDF       *+AR0(1),R0       ; R0=X(I+1) IM
||       STF       R1,*AR1++(IR1)    ; STORE X(I) RE

* STORE LAST VALUE
         LDF       *AR0++(IR0)B,R1   ; LOAD X(N–1) RE
||       STF       R0,*+AR1(1)       ; STORE X(N–1) IM
         STF       R1,*AR1           ; STORE X(N–1) RE

*********************************************************************
*                   FFT ON COLUMNS                                  *
*********************************************************************

WAIT     TSTB      @MASK,IIF
         BZAT      WAIT
         LDI       @SYNCH,AR2        ; ROW/COLUMN SYNCHRONIZATION
         LDI       @PROC,R2
         NOP
         LSH       1,R2              ; OPTIONAL: NOT NEEDED IF COMMON START
                                     ; IS NOT REQUIRED
         AND       0,IIF
t1:      CALL      _syncount

**********************************************************************
*  SET AUTOINITIALIZATION VALUES
**********************************************************************

         LDI       @P01,AR0          ; DMA01
         LDI       @P02,AR1          ; DMA02
         LDI       @P03,AR2          : DMA03
         LDI       @P04,AR4          ; DMA04 LDI 5,IR0
         LDI       @CTRL1,R4         ; SET DMA02 AND DMA01 NEW VALUES
         LSH       1,AR3,R0
         STI       R4,*AR1
         STI       R0,*+AR0(IR0)
||       STI       R0,*+AR1(IR0)     ; SET DMA03 AND DMA04 VALUES
         STI       R0,*+AR2(2)       ; (SRC INDEX)
         STI       R0,*+AR4(2)
         LDI       @CTRL2,R0
         LDI       @CTRL3,R1
         STI       R0,*AR2           ; (CTRL)
||       STI       R1,*AR4
         STI       AR3,*+AR2(3)      ; (COUNTER)
         STI       AR3,*+AR4(3)
         STIK      2,*+AR2(IR0)      ; (DST INDEX)
         STIK      2,*+AR4(IR0)
         STI       AR4,*+AR2(6)      ; (LINK POINTER)
```

```
*********************************************************************
* DMA : – TRANSFER COLUMN 1 TO ON–CHIP RAM (BLOCK1)
*********************************************************************
         LDI       @BLOCK0,AR6
         LDI       @BLOCK1,R7
         LDI       @MYID,R0
         LDI       @NROWS,AR5         ; AR5 = Q = COLUMN COUNTER
         MPYI      AR5,R0             ; R0 = Q*MYNODE
         LSH       1,R0               ; R0 = 2*Q*MYNODE

         LDI       @MATR,AR7
         ADDI      R0,AR7             ; MATRIX POINTER = &MATRIX[2*Q*MYNODE][0]
         LDI       @BLOCK1,R7         ; POINTER TO DMA BUFFER
         LDI       @BLOCK0,AR6        ; POINTER TO FFT BUFFER

         ADDI      2,AR7,R0
         ADDI      1,R0,R1
         STI       R0,*+AR2(1)        ; SRC  ADDRESS (Re PART)
||       STI       R1,*+AR4(1)        ; SRC  ADDRESS (Im PART)
         STI       R7,*+AR2(4)        ; DST  ADDRESS (DMA_BUFFER)
         ADDI      1,R7,R0
         STI       R0,*+AR4(4)        ; DST  ADDRESS (DMA_BUFFER+1)

         LDI       @DMA0,AR0          ; DMA START
         STI       AR2,*+AR0(6)
         STIK      0,*+AR0(3)
         LDI       @CTRL2,R4
         STI       R4,*AR0

*********************************************************************
* CPU : – TRANSFER COLUMN 0 TO ON–CHIP RAM (BLOCK0)
*        – FFT ON COLUMN 0
*********************************************************************
         SUBI3     2,AR3,RC           ; RC=N–2
         LDI       AR7,AR0            ; SOURCE ADDRESS
         LDI       AR6,AR1            ; DESTINATION ADDRESS
         RPTBD      LOOP3
         LSH3      1,AR3,IR1          ; SOURCE OFFSET = 2*N
         LDI       2,IR0              ; DESTINATION OFFSET
         LDF       *+AR0(1),R0        ; R0= X(I) IM

         LDF       *AR0++(IR1),R1     ; X(I) RE & POINTS TO X(I+1)
||       STF       R0, *+AR1(1)       ; STORE X(I) IM
LOOP3    LDF       *+AR0(1),R0        ; R0=X(I+1) IM
||       STF       R1,*AR1++(IR0)     ; STORE X(I) RE

         LAJ       CFFT
         LDF       *AR0,R1            ; LOAD X(N–1) RE
         NOP
         STF       R0,*+AR1(1)        ; STORE X(N–1) IM
||       STF       R1,*AR1            ; STORE X(N–1) RE

*********************************************************************
*  DMA: – MOVES FFT COLUMN (I) (Re PART) TO OFF–CHIP RAM
*        – MOVES FFT COLUMN (I) (Im PART) TO OFF–CHIP RAM
*        – MOVES COLUMN (I+2) (Re PART) TO ON–CHIP RAM
*        – MOVES COLUMN (I+2) (Im PART) TO ON–CHIP RAM
*
*  CPU: – FFT ON COLUMN (I+1)
*********************************************************************
```

```
         LDI       @P02,AR1       ; DMA0
         LDI       @P01,AR0       ; P01
         SUBI      3,AR5          ; AR5=Q–3: (Q–2) DMA TRANSFERS
         ADDI      1H,AR6,R0
B4       TSTB      @MASK,IIF
         BZAT      B4

* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
         STI       R0,*+AR1(1)    ; SOURCE
         ADDI      1H,AR7,R0
         STI       R0,*+AR1(4)    ; DST

* DMA01: BIT–REVERSED TRANSFER OF COLUMN (Re)
         STI       AR6,*+AR0(1)   ; SOURCE
         STI       AR7,*+AR0(4)   ; DST

* DMA03: TRANSFER NEXT COLUMN (Re)
* DMA04: TRANSFER NEXT COLUMN (Im)

         LDI       @P03,AR4       ; AR0 POINTS TO DMA03
         LDI       @P04,AR2       ; AR1 POINTS TO DMA04
         ADDI      2,AR7          ; R0: POINTS TO NEXT COLUMN
         ADDI      2,AR7,R0
         STI       R0,*+AR4(1)    ; SOURCE: (RE)
         AND       0H,IIF         ; CLEAR FLAG
         STI       AR6,*+AR4(4)   ; DESTINATION: BLOCK1(RE)
         ADDI      1,R0           ; POINTS TO IMAGINARY PART
         STI       R0,*+AR2(1)    ; SOURCE: (IM)
         ADDI      1,AR6,R1
         STI       R1,*+AR2(4)    ; DESTINATION: BLOCK0(IM)
         LDI       @DMA0,AR1      ; GIVE THE START
         LDI       R7,R2
         LDI       AR6,R7         ; R7: BLOCK1
         STIK      0,*+AR1(3)
         STI       AR0,*+AR1(6)

* FFT ON CURRENT COLUMN
         LAJ       CFFT
         LDI       R2,AR6         ; AR6: POINTER FOR NEXT FFT
         LDI       @CTRL2,R0
         STI       R0,*AR1        ; START (DMA)
         DBUD      AR5,B4
         LDI       @P02,AR1       ; DMA0
         LDI       @P01,AR0       ; DMA0
         ADDI      1H,AR6,R0

***********************************************************
*  DMA:   TRANSFER LAST FFT RESULT
*  CPU:   FFT ON LAST COLUMN
***********************************************************

         LDI       @P02,AR1       ; DMA0
B5       TSTB      @MASK,IIF
         BZAT      B5
```

```
* DMA02: BIT–REVERSED TRANSFER OF LAST RESULT (Im)
        ADDI      1H,AR6,R0
        STI       R0,*+AR1(1)        ; SOURCE
        ADDI      1H,AR7,R0
        STI       R0,*+AR1(4)        ; DST
        LDI       @P01,AR0           ; P01
        LDI       @CTRL0,R0
        STI       R0,*AR1
        STI       AR6,*+AR0(1)       ; SOURCE
        STI       AR7,*+AR0(4)       ; DST
        LDI       @DMA0,AR1          ; GIVE THE START
        ADDI      2,AR7
        AND       0,IIF
        STIK      0,*+AR1(3)
        STI       AR0,*+AR1(6)
        LDI       @CTRL2,R0
        STI       R0,*AR1            ; START (DMA)

* FFT ON LAST COLUMN
        LAJ       CFFT
        LDI       R7,R2
        LDI       AR6,R7
        LDI       R2,AR6
        SUBI3     2,AR3,RC           ; RC=N–2
        LDI       AR6,AR0            ; SOURCE
        LDI       AR7,AR1            ; DESTINATION
        RPTBD     B6
        LD        AR3,IR0
        LSH3      1,AR3,IR1
        LDF       *+AR0(1),R0        ; R0= X(I) IM

* LOOP
        LDF       *AR0++(IR0)B,R1    ; X(I) RE & POINTS TO X(I+1)
||      STF       R0, *+AR1(1)       ; STORE X(I) IM
B6      LDF       *+AR0(1),R0        ; R0=X(I+1) IM
||      STF       R1,*AR1++(IR1)     ; STORE X(I) RE

* STORE LAST VALUE
B7      TSTB      @MASK,IIF
        BZ        B7
        LDF       *AR0++(IR0)B,R1    ; LOAD X(N–1) RE
||      STF       R0,*+AR1(1)        ; STORE X(N–1) IM
        STF       R1,*AR1            ; STORE X(N–1) RE

        LDI       @TIMER,AR2         ; OPTIONAL: BENCHMARKING (TIME_READ)
        LDI       *+AR2(4),R0        ; TCOMP = R0

t2      B         t2
        .end
```

### SHB.CMD

```
input.obj
shb.obj
spinput.obj
sintab.obj
synch.obj
–m shb.map
–lmylib.lib

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

MEMORY
{
        ROM:      o = 0x00000000 l = 0x1000
        BUF0:     o = 0x002ff800 l = 0x200
        RAM0:     o = 0x002ffa00 1 = 0x200
        BUF1:     o = 0x002ffc00 l = 0x200
        RAM1:     o = 0x002ffe00 l = 0x200
        LM:       o = 0x40000000 l = 0x10000
        GM:       o = 0x80000000 l = 0x20000
}

SECTIONS
{
        INPUT          :{}   >   GM      /* Input data             */
        .text          :{}   >   LM
        .data          :{}   >   RAM1    /* Sine table             */
        STACK          :{}   >   RAM1
        DMA_AUTOINI    :{}   >   RAM1    /* DMA autoinit. values   */
        SYNCH          :{}   >   GM      /* Synchronization        */
}
```

# Appendix C: Parallel 2-D FFT (Distributed-Memory Version)

## C.1. DIS1.C: Distributed-Memory Implementation (C Program) — DMA Used Only for Interprocessor Communication

### *DIS1.C*

```
/*************************************************************************************

DIS1.C :  Parallel 2–dimensional complex FFT (Distributed–memory)
          – single–buffered version
          – DMA is used only for interprocessor communication

Requirements:       P > 0     Q > 0

To run:

          cl30 –v40 –g –mr –o2 dis1.c
          asm30 –v40 sintab.asm
          asm30 –v40 input0.asm
          asm30 –v40 input1.asm
          lnk30 input0.obj dis1.obj disc.cmd –o a0.out
          lnk30 input1.obj dis1.obj disc.cmd –o a1.out

Notes:    1) Before running, initialize the my_node variable to the corresponding
value
          using the ’C40 emulator or an assembly file.

          2) Output: columnwise

*************************************************************************************/
#define   SIZE      4                 /* FFT size                   */
#define   LOGSIZE   2                 /* log (FFT size)             */
#define   P         2                 /* number of processors       */
#define   Q         SIZE/P            /* rows/cols. per processor   */
#define   BLOCK0    0x002ff800        /* on–chip buffer 0           */
#define   DMA0      0x001000a0        /* DMA0 address               */
#define   SWAP(x,y) temp = *x; *x = *y ; *y = temp;
#define   WAIT_DMA(x) while ((0x03c00000 & *x)!=0x02800000)

extern    void cfftc(),               /* C–callable complex FFT     */
          cmove(),                    /* CPU complex move           */
          cmoveb(),                   /* CPU bit–reversed move      */
          exchange(),                 /* set DMA in split mode      */
          set_dma();                  /* set DMA register values    */
extern    float MATRIX[Q][SIZE*2];    /* input matrix               */

float     *block0   = (float *)BLOCK0,
          *MM[Q], *ptr, temp;
int       *dma0     = (int *)DMA0;
int       my_node,
          q         = Q,
          q2        = Q*2,
           i,j,ii,i2,k1;
```

```
#if     (P == 4)
        int       port [P][P]=   { 0,0,4,3,
                                   3,0,0,4,
                                   1,3,0,0,
                                   0,1,3,0};    */Connectivity matrix: processor i
                                                is connected to processor j through
                                                port[i][j]: system specific PPDS  */
#else
        int       port[P][P] =   { 0,0,3,0);/* when P=2 */
#endif


struct NODE {
        int  id;                 /* dst_node ID*q2  */
        int  port;               /* port number to which is connected */
        int  *dma;               /* dma address attached to that port */
        } dnode[P+1];

int     tcomp;

/***********************************************************************************/
main()
{
asm(” or 1800h,st”);

for (i=0;i<Q;i++)   MM[i]=MATRIX[i];     /*  accessing assembly variables  */

/*************************  FFT on rows  ******************************************/
t0:
        time_start(0);                          /* benchmarking (C40 timer)   */
        cmove (&MM[0][0],block0,2,2,SIZE);      /* move row 0 to on–chip RAM  */
        cfftc (block0,SIZE,LOGSIZE);            /* FFT on row 0               */
        cmoveb (block0,&MM[0][0],SIZE,2,SIZE);  /* move back FFT(row 0)       */
        for (j=1;j<P;j++)    {                  /* interprocessor comm.       */
        i= (my_node ^ j);                       /* destination node           */
        dnode[j].id   = (i * q2);               /* destination node * q2      */
        dnode[j].port = port[my_node][i];       /* port to be used */
        dnode[j].dma = dma0 + (dnode[j].port <<4);
        exchange (dnode[j].dma,dnode[j].port,&MM[0][dnode[j].id],q2);
}

for     (i=1;i<q;++i)  {                        /* loop over other rows       */
        cmove (&MM[i][0],block0,2,2,SIZE);      /* move row i to on–chip RAM  */
        cfftc (block0,SIZE,LOGSIZE);            /* FFT on row i               */
        cmoveb (block0,&MM[i][0],SIZE,2,SIZE);  /* move back FFT (row i)      */
for (j=1;j<P;j++)  {                            /* interprocessor comm.       */
        WAIT_DMA(dnode[j].dma);                 /* wait for DMA to finish     */
        exchange (dnode[j].dma,dnode[j].port,&MM[i][dnode[j].id],q2);
        }
}
/**************************   FFT on columns *****************************/
for      (j=1;j<P;j++)   WAIT_DMA(dnode[j].dma); /* wait for DMAs to finish*/

t1:
for  (i=0;i<(q–1);i++) {                        /* loop over (q–1) columns    */
     ptr = &MM[i][i*2]; k1 = 2*(q–i);
     for (j=0;j<P;j++,ptr +=q2)                 /* submatrices transposition  */
     for (ii=2;ii<k1;ii+=2) {
         SWAP((ptr+ii),(ptr+ii*SIZE));          /* exchange Re parts          */
         SWAP((ptr+ii+1),(ptr+ii*SIZE+1));      /* exchange Im parts          */
     }
     cmove (&MM[i][0],block0,2,2,SIZE);         /* FFT on column (i–1)        */
     cfftc (block0,SIZE,LOGSIZE);
     cmoveb (block0,&MM[i][0],SIZE,2,SIZE);

}/*for*/
         cmove (&MM[q–1][0],block0,2,2,SIZE);   /* FFT on last column         */
         cfftc (block0,SIZE,LOGSIZE);
         cmoveb (block0,&MM[q–1][0],SIZE,2,SIZE);

tcomp =time_read(0);                            /* benchmarking               */
t2: ;
} /*main*/
```

## DISC.CMD

```
sintab.obj
–c
–stack 0x0040
–lrts40.lib
–lprts40r.lib
–lmylib.lib
–m disc.map

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
         ROM:      org = 0x00         len = 0x0800
         BUF0:     org = 0x002ff800   len = 0x0400      /* on–chip RAM block 0    */
         RAM1:     org = 0x002ffc00   len = 0x0400      /* on–chip RAM block 1    */
         LM:       org = 0x40000000   len = 0x10000     /* LOCAL MEMORY           */
         GM:       org = 0x80000000   len = 0x20000     /* GLOBAL MEMORY          */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
         INPUT:    {} > LM                          /* INPUT MATRIX               */
         .text:    {} > LM
         .cinit:   {} > RAM1                        /* INITIALIZATION TABLES      */
         .stack:   {} > RAM1                        /* SYSTEM STACK               */
         .bss :    {} > RAM1
         .data:    {} > RAM1                        /* SINE TABLE                 */
}
```

### INPUT0.ASM

```
*********************************************************************************
*
*  INPUT0.ASM : input matrix 2 x 4 (Distributed–memory program)for processor 0
*                 –number of processors in the system: 2
*
*********************************************************************************

        .global   _MATRIX
        .sect     ”INPUT”

_MATRIX

        .float 130.0,90.0    ;[0][0]
        .float 66.0,230.0    ;[0][1]
        .float 205.0,136.0   ;[0][2]
        .float 15.0,187.0    ;[0][3]
        .float 150.0,164.0   ;[1][0]
        .float 222.0,44.0    ;[1][1]
        .float 95.0,243.0    ;[1][2]
        .float 80.0,60.0     ;[1][3]
        .end
```

### INPUT1.ASM

```
*********************************************************************************
*
*  INPUT1.ASM : input matrix 2 x 4 (Distributed–memory program) for processor 1
*              –number of processors in the system: 2
*
*********************************************************************************

        .global   _MATRIX
        .sect     ”INPUT”

_MATRIX

        .float 97.0,36.0     ;[2][0]
        .float 215.0,191.0   ;[2][1]
        .float 209.0,239.0   ;[2][2]
        .float 161.0,22.0    ;[2][3]
        .float 117.0,238.0   ;[3][0]
        .float 203.0,44.0    ;[3][1]
        .float 104.0,187.0   ;[3][2]
        .float 195.0,177.0   ;[3][3]
        .end
```

## C.2. DIS2.C: Distributed-Memory Implementation (C Program) — DMA Used for Interprocessor Communication and Matrix Transposition

### *DIS2.C*

```
/*************************************************************************************

DIS2.C :           Parallel 2–dimensional complex FFT (Distributed–memory)
                   – single–buffered version
                   – DMA is used for interprocessor communication and matrix
                   transposition
Requirements:      P > 0 ; Q > 0
To run:
         cl30 –v40 –g –mr –o2 dis2.c
         asm30 –v40 sintab.asm
         asm30 –v40 input0.asm
         asm30 –v40 input1.asm
         lnk30 input0.obj dis2.obj disc.cmd –o a0.out
         lnk30 input1.obj dis2.obj disc.cmd –o a1.out
Notes:   1) Before running, initialize the my_node variable to the corresponding value
         using the ’C40 emulator or an assembly file.

         2) Output: columnwise

*************************************************************************************/
#define  SIZE           4         /* FFT size                  */
#define  LOGSIZE        2         /* log (FFT size)            */
#define  P              2         /* number of processors      */
#define  Q              SIZE/P    /* rows/cols. per processor  */

#define  BLOCK0    0x002ff800     /* on–chip buffer 0          */
#define  DMA0      0x001000a0     /* DMA0 address              */
#define  SWAP(x,y) temp = *x; *x = *y ; *y = temp;
#define  WAIT_DMA(x) while ((0x03c00000 & *x)!=0x02800000)

extern   void      cfftc(),       /* C–callable complex FFT    */
                   cmove(),       /* CPU complex move          */
                   cmoveb(),      /* CPU bit–reversed move     */
                   exchange(),    /* set DMA in split mode     */
                   set_dma();     /* set DMA register values   */

extern   float     MATRIX[Q][SIZE*2];    /*  input matrix              */
int      MEM[35*P];                      /*  autoinitialization values:
                                         5 set of different values
                                         per processor (5*7)       */

float    *block0   = (float *)BLOCK0,
         array[Q*2], *MM[Q], *ptr, temp;

int      *dma0     = (int *)DMA0,
         *dma[P],
         *mem      = MEM,
         ctrl2     = 0x00c00008,     /* autoinit,TCC=0,DMA low pr.     */
         ctrl3     = 0x00c40004,     /* no autoinit,TCC=1,DMA low pr.  */
         mask      = 0x02000000,
         *mp;
```

```
int      my_node,
         size2 = SIZE*2,
         q = Q,
         q2 = Q*2,
         i,ii,j,i2,k1,k2;
#if      (P == 4)
int      port[P][P] = { 0,0,4,3,
                        3,0,0,4,
                        1,3,0,0,
                        0,1,3,0 };   /*  Connectivity matrix: processor i
                                         is connected to processor j through
                                         port[i][j]:system specific(PPDS)     */
#else

int      port[P][P] = { 0,0,3,0};    /* when P = 2             */
#endif

struct NODE {
            int id;               /* will keep (destination node ID* q2)           */
            int port;             /* port number to which dst node is connected    */
            int *dma;             /* dma address to be used with port              */
                   } dnode[P+1];

int      tcomp;

/**********************************************************************************/
main()
{
asm(” or 1800h,st”);                 /* cache enable              */

for (i=0;i<Q;i++) MM[i]=MATRIX[i];   /* accessing assembly vars   */

/**********************    FFT on rows  *********************************************/
t0:
time_start(0);                              /* benchmarking                */
cmove (&MM[0][0],block0,2,2,SIZE);          /* move row 0 to on–chip RAM  */
cfftc (block0,SIZE,LOGSIZE);                /* FFT on row 0                */
cmoveb (block0,&MM[0][0],SIZE,2,SIZE);      /* move back FFT(row 0)        */
for      (j=1;j<P;j++)    {                 /* interprocessor comm.        */
         i = (my_node ^ j);                 /* destination node            */
         dnode[j].id = (i * q2);            /* destination node * q2       */
         dnode[j].port = port[my_node][i];  /* port to be used             */
         dnode[j].dma = dma0 + (dnode[j].port <<4); /* dma to be used      */
         exchange (dnode[j].dma,dnode[j].port,&MM[0][dnode[j].id],q2);
         }

for      (i=1;i<q;++i)  {                       /* loop over other rows       */
         cmove (&MM[i][0],block0,2,2,SIZE);     /* move row i to on–chip RAM  */
         cfftc (block0,SIZE,LOGSIZE);           /* FFT on row i               */
         cmoveb (block0,&MM[i][0],SIZE,2,SIZE); /* move back FFT (row i)      */
         for (j=1;j<P;j++)    {                 /* interprocessor comm.       */
            WAIT_DMA(dnode[j].dma);             /* wait for DMA to finish     */
            exchange (dnode[j].dma,dnode[j].port,&MM[i][dnode[j].id],q2);
         }
}
```

```
/**************************   FFT on columns *****************************/

for      (j=1;j<P;j++) WAIT_DMA(dnode[j].dma)    /*  wait for DMAs in split mode to
                                                     finish                     */

t1: if (q==1) goto lastcol;                      /* no need for transposition  */

ptr=&MM[0][0];

for      (j=0;j<P;j++,ptr +=q2)                  /* transposition of row/col 0 */
for      (ii=2;ii<q2;ii+=2) {
         SWAP((ptr+ii),(ptr+ii*SIZE));           /* Re part                     */
         SWAP((ptr+ii+1),(ptr+ii*SIZE+1));       /* Im part                     */
         }

/* DMA0 transposes column/row 1 :
[1]: row(Re,Im) –> array(Re,Im)
[2.a]: col(Re) –> row(Re)     [2.b]: col(Im) –> row(Im)
[3.a]: array(Re) –> col(Re)   [3.b]: array(Im) –> col(Im)                      */

mp = mem;          ptr = &MM[1][2];

for      (j=0;j<(P–1);j++) {
         set_dma(mp,ctrl2,ptr,1,(q2–2),array,1,(mp+7));
         set_dma((mp+7),ctrl2,ptr,size2,(q–1),ptr,2,(mp+14));
         set_dma((mp+14),ctrl2,(ptr+1),size2,(q–1),(ptr+1),2,(mp+21));
         set_dma((mp+21),ctrl2,array,2,(q–1),ptr,size2,(mp+28));
         set_dma((mp+28),ctrl2,(array+1),2,(q–1),(ptr+1),size2,(mp+35));
         mp += 35; ptr += q2;
         }

set_dma(mp,ctrl2,ptr,1,(q2–2),array,1,(mp+7));
set_dma((mp+7),ctrl2,ptr,size2,(q–1),ptr,2,(mp+14));
set_dma((mp+14),ctrl2,(ptr+1),size2,(q–1),(ptr+1),2,(mp+21));
set_dma((mp+21),ctrl2,array,2,(q–1),ptr,size2,(mp+28));
set_dma((mp+28),ctrl3,(array+1),2,(q–1),(ptr+1),size2,0);
*(dma0+3) = 0; *(dma0+6) =(int)mem; *dma0=ctrl2;

cmove (&MM[0][0],block0,2,2,SIZE);       /* move column 0 to on–chip            */
cfftc(block0,SIZE,LOGSIZE);              /* FFT on column 0                     */
cmoveb (block0,&MM[0][0],SIZE,2,SIZE);   /* move FFT column 0 off–chip          */

for      (i=2;i<q–1;i++) {
         i2=2*i;
                                      /* Check IIF register to see if DMA0
                                      (unified mode) has finished              */
         asm(”WAIT:  TSTB @_mask,iif”);
         asm(” BZAT WAIT”);
         asm(” ANDN @_mask,iif”);
         mp        = mem;      ptr = &MM[i][i2]; k1 = (q–i); k2 =q2–i2;
         if        (k1>1) {
         for (j=0;j<P;j++)  {        /* DMA transposes row/column i    */
                   *(mp+1)  = *(mp+8)  = *(mp+11) = *(mp+25) = (int)ptr;
                   *(mp+15) = *(mp+18) = *(mp+32) = (int)(ptr+1) ;
                                  /* counter                           */
                   *(mp+10) = *(mp+17) = *(mp+24) = *(mp+31) =k1;
                   *(mp+3) = k2;
                   mp += 35; ptr += q2;  /* points to next submatrix   */
         }
         *(dma0+3) = 0; *(dma0+6) = (int)mem; *dma0=ctrl2;
}/* if */

         cmove (&MM[i–1][0],block0,2,2,SIZE);
         cfftc (block0,SIZE,LOGSIZE);    /* FFT on column (i–1)        */
         cmoveb (block0,&MM[i–1][0],SIZE,2,SIZE);
         }/* for */

lastcol:

         asm(”WAIT2:  TSTB @_mask,iif”);
         asm(” BZAT WAIT2”);
         asm(” ANDN @_mask,iif”);

cmove (&MM[q–2][0],block0,2,2,SIZE);
cfftc (block0,SIZE,LOGSIZE);             /* FFT on column (q–2)        */
cmoveb (block0,&MM[q–2][0],SIZE,2,SIZE);
cmove (&MM[q–1][0],block0,2,2,SIZE);
cfftc (block0,SIZE,LOGSIZE);             /* FFT on last column         */
cmoveb (block0,&MM[q–1][0],SIZE,2,SIZE);

tcomp =time_read(0);                     /* Optional: Benchmarking     */

t2: ;

}/* main */
```

## C.3. DIS2.ASM: Distributed-Memory (’C40 Assembly Program) — DMA Used for Interprocessor Communication and Matrix Transposition

### *DIS2.ASM*

```
**********************************************************************
*
*       DIS2.ASM :  TMS320C40 Parallel  2–dimensional complex FFT
*                  – distributed–memory single–buffered version
*                  – DMAs are used for interprocessor communication
*                    and for matrix transposition
*
*       Routines used: cfft.asm (complex FFT)
*
*       Requirements :       Number of processors = P > 1
*                      Rows/columns per processor = Q >= 4
*
*       To run:
*
*       asm30 –v40 –g –s dis2.asm
*       asm30 –v40 –g –s dpinput.asm
*       asm30 –v40 –g –s ssintab.asm
*       asm30 –v40 –g –s 0.asm
*       asm30 –v40 –g –s 1.asm
*       asm30 –v40 –g –s input0.asm
*       asm30 –v40 –g –s input1.asm
*       lnk30 dis.cmd 0.obj input0.obj –o a0.out
*       lnk30 dis.cmd 1.obj input1.obj –o a1.out
*
**********************************************************************/

        .global N               ; fft size
        .global P               ; number of processors
        .global Q               ; rows/columns per processor
        .global MYNODE          ; processor ID
        .global _PORT           ; port matrix address
        .global _MATRIX         ; input matrix address
        .global _DMAMEM         ; memory address for autoinitialization values
        .global _DMALIST        ; space reserved to store addresses  of the
                                ; DMAs used for interprocessor communication
        .global _CTRLIST        ; space for control register values
                                ; for DMAs used for interprocessor comm.
        .global _DSTQLIST       ; space reserved to store (dst_node*q) values
                                ; to determine the source address for each
                                ; DMA interprocessor communication
        .global _ARRAY          ; buffer to be used in matrix transposition
                                ; using DMA
        .global CFFT            ; 1d–fft subroutine
        .global C2DFFT          ; entry point for execution

_STACK  .usect ”STACK”,10h
        .text
```

```
FFTSIZE  .word     N
PROC     .word     P
NROWS    .word     Q
MYID     .word     MYNODE
PORT     .word     _PORT
MATR     .word     MATRIX
BLOCK0   .word     002FF800H      ; ram block 0
STACK    .word     _STACK         ; stack address
DMA0     .word     001000a0H      ; DMA0 address
DMALIST  .word     _DMALIST
CTRLIST  .word     _CTRLIST
DSTQLIST .word     _DSTQLIST
DMAMEM   .word     _DMAMEM        ; pointer to autoinit. values in memory
SMASK    .word     02000000H      ; to check if DMA unified mode has finished
                                  ; using the IIF register
DMAMASK .word      03C00000H      ; to check if DMA in split mode has finished
                                  ; using the start fields in the DMAs control
                                  ; register
SPLITD   .word     02800000H
CTRL2    .word     00C00008H      ; control register word: autoinit. , TCC = 0
CTRL3    .word     00C40004H      ; control register word: no autoinit., TCC = 1
CONTROL  .word     03C040D4H      ; control register word: split mode for
                                  ; interprocessor communication
ARRAY    .word     _ARRAY
PORT0    .word     00000000H      ; this values help to set DMA control registers
PORT1    .word     00008000H      ; with the corresponding port values for the
PORT2    .word     00010000H      ; port field
PORT3    .word     00018000H
PORT4    .word     00020000H
PORT5    .word     00028000H
PORTS    .word     PORT0
ENABLE   .word     24924955H      ; enable port interrupts to DMAs
TIMER    .word     0100020h       ; Timer 0 address (benchmarking)

C2DFFT   LDP       FFTSIZE        ; load data page pointer
         LDI       2,IR0          ; destination offset
         LDI       2,IR1          ; source offset
         LDI       @STACK,SP      ; initialize the stack pointer

t0:
         LDI       @TIMER,AR2     ; Optional: benchmarking : timer start
         STIK      –1,*+AR2(8)
         LDI       961,R0
         STI       R0,*AR2
         OR        9800h,ST       ; cache enable and set condition flag =1
                                  ; (to enable any primary register to modify
                                  ; condition flags)
         LDI       @FFTSIZE,AR3   ; ar3 = n = matrix size
         LDI       @NROWS,R7      ; r7 = n/p = q = rows/columns per processor
         LDI       @MATR,AR7      ; initialize matrix pointer
         LDI       @BLOCK0,AR6    ; ar6: pointer to the on–chip RAM block that
                                  ; contains the input data for FFT computation
```

```
*********************************************************
*                  FFT ON ROWS
*********************************************************

**********************
*   CPU MOVES ROW 0  *
*   TO ON–CHIP RAM   *
**********************

         SUBI3     2,AR3,RC           ; rc = n–2
         RPTBD     LOOP0
         LDI       AR7,AR0            ; source address  = row 0 = & x(0)
         LDI       AR6,AR1            ; destination address
         LDF       *+AR0(1),R0        ; R0 = x(i) Im

         LDF       *AR0++(IR1),R1     ; x(i) Re & points to x(i+1)
||       STF       R0, *+AR1(1)       ; store x(i) Im
LOOP0    LDF       *+AR0(1),R0        ; R0 = x(i+1) Im
||       STF       R1,*AR1++(IR0)     ; store x(i) Re

**********************
*    FFT ON ROW 0    *
**********************

         LAJ       CFFT               ; call 1d–fft routine (complex FFT)
         LDF       *AR0,R1            ; LOAD X(N–1) RE
         STF       R0,*+AR1(1)        ; STORE X(N–1) IM
         STF       R1,*AR1            ; STORE X(N–1) RE

**********************
*  CPU MOVES ROW 0   *
*  (BIT–REVERSED) TO *
*  EXTERNAL MEMORY   *
**********************

         LDI       AR6,AR0            ; SOURCE
         LDI       AR7,AR1            ; DESTINATION
         SUBI3     2,AR3,RC
         RPTBD     LOOP1
         LDI       AR3,IR0            ; SOURCE OFFSET FOR BIT–REVERSE = N
         LDI       2,IR1              ; DESTINATION OFFSET
         LDF       *+AR0(1),R0
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
LOOP1    LDF       *+AR0(1),R0
||       STF       R1,*AR1++(IR1)
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
         STF       R1,*AR1++(IR1)
```

```
**********************
*   INTERPROCESSOR   *
*   COMMUNICATION    *
*      (DMA)         *
**********************

         LDI       @DMALIST,AR4   ; keeps dma addresses
         LDI       @CTRLIST,AR5   ; keeps dma control registers
                                  ; according to port attached
         LDI       @DSTQLIST,AR6  ; keeps pointer equal to (dest_node* q2)
         LDI       @MYID,R4       ; my node–id
         LDI       @PROC,R3       ; number of processors
         LDI       @NROWS,R7      ; q = (N/P)
         LSH       1,R7,R2        ; r2= 2*q
         MPYI      R3,R4,AR0      ; P*mynode
         ADDI      @PORT,AR0      ; &port[mynode][0]
         LDI       @ENABLE,DIE    ; enable port interrupts to all DMAs
         SUBI      1,R3,IR0       ; j = loop counter = (P–1)


LOOP2
         XOR       IR0,R4,IR1     ; destination node = mynode ^ j
         MPYI      R2,IR1,R0      ; destination node * q2
         STI       R0,*+AR6(IR0)  ; DSTQLIST update
         ADDI      AR7,R0         ; pointer to matrix location to transfer
         LDI       *+AR0(IR1),AR2 ; port[mynode][dest_node]
         MPYI      16,AR2,AR1
         ADDI      @DMA0,AR1      ; DMA address
         STI       AR1,*+AR4(IR0) ; DMALIST update
         STI       R0,*+AR1(1)    ; src primary channel
         STI       R0,*+AR1(4)    ; dst secondary channel
         STI       R2,*+AR1(3)    ; counter primary channel
         STI       R2,*+AR1(7)    ; counter secondary channel
         ADDI      @PORTS,AR2
         LDI       @CONTROL,R0
         OR        *AR2,R0        ; DMA control register
         STI       R0,*+AR5(IR0)  ; CTRLIST update
         SUBI      1,IR0
         BNZD      LOOP2
         STIK      1,*+AR1(2)     ; src index primary channel
         STIK      1,*+AR1(5)     ; src index secondary channel
         STI       R0,*AR1        ; DMA start

*********************************************************
*                 (Q–1) ROWS
*********************************************************

**********************
*   CPU MOVES ROW I  *
*   TO ON–CHIP RAM   *
**********************

         LSH3      1,AR3,R0
         LDI       @BLOCK0,AR6
         ADDI      R0,AR7         ; AR7 POINTS TO ROW 1
         SUBI      2,R7,AR5       ; AR5 = Q–2
         LDI       AR7,AR0        ; SOURCE
         LDI       2,IR1
         SUBI3     2,AR3,RC       ; RC = N–2
```

```
LOOPR    RPTBD     LOOP3
         LDI       AR6,AR1            ; DESTINATION
         LDI       2,IR0              ; DESTINATION OFFSET
         LDF       *+AR0(1),R0        ; R0 = X(I) IM
         LDF       *AR0++(IR1),R1     ; X(I) RE & POINTS TO X(I+1)
||       STF       R0, *+AR1(1)       ; STORE X(I) IM
LOOP3    LDF       *+AR0(1),R0        ; R0 = X(I+1) IM
||       STF       R1,*AR1++(IR0)     ; STORE X(I) RE

**********************
*    FFT ON ROW I    *
**********************

         LAJ       CFFT               ; CALL 1D–FFT (COMPLEX)
         LDF       *AR0,R1            ; LOAD X(N–1) RE
         STF       R0,*+AR1(1)        ; STORE X(N–1) IM
         STF       R1,*AR1            ; STORE X(N–1) RE

**********************
*  CPU MOVES ROW I   *
*  (BIT–REVERSED) TO *
*  EXTERNAL MEMORY   *
**********************

         LDI       AR6,AR0            ; SOURCE
         LDI       AR7,AR1            ; DESTINATION
         SUBI3     2,AR3,RC
         RPTBD     LOOP4
         LDI       AR3,IR0            ; SOURCE OFFSET FOR BIT–REVERSE = N
         LDI       2,IR1              ; DESTINATION OFFSET
         LDF       *+AR0(1),R0
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
LOOP4    LDF       *+AR0(1),R0
||       STF       R1,*AR1++(IR1)
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
         STF       R1,*AR1++(IR1)

**********************
*   WAIT FOR DMAS    *
*     TO FINISH      *
**********************

                   * DMAS DONE
         LDI       @DMALIST,AR4       ; POINTS TO DMALIST
         LDI       @PROC,R3           ; R3 = NUM OF PROCESSORS
         ADDI      R3,AR4
         SUBI      2,R3,RC
         RPTBD     LLP
         LDI       @SPLITD,R0
         LDI       @DMAMASK,R1
         SUBI      1,AR4,AR0          ; AR0 = POINTS TO DMA[0]
         LDI       *AR0––(1),AR2
AGAINP   AND       *AR2,R1,R4
         XOR       R0,R4              ; =0 IF DMA FINISH
LLP      BNZ       AGAINP
```

```
**********************
*   INTERPROCESSOR   *
*   COMMUNICATION    *
*      (DMA)         *
**********************

         LDI       @DMALIST,AR0
         LDI       @CTRLIST,AR1
         LDI       @DSTQLIST,AR2
         LDI       @NROWS,R2
         LSH       1,R2           ; R2 = Q2
         SUBI      1,R3,IR0

LOOP5    LDI       *+AR0(IR0),AR4 ; DMA ADDRESS
         LDI       *+AR2(IR0),R6  ; (DSTNODE*Q2)
         ADDI      AR7,R6         ; POINTS TO MATRIX LOCATION TO TRANSFER
         STI       R6,*+AR4(1)    ; SOURCE  PRIMARY CHANNEL
         STI       R6,*+AR4(4)    ; SOURCE SECONDARY CHANNEL
         LDI       *+AR1(IR0),R0
         SUBI      1,IR0
         BNZD      LOOP5
         STI       R2,*+AR4(3)    ; PRIMARY COUNTER  = Q2
         STI       R2,*+AR4(7)    ; SECONDARY COUNTER  = Q2
         STI       R0,*AR4
         LSH3      1,AR3,R0
         ADDI      R0,AR7
         DBUD      AR5,LOOPR
         LDI       AR7,AR0        ; SOURCE
         LDI       2,IR1          ; SOURCE OFFSET
         SUBI3     2,AR3,RC

*********************************************************
*                     FFT ON COLUMNS
*********************************************************

**********************
*   WAIT FOR DMAS    *
*     TO FINISH      *
**********************

         LDI       @DMALIST,AR4   ; POINTS TO DMALIST
         LDI       @PROC,R3       ; R3 = NUM OF PROCESSORS
         ADDI      R3,AR4
         SUBI      2,R3,RC
         RPTBD     LLN
         LDI       @SPLITD,R0
         LDI       @DMAMASK,R1
         SUBI      1,AR4,AR0      ; AR0 = POINTS TO DMA[0]
         LDI       *AR0––(1),AR2
AGAINN   AND       *AR2,R1,R4
         XOR       R0,R4          ; =0 IF DMA FINISH
LLN      BNZ       AGAINN
```

```
**********************
* CPU TRANSPOSITION  *
*       ROW 0        *
**********************

t1:

        SUBI      1,R3,AR5
        LDI       @MATR,AR7       ; INITIALIZE POINTER TO COL 0
        LSH3      1,AR3,IR0       ; IR0 = 2N
        MPYI      AR5,R2,R4       ; R4  = (P–1)*Q2
        ADDI      AR7,R4          ; R4 = PTR
        ADDI      R2,R4
        SUBI3     2,R7,RC         ; RC = Q–2

LOOP10  RPTBD     LOOP11
        SUBI      R2,R4
        ADDI      2,R4,AR0        ; AR0 = PTR + 2
        ADDI      IR0,R4,AR1      ; AR1 = PTR + 2*SIZE
        LDF       *+AR0(1),R0     ; R0 =IM
||      LDF       *+AR1(1),R6     ; R6 =IM
        STF       R0, *+AR1(1)
||      STF       R6, *+AR0(1)
        LDF       *AR0,R0
||      LDF       *AR1,R6
LOOP11  STF       R0, *AR1++(IR0)
||      STF       R6, *AR0++(IR1)
        DBUD       AR5,LOOP10
        SUBI3     2,R7,RC
        LDI       @CTRL2,R3
        LDI       @DMAMEM,AR4     ; AR4 = MP

**********************
* DMA TRANSPOSITION  *
*       ROW 1        *
**********************

        SUBI      1,R7,R4         ; R4 = Q–1
        LDI       @DMA0,AR0
        LSH3      1,AR3,AR2       ; AR2 = 2N
        ADDI      AR7,AR2,R6      ; R6 = M[1]
        ADDI      2, R6           ; R6 = M[1][2] = FI
        ADDI      1,R6,R10        ; FI+1
        STI       AR4,*+AR0(6)
        STIK      0,*+AR0(3)
        LDI       @PROC,RC
        SUBI      2,RC ; LOOP (P–1) TIMES
        RPTBD     TROW1
        SUBI      2,R2,R8         ; R8 = Q2–2
        LDI       @ARRAY,R5
        ADDI      1,R5,R9         ; ARRAY–1

* MP

        STI       R6,*+AR4(1)     ; SOURCE
        STIK      1,*+AR4(2)      ; SRC INDEX
        STI       R8,*+AR4(3)     ; COUNTER
        STI       R5,*+AR4(4)     ; ARRAY
        STIK      1,*+AR4(5)      ; DST INDEX
        STI       R3,*AR4++(7)    ; CTRL
        STI       AR4,*–AR4(1)    ; LINK POINTER
```

```
* MP+7
        STI      R6,*+AR4(1)        ; SOURCE
        STI      AR2,*+AR4(2)       ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R6,*+AR4(4)        ; DST
        STIK     2,*+AR4(5)         ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

* MP+14
        STI      R10,*+AR4(1)       ; SOURCE
        STI      AR2,*+AR4(2)       ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R10,*+AR4(4)       ; DST
        STIK     2,*+AR4(5)         ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

* MP+21
        STI      R5,*+AR4(1)        ; SOURCE
        STIK     2,*+AR4(2)         ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R6,*+AR4(4)        ; DST
        STI      AR2,*+AR4(5)       ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

* MP+28
        STI      R9,*+AR4(1)        ; SOURCE
        STIK     2,*+AR4(2)         ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R10,*+AR4(4)       ; DST
        STI      AR2,*+AR4(5)       ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

        ADDI     R2,R6
TROW1   ADDI     1,R6,R10           ; FI+1

* MP
        STI      R6,*+AR4(1)        ; SOURCE
        STIK     1,*+AR4(2)         ; SRC INDEX
        STI      R8,*+AR4(3)        ; COUNTER
        STI      R5,*+AR4(4)        ; ARRAY
        STIK     1,*+AR4(5)         ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

* MP+7
        STI      R6,*+AR4(1)        ; SOURCE
        STI      AR2,*+AR4(2)       ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R6,*+AR4(4)        ; DST
        STIK     2,*+AR4(5)         ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER
```

```
* MP+14
        STI      R10,*+AR4(1)       ; SOURCE
        STI      AR2,*+AR4(2)       ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R10,*+AR4(4)       ; DST
        STIK     2,*+AR4(5)         ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

* MP+21
        STI      R5,*+AR4(1)        ; SOURCE
        STIK     2,*+AR4(2)         ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R6,*+AR4(4)        ; DST
        STI      AR2,*+AR4(5)       ; DST INDEX
        STI      R3,*AR4++(7)       ; CTRL
        STI      AR4,*–AR4(1)       ; LINK POINTER

* MP+28
        STI      R9,*+AR4(1)        ; SOURCE
        STIK     2,*+AR4(2)         ; SRC INDEX
        STI      R4,*+AR4(3)        ; COUNTER
        STI      R10,*+AR4(4)       ; DST
        STI      AR2,*+AR4(5)       ; DST INDEX
        LDI      @CTRL3,R0
        STI      R0,*AR4++(7)       ; CTRL
        STI      R3,*AR0            ; START DMA

**********************
*   CPU MOVES COL 0  *
*   TO ON–CHIP RAM   *
**********************

COLUMN0:
        LDI      AR7,AR0            ; SOURCE : AR7 : POINTS TO COL 0
        LDI      AR6,AR1            ; DESTINATION
        SUBI3    2,AR3,RC           ; RC=N–2
        RPTBD    LOOP8
        LDI      2,IR1              ; SOURCE OFFSET
        LDI      2,IR0              ; DESTINATION OFFSET
        LDF      *+AR0(1),R0        ; R0= X(I) IM
        LDF      *AR0++(IR1),R1     ; X(I) RE & POINTS TO X(I+1)
||      STF      R0, *+AR1(1)       ; STORE X(I) IM
LOOP8   LDF      *+AR0(1),R0        ; R0=X(I+1) IM
||      STF      R1,*AR1++(IR0)     ; STORE X(I) RE

**********************
*    FFT ON COL 0    *
**********************


        LAJ      CFFT
        LDF      *AR0,R1            ; LOAD X(N–1) RE
        STF      R0,*+AR1(1)        ; STORE X(N–1) IM
        STF      R1,*AR1            ; STORE X(N–1) RE
```

```
**********************
*  FFT MOVES COL. 0  *
*  (BIT–REVERSED) TO *
*  EXTERNAL MEMORY   *
**********************

         LDI       AR6,AR0            ; SOURCE = BLOCK0
         LDI       AR7,AR1            ; DESTINATION = MATRIX
         SUBI3     2,AR3,RC           ; RC=N–2
         RPTBD     LOOP9
         LDI       AR3,IR0            ; SOURCE OFFSET = IR0 = N  (BIT–REVERSE)
         LDI       2,IR1              ; DESTINATION OFFSET (COLUMNS) = IR1 = 2N
         LDF       *+AR0(1),R0
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)

LOOP9    LDF       *+AR0(1),R0
||       STF       R1,*AR1++(IR1)
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
         STF       R1,*AR1++(IR1)
         LSH3      1,AR3,R0
         ADDI      R0,AR7             ; AR7 POINTS TO COL 1

*********************************************************
*                (Q–2) COLUMNS
*********************************************************

         CMPI      2,R7               ; if Q=2 goto last column
         BZ        LASTCOL

**********************
*   WAIT FOR DMAS   *
*     TO FINISH      *
**********************

         LDI       2,AR5              ; AR5 = I

WAIT     TSTB      @SMASK,IIF
         BZAT      WAIT
         ANDN      @SMASK,IIF
         LDI       @DMA0,AR0
         LSH3      1,R7,R2            ; R2 = Q2
         ADDI      R0,AR7,R6          ; R6 = &M[2]
         LSH3      1,AR5,R0           ; R0 = I2
         ADDI      R0,R6              ; R6 = PTR
         LDI       @DMAMEM,AR4
         STI       AR4,*+AR0(6)
         STIK      0,*+AR0(3)
```

```
**********************
*  TRANSPOSE ALL P   *
*  SECTIONS OF ROW I *
**********************

         LDI       @PROC,R0
         SUBI      1,R0,RC             ; LOOP P TIMES
         RPTBD     TRANSP
         SUBI      AR5,R7,R4           ; R4 = Q–I
         LSH3      1,R4,R3             ; R3 = Q2–I2
         ADDI      1,R6,R10            ; R10= PTR+1
         STI       R6,*+AR4(1)         ; PTR
         STI       R6,*+AR4(8)
         STI       R6,*+AR4(11)
         STI       R6,*+AR4(25)
         STI       R10,*+AR4(15)       ; PTR+1
         STI       R10,*+AR4(18)
         STI       R10,*+AR4(32)
         STI       R4,*+AR4(10)        ; Q–I
         STI       R4,*+AR4(17)
         STI       R4,*+AR4(24)
         STI       R4,*+AR4(31)
         STI       R3,*+AR4(3)         ; Q2–I2
         ADDI      35,AR4              ; MP+=35
         ADDI      R2,R6               ; FI+=Q2

TRANSP   ADDI      1,R6,R10
         LDI       @CTRL2,R0
         STI       R0,*AR0             ; START DMA

**********************
*   CPU MOVES COL I  *
*   TO ON–CHIP RAM   *
**********************

         SUBI3     2,AR3,RC            ; RC=N–2
         LDI       AR7,AR0             ; SOURCE
         LDI       AR6,AR1             ; DESTINATION
         RPTBD     LOOP18
         LDI       2,IR1               ; SOURCE OFFSET = 2N
         LDI       2,IR0               ; DESTINATION OFFSET
         LDF       *+AR0(1),R0         ; R0= X(I) IM
         LDF       *AR0++(IR1),R1      ; X(I) RE & POINTS TO X(I+1)
||       STF       R0, *+AR1(1)        ; STORE X(I) IM

LOOP18   LDF       *+AR0(1),R0         ; R0=X(I+1) IM
||       STF       R1,*AR1++(IR0)      ; STORE X(I) RE

**********************
*    FFT ON COL I    *
**********************

         LAJ       CFFT
         LDF       *AR0,R1             ; LOAD X(N–1) RE
         STF       R0,*+AR1(1)         ; STORE X(N–1) IM
         STF       R1,*AR1             ; STORE X(N–1) RE
```

```
**********************
*  FFT MOVES COL. I  *
*  (BIT–REVERSED) TO *
*  EXTERNAL MEMORY   *
**********************

        LDI       AR6,AR0             ; SOURCE = BLOCK0
        LDI       AR7,AR1             ; DESTINATION = MATRIX
        SUBI3     2,AR3,RC            ; RC=N–2
        RPTBD     LOOP19
        LDI       AR3,IR0             ; SOURCE OFFSET = IR0 = N  (BIT–REVERSE)
        LDI       2,IR1               ; DESTINATION OFFSET
        LDF       *+AR0(1),R0
        LDF       *AR0++(IR0)B,R1
||      STF       R0,*+AR1(1)

LOOP19  LDF       *+AR0(1),R0
||      STF       R1,*AR1++(IR1)
        LSH3      1,AR3,R0            ; R0 = 2*N
        ADDI      1,AR5
        CMPI      R7,AR5
        BND       WAIT
        LDF       *AR0++(IR0)B,R1
||      STF       R0,*+AR1(1)
        STF       R1,*AR1++(IR1)
        ADDI      R0,AR7

*************************************************************************
*                       LAST COLUMN                                     *
*************************************************************************

WAIT2   TSTB      @SMASK,IIF
        BZAT      WAIT2
        ANDN      @SMASK,IIF

LASTCOL:

**********************
* CPU MOVES COL (N–1)*
*   TO ON–CHIP RAM   *
**********************

        LDI       AR7,AR0             ; SOURCE
        LDI       AR6,AR1             ; DESTINATION
        SUBI3     2,AR3,RC            ; RC=N–2
        RPTBD     LOOP28
        LDI       2,IR1               ; SOURCE OFFSET
        LDI       2,IR0               ; DESTINATION OFFSET
        LDF       *+AR0(1),R0         ; R0= X(I) IM
        LDF       *AR0++(IR1),R1      ; X(I) RE & POINTS TO X(I+1)
||      STF       R0, *+AR1(1)        ; STORE X(I) IM

LOOP28  LDF       *+AR0(1),R0         ; R0=X(I+1) IM
||      STF       R1,*AR1++(IR0)      ; STORE X(I) RE
```

```
**********************
*  FFT ON COL (N–1)  *
**********************

         LAJ       CFFT
         LDF       *AR0,R1            ; LOAD X(N–1) RE
         STF       R0,*+AR1(1)        ; STORE X(N–1) IM
         STF       R1,*AR1            ; STORE X(N–1) RE

**********************
* FFT MOVES COL.(N–1)*
*  (BIT–REVERSED) TO *
*  EXTERNAL MEMORY   *
**********************

         LDI       AR6,AR0            ; SOURCE = BLOCK0
         LDI       AR7,AR1            ; DESTINATION = MATRIX
         SUBI3     2,AR3,RC           ; RC=N–2
         RPTBD     LOOP29
         LDI       AR3,IR0            ; SOURCE OFFSET = IR0 = N  (BIT–REVERSE)
         LDI       2,IR1              ; DESTINATION OFFSET
         LDF       *+AR0(1),R0
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)

LOOP29   LDF       *+AR0(1),R0
||       STF       R1,*AR1++(IR1)
         LDF       *AR0++(IR0)B,R1
||       STF       R0,*+AR1(1)
         STF       R1,*AR1++(IR1)
         LDI       @TIMER,AR2         ; OPTIONAL: BENCHMARKING (TIME_READ)
         LDI       *+AR2(4),R0        ; TCOMP = R0


t2       BU        t2

         .end
```

### DPINPUT.ASM

```
***************************************************************************
*
* DPINPUT.ASM :     Input file for distributed–memory program with parallel
*                   system information
*
***************************************************************************

          .global   N             ; FFT size
          .global   M             ; LOG2 FFT
          .global   P             ; Number of processors
          .global   Q             ; Rows per processor
          .global   _PORT
          .global   _ARRAY        ; buffer to be used in matrix transposition
          .global   _DMAMEM       ; memory address for autoinitialization values
          .global   _DMALIST
          .global   _CTRLIST
          .global   _DSTQLIST

N         .set      16            ; FFT size
M         .set      4             ; LOG FFT
P         .set      2             ; number of processors
Q         .set      N/P           ; rows/columns per processor

          .text

*_PORT    .int      0,0,4,3       ; connectivity matrix: processor i is
*         .int      3,0,0,4       ; connected to processor j through port
*         .int      1,3,0,0       ; PORT[i][j]    (P = 4)
*         .int      0,1,3,0

_PORT     .int      0,0,3,0       ; P = 2

_DMAMEM   .space    35*P
_DMALIST  .space    P
_CTRLIST  .space    P
_DSTQLIST .space    P
_ARRAY    .space    2*Q
          .end
```

### SSINTAB.ASM

```
*****************************************************************
*
*   SSINTAB.ASM:    Table with twiddle factors for a 16–point CFFT
*         and data input. File to be linked with the
*         source code for a 16–point, radix–2 FFT.
*
*****************************************************************

          .global   SINE
          .data
SINE      .float    0.000000
          .float    0.382683
          .float    0.707107
          .float    0.923880
COSINE    .float    1.000000
          .float    0.923880
          .float    0.707107
          .float    0.382683
          .float    –0.000000
          .float    –0.382684
          .float    –0.707107
          .float    –0.923880
          .float    –1.000000
          .float    –0.923880
          .float    –0.707107
          .float    –0.382683
          .float    –0.000000
          .float    –0.382684
          .float    –0.707107
          .float    –0.923880
          .end
```

### DIS.CMD

```
dis2.obj
ssintab.obj
dpinput.obj
–lmylib.lib
–m dis.map

MEMORY
{
         ROM:      o = 0x00000000 l = 0x1000
         RAM0:     o = 0x002ff800 l = 0x400
         RAM1:     o = 0x002ffc00 l = 0x400
         LM:       o = 0x40000000 l = 0x10000
         GM:       o = 0x80000000 l = 0x20000
}

SECTIONS
{
         INPUT     :    {}    >   LM
         .text     :    {}    >   LM
         .data     :    {}    >   RAM1
         STACK     :    {}    >   RAM1
}
```

# Appendix D: Mylib.lib Routines

## D.1. CFFT.ASM: Assembly Language FFT Routine

### *CFFT.ASM*

```
**************************************************************************************
*
*       CFFT.ASM : TMS320C40 COMPLEX, RADIX–2, DIF FFT
*
*       GENERIC PROGRAM FOR A LOOPED–CODE RADIX–2 FFT COMPUTATION IN 320C40
*
*       THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 111.
*       THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY.  THE COMPUTATION
*       IS DONE IN–PLACE, BUT THE RESULT IS MOVED TO ANOTHER MEMORY
*       SECTION TO DEMONSTRATE THE BIT–REVERSED ADDRESSING.
*
*       THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A .DATA SECTION.
*       THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE GENERIC
*       NATURE OF THE PROGRAM.  FOR THE SAME PURPOSE, THE SIZE OF THE FFT
*       N AND LOG2(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED
*       DURING LINKING.
*
*       INPUT PARAMETERS:
*                         AR6: INPUT ADDRESS (BLOCK 0/1 ON–CHIP)
*                         R11: RETURN ADDRESS
*
*       REGISTERS MODIFIED:   R0,R1,R2,R3,R4,R5,R6,R8,R9,R10
*                             AR0,AR1,AR2,AR4,AR5
*                             IR0,IR1
*                             RC
*
**************************************************************************************

         .globl    CFFT           ; Entry point for execution
         .globl    N              ; FFT size
         .globl    M              ; LOG2(N)
         .globl    SINE           ; Address of sine table
         .text
*        INITIALIZE

FFTSIZ   .word     N
LOGFFT   .word     M
SINTAB   .word     SINE

CFFT     PUSH      DP
         PUSH      AR5
         LDP       FFTSIZ
         LDI       1,R8           ; Initialize repeat counter of first loop
         LDI       1,AR5          ; Initialize IE index (AR5=IE)
         LDI       @FFTSIZ,R10    ; R10=N
         LSH3      –2,R10,IR1     ; IR1=N/4, pointer for SIN/COS table
         LDI       @LOGFFT,R9     ; R9 holds the remain stage number
         LSH3      1,R10,IR0      ; IR0=2*N (because of real/imag)
         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       AR6,AR0        ; AR0 points to X(I)
         ADDI      R10,AR0,AR2    ; AR2 points to X(L)
```

```
*        FIRST LOOP
         ADDF      *AR0,*AR2,R0       ; R0=X(I)+X(L)
         SUBF      *AR2++,*AR0++,R1   ; R1=X(I)–X(L)
         ADDF      *AR2,*AR0,R2       ; R2=Y(I)+Y(L)
         SUBF      *AR2,*AR0,R3       ; R3=Y(I)–Y(L)
         STF       R2,*AR0– –         ; Y(I)=R2  and...
||       STF       R3,*AR2– –         ; Y(L)=R3
BLK1     STF       R0,*AR0++(IR0)     ; X(I)=R0  and...
||       STF       R1,*AR2++(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      AR6,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,AR2        ; (X(L),Y(L)) pointer
         ADDI      2,AR1
         LDF       *AR4,R6            ; R6=SIN

*  SECOND LOOP
         SUBF      *AR2,*AR0,R2       ; R2=X(I)–X(L)
         SUBF      *+AR2,*+AR0,R1     ; R1=Y(I)–Y(L)
         MPYF      R2,R6,R0           ; R0=R2*SIN
||       ADDF      *+AR2,*+AR0,R3

*                                     ; R3=Y(I)+Y(L)
         MPYF      R1,*+AR4(IR1),R3
||       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      *AR2,*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,*AR2++(IR0)     ; X(L)=R2*COS+R1*SIN
||       STF       R4,*+AR2           ; 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      AR6,AR1,AR0        ; (X(I),Y(I)) pointer
         SUBI      1,R8,RC
         LSH       1,R8               ; Increment loop counter
         BRD       LOOP               ; Next FFT stage (delayed)
         LSH       1,AR5              ; IE=2*IE
         LDI       R10,IR0            ; N1=N2
         SUBI3     1,R8,RC
END      BUD       R11
         POP       AR5
         POP       DP
         NOP
         .end
```

## D.2. CFFTC.ASM: Assembly Language FFT Routine (C-Callable)

### *CFFTC.ASM*

```
******************************************************************************
*
*       CFFTC.ASM : Complex radix–2 DIF 1–D FFT routine (C–callable)
*
*       Generic program for a lopped–code radix–2 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.
*       This data is included in a separate file to preserve the generic
*       nature of the program.  For the same purpose, the size of the FFT
*       N and log2(N) are defined in a .globl directive and specified
*       during linking.
*
*       Calling conventions:
*
*         cfftc((float *)input,int fft_size,int logfft)
*                   ar2 r2 r3
*
*         where     input     :   Complex vector address
*                   fft_size  :   Complex FFT size
*                   logfft    :   logarithm (base 2) of FFT size
*
*         Registers modified: R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10
*                             AR0,AR1,AR6,AR4,AR5
*                             IR0,IR1
*                             RC,DP
*
******************************************************************************

          .globl    SINE              ; Address of sine/cosine table
          .globl    _cfftc            ; Entry point for execution
          .text
SINTAB    .word     SINE

_cfftc:
          LDI       SP,AR0
          PUSH      DP
          PUSH      R4                ; Save dedicated registers
          PUSH      R5
          PUSH      R6
          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
||        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
```

## D.3. CMOVE.ASM: Complex-Vector Move Routine

### *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 length)
*                            ar2 r2 r3 rc rs
*
*       where     src            : Vector Source Address
*                 dst            : Vector Destination Address
*                 src_displ      : Source offset (real)
*                 dst_displ      : Destination offset (real)
*                 length         : Vector length (complex)
*
**************************************************************************************

        .global   _cmove

_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 length
        SUBI      2,RC               ; RC=length–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 length –2
        RPTBD     CMOVE
        LDI       R2,AR1             ; 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
```

## D.4. CMOVEB.ASM: Complex-Vector Bit-Reversed Move Routine

### *CMOVEB.ASM*

```
***************************************************************************************
*
*    CMOVEB.ASM :  TMS320C40 C–callable routine to bit–reversed move a complex
*                  float vector pointed by src, to an address pointed by dst.
*
*    Calling conventions:
*
*    void cmoveb ((float *)src,(float *)dst, int src_displ,int dst_displ,int length)*
*                       ar2 r2 r3 rc rs
*
*    where         src            : Vector Source Address
*                  dst            : Vector Destination Address
*                  src_displ      : Source offset (real)
*                  dst_displ      : Destination offset (real)
*                  length         : Vector length (complex)
*
***************************************************************************************

     .global       _cmoveb

_cmoveb:
         .if       .REGPARM == 0
         LDI       SP,AR0
         LDI       *–AR0(1),AR2       ; Source address
         LDI       *–AR0(4),IR1       ; Destination index (real)
         LDI       *–AR0(5),RC        ; Complex length
         SUBI      2,RC               ; RC=length–2
         RPTBD     CMOVEB
         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 length –2
         RPTBD     CMOVEB
         LDI       R2,AR1             ; source address
         LDI       R3,IR0             ; source index (real)
         LDF       *+AR2(1),R0
         .endif

*        loop
         LDF       *AR2++(IR0)B,R1
||       STF       R0,*+AR1(1)
CMOVEB   LDF       *+AR2(1),R0
||       STF       R1,*AR1++(IR1)

         POP       AR0
         BUD       AR0
         LDF       *AR2++(IR0)B,R1
||       STF       R0,*+AR1(1)
         STF       R1,*AR1
         NOP
         .end
```

## D.5. SET_DMA.ASM: Routine to Set DMA Register Values

```
**************************************************************************************
*
*       SET_DMA.ASM : TMS320C30/’C40 C–callable routine to set DMA register values
*
*       Calling conventions:
*
*       void set_dma   ((int *)dma, int ctrl, (float *)src, int src_index,
*                 int counter, (float *)dst, int dst_index, (int *)dma_link)
*
*       where     dma        :   DMA register address      :   ar2
*                 ctrl       :   Control Register          :   r2
*                 src        :   Source Address            :   r3
*                 src_index  :   Source Address Index      :   rc
*                 counter    :   Transfer Count            :   rs
*                 dst        :   Destination Address       :   re
*                 dst_index  :   Destination Address Index :   stack
*                 dma_link   :   Link Pointer              :   stack
*
**************************************************************************************

        .global   _set_dma
        .text

_set_dma:

        LDI       SP,AR0         ; Points to top of stack
        .if       .REGPARM == 0
        LDI       *–AR0(1),AR2   ; AR2 points to DMA registers
        LDI       *–AR0(2),R2    ; Control register
        LDI       *–AR0(3),R3    ; Source
        LDI       *–AR0(4),RC    ; Source index
        LDI       *–AR0(5),RS    ; Transfer counter
        LDI       *–AR0(6),RE    ; Destination address
        LDI       *–AR0(7),R0    ; Destination index
        LDI       *–AR0(8),R1    ; Link pointer
        .else
        LDI       *–AR0(1),R0    ; Destination index
        LDI       *–AR0(2),R1    ; Link pointer
        .endif

        STI       R3,*+AR2(1)    ; source address
        STI       RC,*+AR2(2)    ; source index
        STI       RS,*+AR2(3)    ; counter
        STI       RE,*+AR2(4)    ; destination address
        POP       AR0
        BUD       AR0
        STI       R0,*+AR2(5)    ; destination index
        STI       R1,*+AR2(6)    ; link pointer
        STI       R2,*AR2        ; control
        .end
```

## D.6. EXCHANGE.ASM: Routine for Interprocessor Communication

### *EXCHANGE.ASM*

```
************************************************************************************
*
*       EXCHANGE.ASM   : TMS320C40 C–callable routine to exchange
*                      two floating point vectors pointed by ”address” in
*                      each processor memory. This routine uses
*                      DMA in split mode with source/destination
*                      synchronization given by OCRDY/ICRDY respectively.
*
*       Calling conventions:
*
*       void exchange  ((int *) dma, int comport, (float *)address, int length)
*
*       where     dma        :   DMA address                  :   ar2
*                 comport    :   Comport number to be used    :   r2
*                 address    :   Floating–point vector address :   r3
*                 length     :   Vector length                :   rc
*
************************************************************************************
*       This routine requires that the communicating ’C4xs enter to the routine at
*       approximately the same time. This can be guaranteed by using a system with a
*       common reset or by using a system with a common reset or by using the PDM
*       (part  of the ’C4x emulator) when you start running the 2D-FFT application.
*       For systems without this capability, use exch2.asm instead of this routine.
************************************************************************************

        .global   _exchange

        .text
CONTROL .word     03C040D4H      ; DMA interrupt, R/W sync, split mode,
                                 ; CPU higher priority
PORT0   .word     00000000H
PORT1   .word     00008000H
PORT2   .word     00010000H
PORT3   .word     00018000H
PORT4   .word     00020000H
PORT5   .word     00028000H
PORTS   .word     PORT0
ENABLE  .word     24924955H      ; Enable interrupts to DMAS

_exchange:
        LDI       SP,AR0         ; Points to top of stack
        PUSH      DP

        .if       .REGPARM == 0

        LDI       *–AR0(1),AR2   ; DMA address
        LDI       *–AR0(2),R2    ; comport address
        LDI       *–AR0(3),R3    ; Memory address
        LDI       *–AR0(4),RC    ; Vector length
        .endif
        LDI       R2,AR1
        LDP       CONTROL
        STI       R3,*+AR2(1)    ; Source primary channel
        STI       RC,*+AR2(3)    ; Primary channel counter
        STI       R3,*+AR2(4)    ; Source secondary channel
        LDI       @CONTROL,R3
        STIK      1H,*+AR2(2)    ; Primary source index
        STIK      1H,*+AR2(5)    ; Secondary source index
        ADDI      @PORTS,AR1     ; Pointing to port to be used

        STI       RC,*+AR2(7)    ; Secondary channel counter
        OR        *AR1,R3        ; Selecting port in DMA control reg.
        STI       R3,*AR2
        LDI       @ENABLE,DIE
        POP       DP
        RETS

        .end
```

### EXCHANGE2.ASM

```
********************************************************************
*
*  EXCHANGE2.ASM :  TMS320C40 C–callable routine to exchange
*                   two floating point vectors pointed by ”address” in
*                   each processor memory. This routine uses
*                   DMA in split mode with source/destination
*                   synchronization given by OCRDY/ICRDY respectively.
*
*  Calling conventions:
*
*  void exchange ((int *) dma, int comport, (float *)address, int lenght)
*
*  where        dma       : DMA address                    : ar2
*               comport   : Comport number to be used      : r2
*               address   : Floating–point vector address  : r3
*               lenght    : Vector lenght (complex)        : rc
*
*******************************************************************
* This routine can be used in multiprocessing systems without a common start
*******************************************************************
        .global _exchange

        .text
CONTROL .word    03C040D4H ; DMA interrupt, R/W synch, split mode,
                           ; CPU higher priority
PORT0   .word    00000000H
PORT1   .word    00008000H
PORT2   .word    00010000H
PORT3   .word    00018000H
PORT4   .word    00020000H
PORT5   .word    00028000H
PORTS   .word    PORT0
ENABLE  .word    24924955H ; Enable interrupts to DMA’S
PORTADR .word    100040h   ; RMP: 8/13/93 :modified for async parallel
                           ; systems

_exchange:
        PUSH    DP

        .if      .REGPARM == 0
        LDI      SP,AR0    ; Points to top of stack
        LDI      *–AR0(1),AR2 ; DMA address
        LDI      *–AR0(2),R2  ; comport adress
        LDI      *–AR0(3),R3  ; Memory adress
        LDI      *–AR0(4),RC  ; Vector lenght
        .endif
        LDP      CONTROL

*** RMP: 8/13/93 :modified for async parallel systems
        MPYI     10h,R2,R0    ; This instructions synchronize the
        ADDI     @PORTADR,R0  ; processors at both end of the comm ports
        ADDI     1,R0,AR0     ; in systems where a common processor
        ADDI     2,R0,AR1     ; start is not offered. This is done by
        STI      R0,*AR1      ; sending/recieving a dummy word.
        LDI      *AR0,R0
***

        LDI      R2,AR1
        STI      R3,*+AR2(1)  ; Source primary channel
        STI      RC,*+AR2(3)  ; Primary channel counter
        STI      R3,*+AR2(4)  ; Source secundary channel
        LDI      @CONTROL,R3
        STIK     1H,*+AR2(2)  ; Primary source index
        STIK     1H,*+AR2(5)  ; Secondary source index
        ADDI     @PORTS,AR1   ; Pointing to port to be used

        STI      RC,*+AR2(7)  ; Secondary channel counter
        OR       *AR1,R3      ; Selecting port in DMA control reg.
        STI      R3,*AR2
        LDI      @ENABLE,DIE
        POP      DP
        RETS
        .end
```

## D.7. SYNCOUNT.ASM: Interprocessor Synchronization Routine

### *SYNCOUNT.ASM*

```
**************************************************************************************
*
*       syncount.asm   : assembly language synchronization routine to provide
*                      a global start for all the processors. Rotating priority
*                      for shared-memory access should be selected. The
*                      processors start with a cycle difference of maximum 3
                       instruction cycles, which for practical purposes is
*                      acceptable. This routine is C–callable and uses register
*                      for parameter passing.
*
*       Calling conventions: void syncount((int *)counter,int value)
                                            ar2            ,r2
*
**************************************************************************************

        .global _syncount
        .text

_syncount:
        .if       .REGPARM == 0
        LDI       SP,AR1
        LDI       *–AR1(1),AR2
        LDI       *–AR1(2),R2
        .endif
        LDII      *AR2,R1
        ADDI      1,R1
        CMPI      R1,R2
        STII      R1,*AR2
        BZ        L1

AGAIN   LDI       *AR2,R1
        CMPI      R1,R2
        BNZ       AGAIN
L1      RETS
        .end
```

# Parallel DSP for Designing Adaptive Filters

**Daniel Chen**
**Digital Signal Processing — Semiconductor Group**
**Texas Instruments Incorporated**

# Transmission of Still and Moving Images Over Narrowband Channels

**Stefan Goss, Wilhelm Vogt, Rodolfo Mann Pelz, Dirk Lappe**
**Communication Research Institute**
**Robert Bosch GmbH**

## Overview

The transmission of pictures over radio channels can be of great benefit to:

- Public authorities, such as the police and emergency services,
- Public transportation, such as railways, airlines and ships,
- Private citizens

Radio networks have a narrow bandwidth and therefore require low transmission rates. Moreover, radio channels are prone to interference, such as that caused by multipath propagation. This report describes channels and transmission methods in existing networks and shows how, by applying complex algorithms for coding images, you can develop a source codec on the basis of a multisignal processor system can be with the aim of achieving a source data rate as low as 8 kbps.

## Networks and Transmission Methods

### Nonpublic Land Mobile Telecommunication Network

The existing Nonpublic Land Mobile Telecommunication Network (NPLMTN) is characterized by narrow-band frequency modulation with a channel separation of 20 kHz in the 450-MHz region. For the considered low data rate video and speech transmission, the existing network structure can be used through application of available commercial equipment. For proper digital modulation methods with a constant envelope, consider the well-known variety of continuous phase modulation types. In this context, Gaussian minimum shift keying (GMSK) [1] is characterized by a relatively high bandwidth efficiency, which is achieved with a pulse shaping filter with Gaussian characteristics. A data rate of 16 kbps can be attained in this application for an ACI of –70 dB, which is a general requirement for single-channel-per-carrier land mobile radio systems.

After the system inherent noncoherent demodulation (limiter + discriminator), a modified maximum likelihood sequence estimation (MLSE) with the Viterbi Algorithm (VA) is performed to obtain the transmitted data sequence. This method takes the effects of nonideal intermediate frequency filters at the demodulator output into account.

Due to the underlying narrow-band transmission — in other words, the signal bandwidth is less than the so called coherence bandwidth of the mobile channel (50–500 kHz) — the propagation system is characterized by time-selective fading. The received signal equals the product of the transmitted signal and a complex stochastic Gaussian process, which exhibits a Rayleigh or Rice distributed envelope and a uniform distributed phase [2].

### Public Switched Telephone Network (PSTN)

You can transmit video and speech in the analog telephone network through commercially available modems with data rates up to 24 kbps (Codex 326XFAST) in a synchronous mode. This high bandwidth efficiency is achieved with trellis coded *m*–ary QAM modulation [3]. The maximum likelihood detection is performed with the VA.

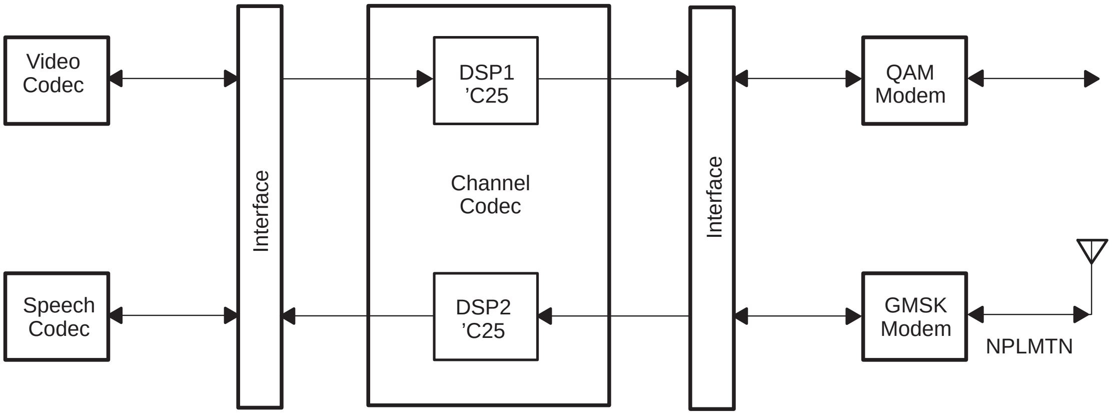
An exact characterization of the underlying propagation medium is a difficult task because a typical telephone channel cannot be defined. A simple model assumes a band-limited nonideal bandpass system and additive white Gaussian noise (AWGN). The linear distortion of the transmitted signal results in intersymbol interference (ISI), in which the error patterns are characterized by error bursts. Figure 1 depicts the system developed for video and speech transmission in the PSTN and NPLMTN.

Other applications of video and speech transmission are the analog and digital cordless telephone systems CT1 and DECT, and the analog and digital mobile radio telephone systems C and D (GSM) [4]. A current field of research includes the future public land mobile telecommunication system (FPLMTS), or the European version UMTS.

## Forward Error Correction (FEC)

The effects of radio channels on data transmission can be compensated for to some extent through application of FEC. Due to the limited resources (e.g., finite data rate), an efficient channel coding is a primary goal. Furthermore, due to the different sensitivity of individual symbols or symbol groups in the source coded data sequences (video, speech) to channel errors, you should devise an unequal error protection. In general, you can do this with block codes, but the application of convolutional codes allows ML decoding with soft decisions and channel state information. Rate compatible punctured convolutional (RCPC) codes [5] support dynamic allocation of redundancy with one encoder and one decoder. In the case of channels with memory, like the mobile channel, an additional interleaver and deinterleaver must be considered. Figure 1 shows two TMS320C25 digital signal processors (DSPs) implementing the corresponding algorithms for channel encoding.

**Figure 1. Digital Transmission System**

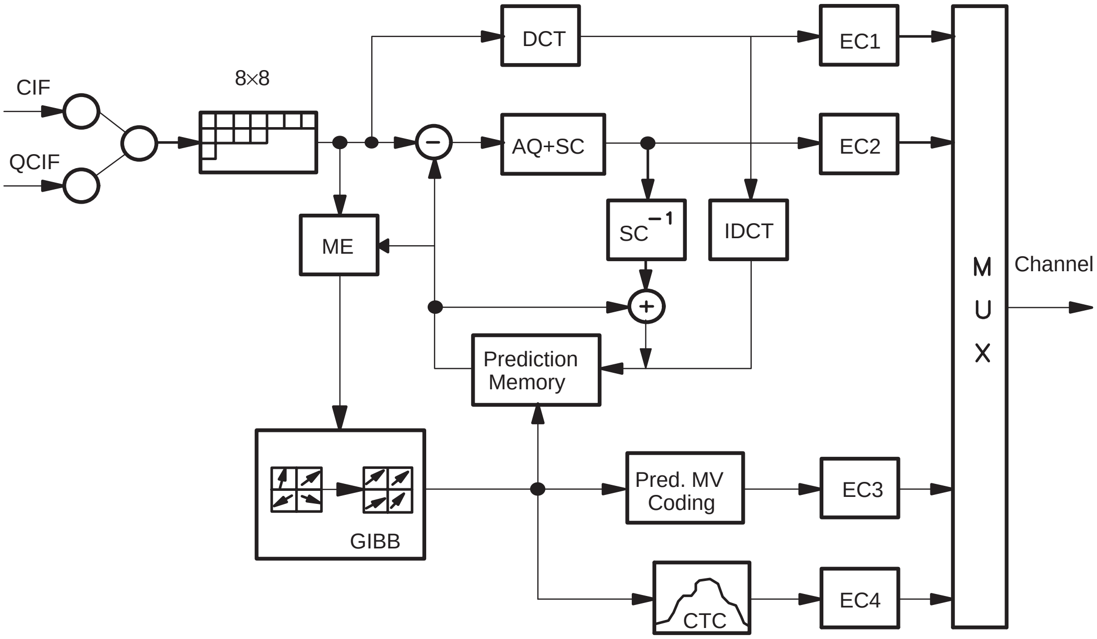


# Image Source Coding

## A Hybrid Codec for Moving Pictures

The source codec for moving pictures in an ISDN environment is committed and standardized as an H.261 recommendation. The data rate is fixed with regard to one B-channel (64 kbps). For data rates between 8 and 16 kbps, more efficient algorithms are necessary. A hybrid source codec for p × 8 kbps is shown in Figure 2. The following text describes this in detail and addresses the main differences with respect to the H.261 codec.

**Figure 2. Hybrid Codec for p × 8 kbps**



The input image format is QCIF (quarter common intermediate format) with a spatial resolution of 180 × 144 pixels for the luminance signal (Y) and 90 × 72 pixels for the color different components (U and V). The temporal resolution is reduced form 50 Hz to 6.25 Hz (factor 8). These operations are carried out in an additional preprocessing stage.

Like an ISDN codec, the hybrid codec is split into the motion estimation part and the coding stage of the prediction error. In the example shown in Figure 2, the QCIF image is the input information for a block generation stage that divides the input image into 396 blocks of 8 × 8 pixels. In the next step, a motion estimation (ME) for every block is performed by calculating a mean square error (MSE) between luminance blocks of the input image (also called original image) and the last coded and decoded image (prediction image). Every block of the original image is matched in a window of 40 × 40 pixels in the prediction image. The window is centered with regard to the block position in the original image. For a fixed number of dedicated positions in the window, the mean square error between the original and the predicted block is computed. The result is the motion vector for the minimum of mean square errors.

For typical videophone applications, the size of moving objects (i. e., persons) is higher than the block size. The minimization of the MSE per block leads to nonhomogeneous vector fields inside objects. The additional postprocessing stage (GIBB) smoothes the vector field with a model-based algorithm [6]. In this case, the maximum probability of the moving direction of objects is computed, starting with the vectors of the ME. This operation leads to a much more homogeneous vector field and to a subjectively better reconstruction and motion compensation in the prediction memory on the coder and the decoder side.

Another advantage is the reduced number of bits for the differential coded motion vectors. This reduction is greater than the additional bits required for coding the larger prediction error.

The coder control stage has the observed value of the bit consumption per frame. The necessary bits for a set of attributes and the coded motion vectors are subtracted from the total number of bits per frame. The remaining bits are used for coding the prediction errors of the blocks.

A DPCM Loop (differential pulse code modulation) performs the coding. The MSE for every block of the original image is sorted and compared with a fixed threshold to obtain an intraframe/interframe decision. The blocks with an MSE above this threshold are intraframe coded. As in an H.261 Codec, the block will be transformed in the frequency domain by a DCT (discrete cosine transformation). A linear quantization of the nonzero coefficients and a run-length coding of zero coefficients reduces the number of bits for the underlying block.

Blocks with an MSE below the intraframe/interframe threshold are interframe coded. After the MVs and the DCT information are transmitted, the remaining bits for this frame are divided by the computed amount of bits per interframe blocks. This gives the number of blocks, which could be coded in this frame.

In contrast to the H.261 concept, when an interframe-DCT is implemented, the block differences are coded in the time domain. This involves an adaptive quantization (AQ) and the coding of special structures (SC) belonging to the block [7].

In a first step, the probability density function of the pixel difference inside the blocks is determined. Next, a three-step symmetric quantizer is devised by computing the thresholds and the replacement values. The three replacement values are transmitted to the decoder.

Every $8 \times 8$ differential block is quantized with this quantizer function and then divided into sixteen $2 \times 2$ blocks. Inside these $2 \times 2$ blocks, only $3^4 = 81$ combinations of replacement values are possible and are called structures. Former subjective investigations have shown that these 81 structures can be represented by only 31. In this case, only a 5-bit wide index must be transmitted to reconstruct the 31 structures on the receiver side. The described process is a special form of vector quantization.

The source-coded sequence is obtained by multiplexing (MUX) the individually coded parameters: mask of moving objects (in blocks), motion vectors, address information of coded blocks, DCT coefficients, replacement values, and quadtree information of structure coded $2 \times 2$ blocks and their indices.

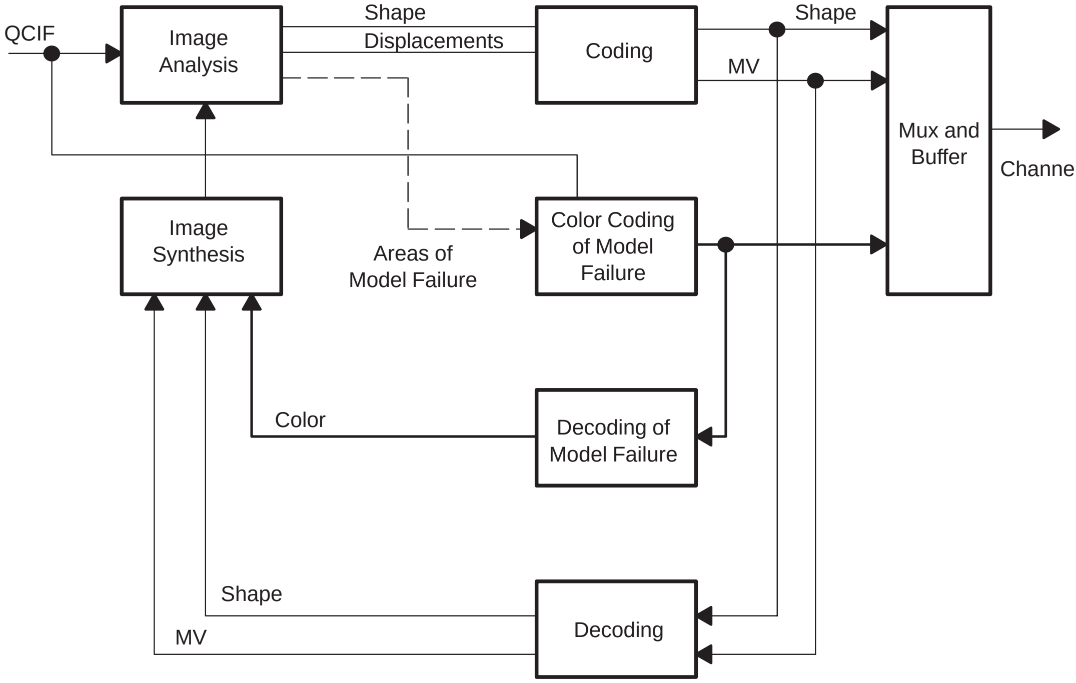
Every parameter group exhibits a different probability density. The introduction of an entropy coder (EC) for every group allows an additional bit reduction up to 50 percent. In typical applications, the entropy coding is realized by different Huffman tables or by means of an arithmetic coder, as in case of the presented codec.

The source decoder is part of the source coder. It consists of the DPCM loop for the interframe coded blocks, an inverse DCT, and the prediction memory. After entropy decoding of the received sequence, the motion compensation is calculated in the prediction memory by using the decoded motion vectors. The reconstructed image is displayed after the decoding of the intraframe-coded blocks (IDCT) and the interframe-coded blocks in $SC^{-1}$ at the coder and the decoder side.

### Advanced Source Codec Architectures

Advanced codec architectures are under investigation worldwide. The main goal is to change from block-oriented to object-oriented algorithms. The main disadvantages of block-oriented codecs are the visible errors like blocking and the mosquito effect [8]. Figure 3 depicts a proposal for an image sequence coding scheme, which is based on an object-oriented analysis-synthesis approach. With this approach, the original input image is decomposed into objects, each described by a set of shape, motion, and color (luminance and chrominance) parameters in the image analysis-synthesis stage. Therefore, different model types (i. e., 2-dimensional, 3-dimensional, rigid, and nonrigid objects) are possible. The areas in which no modeling is possible are denoted as model failure.

**Figure 3. Object-Oriented Analysis-Synthesis Codec for p × 8 kbps**



The shape information has pixel accuracy and is coded by polygon and spline approximation. This leads to a nonvisible shape error of between one and two pixels. The motion information has half-pixel accuracy. For coding the model failure, different methods are under investigation. The (decoded) parameter sets of any object are forced as the input information for the image synthesis stage. If the objects in case of the analysis-synthesis codec are interpreted as a set of blocks, then the analysis-synthesis codec equals the hybrid codec scheme.

## Coding of Still Pictures

The source coding on still images can be executed with the same codec architecture. A motion estimation is not necessary. The input image format is CIF (common intermediate format).

## Speech Coding

Speech source coding is accomplished by means of an LPC (linear predictive coding) codec with a data rate of 2 to 4 kbps. The corresponding algorithms can be found in [9].

## Realization of a Source Codec Based on a Multiprocessor System

In this section, several alternatives for the realization of the source codec presented in *A Hybrid Codec for Moving Pictures* are discussed.

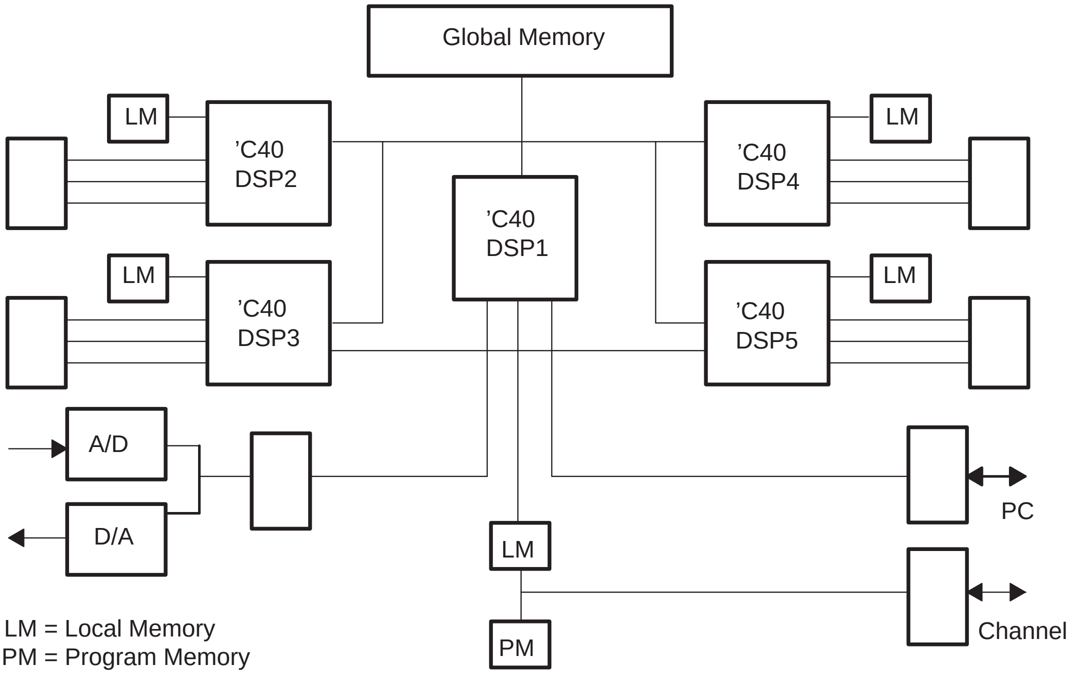
A system for implementation of complex algorithms cannot be conceived with standard ICs, such as ALU and multipliers or with programmable ICs, such as erasable and nonerasable PLDs (programmable logic devices), and LCAs (logic cell arrays), because it would not be compact, reasonable, and cost effective. Moreover, fast prototyping with standard components becomes time consuming as soon as unavoidable modifications become necessary. Even a demonstrator is not practical, because it does not lead to higher integration.

Flexible hardware should involve programmable signal processors. One solution could be the use of specialized single instruction multiple data (SIMD) architectures for high data rates. These processors are supplied with microcode programs for standard video algorithms. Today, available systems that are built with SIMD and a higher hierarchical level with multiple instruction multiple access (MIMD) processors do not have the necessary computational power. To implement the algorithms of the image codec, high-performance digital signal processors and their development tools are required. The DSPs should also be programmable in a high-level language (C) and should be provided with libraries supporting multisignal processor systems. This enables easy system extension with added processors for more computational power and guarantees picture coding in real time.

Figure 4 shows this type of system. It is composed of five ’C40s, which are linked together through their communication ports, and a fast global memory for storing the images [10]. The ports are connected together in the form of a “spoked wheel”, the hub of the wheel being the master processor. Every slave processor has three parallel ports, which can be used to communicate to other ’C40s or to dedicated picture-coding components to form a more complex parallel processor system. The master DSP uses one of its 6 communication ports to communicate with the PC. Another port is used for data transmission between the video codec and A/D and D/A converters. The other four communication ports are tied to the slave processors. The host PC is used during the debug phase as development platform. The JTAG interface ties all processors together and is controlled by emulator software running on the XDS510 board [11]. In a future version, the PC will be used as a control interface to the picture codec. Also, compressed video sequences could be stored on the PC disk for several postprocessing operations.

Before the algorithm is implemented on a multi-DSP system, it must be divided into tasks. A self-written operating system supports the distribution of the tasks on the multi-DSP system. The master DSP controls the process, which is determined by the picture frame rate. Control words are sent over the communication ports. The data is exchanged over the global memory or over the parallel ports. After the completion of the described picture coding system at the end of 1992, further investigations concerning the application of the system in a mobile environment will be conducted.

**Figure 4.  Multi-DSP System**



## References

[1] Murota, K., and Hirade, K. “GMSK Modulation for Digital Radio Telephony”, *IEEE Transactions in Communications*, vol. COM–29, pp. 1044–1050, 1981.

[2] Jakes, W.C. *Microwave Mobile Communications*, John Wiley, 1974.

[3] Ungerböck, G. “Channel Coding with Multilevel/Phase Signals”, *IEEE Transactions in Information Theory*, 1982, pp. 55–67.

[4] Mann Pelz, R., and Biere, D. “Video and Speech Transmission in Mobile Telecommunication Systems”, *Nachrichtentechnik Elektronik*, vol. 1, pp. 7–12, 1992.

[5] Hagenauer, J. “Rate Compatible Punctured Convolutional Codes (RCPC) and Their Applications”, *IEEE Transactions in Communications*, vol. COM–36, pp. 389–400, 1988.

[6] Stiller, C. “Motion Estimation for Coding of Moving Video at 8 kbit/s with Gibbs Modeled Vectorfield Smoothing”, *Proc. SPIE, Lausanne*, pp. 468–476, 1990.

[7] Amor, H. “Quellencodierung der Feinsturkturkomponente hochaufgelöster Bilder”, *Fernseh und Kinotechnik* 37, pp. 15– 20, 1983.

[8] Musmann, H.G, Hötter, H., and Ostermann, J. “Object-Oriented Analysis-Synthesis Coding of Moving Images”, *Image Communication*, vol. 1, 1989.

[9] Tremain, T. E. “Government Standard Linear Predictive Coding Algorithm: LPC–10”, *Speech Technology*, pp. 40–49, 1982.

[10] *MPS40 Hardware Reference*, SKALAR Computer GmbH, Göttingen.

[11] *TMS320C4x User’s Guide*, Texas Instruments, 1991.

# Optical Quality Assurance With Parallel Processors

**Ulrich Dumschat**
**Hema Elektronik**

## Introduction

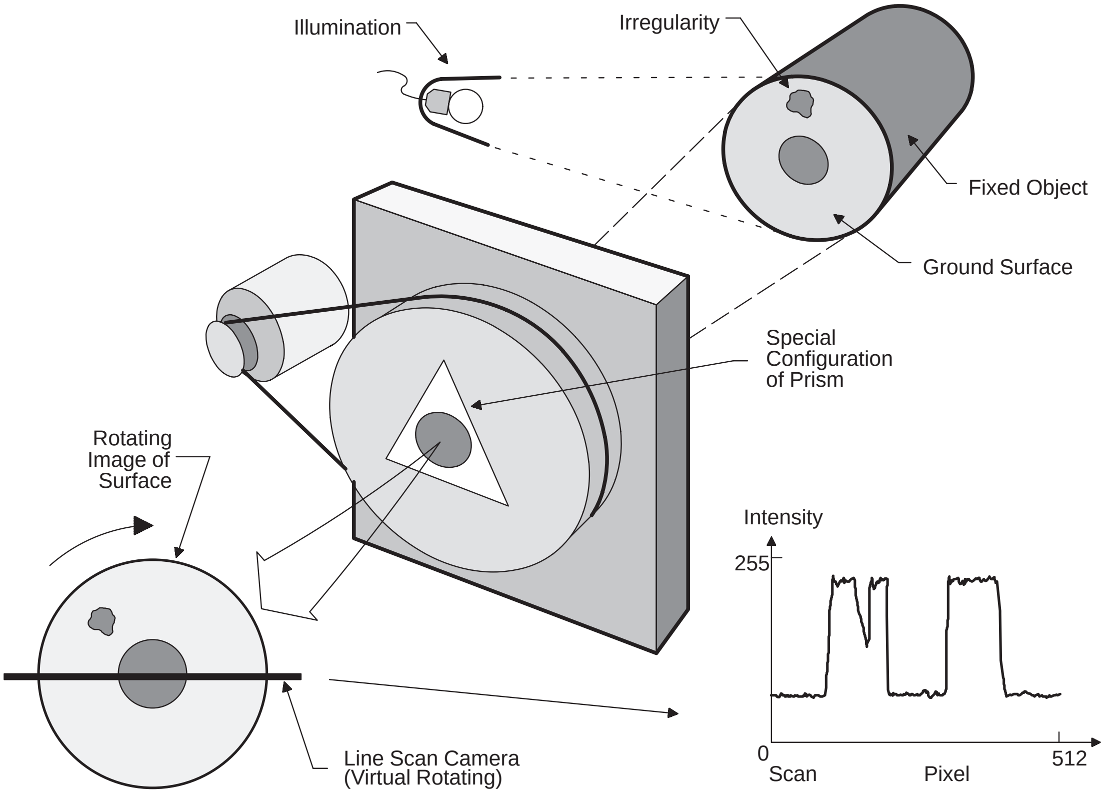
Within the field of industrial production, quality assurance is an important component, which must fulfill increasing requirements. The quality of production machines and the production speeds require that high-performance systems be able to control products in real time (e.g., 20 parts per second or several meters per second). This article describes a way to assure quality on the basis of digital imaging and signal processing with parallel signal processors for machine communication.

Due to its high computing rate and its capability to communicate via six parallel high-speed interfaces (20 Mbytes/second each), the first parallel digital signal processor, the Texas Instruments TMS320C40, is particularly suitable for image processing and forms the basis of the design. Transputers, on the other hand, are used mainly as flexible and high-performance machine controllers.

## Overview

The application this paper describes is a system for securing the quality of surfaces — for example, those on front surfaces of roller bearings. The front surface is captured by means of a line scan camera with a maximum of 400 scans (512 pixels) per surface. The pixel data per line are analyzed and classified by algorithms. A rotating prism virtually rotates the surface to be examined. The line scan camera and the object to be tested are mechanically fixed during data acquisition. During the test procedure, the object is illuminated homogeneously. The line scan camera captures differences in light (256 gray steps) resulting from the surface. This application is illustrated in Figure 1.

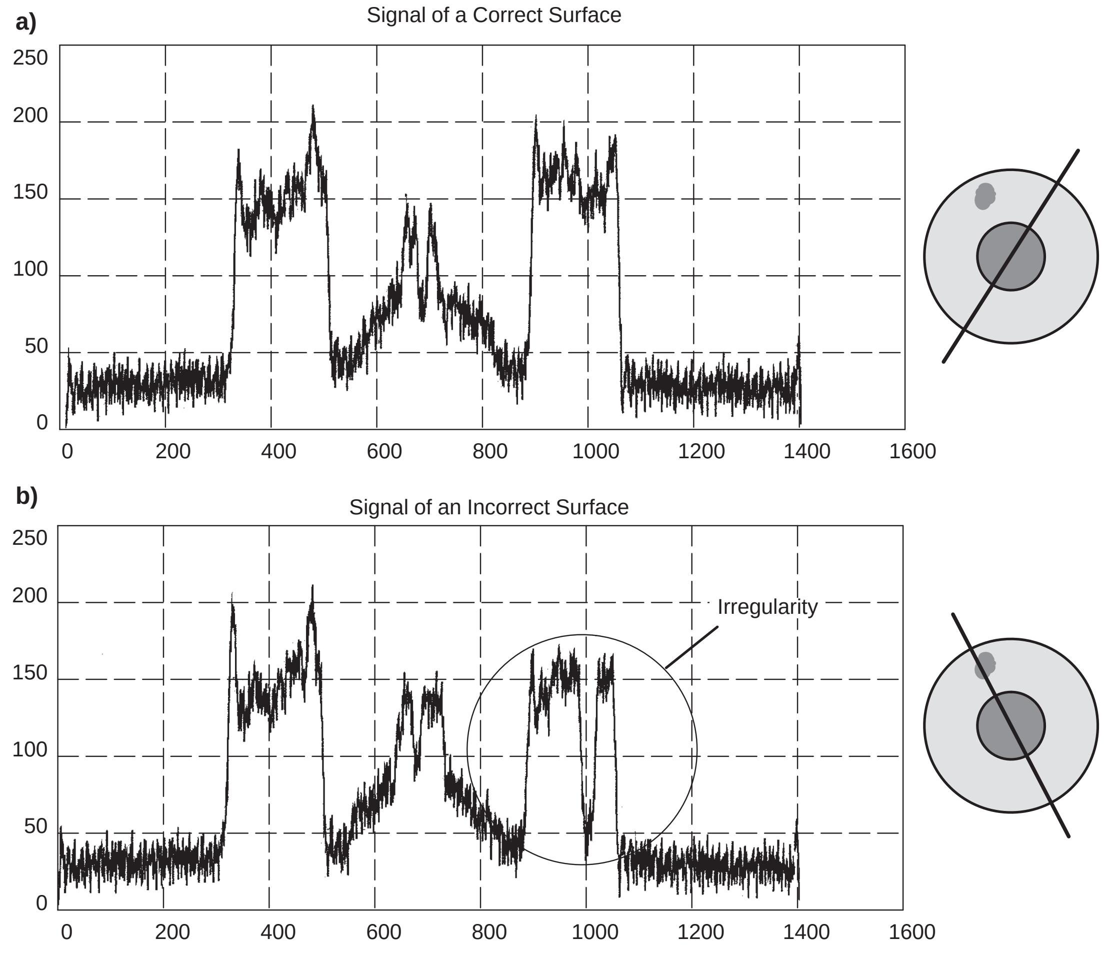
**Figure 1. Surface Quality Scan Application**


Illumination
Irregularity
Fixed Object
Ground Surface
Special Configuration of Prism
Rotating Image of Surface
Intensity
255
0
512
Scan
Pixel
Line Scan Camera (Virtual Rotating)

The signals of the line scan camera show characteristic shapes. Figure 2(a) shows a scan through a correct surface; Figure 2(b) shows a curve of an incorrect signal. This irregularity should be recognized and classified by the digital quality assurance system.

So far, this application has been built with analog hardware components (comparators, filters), revealing the common defects, such as aging of the components (parameter drift) and dependency on temperature. Nonrelevant zones of the signal (regardless of the object) cannot be extracted by the analog technology. Although the speed of common analog computer components is advantageous, digital processors are more flexible. Different mathematical algorithms can be implemented for classification. All results are reproducible. Nonrelevant zones can be extracted before the signal analysis. The computer performance will thus be concentrated on the essential test fields. The most important benefit is that you will be able to increase the system's performance later by improving the algorithms.

**Figure 2.  Measured Signal**



The time requirements of a digital classification system are calculated with the following formula:

f(pixel clock of the line scan camera) = No. of pixels/s × No. of scans

- pixel clock of the line scan camera : 10 MHz (maximum)
- 5–20 objects/s
- 512 pixels (8 bits) × 400 scans (20 objects) = 4,096 Mbytes/s (average data rate)
- 10 Mbytes/s (peak data rate)
- TMS320C40 for image and signal processing
- Transputers for machine control

The number of scans per object, multiplied by the number of pixels per scan within one second is a function of the machine cycle. Within one machine cycle, the data of an object are recorded, analyzed, and classified. The required number of objects is a maximum of 20 per second. The pixel clock of the line scan camera is specified with 10 MHz. The required number of scans (400 scans) multiplied by the resolution of the line scan camera (512 pixels) for 20 objects, results in an average data rate of 4,096 Mbytes/s. On average, about 4 megasamples (4 Mbytes) per second must be transferred from the camera to the computer unit. The maximum data rate is calculated from the pixel clock of the line scan camera and amounts to 10 megasamples (10 Mbytes) per second. With conventional processors, this data rate cannot be transferred in real time from the line scan camera to the computing unit.

## The Transputer T805 and the TMS320C40 DSP

The Texas Instruments TMS320C40 digital signal processor is not only able to manipulate big data quantities (275 MOPS and 50 MFLOPS), but also transfers up to 20 Mbytes/s per communication link. Inmos transputers control the recording, manipulation, and display of objects.

The ’C40 parallel DSP shows structures similar to those of a transputer. The performance, however, is 10 to 30 times better than that of a T805. The high-speed links are essential characteristics of parallel processors. The principle of the communicating sequential processes (CSP) model has been developed by HOARE and is the basis of parallel processing. Several software processes are running on one or more processors and communicate via the fast links for data exchange or via soft channels on one chip.

The parallel processors (the ’C40 and the transputer) are connected to each other via an Inmos-Link Adapter or dual-port RAM. The connection to the transputer world is most easily done via a transputer link adapter. This link adapter is implemented on the DSP1 (’C40 board) and converts two ’C40 links to the serial transputer link (one transputer link consists of link-in and link-out). Due to its serial data transmission, the transputer link represents a bottleneck. The data transfer between the DSP and the transputer is restricted to about 1 Mbyte/s by the serial link. But for parameters, configuration, and results, this data rate is adequate.

An increased data transfer rate is obtained by means of dual-port RAM, whereby the DSP 1 card is plugged into a 4-fold transputer card (TR3-N) as a piggyback board. The communication rate via DPRAM amounts to about 15 Mbytes/s. Figure 3 shows block diagrams of the T805 and the TMS320C40, and Figure 4 shows the interconnection.

**Figure 3. Transputer T805 and the DSP TMS320C40**

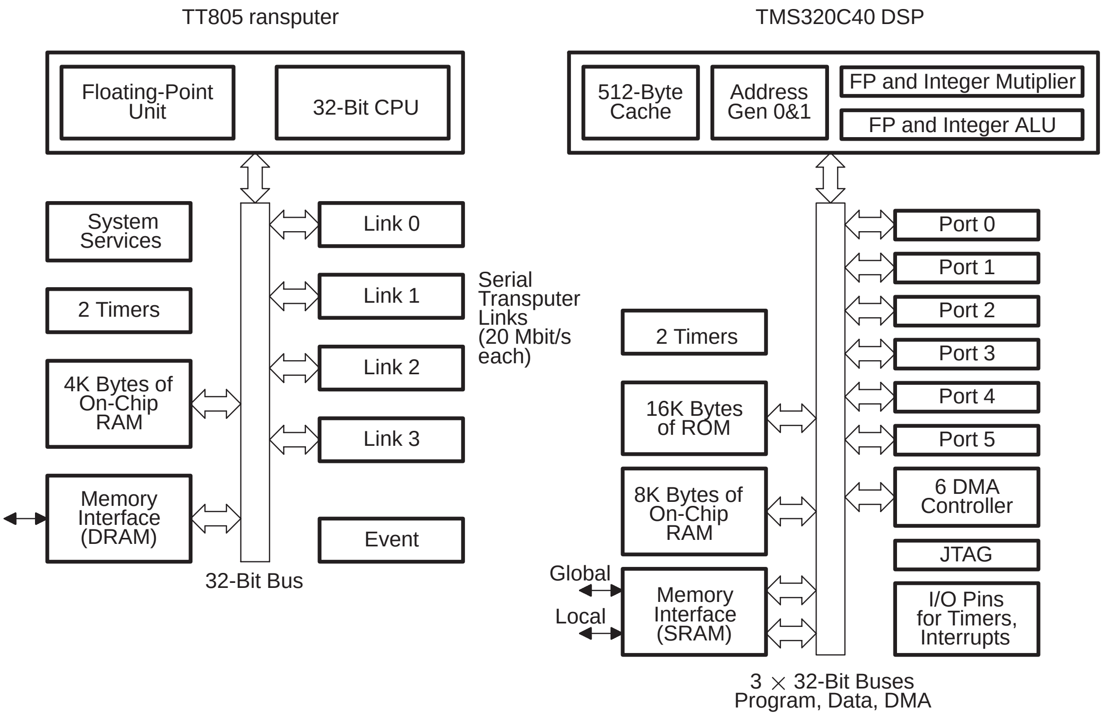
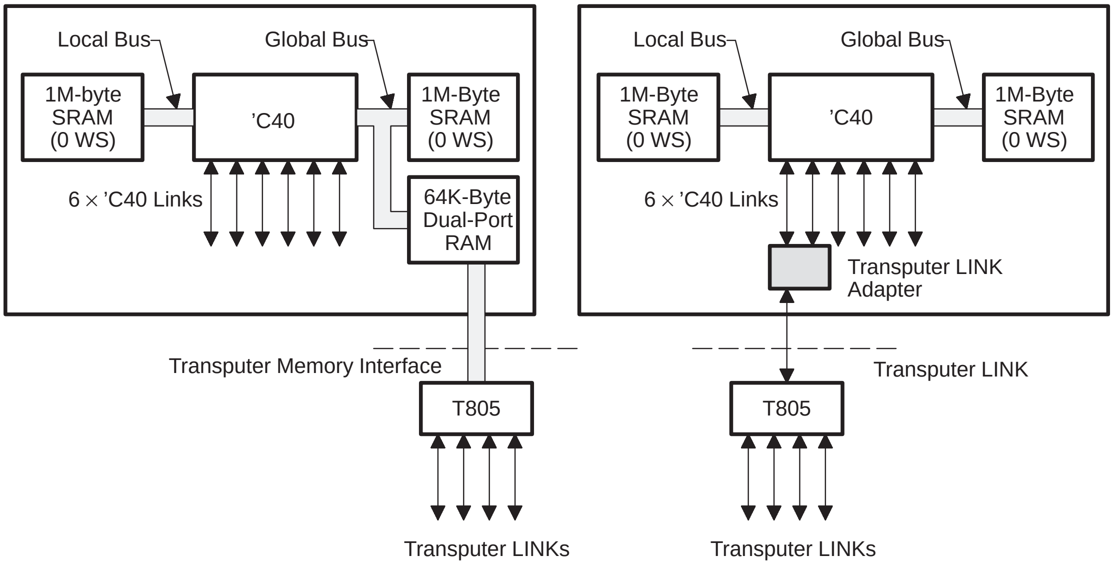


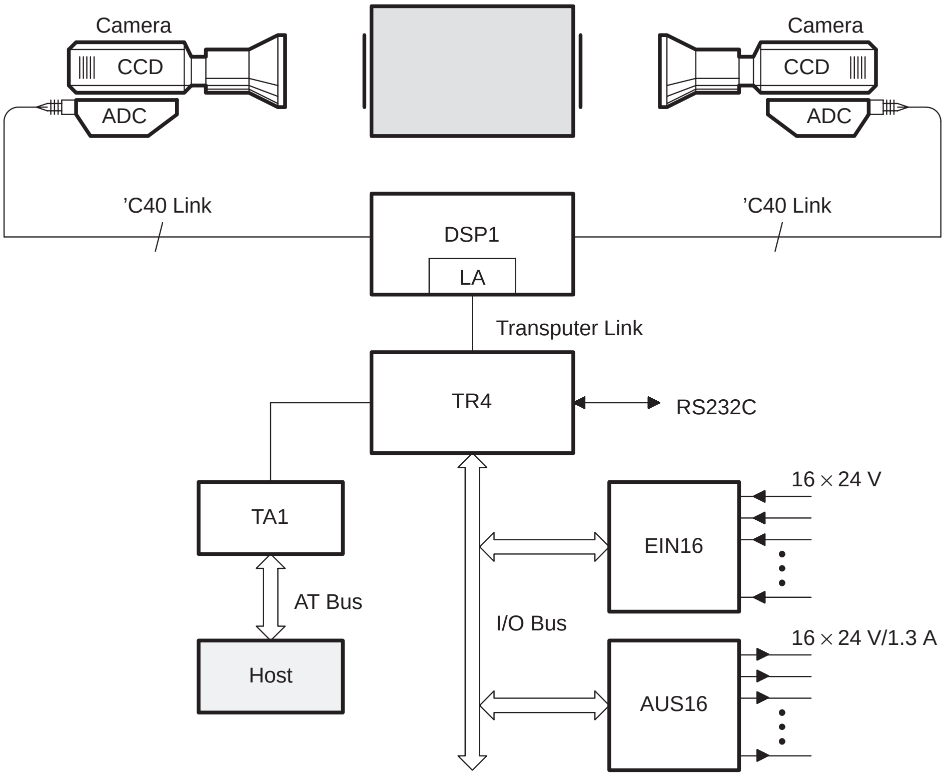
**Figure 4. Interfacing the TMS320C40 and the Transputer**



## Hardware

The complete hardware concept of the quality assurance system is composed of the individual components for signal/image processing and control shown in Figure 5. The signal and image processing is designed for two test surfaces; that is, two cameras simultaneously examine two front surfaces, which are further analyzed and classified by a DSP. The cameras are each equipped with an 8-bit A/D converter, enabling the pixel data to be transferred directly to the ’C40 via the ’C40 link. The classification result is transferred in the form of command signals to the command section via the transputer link. The command section consists of a transputer card (TR4), on one side controlling digital inputs and outputs and on the other side managing the connection to the host computer. Via the digital inputs, signals from the machine are received (the object is ready for scanning); via the outputs, the machine is influenced (locked during scanning). The host computer (PC) represents the interface between user and application.

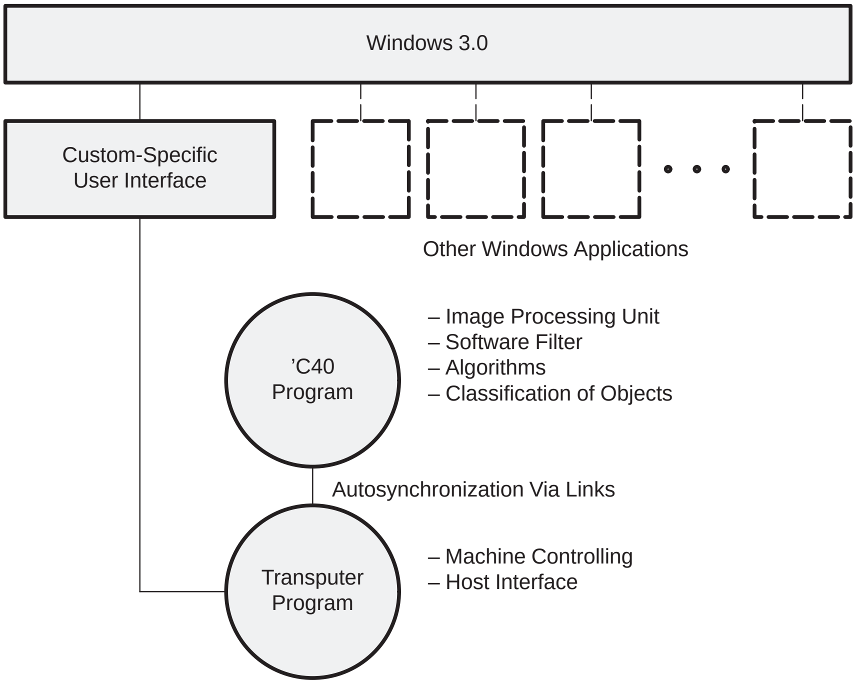
**Figure 5. Overall Concept on Hardware**



## Software

The software is a Windows 3.0 application and is shown in Figure 6. This interface was chosen because it is user friendly. The customer-specific Windows 3.0 application initiates the transputer program. The transputer software carries out the machine command/control and produces the connection to the host. The software on the ’C40 is loaded from the transputer to the ’C40 and effects the classification by means of digital filters, different algorithms, and different types of image/signal processing. Both software modules (transputer and ’C40) synchronize and communicate via the ’C40/transputer link.

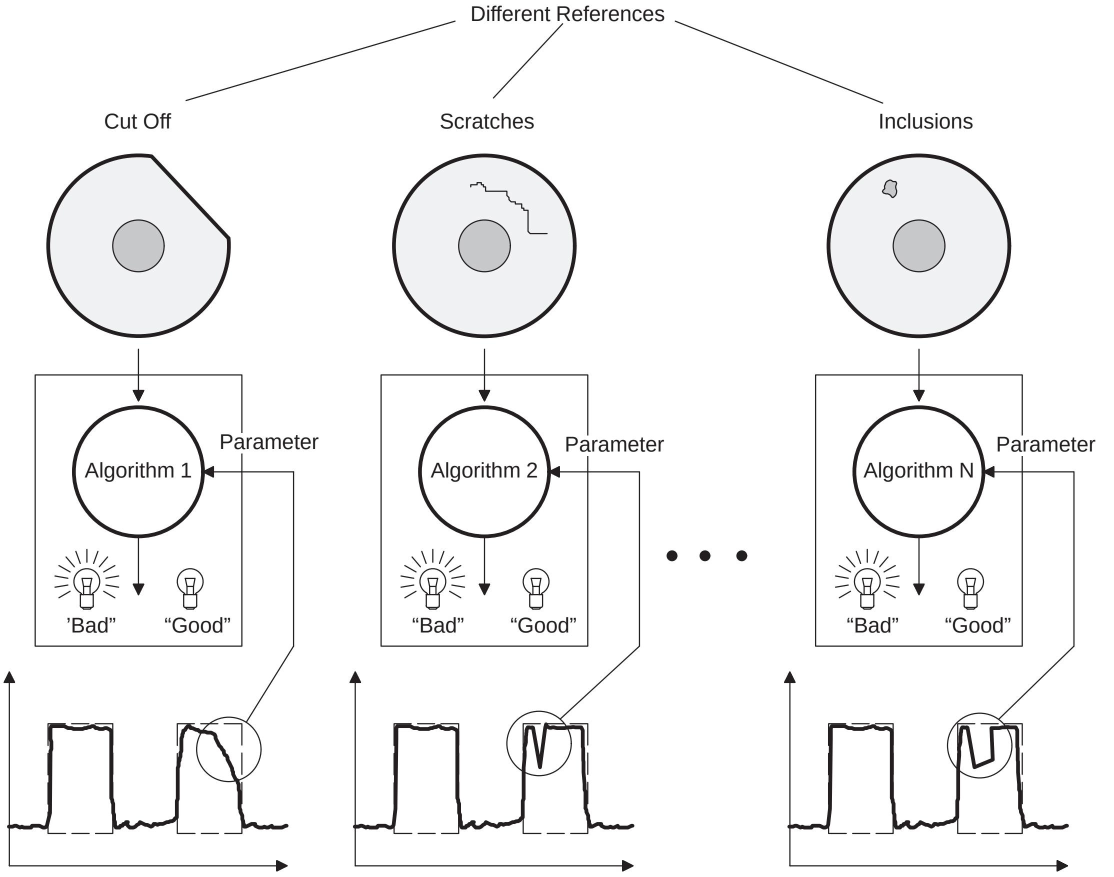
**Figure 6.  Overall Concept on Software**



## Algorithms

Before the classification of different objects can be carried out, the parameters of the implemented classification algorithms must be fixed. User parameters condition the algorithms to search for specific defects and return a “good” or “bad” result, as shown in Figure 7.

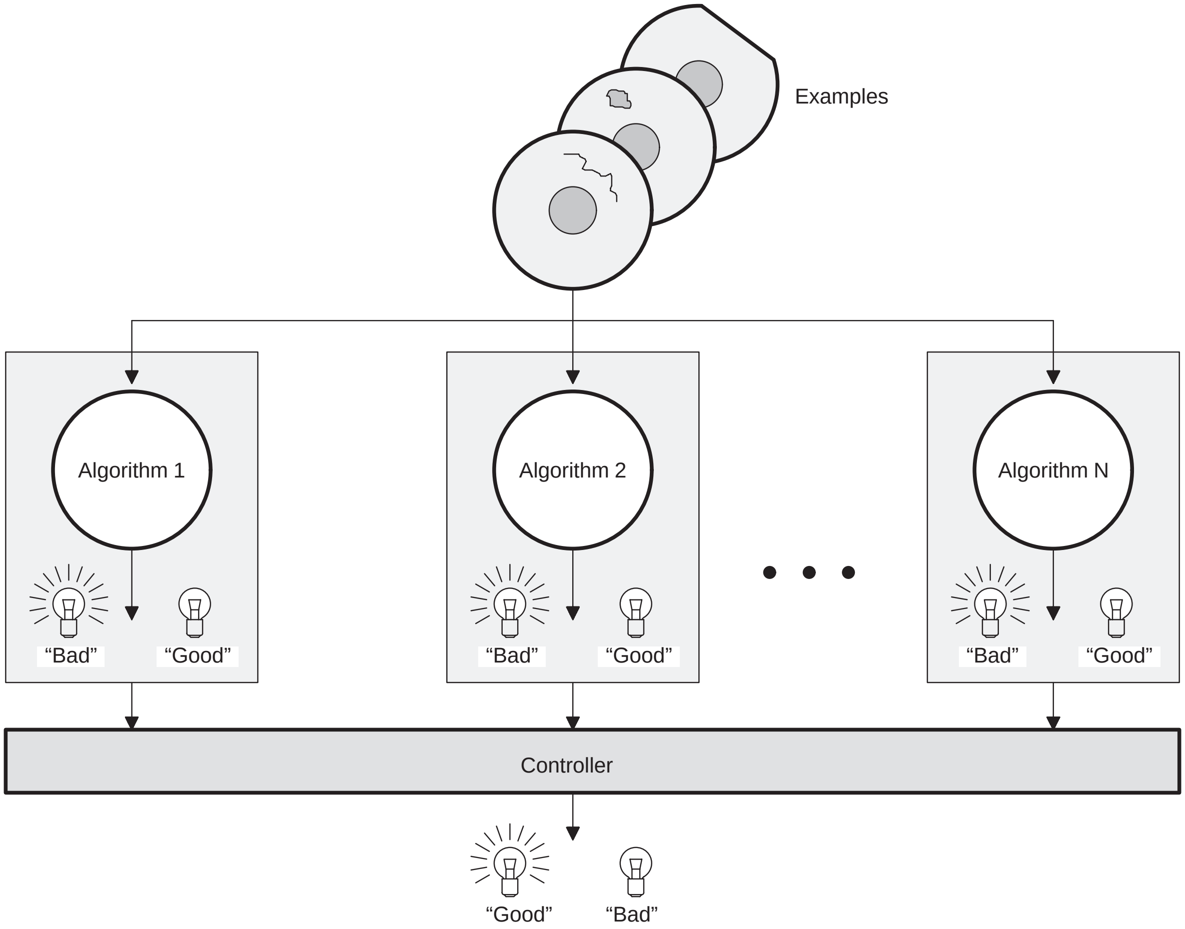
**Figure 7. Setup of Algorithms**



## Object Recognition

In the classification phase, different objects are now transmitted in real time to the application. The results of the different algorithm modules are collected and evaluated by an overall controller. On the basis of these partial statements, the controller produces a general statement on the quality of the tested part. See Figure 8.

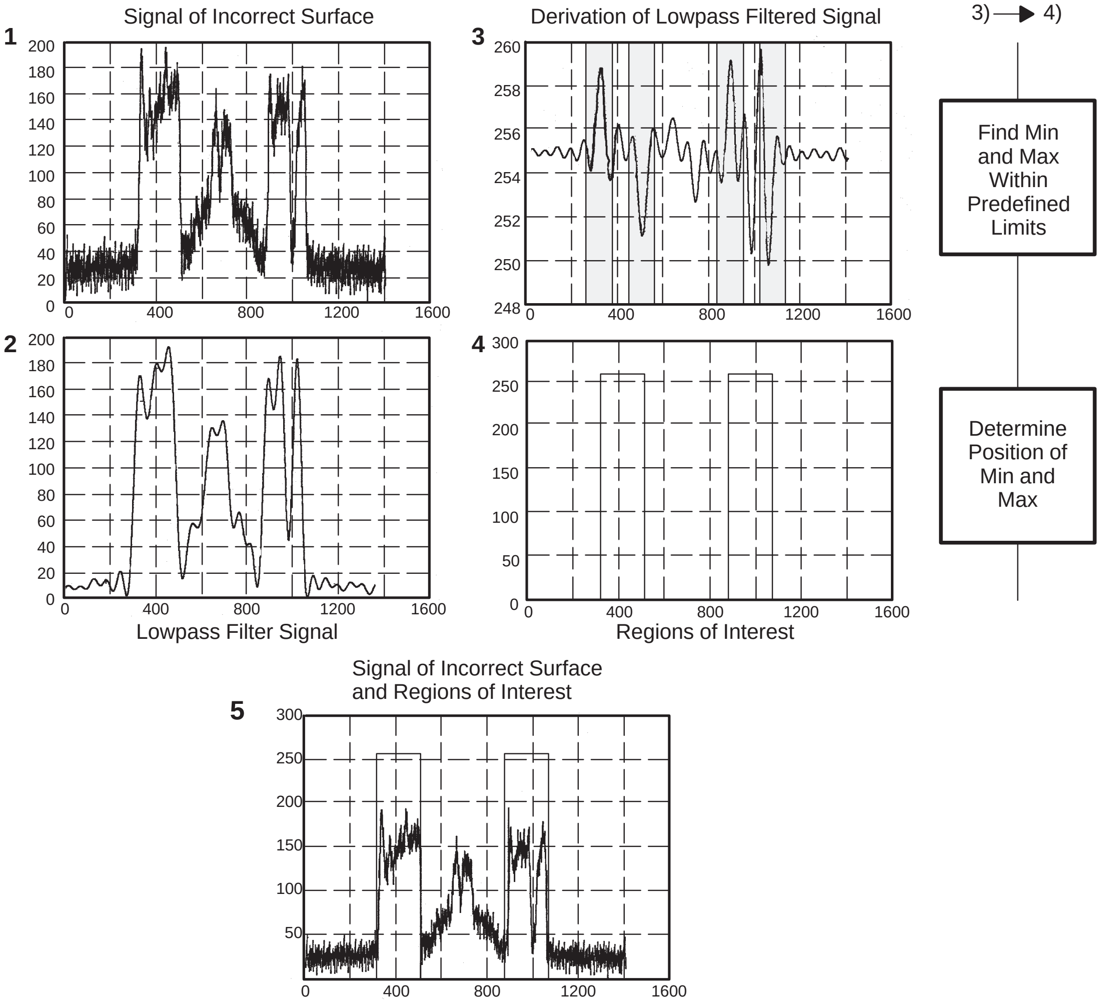
**Figure 8.  Recognition of Objects**



The signal curves in Figure 9 show an example of separating relevant signal sections by means of digital signal processing. The two plateaus of an incorrect surface signal should be separated. Ideally, the signal is low-pass filtered to remove noise. The low-pass filtered signal will be differentiated to extract the edges. Within predefined limits, minimum and maximum are now determined; their positions will give information on the edge points of the plateaus.

Different signal processing algorithms (e.g., FFTs) can now be applied to these separated regions of interest.

## Figure 9.  Signal Filtering, Derivation and Max/Min Calculation



## Conclusion

This example shows an application with parallel processors in industrial environments, where communication processors with DSP characteristics are required for signal and image processing. The main advantage of parallel processors is that the performance is made scalable by adding more processors. The interprocessor communication is guaranteed by high-speed interfaces, which transfer data independently of the CPU.