TEXAS INSTRUMENTS

Digital Signal Processing Applications with the TMS320 Family

Theory, Algorithms, and Implementations

Volume 3

1990 Digital Signal Processor Products

Digital Signal Processing Applications with the TMS320 Family

Volume 3

Edited by
Panos Papamichalis, Ph.D.
Digital Signal Processing
Semiconductor Group
Texas Instruments

TEXAS INSTRUMENTS

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products 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. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that 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.

TRADEMARKS

ADI and *AutoCAD* are trademarks of Autodesk, Inc.
Apollo and *Domain* are trademarks of Apollo Computer, Inc.
ATVista is a trademark of Truevision, Inc.
CodeView, MS-Windows, MS, and *MS-DOS* are trademarks of Microsoft Corp.
DEC, Digital DX, VAX, VMS, and *Ultrix* are trademarks of Digital Equipment Corp.
DGIS is a trademark of Graphic Software Systems, Inc.
EPIC, *XDS, TIGA,* and *TIGA-340* are trademarks of Texas Instruments, Inc.
GEM is a trademark of Digital Research, Inc.
*GSS*CGI* is a trademark of Graphic Software Systems, Inc.
HPGL is a registered trademark of Hewlett-Packard Co.
Macintosh and *MPW* are trademarks of Apple Computer Corp.
NEC is a trademark of NEC Corp.
PC-DOS, PGA, and *Micro Channel* are trademarks of IBM Corp.
PEPPER is a registered trademark of Number Nine Computer Corp.
PM is a trademark of Microsoft Corp.
PostScript is a trademark of Adobe Systems, Inc.
RTF is a trademark of Microsoft Corp.
Sony is a trademark of Sony Corp.
Sun 3, *Sun Workstation*, *SunView, SunWindows,* and *SPARC* are trademarks of Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T Bell Laboratories.

CONTENTS

PART V. COMPUTERS

PART VI. TOOLS

Foreword

Much has happened in the TMS320 Family since Volume 1 of *Digital Signal Processing Applications with the TMS320 Family* was published, and Volumes 2 and 3 are a timely update to the family history.

The DSP microcomputers keep changing the perspective of the systems designers by offering more computational power and better interfacing capabilities. The steps of change are coming more quickly, and the potential impact is greater and greater. Because things change so rapidly in this area, there is a pressing need for ways to quickly learn how to utilize the new technology. These new volumes respond to that need.

As with Volume 1, the purpose of these books is to teach us about the issues and techniques that are important in implementing digital signal processing systems using microprocessors in the TMS320 Family. Volume 2 highlights the TMS320C25; and Volume 3, the TMS320C30 chip. A large part of the books is devoted to such matters as characteristics of the TMS320C25 and TMS320C30 chips, useful program code for implementing special DSP functions, and details on interfacing the new chips to external devices. The remainder of the books illustrates how these chips can be used in communications, control, and computer graphics applications.

What these two volumes make clear is how remarkably fast the field of DSP microcomputing is evolving. IC technologists and designers are simply packing more and more of the right kind of computing power into affordable microprocessor chips. The high-speed floating-point computing power and huge address spaces of chips like the TMS320C30 open the door to a whole new class of applications that were difficult or impractical with earlier generations of fixed-point DSP chips. The signal processing theorists and system designers are clearly being challenged to match the creativity of the chip designers.

The present books differ from Volume 1 in the inclusion of a small section on tools. This is a hopeful sign, because it is progress in this area that is likely to have the greatest impact on speeding the widespread application of DSP microprocessors. While useful design tools are beginning to emerge, much more can be done to help system designers manage the complexity of sophisticated DSP systems, which often involve a unique combination of theory, numerical and symbolic processing algorithms, real-time programming, and multiprocessing. No doubt future volumes of *Digital Signal Processing Applications with the TMS320 Family* will have more to say about this important topic. Until then, Volumes 2 and 3 have much useful information to help system designers keep up with the TMS320 Family.

Ronald W. Schafer
Atlanta, Georgia
November 14, 1989

Preface

The newer, floating-point DSP devices, such as the TMS320C30, have brought an added dimension to DSP applications. With the TMS20C30, programming is much easier because the designer does not have to worry about dynamic range and accuracy issues. An algorithm implemented in floating-point in a high-level language can be easily ported to such a device. The new architecture contains other features, besides the floating point capability, that simplify programming. Some of these features (such as the software stack, the large register file, etc.) were added to facilitate the development of high-level language compilers. Currently, C and Ada compilers have been introduced. In addition, Spectron Microsystems introduced an operating system for DSPs (called SPOX) that further facilitates the development of algorithms on the DSP devices.

Volume 3 of *Digital Signal Processing Applications with the TMS320 Family* contains application reports primarily on the third generation of the TMS320 Family (floating-point devices). This book is a continuation of Volumes 1 and 2 in the sense that it addresses the same needs of the designer. The designer still has the task of selecting the DSP device with the appropriate cost, performance, and support, developing the DSP algorithm that will solve the problem, and implementing the algorithm on the processor. This volume tries to help by bringing the designer up to date on the applications of newer processors or in different applications of earlier processors.

The objectives remain the same as in earlier volumes. First, the application reports supply examples of device use and serve as tutorials in programming the devices. Of course, the same purpose is served on a more elementary basis by the software and hardware applications sections of the corresponding user's guides. Second, since the source code of each application is provided with the report, the designer can take it intact (or extract a portion of it) and place it in the application.

It is assumed that the reader has exposure to the TMS320 devices or, at least, has the necessary manuals (such as the appropriate TMS320 user's guides) that will help the reader understand the explanations in the reports. The reports themselves include as references the necessary background material. Additionally, the Introduction gives a brief overview of the available devices at the time of the writing and points to the source of more information.

The reports are grouped by application area. The term *report* is used here in a broad sense, since some articles from technical publications are also included. The authors of the reports are either the digital signal processing engineering staff of the Texas Instruments Semiconductor Group (including both field and factory personnel, and summer students) or third parties.

The source code associated with the reports is also available in electronic form, and the reader can download it from the TI DSP Electronic Bulletin Board (telephone (713) 274–2323). If more information is needed, the DSP Hotline can be called at (713) 274–2320.

The editor thanks all the authors and the reviewers for their contribution to this volume of application reports.

Panos E. Papamichalis, Ph.D.
Senior Member of Technical Staff

Part I. Introduction

TMS320 Family and Book Overview

Digital signal processors have found applications in areas where they were not even considered a few years ago. The two major reasons for such proliferation are an increase in processor performance and a reduction in cost. Volume 3 of *Digital Signal Processing Applications with the TMS320 Family* presents a set of application reports primarily on the TMS320C30, the third-generation TMS320 device.

Organization of the Book

The material in this book is grouped by subject area:

- Introduction
- Digital Signal Processing Routines
- DSP Interface Techniques
- Telecommunications
- Computers
- Tools
- Bibliography

The **Introduction** contains this overview and two review articles. The first article gives a general description of the TMS320 family and is reprinted from a special issue of the *IEEE Proceedings*, while the second article discusses the TMS320C30 device and is reprinted from the *IEEE Micro Magazine*. The overview points out how the TMS320 family has grown since the two articles were published and also introduces newer devices.

The five articles in the **Digital Signal Processing Routines** section present useful algorithms, such as the FFT, the Discrete Cosine Transform, etc., that are implemented on the TMS320C30. Two of the reports also consider implementations on the TMS320C25.

The section on **DSP Interface Techniques** contains an article on interfacing the TMS320C30 with external hardware, such as memories and A/D and D/A converters, and an article on a hardware implementation of a floating-point converter between the IEEE and the TMS320C30 formats.

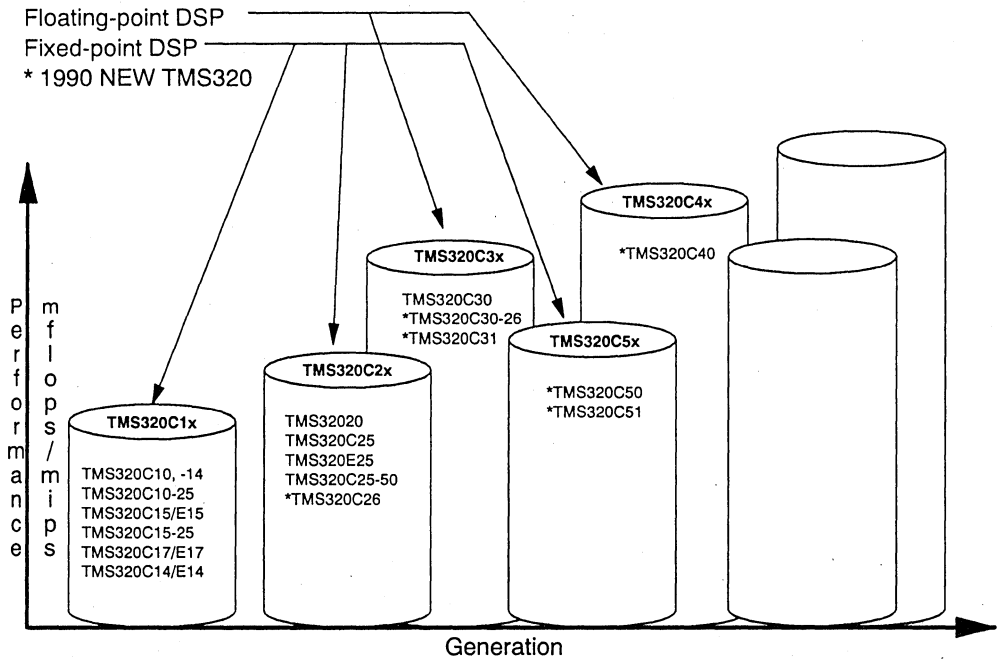
The following three sections contain one article each. In the **Telecommunications** section, an implementation of the government-standard CELP speech-coding algorithm is presented. The **Computers** section contains an article on 3-D graphics systems, which shows examples of using the TMS320C30 device for graphics problems. In the **Tools** section, the article gives a functional description of the TMS320C30 Application Board that is part of the hardware emulator for that device.

The **Bibliography** section contains a list of articles mentioning DSP implementations using TMS320 devices. The different titles are listed chronologically and are grouped by subject. The list is not exhaustive, but it gives pointers for pursuing practical implementations in representative application areas.

The TMS320 Family of Processors

The TMS320 Family of digital signal processors started with the TMS32010 in 1982, but it has been expanded to encompass five generations (at the time of this writing) with devices in each generation. Figure 1 shows this progression through the generations. The TMS320 devices can be grouped in two broad categories: fixed-point and floating-point devices. As implied by Figure 1, the first, second, and fifth generations are the fixed-point devices, while the third and the fourth generations (the latest one under development) support floating-point arithmetic.

Figure 1. TMS320 Family Roadmap



The following article, "The TMS320 Family of Digital Signal Processors," by Lin, et. al., is reprinted from the Proceedings of the IEEE and gives an overview of the TMS320 family. Since additional devices have been developed from the time the article was written, this section highlights these newer devices. Table 1 shows a comprehensive list of the currently available TMS320 devices and their salient characteristics.

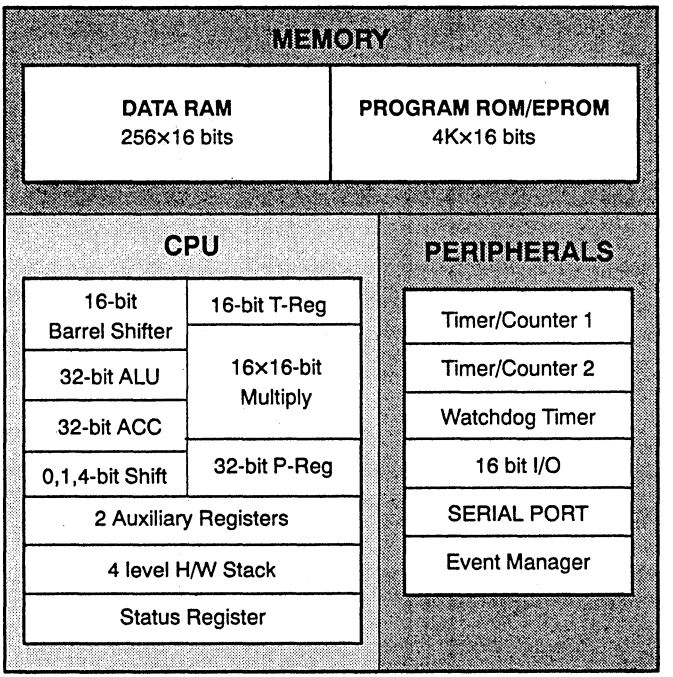
Table 1. TMS320 Family Overview

Gen	Device	Data Type	Cycle Time (ns)	Memory				I/O			On-Chip Timers	Package
				RAM	On-Chip ROM	EPROM	Off-Chip	Parallel	Serial	DMA		
1st	TMS320C10 ¶	Integer	200	144	1.5K		4K	8x16				DIP/PLCC
	TMS320C10-25	Integer	160	144	1.5K		4K	8x16				DIP/PLCC
	TMS320C10-14	Integer	280	144	1.5K		4K	8x16				DIP/PLCC
	TMS320E14	Integer	160	256		4K	4K	7x16	1		4	CERQUAD
	TMS320C15 ¶	Integer	200	256	4K		4K	8x16				DIP/PLCC
	TMS320C15-25 ¶	Integer	160	256	4K		4K	8x16				DIP/PLCC
	TMS320E15 ¶	Integer	200	256		4K	4K	8x16				DIP/CERQUAD
	TMS320E15-25	Integer	160	256		4K	4K	8x16				DIP/CERQUAD
	TMS320C17	Integer	200	256	4K		4K	6x16	2		1	DIP/PLCC
	TMS320E17	Integer	200	256		4K	4K	6x16	2		1	DIP/CERQUAD
2nd	TMS32020 ¶	Integer	200	544			128K	16x16	1	†	1	PGA
	TMS320C25 ¶	Integer	100	544	4K		128K	16x16	1	†	1	PGA/PLCC
	TMS320C25-50 ¶	Integer	80	544	4K		128K	16x16	1	†	1	PGA/PLCC
	TMS320E25 ¶	Integer	100	544		4K	128K	16x16	1	†	1	CERQUAD
	TMS320C26	Integer	100	1.5K	256		128K		1	†	1	PLCC
3rd	TMS320C30 ¶	Float Pt	60	2K	4K		16M	16Mx32	2	‡	2	PGA
5th	TMS320C50 ¶	Integer	50	8.5K	2K		128K	16x16	1	†	1	CLCC

† External DMA
‡ External/Internal DMA
¶ For information on military versions of these devices, contact your local TI sales office.

The additions to the first generation are the TMS320C14 and the TMS320E14; the latter is identical with the former, except that the latter's on-chip program memory is EPROM. The TMS320C14/E14 devices have features that make them suitable for control applications. Figure 2 shows the components of these devices. The memory and the CPU are identical to TMS320C15/E15, while the peripherals reflect the orientation of the devices toward control.

Figure 2. TMS320C14/E14 Key Features



Some of the key features of the TMS320C14/E14 are:

- 160-ns instruction cycle time
- Object-code-compatible with the TMS320C15
- Four 16-bit timers
  - Two general-purpose timers
  - One watchdog timer
  - One baud-rate generator
- 16 individual bit-selectable I/O pins
- Serial port/USART with codec-compatible mode
- Event manager with 6-channel PWM D/A
- CMOS technology, 68-pin CERQUAD

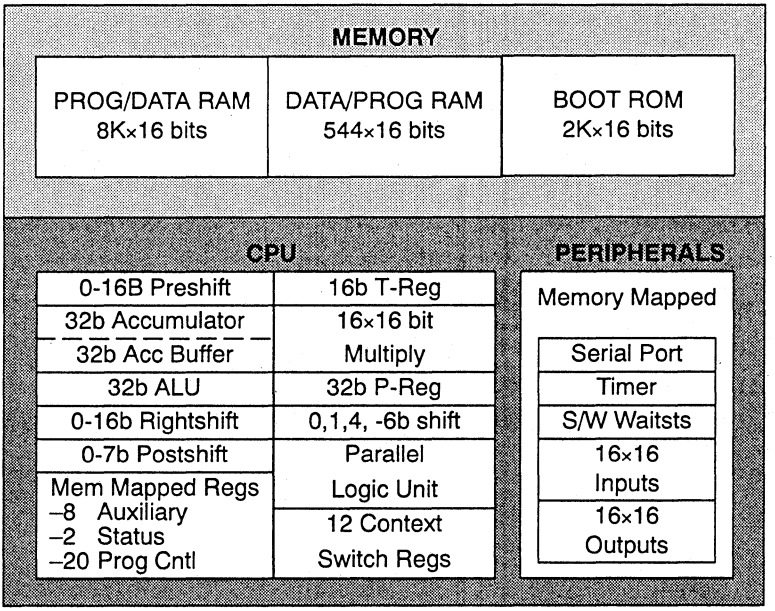
The additions to the second generation are the TMS320E25, the TMS320C25-50, and the TMS320C26. The TMS320E25 is identical to the TMS320C25, except that the 4K-word on-chip program memory is EPROM. Since increased speed is very important for the real-time implemen-

tation of certain applications, the TMS320C25-50 was designed as a faster version of the TMS320C25 and has a clock frequency of 50 MHz instead of 40 MHz.

The TMS320C26 is a modification of the TMS320C25 in which the program ROM has been exchanged for RAM. The memory space of the TMS320C26 has 1.5K words of on-chip RAM and 256 words of on-chip ROM, making it ideal for applications requiring larger RAM but minimal external memory.

A new generation of higher-performance fixed-point processors has been introduced in the TMS320 Family: the TMS320C5x devices. This generation shares many features with the first and the second generations, but it also encompasses significant new features. Figure 3 shows the basic components of the first device in that generation, the TMS320C50.

Figure 3. TMS320C50 Key Features



Some of the important features of the TMS320C50 are listed below:

- Source code is upward compatible with the TMS320C1x/C2x devices
- 50/35-ns instruction cycle time
- 8K words of on-chip program/data RAM
- 2K words boot ROM
- 544 words of data/program RAM
- 128K words addressable total memory
- Enhanced general-purpose and DSP-specific instructions
- Static CMOS, 84-pin CERQUAD
- JTAG serial scan path

The software and hardware development tools for the TMS320 family make the development of applications easy. Such tools include assemblers, linkers, simulators, and C compilers for the software. They include evaluation modules, software development boards, and extended development systems for hardware. These tools are mentioned in the following paper by Lin, et. al. The interested reader can find much more information in the additional literature that is published by Texas Instruments and mentioned in the next section. In particular, the *TMS320 Family Development Support Reference Guide* is an excellent source.

One important addition to the list of tools is the SPOX operating system, developed by Spectron Microsystems. SPOX permits you to write an application in a high-level language (C) and run it on actual DSP hardware. The operating system of SPOX hides the details of the interface from you and lets you concentrate on your algorithm while running it at supercomputer speeds on the TMS320C30.

References

Texas Instruments publishes an extensive bibliography to help designers use the TMS320 devices effectively. Besides the user's guides for corresponding generations, there are manuals for the software and the hardware tools. The *TMS320 Family Development Support Reference Guide* is particularly useful because it provides information, not only on development tools offered by TI, but also on those produced by third parties. Here is a partial list of the literature available (the literature number is in parentheses)

- *TMS320 Family Development Support Reference Guide* (SPRU011A)
- *TMS320C1x User's Guide* (SPRU013A)
- *TMS320C2x User's Guide* (SPRU014)
- *TMS320C3x User's Guide* (SPRU031)
- *TMS320C1x/TMS320C2x Assembly Language Tools User's Guide* (SPRU018)
- *TMS320C30 Assembly Language Tools User's Guide* (SPRU035)
- *TMS320C25 C Compiler Reference Guide* (SPRU024)
- *TMS320C30 C Compiler Reference Guide* (SPRU034)
- *Digital Signal Processing Applications with the TMS320 Family, Volume 1* (SPRA012)
- *Digital Signal Processing Applications with the TMS320 Family, Volume 2* (SPRA016)

You can request this literature by calling the Customer Response Center at 1–800–232–3200, or the DSP Hotline at 1–713–274–2320.

Contents of Other Volumes of the Application Book

Volume 1

Part I. Digital Signal Processing and the TMS320 Family

- Introduction
- The TMS320 Family

Part II. Fundamental Digital Signal Processing Operations

- Digital Signal Processing Routines

- Implementation of FIR/IIR Filters with the TMS32010/TMS32020
- Implementation of Fast Fourier Transform Algorithms with the TMS32020
- Companding Routines for the TMS32010/TMS32020
- Floating-Point Arithmetic with the TMS32010
- Floating-Point Arithmetic with the TMS32020
- Precision Digital Sine-Wave Generation with the TMS32010
- Matrix Multiplication with the TMS32010 and TMS32020

• DSP Interface Techniques
- Interfacing to Asynchronous Inputs with the TMS32010
- Interfacing External Memory to the TMS32010
- Hardware Interfacing to the TMS32020
- TMS32020 and MC68000 Interface

Part III. Digital Signal Processing Applications

• Telecommunications
- Telecommunications Interfacing to the TMS32010
- Digital Voice Echo Canceller with a TMS32020
- Implementation of the Data Encryption Standard Using the TMS32010
- 32K-bit/s ADPCM with the TMS32010
- A Real-Time Speech Subband Coder Using the TMS32010
- Add DTMF Generation and Decoding to DSP-μP Designs

• Computers and Peripherals

• Speech Coding/Recognition
- A Single-Processor LPC Vocoder
- The Design of an Adaptive Predictive Coder Using a Single-Chip
- Digital Signal Processor
- Firmware-Programmable C Aids Speech Recognition

• Image/Graphics
- A Graphics Implementation Using the TMS32020 and TMS34061

• Digital Control
- Control System Compensation and Implementation with the TMS32010

Volume 2

Part I. Introduction

• Book Overview

• The TMS320 Family of DSP

• The Texas Instruments TMS320C25 Digital Signal Microcomputer

Part II. Digital Signal Interface Techniques

• Hardware Interfacing to the TMS320C2x

• Interfacing the TMS320 Family to the TLC32040 Family

• ICC Requirements of the TMS320C25

• An Implementation of a Software UART Using the TMS320C25

• TMS320C17/E17 and TMS370 Serial Interface

Part III. Data Communications

- Theory and Implmentation of a Split-Band Modem Using the TMS320C17
- Implementation of an FSK Modem Using the TMS320C17
- An All-Digital Automatic Gain Control

Part IV. Telecommunications

- General Purpose Tone Decoding and DTMF Detection

Part V. Control

- Digital Control

Part VI. Tools

- TMS320 Algorithm Debugging Techniques

The TMS320 Family of Digital Signal Processors

Kun-Shan Lin
Gene A. Frantz
Ray Simar, Jr.

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

Reprinted from
PROCEEDINGS OF THE IEEE
Vol. 75, No. 9, September 1987

The TMS320 Family of Digital Signal Processors

KUN-SHAN LIN, MEMBER, IEEE, GENE A. FRANTZ, SENIOR MEMBER, IEEE, AND RAY SIMAR, JR.

This paper begins with a discussion of the characteristics of digital signal processing, which are the driving force behind the design of digital signal processors. The remainder of the paper describes the three generations of the TMS320 family of digital signal processors available from Texas Instruments. The evolution in architectural design of these processors and key features of each generation of processors are discussed. More detailed information is provided for the TMS320C25 and TMS320C30, the newest members in the family. The benefits and cost-performance tradeoffs of these processors become obvious when applied to digital signal processing applications, such as telecommunications, data communications, graphics/image processing, etc.

Digital Signal Processing Characteristics

Digital signal processing (DSP) encompasses a broad spectrum of applications. Some application examples include digital filtering, speech vocoding, image processing, fast Fourier transforms, and digital audio [1]–[10]. These applications and those considered digital signal processing have several characteristics in common:

- mathematically intensive algorithms,
- real-time operation,
- sampled data implementation,
- system flexibility.

To illustrate these characteristics in this section, we will use the digital filter as an example. Specifically, we will use the Finite Impulse Response (FIR) filter which in the time domain takes the general form of

$$y(n) = \sum_{i=1}^{N} a(i) * x(n - i) \qquad (1)$$

where $y(n)$ is the output sample at time n, $a(i)$ is the ith coefficient or weighting factor, and $x(n - i)$ is the $(n - i)$th input sample.

With this example in mind, we can discuss the various characteristics of digital signal processing: mathematically intensive algorithms, real-time processing, sampled data implementation, and system flexibility. First, let us look at the concept of mathematically intensive algorithms.

Manuscript received October 6, 1986; revised March 27, 1987.
The authors are with the Semiconductor Group, Texas Instruments Inc., Houston, TX 77521-1445, USA.
IEEE Log Number 8716214.

Mathematically Intensive Algorithms

From (1), we can see that to generate every $y(n)$, we have to compute N multiplications and additions or sums of products. This computation makes it mathematically intensive, especially when N is large.

At this point it is worthwhile to give the FIR filter some physical significance. An FIR filter is a common technique used to eliminate the erratic nature of stock market prices. When the day-to-day closing prices are plotted, it is sometimes difficult to obtain the desired information, such as the trend of the stock, because of the large variations. A simple way of smoothing the data is to calculate the average closing values of the previous five days. For the new average value each day, the oldest value is dropped and the newest value added. Each daily average value (average (n)) would be the sum of the weighted value of the latest five days, where the weighting factors ($a(i)$'s) are 1/5. In equation form, the average is determined by

$$\text{average } (n) = \frac{1}{5} * d(n - 1) + \frac{1}{5} * d(n - 2) + \frac{1}{5} * d(n - 3) + \frac{1}{5} * d(n - 4) + \frac{1}{5} * d(n - 5) \qquad (2)$$

where $d(n - i)$ is the daily stock closing price for the $(n - i)$th day. Equation (2) assumes the same form as (1). This is also the general form of the convolution of two sequences of numbers, $a(i)$ and $x(i)$ [5], [6]. Both FIR filtering and convolution are fundamental to digital signal processing.

Real-Time Processing

In addition to being mathematically intensive, DSP algorithms must be performed in real time. Real time can be defined as a process that is accomplished by the DSP without creating a delay noticeable to the user. In the stock market example, as long as the new average value can be computed prior to the next day when it is needed, it is considered to be completed in real time. In digital signal processing applications, processes happen faster than on a daily basis. In the FIR filter example in (1), the sum of products must

be computed usually within hundreds of microseconds before the next sample comes into the system. A second example is in a speech recognition system where a noticeable delay between a word being spoken and being recognized would be unacceptable and not considered real-time. Another example is in image processing, where it is considered real-time if the processor finishes the processing within the frame update period. If the pixel information cannot be updated within the frame update period, problems such as flicker, smearing, or missing information will occur.

Sampled Data Implementation

The application must be capable of being handled as a sampled data system in order to be processed by digital processors, such as digital signal processors. The stock market is an example of a sampled data system. That is, a specific value (closing value) is assigned to each sample period or day. Other periods may be chosen such as hourly prices or weekly prices. In an FIR filter as shown in (1), the output $y(n)$ is calculated to be the weighted sum of the previous N inputs. In other words, the input signal is sampled at periodic intervals (1 over the sample rate), multiplied by weighting factor $a(i)$, and then added together to give the output result of $y(n)$. Examples of sample rates for some typical sampled data applications [2], [4] are shown in Table 1.

Table 1 Sample Rates versus Applications

Application	Nominal Sample Rate
Control	1 kHz
Telecommunications	8 kHz
Speech processing	8–10 kHz
Audio processing	40–48 kHz
Video frame rate	30 Hz
Video pixel rate	14 MHz

In a typical DSP application, the processor must be able to effectively handle sampled data in large quantity and also perform arithmetic computations in real time.

System Flexibility

The design of the digital signal processing system must be flexible enough to allow improvements in the state of the art. We may find out after several weeks of using the average stock price as a means of measuring a particular stock's value that a different method of obtaining the daily information is more suited to our needs, e.g., using different daily weightings, a different number of periods over which to average, or a different procedure for calculating the result. Enough flexibility in the system must be available to allow for these variations. In many of the DSP applications, techniques are still in the developmental phase, and therefore the algorithms tend to change over time. As an example, speech recognition is presently an inexact technique requiring continual algorithmic modification. From this example we can see the need for system flexibility so that the DSP algorithm can be updated. A programmable DSP system can provide this flexibility to the user.

HISTORICAL DSP SOLUTIONS

Over the past several decades, digital signal processing machines have taken on several evolutions in order to incorporate these characteristics. Large mainframe computers were initially used to process signals in the digital domain. Typically, because of state-of-the-art limitations, this was done in nonreal time. As the state of the art advanced, array processors were added to the processing task. Because of their flexibility and speed, array processors have become the accepted solution for the research laboratory, and have been extended to end-applications in many instances. However, integrated circuit technology has matured, thus allowing for the design of faster microprocessors and microcomputers. As a result, many digital signal processing applications have migrated from the array processor to microprocessor subsystems (i.e., bit-slice machines) to single-chip integrated circuit solutions. This migration has brought the cost of the DSP solution down to a point that allows pervasive use of the technology. The increased performance of these highly integrated circuits has also expanded DSP applications from traditional telecommunications to graphics/image processing, then to consumer audio processing.

A recent development in DSP technology is the single-chip digital signal processor, such as the TMS320 family of processors. These processors give the designer a DSP solution with its performance attainable only by the array processors a few years ago. Fig. 1 shows the TMS320 family in graphical form with the y-axis indicating the hypothetical performance and the x-axis being the evolution of the semiconductor processing technology. The first member of the family, the TMS32010, was disclosed to the market in 1982 [11], [12]. It gave the system designer the first microcomputer capable of performing five million DSP operations per second (5 MIPS), including the add and multiply functions [13] required in (1). Today there are a dozen spinoffs from the TMS32010 in the first generation of the TMS320 family. Some of these devices are the TMS320C10, TMS320C15, and TMS320C17 [14]. The second generation of devices include the TMS32020 [15] and TMS320C25 [16]. The TMS320C25 can perform 10 MIPS [16]. In addition, expanded memory space, combined single-cycle multiply/accumulate operation, multiprocessing capabilities, and expanded I/O functions have given the TMS320C25 a 2 to 4 times performance improvement over its predecessors. The third generation of the TMS320 family of processors, the TMS320C30 [26], [27], has a computational rate of 33 million DSP floating-point operations per second (33 MFLOPS). Its performance (speed, throughput, and precision) has far exceeded the digital signal processors available today and has reached the level of a supercomputer.

It we look closely at the TMS320 family as shown in Fig. 1, we can see that devices in the same generation, such as the TMS320C10, TMS320C15, and TMS320C17, are assembly object-code compatible. Devices across generations, such as the TMS320C10 and TMS320C25, are assembly source-code compatible. Software investment on DSP algorithms therefore can be maintained during the system upgrade. Another point is that since the introduction of the TMS32010, semiconductor processing technology has emerged from 3-μm NMOS to 2-μm CMOS to 1-μm CMOS.

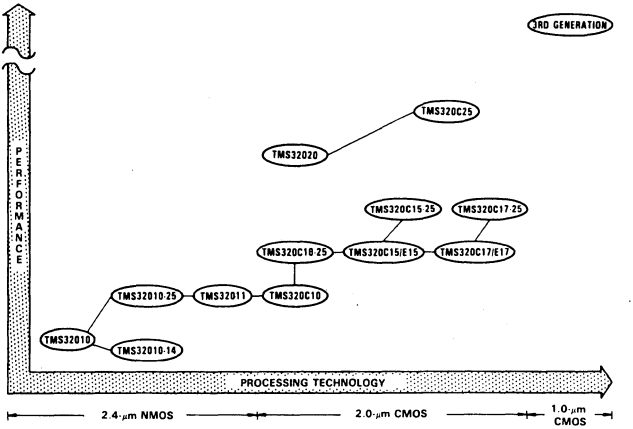


Fig. 1. The TMS320 family of digital signal processors.

The TMS320 generations of processors have also taken the same evolution in processing technology. Low power consumption, high performance, and high-density circuit integration are some of the direct benefits of this semiconductor processing evolution.

From Fig. 1, it can be observed that various DSP building blocks, such as the CPU, RAM, ROM, I/O configurations, and processor speeds, have been designed as individual modules and can be rearranged or combined with other standard cells to meet the needs of specific applications. Each of the three generations (and future generations) will evolve in the same manner. As applications become more sophisticated, semicustom solutions based on the core CPU will become the solution of choice. An example of this approach is the TMS320C17/E17, which consists of the TMS320C10 core CPU, expanded 4K-word program ROM (TMS320C17) or EPROM (TMS320E17), enlarged data RAM of 256 words, dual serial ports, companding hardware, and a coprocessor interface. Furthermore, as integrated circuit layout rules move into smaller geometry (now at 2 μm, rapidly going to 1 μm), not only will the TMS320 devices become smaller in size, but also multiple CPUs will be incorporated on the same device along with application-specific I/O to achieve low-cost integrated system solutions.

Basic TMS320 Architecture

As noted previously, the underlying assumption regarding a digital signal processor is fast arithmetic operations and high throughput to handle mathematically intensive algorithms in real time. In the TMS320 family [11]–[17], [26], [27], this is accomplished by using the following basic concepts:

- Harvard architecture,
- extensive pipelining,
- dedicated hardware multiplier,
- special DSP instructions,
- fast instruction cycle.

These concepts were designed into the TMS320 digital signal processors to handle the vast amount of data characteristic of DSP operations, and to allow most DSP operations to be executed in a single-cycle instruction. Furthermore, the TMS320 processors are programmable devices, providing the flexibility and ease of use of general-purpose microprocessors. The following paragraphs discuss how each of the above concepts is used in the TMS320 family of devices to make them useful in digital signal processing applications.

Harvard Architecture

The TMS320 utilizes a modified Harvard architecture for speed and flexibility. In a strict Harvard architecture [18], [19], the program and data memories lie in two separate spaces, permitting a full overlap of instruction fetch and execution. The TMS320 family's modification of the Harvard architecture further allows transfer between program and data spaces, thereby increasing the flexibility of the device. This architectural modification eliminates the need for a separate coefficient ROM and also maximizes the processing power by maintaining two separate bus structures (program and data) for full-speed execution.

Extensive Pipelining

In conjunction with the Harvard architecture, pipelining is used extensively to reduce the instruction cycle time to its absolute minimum, and to increase the throughput of the processor. The pipeline can be anywhere from two to four levels deep, depending on which processor in the family is used. The TMS320 family architecture uses a two-level pipeline for its first generation, a three-level pipeline for its second generation, and a four-level pipeline for its third generation of processors. This means that the device is processing from two to four instructions in parallel, and each instruction is at a different stage in its execution. Fig. 2 shows an example of a three-level pipeline operation.

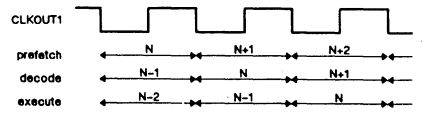


Fig. 2. Three-level pipeline operation.

In pipeline operation, the prefetch, decode, and execute operations can be handled independently, thus allowing the execution of instructions to overlap. During any instruction cycle, three different instructions are active, each at a different stage of completion. For example, as the *N*th instruction is being prefetched, the previous $(N - 1)$th instruction is being decoded, and the previous $(N - 2)$th instruction is being executed. In general, the pipeline is transparent to the user.

Dedicated Hardware Multiplier

As we saw in the general form of an FIR filter, multiplication is an important part of digital signal processing. For each filter tap (denoted by *i*), a multiplication and an addition must take place. The faster a multiplication can be performed, the higher the performance of the digital signal processor. In general-purpose microprocessors, the multiplication instruction is constructed by a series of additions, therefore taking many instruction cycles. In comparison, the characteristic of every DSP device is a dedicated multiplier. In the TMS320 family, multiplication is a single-cycle instruction as a result of the dedicated hardware multiplier. If we look at the arithmetic for each tap of the FIR filter to be performed by the TMS32010, we see that each tap of the filter requires a multiplication (MPY) instruction.

```
LT     ;LOAD MULTIPLICAND INTO T REGISTER
DMOV   ;MOVE DATA IN MEMORY TO DO DELAY
MPY    ;MULTIPLY
APAC   ;ADD MULTIPLICATION RESULT TO ACC
```

The other three instructions are used to load the multiplier circuit with the multiplicand (LT), move the data through the filter tap (DMOV), and add the result of the multiplication (stored in the product register) to the accumulator (APAC). Specifically, the multiply instruction (MPY) loads the multiplier into the dedicated multiplier and performs the multiplication, placing the result in a product register. Therefore, if a 256-tap FIR filter is used, these four instructions are repeated 256 times. At each sample period, 256 multiplications must be performed. In a typical general-purpose microprocessor, this requires each tap to be 30 to 40 instruction cycles long, whereas in the TMS320C10, it is only four instruction cycles. We will see in the next section how special DSP instructions reduce the time required for each FIR tap even further.

Special DSP Instructions

Another characteristic of DSP devices is the use of special instructions. We were introduced to one of them in the previous example, the DMOV (data move) instruction. In digital signal processing, the delay operator (z^{-1}) is very important. Recalling the stock market example, during each new sample period (i.e., each new day), the oldest piece of data (the closing price five days ago) was dropped and a new one (today's closing price) was added. Or, each piece of the old data is delayed or moved one sample period to make room for the incoming most current sample. This delay is the function of the DMOV instruction. Another special instruction in the TMS32010 is the LTD instruction. It executes the LT, DMOV, and APAC instructions in a single cycle. The LTD and MPY instruction then reduce the number of instruction cycles per FIR filter tap from four to two. In the second-generation TMS320, such as the TMS320C25, two more special instructions have been included (the RPT and MACD instructions) to reduce the number of cycles per tap to one, as shown in the following:

```
RPTK  255  ;REPEAT THE NEXT INSTRUCTION 256 TIMES
           (N + 1)
MACD       ;LT, DMOV, MPY, AND APAC
```

Fast Instruction Cycle

The real-time processing capability is further enhanced by the raw speed of the processor in executing instructions. The characteristics which we have discussed, combined with optimization of the integrated circuit design for speed, give the DSP devices instruction cycle times less than 200 ns. The specific instruction cycle times for the TMS320 family are given in Table 2. These fast cycle times have made the TMS320 family of processors highly suited for many real-time DSP applications. Table 1 showed the sample rates for some typical DSP applications. This table can be combined with the cycle times indicated in Table 2 to show how many instruction cycles per sample can be achieved by the various generations of the TMS320 for real-time applications (see Fig. 3).

Table 2 TMS320 Cycle Times

Device	Cycle Time (ns)
TMS320C10*	160-200
TMS32020	160-200
TMS320C25	100-125
TMS320C30	60-75

*The same cycle time applies to all of the first-generation processors.

As we can see from Fig. 3, many instruction cycles are available to process the signal or to generate commands for real-time control applications. Therefore, for simple control applications, the general-purpose microprocessors or controllers would be adequate. However, for more mathematically intensive control applications, such as robotics and adaptive control, digital signal processors are much better suited [24]. The number of available instruction cycles is reduced as we increase the sample rate from 8 kHz for typical telecommunication applications to 40-48 kHz for audio processing. Since most of these real-time applications require only a few hundreds of instructions per sample (such as ADPCM [4], and echo cancelation [4]), this is within the reach of the TMS320. For higher sample rate applications, such as video/image processing, digital signal processors available today are not capable of handling the processing of the real-time video data. Therefore, for these

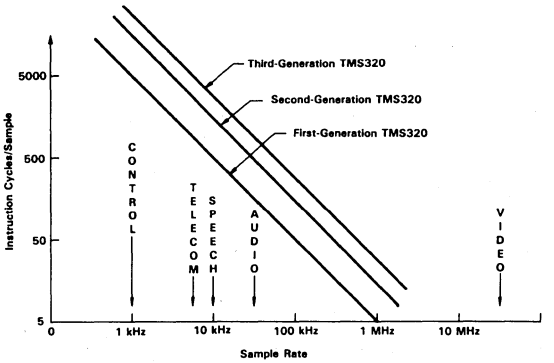


Fig. 3. Number of instruction cycles/sample versus sample rate for the TMS320 family.

types of applications, multiple digital signal processors and frame buffers are usually required. From Fig. 3, it can also be seen that for slower speed applications, such as control, the first-generation TMS320 provides better cost-performance tradeoffs than the other processors. For high sample rate applications, such as video/image processing, the second and third generations of the TMS320 with their multiprocessing capabilities and high throughput are better suited.

Now that we have discussed the basic characteristics of digital signal processors, we can concentrate on specific details of each of the three generations of the TMS320 family devices.

The First Generation of the TMS320 Family

The first generation of the TMS320 family includes the TMS32010 [13], and TMS32011 [17], which are processed in 2.4-μm NMOS technology, and the TMS320C10 [13], TMS320C15/E15 [14], and TMS320C17/E17 [14], processed in 1.8-μm CMOS technology. Some of the key features of these devices are [14] as follows:

- Instruction cycle timing:
  - -160 ns
  - -200 ns
  - -280 ns.
- On-chip data RAM:
  - -144 words
  - -256 words (TMS320C15/E15, TMS320C17/E17).
- On-chip program ROM:
  - -1.5K words
  - -4K words (TMS320C15, TMS320C17).
- 4K words of on-chip program EPROM (TMS320E15, TMS320E17).
- External memory expansion up to 4K words at full speed.
- 16 × 16-bit parallel multiplier with 32-bit result.
- Barrel shifter for shifting data memory words into the ALU.
- Parallel shifter.
- 4 × 12-bit stack that allows context switching.
- Two auxiliary registers for indirect addressing.
- Dual-channel serial port (TMS32011, TMS320C17, TMS320E17).
- On-chip companding hardware (TMS32011, TMS320C17, TMS320E17).
- Coprocessor interface (TMS320C17, TMS320E17).
- Device packaging
  - -40-pin DIP
  - -44-pin PLCC.

TMS320C10

The first generation of the TMS320 processors is based on the architecture of the TMS32010 and its CMOS replica, the TMS320C10. The TMS32010 was introduced in 1982 and was the first microcomputer capable of performing 5 MIPS. Since the TMS32010 has been covered extensively in the literature [4], [11]–[14], we will only provide a cursory review here. A functional block diagram of the TMS320C10 is shown in Fig. 4.

As shown in Fig. 4, the TMS320C10 utilizes the modified Harvard architecture in which program memory and data memory lie in two separate spaces. Program memory can reside both on-chip (1.5K words) or off-chip (4K words). Data memory is the 144 × 16-bit on-chip data RAM. There are four basic arithmetic elements: the ALU, the accumulator, the multiplier, and the shifters. All arithmetic operations are performed using two's-complement arithmetic.

ALU: The ALU is a general-purpose arithmetic logic unit that operates with a 32-bit data word. The unit can add, subtract, and perform logical operations.

Accumulator: The accumulator stores the output from the ALU and is also often an input to the ALU. It operates with a 32-bit word length. The accumulator is divided into a high-order word (bits 31 through 16) and a low-order word (bits 15 through 0). Instructions are provided for storing the high- and low-order accumulator words in data memory (SACH for store accumulator high and SACL for store accumulator low).

Multiplier: The 16 × 16-bit parallel multiplier consists of three units: the T register, the P register, and the multipler array. The T register is a 16-bit register that stores the multiplicand, while the P register is a 32-bit register that stores the product. In order to use the multiplier, the multiplicand

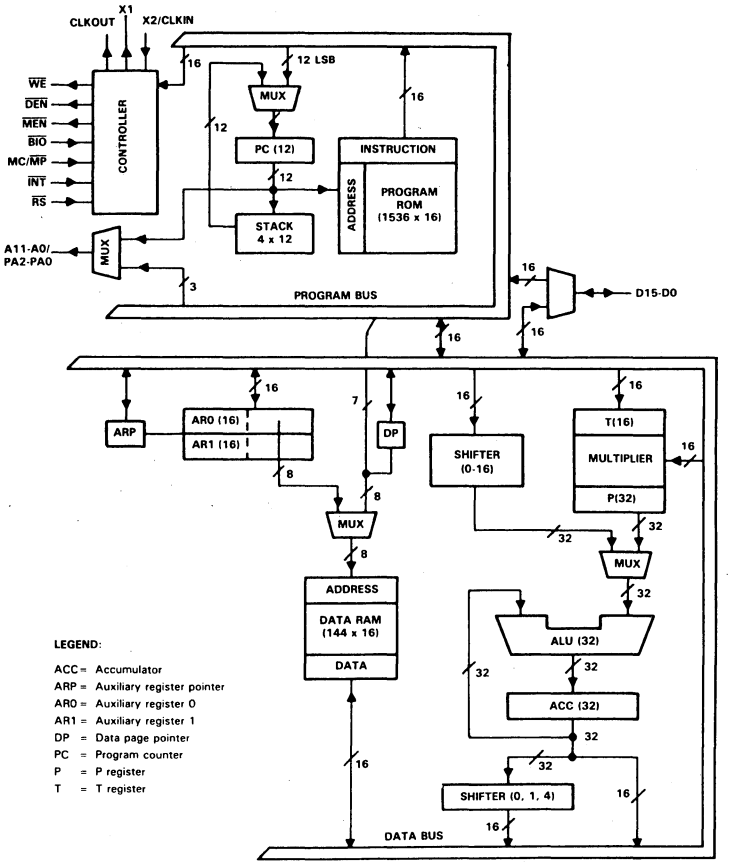


Fig. 4. TMS320C10 functional block diagram.

must first be loaded into the T register from the data RAM by using one of the following instructions: LT, LTA, or LTD. Then the MPY (multiply) or the MPYK (multiply immediate) instruction is executed. The multiply and accumulate operations can be accomplished in two instruction cycles with the LTA/LTD and MPY/MPYK instructions.

Shifters: Two shifters are available for manipulating data: a barrel shifter and a parallel shifter. The barrel shifter performs a left-shift of 0 to 16 bits on all data memory words that are to be loaded into, subtracted from, or added to the accumulator. The parallel shifter, activated by the SACH instruction, can execute a shift of 0, 1, or 4 bits to take care of the sign bits in two's-complement arithmetic calculations.

Based on the architecture of the TMS32010/C10, several spinoffs have been generated offering different processor speeds, expanded memory, and various I/O integration. Currently, the newest members in this generation are the TMS320C15/E15 and the TMS320C17/E17 [14].

TMS320C15/E15

The TMS320C15 and TMS320E15 are fully object-code and pin-for-pin compatible with the TMS32010 and offer expanded on-chip RAM of 256 words and on-chip program ROM (TMS320C15) or EPROM (TMS320E15) of 4K words. The TMS320C15 is available in either a 200-ns version or a 160-ns version (TMS320C15-25).

TMS320C17/E17

The TMS320C17/E17 is a dedicated microcomputer with 4K words of on-chip program ROM (TMS320C17) or EPROM (TMS320E17), a dual-channel serial port for full-duplex serial communication, on-chip companding hardware (u-law/A-law), a serial port timer for stand-alone serial communication, and a coprocessor interface for zero glue interface between the processor and any 4/8/16-bit microprocessor. The TMS320C17/E17 is also object-code compatible with the TMS32010 and can use the same development tools. The

Table 3 TMS320 First-Generation Processors

TMS320 Devices	Instruction Cycle Time (ns)	Process	On-Chip Prog ROM (words)	On-Chip Prog EPROM (words)	On-Chip Data RAM (words)	Off-Chip Prog (words)	Ref
TMS32010	200	NMOS	1.5K		144	4K	[13]
TMS32010-25	160	NMOS	1.5K		144	4K	[13]
TMS32010-14	280	NMOS	1.5K		144	4K	[13]
TMS32011	200	NMOS	1.5K		144		[17]
TMS320C10	200	CMOS	1.5K		144	4K	[13]
TMS320C10-25	160	CMOS	1.5K		144	4K	[13]
TMS320C15	200	CMOS	4.0K		256	4K	[13]
TMS320C15-25	160	CMOS	4.0K		256	4K	[14]
TMS320E15	200	CMOS		4.0K	256	4K	[14]
TMS320C17	200	CMOS	4.0K		256		[14]
TMS320C17-25	160	CMOS	4.0K		256		[14]
TMS320E17	200	CMOS		4.0K	256		[14]

device is based on the TMS320C10 core CPU with added peripheral memory and I/O modules added on-chip. The TMS320C17/E17 can be regarded as a semicustom DSP solution suited for high-volume telecommunication and consumer applications.

Table 3 provides a feature comparison of all members of the first-generation TMS320 processors. References to more detailed information on these processors are also provided.

The Second Generation of the TMS320 Family

The second-generation TMS320 digital signal processors includes two members, the TMS32020 [15] and the TMS320C25 [16]. The architecture of these devices has been evolved from the TMS32010, the first member of the TMS320 family. Key features of the second-generation TMS320 are as follows:

- Instruction cycle timing:
  - -100 ns (TMS320C25)
  - -200 ns (TMS32020).
- 4K words of on-chip masked ROM (TMS320C25).
- 544 words of on-chip data RAM.
- 128K words of total program data memory space.
- Eight auxiliary registers with a dedicated arithmetic unit.
- Eight-level hardware stack.
- Fully static double-buffered serial port.
- Wait states for communication to slower off-chip memories.
- Serial port for multiprocessing or interfacing to codecs.
- Concurrent DMA using an extended hold operation (TMS320C25).
- Bit-reversed addressing modes for fast Fourier transforms (TMS320C25).
- Extended-precision arithmetic and adaptive filtering support (TMS320C25).
- Full-speed operation of MAC/MACD instructions from external memory (TMS320C25).
- Accumulator carry bit and related instructions (TMS320C25).
- 1.8-μm CMOS technology (TMS320C25):
  - -68-pin grid array (PGA) package.
  - -68-pin lead chip carrier (PLCC) package.
- 2.4-μm NMOS technology (TMS32020):
  - -68-pin PGA package.

TMS320C25 Architecture

The TMS320C25 is the latest member in the second generation of TMS320 digital signal processors. It is a pin-compatible CMOS version of the TMS32020 microprocessor, but with an instruction cycle time twice as fast and the inclusion of additional hardware and software features. The instruction set is a superset of both the TMS32010 and TMS32020, maintaining source-code compatibility. In addition, it is completely object-code compatible with the TMS32020 so that TMS32020 programs run unmodified on the TMS320C25.

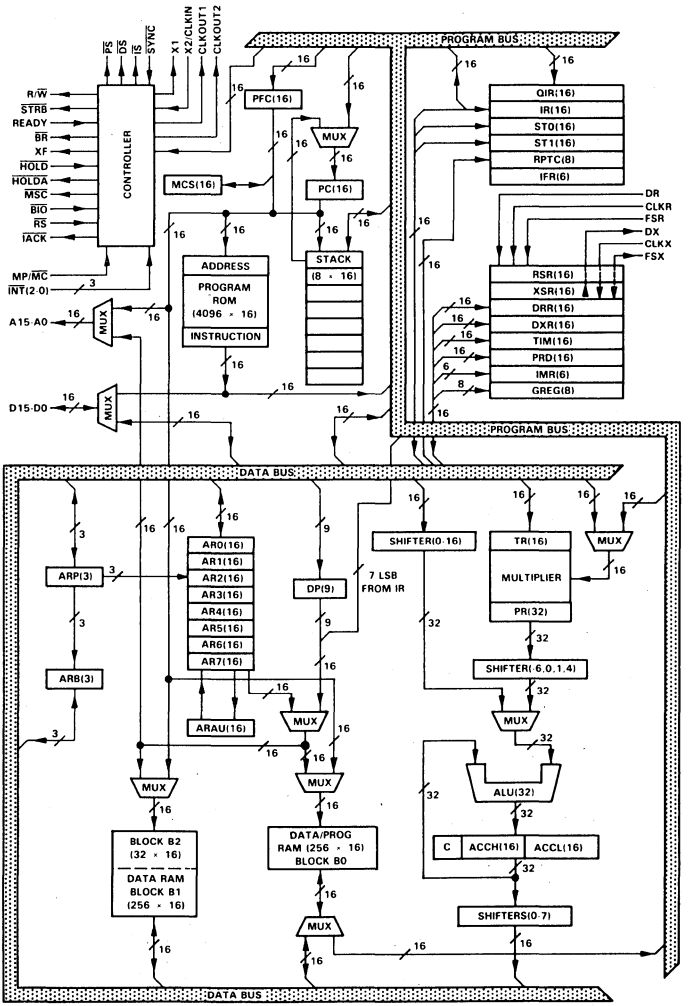
The 100-ns instruction cycle time provides a significant throughput advantage for many existing applications. Since most instructions are capable of executing in a single cycle, the processor is capable of executing ten million instructions per second (10 MIPS). Increased throughput on the TMS320C25 for many DSP applications is attained by means of single-cycle multiply/accumulate instructions with a data move option (MAC/MACD), eight auxiliary registers with a dedicated arithmetic unit, instruction set support for adaptive filtering and extended-precision arithmetic, bit-reversal addressing, and faster I/O necessary for data-intensive signal processing.

Instructions are included to provide data transfers between the two memory spaces. Externally, the program and data memory spaces are multiplexed over the same bus so as to maximize the address range for both spaces while minimizing the pin count of the device. Internally, the TMS320C25 architecture maximizes processing power by maintaining two separate bus structures, program and data, for full-speed execution.

Program execution in the device takes the form of a three-level instruction fetch-decode-execute pipeline (see Fig. 2). The pipeline is essentially invisible to the user, except in some cases where it must be broken (such as for branch instructions). In this case, the instruction timing takes into account the fact that the pipeline must be emptied and refilled. Two large on-chip data RAM blocks (a total of 544 words), one of which is configurable either as program or data memory, provide increased flexibility in system design. An off-chip 64K-word directly addressable data memory address space is included to facilitate implementations of DSP algorithms. The large on-chip 4K-word masked ROM can be used for cost-reduced systems, thus providing for a true single-chip DSP solution. The remainder of the 64K-word program memory space is located externally. Large

programs can execute at full speed from this memory space. Programs may also be downloaded from slow external memory to on-chip RAM for full-speed operation. The VLSI implementation of the TMS320C25 incorporates all of these features as well as many others such as a hardware timer, serial port, and block data transfer capabilities.

A functional block diagram of the TMS320C25, shown in Fig. 5, outlines the principal blocks and data paths within



LEGEND:

ACCH	= Accumulator high	IFR	= Interrupt flag register	PC	= Program counter
ACCL	= Accumulator low	IMR	= Interrupt mask register	PFC	= Prefetch counter
ALU	= Arithmetic logic unit	IR	= Instruction register	RPTC	= Repeat instruction counter
ARAU	= Auxiliary register arithmetic unit	MCS	= Microcall stack	GREG	= Global memory allocation register
ARB	= Auxiliary register pointer buffer	QIR	= Queue instruction register	RSR	= Serial port receive shift register
ARP	= Auxiliary register pointer	PR	= Product register	XSR	= Serial port transmit shift register
DP	= Data memory page pointer	PRD	= Period register for timer	AR0-AR7	= Auxiliary registers
DRR	= Serial port data receive register	TIM	= Timer	ST0,ST1	= Status registers
DXR	= Serial port data transmit register	TR	= Temporary register		

Fig. 5. TMS320C25 functional block diagram.

the processor. The diagram also shows all of the TMS320C25 interface pins.

In the following architectural discussions on the memory, central arithmetic logic unit, hardware multiplier, control operations, serial port, and I/O interface, please refer to the block diagram shown in Fig. 5.

Memory Allocation: The TMS320C25 provides a total of 4K 16-bit words of on-chip program ROM and 544 16-bit words of on-chip data RAM. The RAM is divided into three separate Blocks (B0, B1, and B2). Of the 544 words, 256 words (block B0) are configurable as either data or program memory by CNFD (configure data memory) or CNFP (configure program memory) instructions provided for that purpose; 288 words (blocks B1 and B2) are always data memory. A data memory size of 544 words allows the TMS320C25 to handle a data array of 512 words while still leaving 32 locations for intermediate storage. The TMS320C25 provides 64K words of off-chip directly addressable data memory space as well as a 64K-word off-chip program memory space.

A register file containing eight Auxiliary Registers (AR0–AR7), which are used for indirect addressing of data memory and for temporary storage, increase the flexibility and efficiency of the device. These registers may be either directly addressed by an instruction or indirectly addressed by a 3-bit Auxiliary Register Pointer (ARP). The auxiliary registers and the ARP may be loaded from either data memory or by an immediate operand defined in the instruction. The contents of these registers may also be stored into data memory. The auxiliary register file is connected to the Auxiliary Register Arithmetic Unit (ARAU). Using the ARAU accessing tables of information does not require the CALU for address manipulation, thus freeing it for other operations.

Central Arithmetic Logic Unit (CALU): The CALU contains a 16-bit scaling shifter, a 16 × 16-bit parallel multiplier, a 32-bit Arithmetic Logic Unit (ALU), and a 32-bit accumulator. The scaling shifter has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. This shifter produces a left-shift of 0 to 16 bits on the input data, as programmed in the instruction. Additional shifters at the outputs of both the accumulator and the multiplier are suitable for numerical scaling, bit extraction, extended-precision arithmetic, and overflow prevention.

The following steps occur in the implementation of a typical ALU instruction:

1) Data are fetched from the RAM on the data bus.
2) Data are passed through the scaling shifter and the ALU where the arithmetic is performed.
3) The result is moved into the accumulator.

The 32-bit accumulator is split into two 16-bit segments for storage in data memory: ACCH (accumulator high) and ACCL (accumulator low). The accumulator has a carry bit to facilitate multiple-precision arithmetic for both addition and subtract instructions.

Hardware Multiplier: The TMS320C25 utilizes a 16 × 16-bit hardware multiplier, which is capable of computing a 32-bit product during every machine cycle. Two registers are associated with the multiplier:

- a 16-bit Temporary Register (TR) that holds one of the operands for the multiplier, and
- a 32-bit Product Register (PR) that holds the product.

The output of the product register can be left-shifted 1 or 4 bits. This is useful for implementing fractional arithmetic or justifying fractional products. The output of the PR can also be right-shifted 6 bits to enable the execution of up to 128 consecutive multiple/accumulates without overflow. An unsigned multiply (MPYU) instruction facilitates extended-precision multiplication.

I/O Interface: The TMS320C25 I/O space consists of 16 input and 16 output ports. These ports provide the full 16-bit parallel I/O interface via the data bus on the device. A single input (IN) or output (OUT) operation typically takes two cycles; however, when used with the repeat counter, the operation becomes single-cycle. I/O devices are mapped into the I/O address space using the processor's external address and data buses in the same manner as memory-mapped devices. Interfacing to memory and I/O devices of varying speeds is accomplished by using the READY line.

A Direct Memory Access (DMA) to external program/data memory is also supported. Another processor can take complete control of the TMS320C25's external memory by asserting $\overline{\text{HOLD}}$ low, causing the TMS320C25 to place its address, data, and control lines in the high-impedance state. Signaling between the external processor and the TMS320C25 can be performed using interrupts. Two modes of DMA are available on the device. In the first, execution is suspended during assertion of $\overline{\text{HOLD}}$. In the second "concurrent DMA" mode, the TMS320C25 continues to execute its program while operating from internal RAM or ROM, thus greatly increasing throughput in data-intensive applications.

TMS320C25 Software

The majority of the TMS320C25 instructions (97 out of 133) are executed in a single instruction cycle. Of the 36 instructions that require additional cycles of execution, 21 involve branches, calls, and returns that result in a reload of the program counter and a break in the execution pipeline. Another seven of the instructions are two-word, long-immediate instructions. The remaining eight instructions support I/O, transfers of data between memory spaces, or provide for additional parallel operation in the processor. Furthermore, these eight instructions (IN, OUT, BLKD, BLKP, TBLR, TBLW, MAC, and MACD) become single-cycle when used in conjunction with the repeat counter. The functional performance of the instructions exploits the parallelism of the processor, allowing complex and/or numerically intensive computations to be implemented in relatively few instructions.

Addressing Modes: Since most of the instructions are coded in a single 16-bit word, most instructions can be executed in a single cycle. Three memory addressing modes are available with the instruction set: direct, indirect, and immediate addressing. Both direct and indirect addressing are used to access data memory. Immediate addressing uses the contents of the memory addressed by the program counter.

When using direct addressing, 7 bits of the instruction word are concatenated with the 9 bits of the data memory page pointer (DP) to form the 16-bit data memory address. With a 128-word page length, the DP register points to one of 512 possible data memory pages to obtain a 64K total data memory space. Indirect addressing is provided by the aux-

iliary registers (AR0–AR7). The seven types of indirect addressing are shown in Table 4. Bit-reversed indexed addressing modes allow efficient I/O to be performed for the resequencing of data points in a radix-2 FFT program.

Table 4 Addressing Modes of the TMS320C25

Addressing Mode	Operation
OP A	direct addressing
OP * (,NARP)	indirect; no change to AR.
OP *+(,NARP)	indirect; current AR is incremented.
OP *−(,NARP)	indirect; current AR is decremented.
OP *0+(,NARP)	indirect; AR0 is added to current AR.
OP *0−(,NARP)	indirect; AR0 is subtracted from current AR.
OP *BR0+(,NARP)	indirect; AR0 is added to current AR (with reverse carry propagation).
OP *BR0−(,NARP)	indirect; AR0 is subtracted from current AR (with reverse carry propagation).

Note: The optional NARP field specifies a new value of the ARP.

TMS320C25 System Configurations

The flexibility of the TMS320C25 allows systems configurations to satisfy a wide range of application requirements [16]. The TMS320C25 can be used in the following configurations:

- a stand-alone system (a single processor using 4K words of on-chip ROM and 544 words of on-chip RAM),
- parallel multiprocessing systems with shared global data memory, or
- host/peripheral coprocessing using interface control signals.

A minimal processing system is shown in Fig. 6 using external data RAM and PROM/EPROM. Parallel multiprocessing and host/peripheral coprocessing systems can be designed by taking advantage of the TMS320C25's direct memory access and global memory configuration capabilities.

In some digital processing tasks, the algorithm being implemented can be divided into sections with a distinct processor dedicated to each section. In this case, the first and second processors may share global data memory, as well as the second and third, the third and fourth, etc. Arbitration logic may be required to determine which section of the algorithm is executing and which processor has access to the global memory. With multiple processors dedicated to distinct sections of the algorithm, throughput can be increased via pipelined execution. The TMS320C25 is capable of allocating up to 32K words of data memory as global memory for multiprocessing applications.

The Third Generation of the TMS320 Family

The TMS320C30 [26]–[27] is Texas Instruments third-generation member of the TMS320 family of compatible digital signal processors. With a computational rate of 33 MFLOPS (million floating-point operations per second), the TMS320C30 far exceeds the performance of any programmable DSP available today. Total system performance has been maximized through internal parallelism, more than twenty-four thousand bytes of on-chip memory, single-cycle floating-point operations, and concurrent I/O. The total system cost is minimized with on-chip memory and on-chip peripherals such as timers and serial ports. Finally, the user's system design time is dramatically reduced with the availability of the floating-point operations, general-purpose instructions and features, and quality development tools.

The TMS320C30 provides the user with a level of performance that, at one time, was the exclusive domain of supercomputers. The strong architectural emphasis of providing a low-cost system solution to demanding arithmetic algorithms has resulted in the architecture shown in Fig. 7.

The key features of the TMS320C30 [26], [27] are as follows:

- 60-ns single-cycle execution time, 1-μm CMOS.
- Two 1K × 32-bit single-cycle dual-access RAM blocks.
- One 4K × 32-bit single-cycle dual-access ROM block.
- 64 × 32-bit instruction cache.
- 32-bit instruction and data words, 24-bit addresses.
- 32/40-bit floating-point and integer multiplier.
- 32/40-bit floating-point, integer, and logical ALU.
- 32-bit barrel shifter.
- Eight extended-precision registers.
- Two address-generators with eight auxiliary registers.
- On-chip Direct Memory Access (DMA) controller for concurrent I/O and CPU operation.
- Peripheral bus and modules for easy customization.
- High-level language support.
- Interlocked instructions for multiprocessing support.
- Zero overhead loops and single-cycle branches.

The architecture of the TMS320C30 is targeted at 60-ns and faster cycle times. To achieve such high-performance

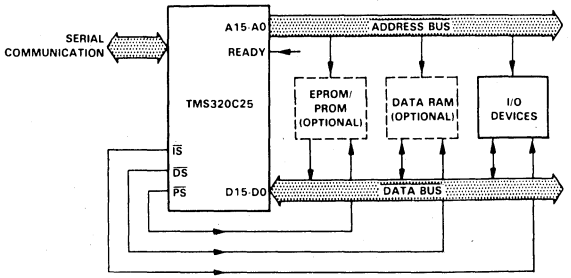


Fig. 6. Minimal processing system with external data RAM and PROM/EPROM.

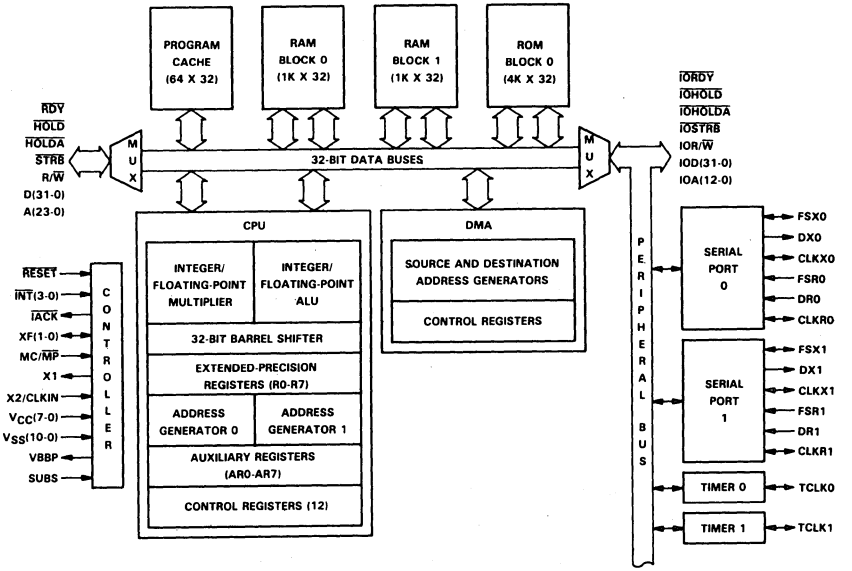


Fig. 7. TMS320C30 functional block diagram.

goals while still providing low-cost system solutions, the TMS320C30 is designed using Texas Instruments state-of-the-art 1-μm CMOS process. The TMS320C30 's high system performance is achieved through a high degree of parallelism, the accuracy and precision of its floating-point units, its on-chip DMA controller that supports concurrent I/O, and its general-purpose features. At the heart of the architecture is the Central Processing Unit (CPU).

The CPU

The CPU consists of the following elements: floating-point/integer multiplier; ALU for performing floating-point, integer, and logical operations; auxiliary register arithmetic units; supporting register file, and associated buses. The multiplier of the CPU performs floating-point and integer multiplication. When performing floating-point multiplication, the inputs are 32-bit floating-point numbers, and the result is a 40-bit floating-point number. When performing integer multiplication, the input data is 24 bits and yields a 32-bit result. The ALU performs 32-bit integer, 32-bit logical, and 40-bit floating-point operations. Results of the multiplier and the ALU are always maintained in 32-bit integer or 40-bit floating-point formats. The TMS320C30 has the ability to perform, in a single cycle, parallel multiplies and adds (subtracts) on integer or floating-point data. It is this ability to perform floating-point multiplies and adds (subtracts) in a single cycle which give the TMS320C30 its peak computational rate of 33 MFLOPS.

Floating-point operations provide the user with a convenient and virtually trouble-free means of performing computations while maintaining accuracy and precision. The TMS320C30 implementation of floating-point arithmetic allows for floating-point operations at integer speeds. The floating-point capability allows the user to ignore, to a large extent, problems with overflow, operand alignment, and other burdensome tasks common to integer operations.

The register file contains 28 registers, which may be operated upon by the multiplier and ALU. The first eight of these registers (R0–R7) are the extended-precision registers, which support operations on 40-bit floating-point numbers and 32-bit integers.

The next eight registers (AR0–AR7) are the auxiliary registers, whose primary function is related to the generation of addresses. However, they also may be used as general-purpose 32-bit registers. Two auxiliary register arithmetic units (ARAU0 and ARAU1) can generate two addresses in a single cycle. The ARAUs operate in parallel with the multiplier and ALU. They support addressing with displacements, index registers (IR0 and IR1), and circular and bit-reversed addressing.

The remaining registers support a variety of system functions: addressing, stack management, processor status, block repeat, and interrupts.

Data Organization

Two integer formats are supported on the TMS320C30: a 16-bit format used for immediate integer operands and a 32-bit single-precision integer format.

Two unsigned-integer formats are available: a 16-bit format for immediate unsigned-integer operands and a 32-bit single-precision unsigned-integer format.

The three floating-point formats are assumed to be normalized, thus providing an extra bit of precision. The first

is a 16-bit short floating-point format for immediate floating-point operands, which consists of a 4-bit exponent, 1 sign bit, and an 11-bit fraction. The second is a single-precision format consisting of an 8-bit exponent, 1 sign bit, and a 23-bit fraction. The third is an extended-precision format consisting of an 8-bit exponent, 1 sign bit, and a 31-bit fraction.

The total memory space of the TMS320C30 is 16M (million) × 32 bits. A machine word is 32 bits, and all addressing is performed by word. Program, data, and I/O space are contained within the 16M-word address space.

RAM blocks 0 and 1 are each 1K × 32 bits. The ROM block is 4K × 32 bits. Each RAM block and ROM block is capable of supporting two data accesses in a single cycle. For example, the user may, in a single cycle, access a program word and a data word from the ROM block.

The separate program data, and DMA buses allow for parallel program fetches, data reads and writes, and DMA operations. Management of memory resources and busing is handled by the memory controller. For example, a typical mode of operation could involve a program fetch from the on-chip program cache, two data fetches from RAM block 0, and the DMA moving data from off-chip memory to RAM block 1. All of this can be done in parallel with no impact on the performance of the CPU.

A 64 × 32-bit instruction cache allows for maximum system performance with minimal system cost. The instruction cache stores often repeated sections of code. The code may then be fetched from the cache, thus greatly reducing the number of off-chip accesses necessary. This allows for code to be stored off-chip in slower, lower cost memories. Also, the external buses are freed, thus allowing for their use by the DMA or other devices in the system.

DMA

The TMS320C30 processes an on-chip Direct Memory Access (DMA) controller. The DMA controller is able to perform reads from and writes to any location in the memory map without interfering with the operation of the CPU. As a consequence, it is possible to interface the TMS320C30 to slow external memories and peripherals (A/Ds, serial ports, etc.) without affecting the computational throughput of the CPU. The result is improved system performance and decreased system cost.

The DMA controller contains its own address generators, source and destination registers, and transfer counter. Dedicated DMA address and data buses allow for operation with no conflicts between the CPU and DMA controller.

The DMA controller responds to interrupts in a similar way to the CPU. This ability allows the DMA to transfer data based upon the interrupts received. Thus I/O transfers that would normally be performed by the CPU may instead be performed by the DMA. Again, the CPU may continue processing data while the DMA receives or transmits data.

Peripherals

All peripheral modules are manipulated through memory-mapped registers located on a dedicated peripheral bus. This peripheral bus allows for the straightforward addition, removal, and creation of peripheral modules. The initial TMS320C30 peripheral library will include timers and serial ports. The peripheral library concept allows Texas Instruments to create new modules to serve a wide variety of applications. For example, the configuration of the TMS320C30 in Fig. 7 includes two timers and two serial ports.

Timers: The two timer modules are general-purpose timer/event counters, with two signaling modes and internal or external clocking.

Available to each timer is an I/O pin that can be used as an input clock to the timer or as an output signal driven by the timer. The pin may also be configured as a general-purpose I/O pin.

Serial Ports: The two serial ports are modular and totally independent. Each serial port can be configured to transfer 8, 16, 24, or 32 bits of data per frame. The clock for each serial port can originate either internally or externally. An internally generated divide-down clock is provided. The pins of the serial ports are configurable as general-purpose I/O pins. A special handshake mode allows TMS320C30s to communicate over their serial ports with guaranteed synchronization. The serial ports may also be configured to operate as timers.

External Interfaces

The TMS320C30 provides two external interfaces: the parallel interface and the I/O interface. The parallel interface consists of a 32-bit data bus, a 24-bit address bus, and a set of control signals. The I/O interface consists of a 32-bit data bus, a 13-bit address bus, and a set of control signals. Both ports support an external ready signal for wait-state generation and the use of software-controlled wait states.

The TMS320C30 supports four external interrupts, a number of internal interrupts, and a nonmaskable external reset signal. Two dedicated, general-purpose, external I/O flags, XF0 and XF1, may be configured as input or output pins under software control. These pins are also used by the interlocked instructions to support multiprocessor communication.

Pipelining In the TMS320C30

The operation of the TMS320C30 is controlled by five major functional units. The five major units and their function are as follows:

- *Fetch Unit* (F) which controls the program counter updates and fetches of the instruction words from memory.
- *Decode Unit* (D) which decodes the instruction word and controls address generation.
- *Read Unit* (R) which controls the operand reads from memory.
- *Execute Unit* (E) which reads operands from the register file, performs the necessary operation, and writes results back to the register file and memory.
- *DMA Channel* (DMA) which reads and writes memory concurrently with CPU operation.

Each instruction is operated upon by four of these stages; namely, fetch, decode, read, and execute. To provide for maximum processor throughput these units can perform in parallel with each unit operating on a different instruction. The overlapping of the fetch, decode, read, and execute operations of different instructions is called pipelining. The DMA controller runs concurrently with these units. The pipelining of these operations is key to the high per-

formance of the TMS320C30. The ability of the DMA to move data within the processor's memory space results in an even greater utilization of the CPU with fewer interruptions of the pipeline which inevitably yields greater performance.

The pipeline control of the TMS320C30 allows for extremely high-speed execution rate by allowing an effective rate of one execution per cycle. It also manages pipeline conflicts in a way that makes them transparent to the user.

While the pipelining of the different phases of an instruction is key to the performance of the TMS320C30, the designers felt it essential to avoid pipelining the operation of the multiplier or ALU. By ruling out this additional level of pipelining it was possible to greatly improve the processor's useability.

Instructions

The TMS320C30 instruction set is exceptionally well suited to digital signal processing and other numerically intensive applications. The TMS320C30 also possesses a full complement of general-purpose instructions. The instruction set is organized into the following groups:

- load and store instructions;
- two-operand arithmetic instructions;
- two-operand logical instructions;
- three-operand arithmetic instructions;
- three-operand logic instructions;
- parallel operation instructions;
- arithmetic/logical instruction with store instructions;
- program control instructions;
- interlocked operations instructions.

The load and store instructions perform the movement of a single word to and from the registers and memory. Included is the ability to load a register conditionally. This operation is particularly useful for locating the maximum and minimum of a set of data.

The two-operand arithmetic and logical instructions consist of a complete set of arithmetic instructions. They have two operands; src and dst for source and destination, respectively. The src operand may come from memory, a register, or be part of the instruction word. The dst operand is always a register. This portion of the instruction set includes floating-point integer and logical operations, support of multiprecision arithmetic, and 32-bit arithmetic and logical shifts.

The three-operand arithmetic and logical instructions are a subset of the two-operand arithmetic and logical instructions. They have three operands: two src operands and a dst operand. The src operands may come from memory or a register. The dst operand is always a register. These instructions allow for the reading of two operands from memory and/or the CPU register file in a single cycle.

The parallel operation instructions allow for a high degree of parallelism. They support very flexible, parallel floating-point and integer multiplies and adds. They also include the ability to load two registers in parallel.

The arithmetic/logical and store instructions support a high degree of parallelism, thus complementing the parallel operation instructions. They allow for the performance of an arithmetic or logical instruction between a register and an operand read from memory, in parallel with the storing of a register to memory. They also provide for extremely rapid operations on blocks of memory.

The program control instructions consist of all those operations that affect the program flow. This section of the instruction set includes a set of flexible and powerful constructs that allow for software control of the program flow. These fall into two main types: repeat modes and branching.

For many algorithms, there is an inner kernel of code where most of the execution time is spent. The repeat modes of the TMS320C30 allow for the implementation of zero overhead looping. Using the repeat modes allows these time-critical sections of code to be executed in the shortest possible time. The instructions supporting the repeat modes are RPTB (repeat a block of code) and RPTS (repeat a single instruction). Through the use of the dedicated stack-pointer, block repeats (RPTBs) may be nested.

The branching capabilities of the TMS320C30 include two main subsets: standard and delayed branches. Standard branches, as in any pipelined machine that comprehends them, empty the pipeline to guarantee correct management of the program counter. This results in a branch requiring, in the case of the TMS320C30, four cycles to execute. Included in this subset are calls and returns. A standard branch (BR) is illustrated below.

```
        BR    THREE  ; standard branch.
        MPYF         ; not executed.
        ADDF         ; not executed.
        SUBF         ; not executed.
        AND          ; not executed.
        .
        .
        .
THREE   MPYF         ; fetched 3 cycles after BR
                       is fetched.
        .
        .
        .
```

Delayed branches do not empty the pipe, but rather, guarantee that the next three instructions will be fetched before the program counter is modified by the branch. The result is a branch that only requires a single cycle. Every delayed branch has a standard branch counterpart. A delayed branch (BRD) is illustrated below.

```
        BRD   THREE  ; delayed branch.
        MPYF         ; executed.
        ADDF         ; executed.
        SUBF         ; executed.
        AND          ; not executed.
        .
        .
        .
THREE   MPYF         ; fetched after SUBF fetched.
        .
        .
        .
```

The combination of the repeat modes, standard branches, and delayed branches provides the user with a set of programming constructs which are well suited to a wide range of performance requirements.

The program control instructions also include conditional calls and returns. The decrement and branch conditionally instruction allows for efficient loop control by combining the comparison of a loop counter to zero with

the check of condition flags, i.e., floating-point overflow. The condition codes available include unsigned and signed comparisons, comparisons to zero, and comparisons based upon the status of individual condition flags. These conditions may be used with any of the conditional instructions.

The interlocked operations instructions support multiprocessor communication. Through the use of external signals, these instructions allow for powerful synchronization mechanisms, such as semaphores, to be implemented. The interlocked operations use the two external flag pins, XF0 and XF1. XF0 signals an interlocked-operation request and XF1 acts as an acknowledge signal for the requested interlocked operation. The interlocked operations include interlocked loads and stores. When an interlocked operation is performed the external request and acknowledge signals can be used to arbitrate between multiple processors sharing memory, semaphores, or counters.

Development and Support Tools

Digital signal processors are essentially application-specific microprocessors (or microcomputers). Like any other microprocessor, no matter how impressive the performance of the processor or the ease of interfacing, without good development tools and technical support, it is very difficult to design it into the system. In developing an application, problems are encountered and questions are asked. Oftentimes the tools and vendor support provided to the designer are the difference between the success and failure of the project.

The TMS320 family has a wide range of development tools available [25]. These tools range from very inexpensive evaluation modules for application evaluation and benchmarking purposes, assembler/linkers, and software simulators, to full-capability hardware emulators. A brief summary of these support tools is provided in the succeeding subsections.

Software Tools

Assembler/linkers and software simulators are available on PC and VAX for users to develop and debug TMS320 DSP algorithms. Their features are described as follows:

Assembler/Linker: The Macro Assembler translates assembly language source code into executable object code. The Linker permits a program to be designed and implemented in separate modules that will later be linked together to form the complete program.

Simulator: The Simulator simulates operations of the device in software to allow program verification and debug. The simulator uses the object code produced by the Macro Assembler/Linker.

C Complier: The C Compiler is a full implementation of the standard Kernighan and Ritchie C as defined in *The C Programming Language* [28]. The compiler supports the insertion of assembly language code into the C source code. The user may also write functions in assembly language, and then call these functions from the C source. Similarly, C functions may be called from assembly language. Variables defined in the C source may be accessed in assembly language modules and *vice versa*. The result is a complier that allows the user to tailor the amount of high-level programming versus the amount of assembly language according to his application. The C compiler is supported on the TMS320C25 and the TMS320C30.

Hardware Tools

Evaluation modules and emulation tools are available for in-circuit emulation and hardware program debugging for developing and testing DSP algorithms in a real product environment.

Evaluation Module (EVM): The EVM is a stand-alone single-board module that contains all of the tools necessary to evaluate the device as well as provide basic in-circuit emulation. The EVM contains a debug monitor, editor, assembler, reverse assembler, and software communications to a host computer or a line printer.

SoftWare Development System (SWDS): The SoftWare Development System is a PC plug-in card with similar functionality of the EVM.

Emulator (XDS): The eXtended Development System provides full-speed in-circuit emulation with real-time hardware breakpoint/trace and program execution capability from target memory. By setting breakpoints based on internal conditions or external events, execution of the program can be suspended and the XDS placed into the debug mode. In the debug mode, all registers and memory locations can be inspected and modified. Full-trace capabilities at full speed and a reverse assembler that translates machine code back into assembly instructions are included. The XDS system is designed to interface with either a terminal or a host computer. In addition to the above design tools, other development support is available [25]:

Applications

The TMS320 is designed for real-time DSP and other computation-intensive applications [4]. In these applications, the TMS320 provides an excellent means for executing signal processing algorithms such as fast Fourier transforms (FFTs), digital filters, frequency synthesis, correlation, and convolution. The TMS320 also provides for more general-purpose functions via bit-manipulation instructions, block data move capabilities, large program and data memory address spaces, and flexible memory mapping.

To introduce applications performed by the TMS320, digital filters will be used as examples. The remaining portion of this section will briefly cover applications, and conclude by showing some benchmarks.

Digital Filtering

As discussed several times in this paper, the FIR filter is simply the sum of products in a sampled data system. This was shown in (1). A simple implementation of the FIR filter uses the MACD instruction (multiply/accumulate and data move) for each filter tap, with the RPT/RPTK instruction repeating the MACD for each filter tap. As we saw earlier, a 256-tap FIR filter can be implemented by using the following two instructions:

```
RPTK   255
MACD   *-,COEFFP
```

In this example, the coefficients may be stored anywhere in program memory (reconfigurable on-chip RAM, on-chip ROM, or external memories). When the coefficients are

stored in on-chip ROM or externally, the entire on-chip data RAM may be used to store the sample sequence. This allows filters of up to 512 taps to be implemented. Execution of the filter will be at full speed or 100 ns per tap as long as the memory supports full-speed execution (either on-chip RAM or high-speed external RAM).

Up to this point, it has been assumed that the filter coefficients are fixed from sample to sample. If the coefficients are adapted or updated with time, such as in adaptive filters for echo cancelation [4], [20], then the DSP algorithm requires a greater computational capacity from the processor. The requirement to adapt each of the coefficients, usually with each sample, is accomplished by three instructions (MPYA or MPYS, ZALR, and SACH) on the TMS320C25 [16]. A means of adapting the coefficients is the least-mean-square (LMS) algorithm given by the following equation:

$$b_k(i + 1) = b_k(i) + 2B[e(i) * x(i - k)]$$

where $b_k(i + 1)$ is the weighting coefficient for the next sample period, $b_k(i)$ is the weighting coefficient for the present sample period, B is the gain factor or adaptation step size, $e(i)$ is the error function, and $x(i - k)$ is the input of the filter.

In an adaptive filter, it is important to update the coefficients $b_k(i)$ in order to minimize the error function $e(i)$, which is the difference between the output of the filter and a reference signal. Quantization errors are critical to the performance of the filter when updating the coefficients and can be minimized if the result is obtained by rounding rather than truncating. For each coefficient in the filter at a given point in time, the factor $2*B*e(i)$ is a constant. This factor can then be computed once and stored in the T register for each of the updates. Thus the computational requirement has become one multiply/accumulate plus rounding. Without the new instructions, the adaptation of each coefficient is five instructions corresponding to five clock cycles. This is shown in the following instruction sequence:

```
LRLK   AR2,COEFFD   ; LOAD ADDRESS OF
                      COEFFICIENTS.
LRLK   AR3,LASTAP   ; LOAD ADDRESS OF DATA
                      SAMPLES.
LARP   AR2
LT     ERRF         ; errf = 2*B*e(i)
  .
  .
  .
ZALH   *,AR3        ; ACC = bk(i)*2**16
ADD    ONE, 15      ; ACC = bk(i)*2**16 + 2**15
MPY    *-,AR2
APAC                ; ACC = bk(i)*2**16
                            + errf*x(i-k) + 2**15
SACH   *+           ; SAVE bk(i+1).
  .
  .
  .
```

When the MPYA and ZALR instructions are used, the adaptation reduces to three instructions corresponding to three clock cycles, as shown in the following instruction sequence. Note that the processing order has been slightly changed to incorporate the use of the MPYA instruction. This is due to the fact that the accumulation performed by the MPYA is the accumulation of the previous product.

```
LRLK   AR2,COEFFD   ; LOAD ADDRESS OF
                      COEFFICIENTS.
LRLK   AR3,LASTAP   ; LOAD ADDRESS OF DATA
                      SAMPLES.
LARP   AR2
LT     ERRF         ; errf = 2*B*e(i)
  .
  .
  .
ZALR   *,AR3        ; ACC = bk(i)*2**16 + 2**15
MPYA   *-,AR2       ; ACC = bk(i)*2**16
                            + errf*x(i-k) + 2**15
*                   ; PREG = errf*x(i-k+1)
SACH   *+           ; SAVE bk(i+1).
  .
  .
  .
```

The adaptive filter coefficient update can further be simplified using the TMS320C30 [27] as shown below. The first instruction defines the number of times to repeat the kernel. The second instruction is the repeat-block instruction (RPTB). The RPTB instruction allows the iterations of the kernel to be performed with zero overhead looping. The kernel assumes that the error term is stored in register R0. It is important to note that all of the calculations are performed in floating-point arithmetic. The MPYF3 is a three-operand floating-point multiply of the input sample $x(i - k)$, which is stored in memory by the error term errf. The next step is a three-operand floating-point add (ADDF3) of the change in the filter tap to the filter tap in parallel with the store (STF) of the previously updated filter tap. That is, the store (STF) is to be performed in parallel with ADDF3. Thus the number of cyles for a floating-point adaptation is only two.

```
        LDI     N,RC            ; load length N in-
                                  to block repeat
                                  counter
        RPTB    adapt           ; repeat the adap-
                                  tation loop N+1
                                  times
        MPYF3   *++AR0(1),R0,R1 ; errf * x(i-k) → R1
adapt:
        ADDF3   *+AR1(1),R1,R2  ; b(k,i) + errf * x(i-k)
                                  → R2
||      STF     R2,*AR1++(1)    ; R2 → b(k-1,i)
```

Since we have discussed the application of digital filtering, we can now describe several applications in the areas of telecommunications, graphics/image processing, high-speed control, instrumentation, and numeric processing, and then conclude this section with several benchmarks. If more detail is needed on any of these applications, the reader is referred to [4].

Telecommunications Applications

Many aspects of the telecommunications network can take advantage of the TMS320. As telecommunications evolves more toward an all-digital network, DSP will become even more utilized [23]. Several typical uses of the TMS320 are discussed.

Echo Canceler: In echo cancellation [4], [20], an adaptive FIR filter performs the modeling routine and signal modifications to adaptively cancel the echo caused by the impedance mismatches in the telephone transmission lines.

For this application, a large on-chip RAM of 544 words and on-chip ROM of 4K words on the TMS320C25 provides for a 256-tap adaptive filter (32-ms echo cancellation) to be executed in a single chip without external data or program memory.

High-Speed Modems: The TMS320 can perform numerous functions such a modulation/demodulation, adaptive equalization, and echo cancellation [21], [22]. For lower speed modems, such as Bell 212A and V.22 bis modems, the TMS320C17 provides the most cost-effective single-chip solution to these applications. For higher speed modems, such as the V.32, requiring more processing power and multiprocessing capabilities, the TMS320C25 and TMS320C30 are the designer's choice.

Voice Coding: Voice-coding techniques [3], [4], such as full-duplex 32-kbit/s ADPCM (CCITT G.721), CVSD, 16-kbit/s subband coders, and LPC, are frequently used in voice transmission and storage. Arithmetic speed, normalization, and the bit-manipulation capability of the TMS320 provide for implementation of these functions, usually in a single chip. For example, the TMS320C17 can be used as a single-chip ADPCM [4], subband [4], or LPC [4] coder. An application of voice coding is an ADPCM transcoder implemented in half-duplex on a single TMS320C17 or full-duplex on a TMS320C25 for telecommunication multiplexing applications. Another example is a secure-voice communication system, requiring voice coding, as well as data encryption and transmission over a public-switched network via a modem; the TMS320C25 offers an ideal solution.

Graphics/Image Processing Applications

In graphics and image processing applications [4], the ability to interface with a host processor is important. Both the TMS320C30 and the TMS320C25 multiprocessor interface enable them to be used in a variety of host/coprocessor configurations [4]. Graphics and image processing applications can use the large directly addressable external data space and global memory capability to allow graphical images in memory to be shared with a host processor, thus minimizing unnecessary data transfers. The indexed indirect addressing modes allow matrices to be processed row-by-row when performing matrix multiplication for three-dimensional image rotations, translations, and scaling.

The TMS320C30 has a number of features that support graphics and image processing extremely well. The floating-point capabilities allow for extremely precise computation of perspective transformations. They also support more sophisticated algorithms such as shading and hidden line removal, operations which are computationally intensive.

The large address space allows for straightforward addressing of large images or displays. The flexible addressing registers, coupled with the integer multiply, support powerful addressing of multiple-dimensional arrays. Vector-oriented instructions allow the user to efficiently manipulate large blocks of memory. Finally, the on-chip DMA controller allows the user to easily overlap the processing of data with its I/O.

High-Speed Control

High-speed control applications [4], [24] use the TMS320C17 and TMS320C25 general-purpose features for bit-test and logical operations, timing synchronization, and high data-transfer rate (ten million 16-bit words per second). Both devices can be used in closed-loop systems for control signal conditioning, filtering, high-speed computing, and multichannel multiplexing capabilities. The following demonstrates two typical control applications:

Disk Control: Digital filtering in a closed-loop actuation mechanism positions the read/write heads over the disk surface. Supplemented with many general-purpose features, the TMS320 can replace costly bit-slice/custom/analog solutions to perform such tasks as compensation, filtering, fine/coarse tuning, and other signal conditioning algorithms.

Robotics: Digital signal processing and bit-manipulation power, coupled with host interface, allow the TMS320C25 to be useful in robotics control [24]. The TMS320C25 can replace both the digital controllers and analog signal processing hardware for communication to a central host processor and for the performance of numerically intensive control functions.

Instrumentation

Instrumentation, such as spectrum analyzers and various high-speed/high-precision instruments, often requires a large data memory space and the high performance of a digital signal processor. The TMS320C25 and TMS320C30 are capable of performing very long-length FFTs and generating precision functions with minimal external hardware.

Numeric Processing

Numeric and array processing applications benefit from TMS320 performance. High throughput resulting from features, such as a fast cycle time and an on-chip hardware multiplier, combined with multiprocessing capabilities and data memory expansion, provide for a low-cost, easy-to-use replacement for a typical bit-slice solution. The TMS320C30's floating-point precision, high throughput, and interface flexibility are excellent for this application.

TMS320 Benchmarks

To complete the discussion on the applications that the TMS320 can perform, we will provide some benchmarks. The TMS320 has demonstrated impressive benchmarks in performing some of the common DSP routines and system applications. Table 5 shows typical TMS320 benchmarks [4].

Table 5 TMS320 Family Benchmarks

DSP Routines/Applications	First Generation	Second Generation	Third Generation
FIR filter tap	400 ns	100 ns	60 ns
256-tap FIR sample rate	9.25 kHz	37 kHz	>60 kHz
LMS adaptive FIR filter tap	700 ns	400 ns	180 ns
256-tap adaptive FIR filter sample rate	5.4 kHz	9.5 kHz	>20 kHz
Bi-quad filter element (five multiplies)	2 μs	1 μs	360 ns
Echo canceler (single chip)	8 ms	32 ms	>64 ms

Summary

This paper has discussed characteristics of digital signal processing and how these characteristics have influenced the architectural design of the Texas Instruments TMS320 family of digital signal processors. Three generations of the

TMS320 family were covered, and their support tools necessary to develop end-applications were briefly reviewed. The paper concluded with an overview of digital signal processing applications using these devices.

References

[1] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[2] A. V. Oppenheim, Ed., *Applications of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[3] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[4] K. Lin, Ed., *Digital Signal Processing Applications with the TMS320 Family*. Englewood Cliffs, NJ: Prentice-Hall, 1987

[5] A. V. Oppenhiem and R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[6] C. Burrus and T. Parks, *DFT/FFT and Convolution Algorithms*. New York, NY: Wiley, 1985.

[7] T. Parks and C. Burrus, *Digital Filter Design*. New York, NY: Wiley, 1987.

[8] J. Treichler, C. Johnson, and M. Larimore, *A Practical Guide to Adaptive Filter Design*. New York, NY: Wiley, 1987.

[9] P. Papamichalis, *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[10] R. Morris, *Digital Signal Processing Software*. Ottawa, Ont., Canada: DSPS Inc., 1983.

[11] K. McDonough, E. Caudel, S. Magar, and A. Leigh, "Microcomputer with 32-bit arithmetic does high-precision number crunching," *Electronics*, pp. 105–110, Feb. 24, 1982.

[12] S. Magar, E. Caudel, and A. Leigh, "A Microcomputer with digital signal processing capability," in *1982 Int. Solid State Conf. Dig. Tech. Pap.*, pp. 32–33, 284, 285.

[13] *First Generation TMS320 User's Guide*. Houston, TX: Texas Instruments Inc., 1987.

[14] *TMS320 First-Generation Digital Signal Processors Data Sheet*. Houston, TX: Texas Instruments Inc., 1987.

[15] *TMS32020 User's Guide. Houston, TX: Texas Instruments Inc., 1985.*

[16] *TMS320C25 User's Guide*. Houston, TX: Texas Instruments Inc., 1986.

[17] *TMS32011 User's Guide*. Houston, TX: Texas Instruments Inc., 1985.

[18] H. Cragon, "The elements of single-chip microcomputer architecture," *Comput. Mag.*, vol. 13, no. 10, pp. 27–41, Oct. 1980.

[19] S. Rosen, "Electronic computers: A historical survey," *Comput. Surv.*, vol. 1, no. 1, Mar. 1969.

[20] M. Honig and D. Messerschmitt, *Adaptive Filters*. Dordrecht, The Netherlands: Kluwer, 1984.

[21] R. Lucky *et al.*, *Principles of Data Communication*. New York, NY: McGraw-Hill, 1965.

[22] P. Van Gerwen *et al.*, "Microprocessor implementation of high speed data modems," *IEEE Trans. Commun.*, vol. COM-25, pp. 238–249, 1977.

[23] M. Bellanger, "New applications of digital signal processing in communications," *IEEE ASSP Mag.*, pp. 6–11, July 1986.

[24] Y. Wang, M. Andrews, S. Butner, and G. Beni, "Robot-controller system," in *Proc. Symp. on Incremental Motion Control Systems and Devices*, pp. 17–26, June 1986.

[25] *TMS320 Family Development Support Reference Guide*. Houston, TX: Texas Instruments Inc., 1986.

[26] R. Simar, T. Leigh, P. Koeppen, J. Leach, J. Potts, and D. Blalock, "A 40 MFLOPS digital signal processor: The first supercomputer on a chip," in *Proc. IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing*, Apr. 1987.

[27] *TMS320C30 User's Guide*. Houston, TX: Texas Instruments Inc., 1987.

[28] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

The TMS320C30 Floating-Point Digital Signal Processor

Panos Papamichalis
Ray Simar, Jr.

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

Reprinted from
IEEE MICRO MAGAZINE
Vol. 8, No. 6, December 1988

The TMS320C30 Floating-Point Digital Signal Processor

Digital signal processors have significantly impacted the way we bring real-time implementations of sophisticated DSP algorithms to life. What was once only a laboratory curiosity that required large computers or specialized, bulky, and expensive hardware is now incorporated into low-cost consumer products. The rapid advancement of programmable DSPs since their commercial introduction in the early 1980s lets us satisfy the needs of very demanding applications. Implementation of basic DSP functions, such as digital filters and fast Fourier transforms, has been integrated into advanced system solutions involving speech algorithms, image processing, and control applications. The variety of the applications increases every day as researchers, developers, and entrepreneurs discover new areas in which DSP devices can be used. At the same time, the design of new devices incorporates features that make such implementations easier.

The Texas Instruments family of TMS320 DSPs[1] evolved with the expanding needs of the DSP applications and currently encompasses over 17 devices. The TMS320 family consists of three generations of devices. The first two generations are 16-bit, fixed-point-arithmetic devices while the third one, represented by the TMS320C30 and explained in detail here, is a 32-bit, floating-point device. Architecturally, the TMS320 family, like most DSP devices, relies on multiple Harvard buses. In the first two generations, we expanded the basic Harvard architecture to permit communication between the program and data spaces. In the third generation, we unified the two spaces to form an organization that encompasses the advantages of both the Harvard and the von Neumann architectures.

Overview of the TMS320C30

The 320C30 is a fast processor (16.7 million instructions per second for an instruction cycle time of 60 nanoseconds) with a large memory space (16 million 32-bit words) and floating-point-arithmetic capabilities. This last feature is a major trend in new DSP devices, which was developed to answer the need for quicker, more accurate solutions to numerical problems. DSP algorithms, being very intensive numerically, cause a designer to worry about overflows and the accuracy of results. The introduction of floating-point capabilities eliminates these difficulties.

Panos Papamichalis
Ray Simar, Jr.

Texas Instruments

In the 320C30, a chip design with 1-μm geometries produces instruction cycle times lower than those achieved with the fixed-point devices of the first two generations. In addition, the design produces a controlled increase in die size that results more from the extended on-chip memory spaces than from the floating-point capabilities.

The pipelined architecture of the 320C30 permits the higher throughput achieved by the device, as we explain later. Yet, programmers do not have to worry about the pipeline when writing the code. We can describe the design philosophy of the 320C30 (as well as all the other devices in the TMS320 family) as an "interlocked" or "hidden-pipeline" approach. When writing the program, programmers can assume that the result of any instruction will be available for the next instruction. Most of the instructions execute in one machine cycle. If a conflict arises between executing an instruction in one cycle and having the data available for the next instruction, the device automatically inserts the necessary delay to eliminate the conflict. Since this delay could result in loss of performance, we provide development tools that identify where such conflicts occur. With this data, programmers can rearrange and optimize code.

Many applications, such as graphics and image processing, are difficult to implement on the earlier DSP devices because they require a large memory space. To satisfy this need, the 320C30 provides a total memory space of 16 million 32-bit words, memory several orders of magnitude larger than the fixed-point devices. Furthermore, it contains significantly increased on-chip memory: six thousand 32-bit words of RAM and ROM. The desire to have a device capable of offering system-level solutions to the implemented algorithms guided the design decision to increase on-chip memory. In other words, the 320C30 attempts to offer the capability of implementing an algorithm with as little peripheral circuitry as possible.

Along the same lines, the 320C30 contains a peripheral bus on which on-chip peripherals can be attached using a memory-mapped approach. Currently available peripherals include two serial ports, two timers, and a DMA controller. The modularity of the design permits easy change, addition, or deletion of peripherals to accommodate different needs. For instance, if a μ-law-to-linear format converter or a gate array is more important than one of the timers for certain applications, a user can make the change without impacting the core of the device.

As the power of the DSP devices increases, so does the sophistication of the algorithms that are implemented. The implication is that constructing and debugging an algorithm at the assembly-language level becomes a more and more tedious task. To address that problem, we provide the 320C30 development tools, which include a high-level-language compiler and a DSP operating system. The extended memory space, the software stack, and the large on-chip register file also facilitate such a development. We've already introduced a C compiler and announced an Ada compiler. We expect compiler availability to change significantly the way DSP algorithms are ported to DSP devices. With these tools, programmers can develop the algorithms on large computers, requiring at the most only selective optimization when they incorporate the algorithm on the 320C30.

Here, we describe the 320C30 architecture in detail, discussing both the internal organization of the device and the external interfaces. We also explain the pipeline structure, addressing software-related issues and constructs, and examine the development tools and support. Finally, we present examples of applications.

Architecture of the 320C30

Studying the architecture of the device helps in understanding how the different components contribute toward a high-throughput system. The interaction and the efficient use of the parts can contribute to very effective programming. Another very important aspect to consider is the system cost of the application. We designed the device to incorporate on-chip features that minimize the amount and the cost of external logic, thus leading to very compact and cost-effective solutions. These advantages become explicit when looking at the architecture in detail. The internal structure of the 320C30, as shown in Figure 1, consists of the

- on-chip memory and cache,
- CPU with register file,
- peripheral bus and peripherals, and
- interconnecting buses.

See Figure 2 for the die photograph. To interface with the external world, the 320C30 provides pins corresponding to

- two buses (primary and expansion),
- two serial ports and two timers,
- four external interrupt signals,
- two external flags, and
- hold and hold-acknowledge signals.

In addition, other pins exist for address and data strobs, power, and so on.

The overall architecture of the device is a Harvard type in the sense that internally and externally it has multiple buses to access program instructions, data, or perform DMA transfers. However, it also has a von Neumann flavor since the memory space is unified, and there is no separation of program and data spaces. As a result, the user can choose to locate programs and data at any desired location.

Some of the major features of the 320C30 are:

- a 60-ns cycle time that results in execution of over 16 million instructions per second (MIPS) and over 33 million floating-point operations per second (Mflops);
- 32-bit data buses and 24-bit address buses for a 16M-word overall memory space;
- dual-access, 4K × 32-bit on-chip ROM and 2K × 32-bit on-chip RAM;

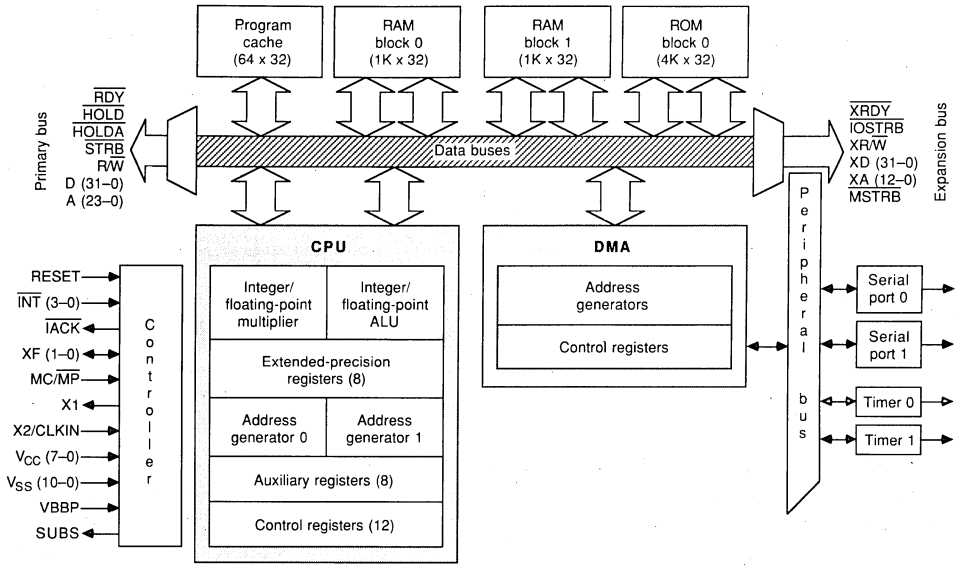


Figure 1. Block diagram of the TMS320C30 architecture.

- a 64 × 32-bit program cache;
- a 32-bit integer/40-bit floating-point multiplier and ALU;
- eight extended-precision registers, eight auxiliary registers, and 12 control and status registers;
- generally single-cycle instructions;
- integer, floating-point, and logical operations;
- two- and three-operand instructions;
- an on-chip DMA controller; and
- fabrication in 1-μm CMOS technology and packaging in a 180-pin package.

Memory organization. The 320C30 provides 4K 32-bit words of on-chip ROM, and 2K 32-bit words of on-chip RAM. The on-chip ROM is mapped into the first 4K of the overall memory map; it is accessed when the processor operates in the microcomputer mode. Location 0 of the memory map holds the reset vector, and adjacent locations hold other interrupt vectors. In microprocessor mode, the reset vector resides in external memory, and on-chip ROM is not accessed. The 2K on-chip RAM consists physically of two segments of 1K words each. These two segments of RAM are mapped into adjacent sections of the memory. Figure 3 on the next page shows the arrangement of the on-chip memory, as well as the cache, buses, and two external interfaces/buses, which we examine later.

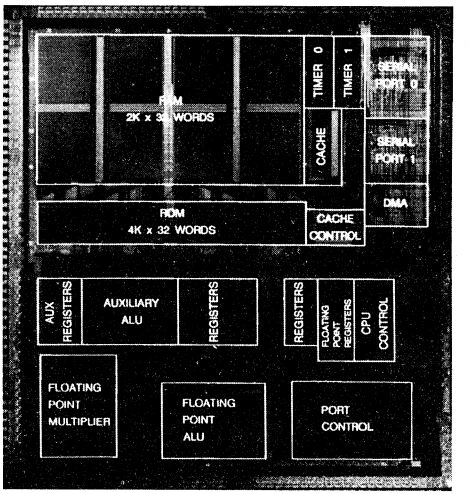


Figure 2. Die photograph of the 320C30.

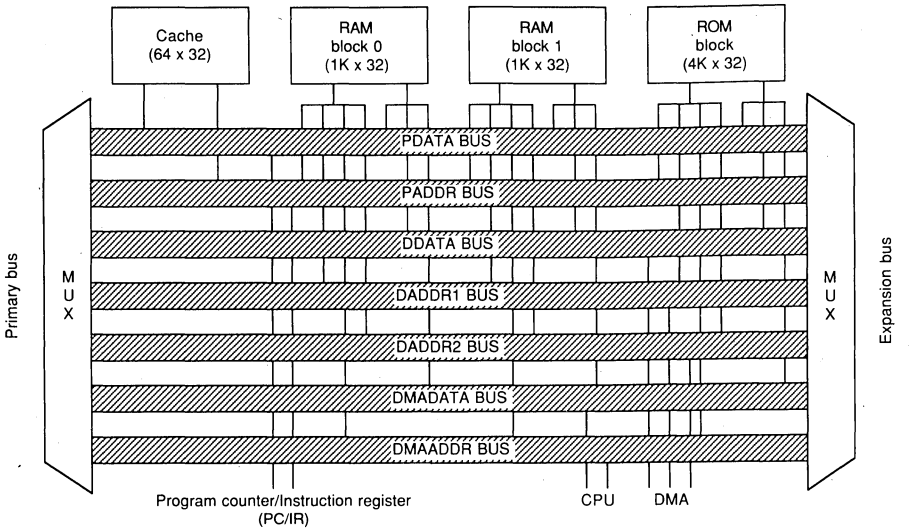


Figure 3. On-chip memory, cache, and buses.

The internal memory (both ROM and RAM) supports two accesses for reads and/or writes in one cycle. This key feature permits high throughput and ease of programming, since it makes possible three-operand instructions with two operands residing in the memory. Notice that, to support this feature, we include two buses dedicated to data addresses (DADDR1, DADDR2) and one bus to carry the data (DDATA). There are also separate program buses, PDATA and PADDR.

The address buses are 24 bits wide, indicating that the overall memory space is 16 million (32-bit) words. We believe this large space will facilitate implementation of algorithms in image processing applications that often require large amounts of memory. The unified memory space offers flexibility in placing program and data. But it also permits optimal use of the memory space as a trade-off between program and data.

An important addition to the architecture is the 64-word instruction cache. To reduce the overall system cost of applications, system designers often use slower (and cheaper) external memories, a tactic that could slow down the processor and degrade the performance. The instruction cache addresses this problem by storing on-chip instructions that have been fetched previously. Its main advantage becomes obvious when loops must be executed. In this case, the first time the instructions are fetched, they are also stored in the cache. Any subsequent execution of the loop does not access external memory but fetches instructions from the cache, resulting in higher speed and making the external buses available for data transfers.

The cache is segmented into two sections of 32 words each that are transparent to users. A user can, however, control the operation of the cache by manipulating three control bits that are contained in the status register of the CPU. Each control bit is dedicated to a specific operation: cache enable/disable, cache freeze, and cache clear. When a cache miss occurs, that is, when the next instruction is not included in the cache, the instruction is brought in and also stored in the cache. The two cache sections are updated on a least recently used basis.

CPU organization. The CPU consists of the ALU (arithmetic logic unit), the hardware multiplier, and the register file. These units are shown in Figure 4.

The register file consists of

- eight 40-bit-wide, extended-precision registers R0 through R7,
- eight 32-bit auxiliary registers AR0 through AR7, and
- twelve 32-bit control registers.

The extended-precision registers function as accumulators and can handle both floating-point and integer numbers. When they are used for floating-point numbers, the top eight bits represent the exponent and the bottom 32 bits the mantissa of the number. In their integer format, registers R0 through R7 use only their bottom 32 bits, keeping the top 8 bits unchanged in any integer or logical operation.

The eight auxiliary registers AR0 through AR7 can function as memory pointers in indirect addressing, as loop counters, or as general-purpose registers in integer arithmetic or logical operations. Associated with these registers are two auxiliary register arithmetic units (ARAU) that generate two memory addresses in parallel for the instructions that need them. The flexibility of indirect addressing increases even further when two index registers are used in conjunction with the auxiliary registers, as we discuss later.

The register file contains 12 control registers designated for specific functions. If the control registers are not used for these functions, they can be treated as general-purpose registers in integer arithmetic and logical operations. Examples of such control registers are the

- status register,
- index registers,
- stack pointer,
- interrupt mask and interrupt flag registers, and
- repeat-block registers.

In particular, the stack-pointer register points to the software stack. The user has the flexibility of designating where the stack resides, and even of changing its location during the program execution. This feature also makes the stack of essentially unlimited depth and permits its usage not only for storing the program counter during subroutine calls but also for passing arguments to subroutines. Such an arrangement is particularly convenient in the development of compilers, and we have used it extensively in the 320C30's optimizing C compiler.

The ALU performs floating-point, integer, and logical operations. The ALU always stores the result in the register file, but the input can come either from the register file or from memory, or it can be an immediate value.

In the case of floating-point arithmetic, the input to the ALU can originate from either a 40-bit extended-precision register or a 32-bit memory datum. Registers R0 through R7 store the 40-bit-word result. On the other hand, in integer arithmetic, both input and output are 32-bit numbers, and the output can move to either the lower 32 bits of the R0 through R7 registers or to any other register in the register file.

The single-cycle hardware multiplier has been an integral part of DSPs because any real-time application relies on the fast execution of multiplies. Following the same distinction as in the previous paragraph on the ALU, the multiplier performs both floating-point and integer multiplications. The 32-bit inputs to a floating-point multiplication yield a 40-bit-wide result for storage in one of the extended-precision registers.

In both the ALU and the multiplier the results of the operations are automatically normalized, thus handling any overflows of the mantissa. If there is an exponent overflow, the result is saturated in the direction of overflow and the overflow flag is set. Underflows are handled by setting the result to zero and setting an underflow flag.

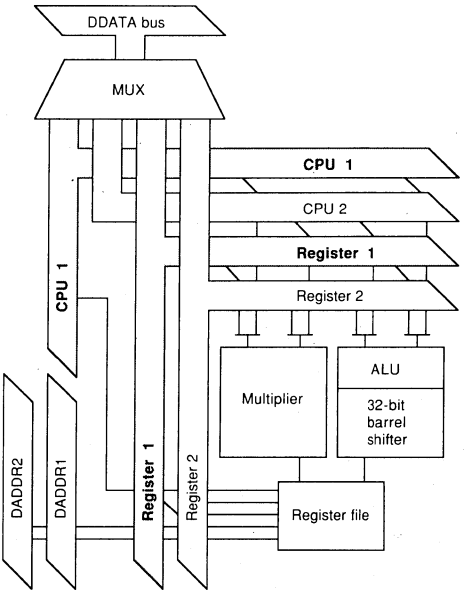


Figure 4. The 320C30 central processing unit.

Buses and peripherals. Figure 3 shows that multiple on-chip buses handle program, data, and DMA operations in parallel. The device contains separate address and data buses for these three operations, with the data having two address buses to accommodate the access of multiple operands from the memory in one cycle. Also, separate buses lead to the register file. The rule to remember is that, in one cycle, up to two data memory accesses are permitted for any on-chip memory block. This multiplicity of buses eliminates bottlenecks. The user can maximize the throughput of the device by a judicious combination of the on-chip memory with the two external buses (the primary bus and the expansion bus).

The primary bus contains a 24-bit address bus and a 32-bit data bus. Its true space, though, is 16M words minus the on-chip memory and the expansion bus. The primary bus can be placed in high impedance when the device is put on hold. To facilitate its interfacing with slow memories, the 320C30 offers programmable wait states (up to seven) as well as an external ready signal.

The expansion bus contains a 13-bit address bus and a 32-bit data bus. It has two strobes, one for memory and one for I/O accesses. In other words, the memory space of the

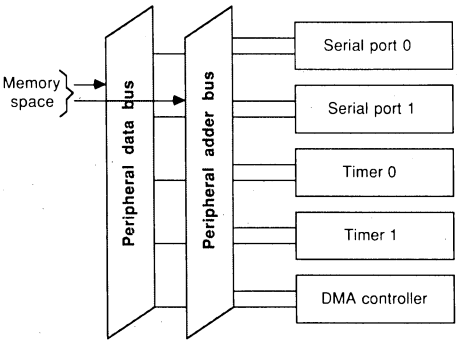


Figure 5. Peripheral bus and peripherals.

expansion bus is two segments of 8K words each, one segment mapped as regular memory and the other one mapped as I/O. Like the primary bus, the expansion bus has up to seven software-programmable wait states.

A major innovation in the 320C30—to support system-level solutions and to help in adapting the device to changing needs—is the peripheral bus shown in Figures 1 and 5. The peripheral bus supplies a way of expanding or varying the interface with the outside world without changing the core of the device. All of the peripherals attached to this bus are mapped to memory, and they can be replaced by others with a minimal effort if certain applications have different demands.

Currently, we have implemented a DMA controller, two serial ports, and two timers as peripherals. The DMA controller performs reads from and writes to any location in the 320C30 memory map without interfering with the operation of the CPU. The DMA controller contains its own address generators, source and destination address registers, and transfer counter. The two modular and totally independent serial ports are identical with a complementary set of control registers. Each serial port can be configured to transfer 8, 16, 24, or 32 bits of data per word, with each port clock originating either internally or externally. The pins of the serial ports are configurable as general-purpose I/O pins, while the serial ports can also be configured and used as timers.

The two 320C30 timer modules function as general-purpose timer/event counters; each have two signaling modes and internal or external clocking. Available to each timer is an I/O pin for use as an input clock to the timer, as an output signal driven by the timer, or as a general-purpose pin.

Software

The software features of a programmable DSP are probably the most important features because they determine the effectiveness of the implementation. Typically, the user first develops an application on a large computer using a high-level language and, once it is working satisfactorily, ports it to a DSP device. The software features of the 320C30 that we discuss include the integer and floating-point number representations, addressing modes, pipeline effects, and different types of instructions and constructs.

Integer and floating-point formats. A 32-bit, twos-complement notation represents the integers. In addition to this single-precision format, we have a short format, consisting of 16-bit, twos-complement numbers used only for immediate operands. Every instruction of the 320C30 consists of one 32-bit word.

We use three formats for floating-point numbers: short, single precision, and extended precision. The single-precision, 32-bit-wide format assigns 24 bits to the mantissa and 8 bits to the exponent. The exponent occupies the 8 most significant bits, and it is represented in twos-complement notation, taking values between −128 and 127. The exponent value −128 is the result reserved to represent zero.

The mantissa, placed at the 24 least significant bits of a 32-bit number, is normalized to a number with an absolute value between 1.0 and 2.0. Since the mantissa is represented in a normalized, twos-complement notation, the leftmost bit, which corresponds to the sign, and its adjacent bit will always be the complement of each other. As a result, only the sign bit is represented, with the most significant bit suppressed. In other words, the mantissa contains 24 significant bits plus the sign bit, with the most significant bit implied.

Addressing modes. The 320C30 supports several addressing modes that allow the user to access data from memory, registers, and the instruction word. The basic addressing modes are

- register,
- direct,
- indirect,
- short immediate,
- long immediate, and
- PC relative.

In register mode the operand is placed into a CPU register that is explicitly specified in an instruction. In direct mode the data memory address is formed by preceding the 16 least significant bits of the instruction word with the 8 least significant bits of the data page pointer. To keep all instructions one word long, we store only the 16 least significant bits from the address in the instruction word; the rest become the data page pointer. This restriction implies that in direct addressing the memory space is segmented into 256 pages of 64K words each.

Table 1.
Addressing modes of the 320C30.

Mode	Example	Operation	Description
Register	ADDF R0,R1		Operand in R0
Direct	ADDF @MEM, R1	Addr = MEM	Operand in MEM
Short immediate	ADDF 3.14,R1		Operand = 3.14
Long immediate	BR LABEL		Branch to LABEL
PC relative	BGE LABEL		Branch to LABEL
Indirect	ADDF *+AR0(di),R1	Addr = AR0 + di	Predisplacement add without modification
Indirect	ADDF *−AR0(di),R1	Addr = AR0 − di	Predisplacement subtract without modification
Indirect	ADDF *++AR0(di),R1	Addr = AR0 + di AR0 = AR0 + di	Predisplacement add and modify
Indirect	ADDF *−−AR0(di),R1	Addr = AR0 − di AR0 = AR0 − di	Predisplacement subtract and modify
Indirect	ADDF *AR0++(di),R1	Addr = AR0 AR0 = AR0 + di	Postdisplacement add and modify
Indirect	ADDF *AR0−−(di),R1	Addr = AR0 AR0 = AR0 − di	Postdisplacement subtract and modify
Indirect	ADDF *AR0++(di)%,R1	Addr = AR0 AR0 = circ(AR0 + di)	Postdisplacement add and circular modify
Indirect	ADDF *AR0−−(di)%,R1	Addr = AR0 AR0 = circ(AR0−di)	Postdisplacement subtract and circular modify
Indirect	ADDF *AR0++(IR0)B,R1	Addr = AR0 AR0 = B(AR0 + IR0)	Postindex (IR0) add and bit-reversed modify

di is an integer between 0 and 255 or one of the index registers IR0 and IR1.

Indirect addressing, the most versatile of all the modes, specifies the address of an operand in memory through the contents of an auxiliary register. As an option, the contents of the register can be modified by constant displacements or by the contents of the index registers. Table 1 lists all of the addressing modes, with particular emphasis on indirect addressing modes.

An instruction explicitly specifies the auxiliary register used for indirect addressing. The user can modify it by a constant displacement taking values 0 to 255 or by the contents of one of the two index registers IR0 or IR1. The modification can take place before or after accessing the memory. In the case of premodification, the user has the option to change the contents of the auxiliary register either permanently or temporarily. The notation used for such modifications is reminiscent of the C-language syntax.

Two special forms of indirect addressing that are particularly useful are bit-reversed and circular addressing. Bit-reversed addressing is used with the fast Fourier transform to compensate for the fact that normally ordered data at the input of the transform are scrambled at output (bit-reversed order). To avoid moving the data around to place them in the proper order, bit-reversed addressing accesses the data in scrambled order for any subsequent operation.

Circular addressing implements circular buffers. Such buffers are very convenient for use in digital-filtering operations. In circular addressing, BK, one of the control registers, specifies the size of the block. Then, when the user modifies the contents of an auxiliary register (pointing within that block) in a circular fashion, the final value is tested to determine if it is still within the block. If it is not, it is wrapped around using modulo arithmetic.

The short-immediate mode encodes immediate, 16-bit-long operands of arithmetic operations. The long-immediate mode encodes program control instructions (branch instructions) for which it is useful to have a 24-bit absolute address contained in the instruction word. Finally, the PC-relative addressing also applies to program control instructions and uses the difference from the present location of the PC counter rather than an absolute address. The last two

modes are transparent to the user. The user specifies the branching label wanted, and the assembler assigns the appropriate addressing mode.

Pipeline. To achieve the high throughput of the device, the 320C30 uses a four-phase pipeline with five major functional units operating in parallel. These five units are

- instruction fetching,
- instruction decoding and address generation,
- operand reads,
- instruction execution, and
- DMA transfer.

Figure 6 shows diagrammatically how the pipeline operates on successive instructions. When the pipeline is full, an instruction completes the execution phase every 60-ns machine cycle.

Occasionally conflicts may arise, as in the case of a loaded auxiliary register that needs to be used for indirect addressing in the next instruction. To handle such cases, we established a priority between the different units, giving DMA the lowest priority. Among the others, an Execute instruction has the highest and a Fetch instruction the lowest priority.

In programming the device, the user does not have to worry about the pipeline conflicts, which do not occur that often anyway. When a conflict does occur, the device automatically inserts the necessary extra cycle(s) to make the instructions behave as expected. In most cases, this arrangement will be sufficient for successful operation. For time-critical operations, though, it may be necessary to remove the extra cycles caused by pipeline conflicts. The user can make this correction by rearranging the instructions of the program. To do so, the user must determine how to identify the locations where insertions occur. For that purpose, the development tools (simulator, emulators) contain a tracing feature that can display the pipeline. In this trace, any conflicts are immediately identified, and then the user can take steps to correct the problem.

Instruction set features. The instruction set of the 320C30 supports both two- and three-operand instructions. In all arithmetic instructions (except Store), the destination is a register in the register file. The source operands can come from memory or from a register or, in the case of two-operand instructions, can be part of the instruction word.

A unique feature of the 320C30 is the set of instructions in which operations execute in parallel. This construct permits a high degree of concurrency and execution of any arithmetic or logical instruction in parallel with a Store instruction. It also supports parallel multiplies and adds, as well as parallel loading and storing of two registers. Parallel multiply and adds lead to the peak performance of 33 Mflops. Executing the Store instruction at the same time with another arithmetic operation essentially permits this kind of data movement without a penalty. As an example, the following instruction adds the contents of memory pointed to by AR1 (indicated by *AR1) to register R0 (treating them as floating-point numbers) and places the result in register R1. In parallel with that process, the original contents of R1 are stored in the memory location indicated by AR3.

```
        ADDF    *AR1,R0,R1
||      STF     R1,*AR3
```

When executing a branch instruction, the pipeline must be flushed since the path followed after the branch is data dependent. As a result, a regular branch instruction is more costly than other instructions, taking four cycles to complete. This overhead may be unacceptable in some time-critical applications. To alleviate this problem and to offer more flexibility to the programmer, the 320C30 contains a set of delayed branches that complement the set of standard branches. In a delayed branch, the three instructions following the branch instruction execute whether the branch is taken or not taken. As a result, the delayed branch ends up taking only one cycle to execute. The same approach can be used even when there are less than three such instructions, by adding NOPs (no operations). The branch will still take less than four cycles.

The greatest cost of branching occurs during the execution of loops. In looping, a counter is decremented and compared to zero at the end of the loop. If it is not zero, a branch is taken to the beginning of the loop. The 320C30 offers a special arrangement that implements loops with no

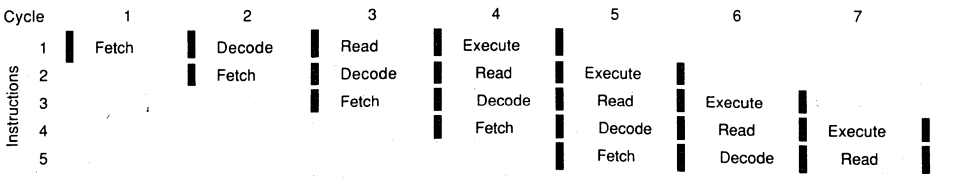


Figure 6. Pipeline of 320C30 instructions.

User-friendly development tools offer extra support: an optimizing C compiler and a DSP operating system.

overhead. The two instructions RPTB (repeat block) and RPTS (repeat single) realize this arrangement. The format of the RPTB instruction is:

```
RPTB        LABEL

(put instructions here)

LABEL  (last instruction)
```

Associated with the repeat-block construct are three of the 12 control registers in the register file. One register indicates the beginning of the block, the second indicates the end of the block, and the third acts as the repeat counter. The assembler automatically assigns values to the first two registers. They contain the address of the instruction immediately below RPTB, and the address of LABEL respectively. Users should initialize the repeat counter before entering the loop. In terms of execution time, this arrangement behaves as if the loop were implemented with straight-line code.

The instruction RPTS has the format

```
RPTS    count
```

and it repeats the following instruction "count" times. It differs from RPTB in that it

- applies to only one instruction;
- does not refetch the instruction for every execution, but keeps it in the instruction register thus freeing the buses for data transfers, and
- is not interruptible.

Table 2 on the next page is a sample of the instructions available on the 320C30. Although we included a rich set of instructions for both DSP and general-purpose processing, the perceived size of the instruction set is much smaller. The reason is that a symmetry exists between integer and floating-point instructions, between instructions with two or three operands, and between single and parallel instructions. For instance, addition is represented by ADDI, ADDF, or ADDC in the case of adding integers, floating-point numbers, or adding with a carry. The three-operand instructions have the same form, with a 3 appended at the end (ADDF3). All of the multiplier and ALU operations can be performed in parallel with a Store instruction, and such instructions take the form of the following example:

```
        ADDF3    *AR0,R1,R2
||      STF      R0,*AR1
```

Furthermore, two loads or two stores can execute in parallel, as is also the case with a multiply and an add or a multiply and a subtract. The design of the instruction set has been guided by a desire to ease programming efforts. The execution results of an instruction are always available for use in the instruction that follows.

Besides the regular arithmetic and logical instructions, the 320C30 includes instructions to handle the software stack, internal and external interrupts, and branches and subroutine calls. Conditional loads and calls make the programming more compact and efficient, while special instructions (called interlocked instructions) can be used in multiprocessor environments.

Development tools and support

The newer DSP devices offer increased processing power that permits the implementation of more complicated and demanding algorithms. However, as the complexity of the algorithm increases, the task of debugging the implementation becomes more difficult. The 320C30 addresses this problem by providing user-friendly development tools and offering extra support in the form of an optimizing C compiler and a DSP operating system.

The assembler translates assembly-language source files into machine-language object files. Source files can contain instructions, assembler directives, and macro directives. Assembler directives control various aspects of the assembly process such as the source-listing format, symbol definition, and method of placing the source code into sections. Macro directives permit a concise representation of groups of instructions that occur frequently.

The linker combines object files into one executable object module. As it creates the executable module, the linker performs relocation operations and resolves external references. The linker accepts relocatable COFF (Common Object File Format) object files, created by the assembler, as input. It can also accept archive library members and output modules created by a previous linker run. Linker directives allow the user to combine object-file sections, bind sections or symbols to specific addresses or within specific portions of 320C30 memory, and define or redefine global symbols. An associated archiver can create macro or object-file libraries.

The software simulator is a very important tool for debugging 320C30 programs. Its interface consists of a screen broken into windows that display the internal registers, the reverse-assembled program, and a versatile window where memory, breakpoints, and a wealth of other information can be displayed. The same interface (modified to accommodate some special features) is also used with the hardware emulator. The major features of the simulator include:

- Simulation of the entire 320C30 instruction set and the

Table 2.
Instructions for the 320C30.

Instruction		Description	Instruction		Description
Load and store instructions					
LDE		Load floating-point exponent	POP		Pop integer from stack
LDF		Load floating-point value	POPF		Pop floating-point value from stack
LDF*cond*		Load floating-point value conditionally	PUSH		Push integer on stack
LDI		Load integer	PUSHF		Push floating-point value on stack
LDI*cond*		Load integer conditionally	STF		Store floating-point value
LDM		Load floating-point mantissa	STI		Store integer
Two-operand instructions					
ABSF		Absolute value of a floating-point number	NORM		Normalize floating-point value
ABSI		Absolute value of an integer	NOT		Bitwise logical-complement
ADDC	†	Add integers with carry	OR	†	Bitwise logical-OR
ADDF	†	Add floating-point values	RND		Round floating-point value
ADDI	†	Add integers	ROL		Rotate left
AND	†	Bitwise logical-AND	ROLC		Rotate left through carry
ANDN	†	Bitwise logical-AND with complement	ROR		Rotate right
ASH	†	Arithmetic shift	RORC		Rotate right through carry
CMPF	†	Compare floating-point values	SUBB	†	Subtract integers with borrow
CMPI	†	Compare integers	SUBC		Subtract integers conditionally
FIX		Convert floating-point value to integer	SUBF		Subtract floating-point values
FLOAT		Convert integer to floating-point value	SUBI		Subtract integer
LSH	†	Logical shift	SUBRB		Subtract reverse integer with borrow
MPYF	†	Multiply floating-point values	SUBRF		Subtract reverse floating-point value
MPYI	†	Multiply integers	SUBRI		Subtract reverse integer
NEGB		Negate integer with borrow	TSTB	†	Test bit fields
NEGF		Negate floating-point value	XOR	†	Bitwise exclusive-OR
NEGI		Negate integer			
Program control instructions					
B*cond*		Branch conditionally (standard)	IDLE		Idle until interrupt
B*cond*D		Branch conditionally (delayed)	NOP		No operation
BR		Branch unconditionally (standard)	RETI*cond*		Return from interrupt conditionally
BRD		Branch unconditionally (delayed)	RETS*cond*		Return from subroutine conditionally
CALL		Call subroutine	RPTB		Repeat block of instructions
CALL*cond*		Call subroutine conditionally	RPTS		Repeat single instruction
DB*cond*		Decrement and branch conditionally (standard)	SWI		Software interrupt
DB*cond*D		Decrement and branch conditionally (delayed)	TRAP*cond*		Trap conditionally

† Two- and three-operand versions

key peripheral features;

- Command entry from either menu-driven keystrokes (menu mode) or from line commands (line mode);
- Help menus for all screen modes;
- Quick storage and retrieval of simulation parameters from files to facilitate preparation for individual sessions;
- Reverse assembly allowing editing and reassembly of source statements;
- Multiple execution modes;
- Trace expressions that are easy to define;
- Trace execution that can display designated expression values, cache memory, and the instruction pipeline; and
- Breakpoints that can occur on address read, write, or both, on address execute, and on expression valid.

Perhaps the most important trend with the newer DSPs is the availability of high-level-language compilers. The presence of C and Ada compilers in the 320C30 is not an accident since the 320C30 was designed with a compiler in mind. We expect this path to a high-level language to make the porting of application programs from large computers much easier. The algorithm can be developed almost entirely on a large computer and then converted to the 320C30 assembly language by compilation.

The C compiler for the 320C30 has exceptional efficiency,[2] which makes a good C program almost as effective as the assembly-language program. The C compiler will be sufficient for most applications. The exception is time-critical applications. In such cases one can use the fact that most DSP algorithms spend the vast majority of the execution time on a small section of the code. (Researchers often mention the 90/10 rule: 90 percent of the time is spent on 10 percent of the code.) Under these circumstances, the user can optimize execution by creating very fast assembly-language routines that implement the time-critical sections, and call them from C as regular C functions. To achieve this, we define the C function interface very precisely so that users can create their own routines. The C-compiler package comes with a library of general-purpose mathematical, interface, and I/O functions.

Besides this method of optimizing the performance of the C language, two more methods can be used. The first one is based on the fact that the output of the compiler is an assembly-language program. The user can edit this program and optimize it by rearranging the instructions. The second method is to use the "asm" directive supported by the C compiler. The arguments of this directive are passed to the output of the compilation without any alteration so that the user can insert assembly-language instructions into the middle of the C program.

A key part of the 320C30 development environment is Spox, the first real-time operating-system for a single-chip DSP. Spox, developed by Spectron Microsystems, extends the core C language with a library of standard I/O routines and, most importantly, a DSP math package. One of Spox's unique features is that it provides users with software objects that are especially suited for DSP. Some of these objects are vectors, matrices, filters, and streams. The math package and these software objects are carefully designed to take full advantage of the capabilities of the 320C30. Spox also supports multitasking, thus allowing the user to easily implement the more complex control structures that are becoming essential for DSP systems.

Perhaps the most important trend with the newer DSPs is the availability of high-level-language compilers.

By providing a complete software development environment that includes compilers and operating systems along with the more-traditional tools such as assemblers and linkers, we allow the user to move from system conception to system implementation in the shortest possible time.

The next level of development tools includes the hardware emulators for debugging target hardware or determining the performance of an algorithm on the 320C30 device itself. The XDS1000 is a real-time, in-circuit emulator/software development tool based on the 320C30. Besides these tools from Texas Instruments, other companies offer related support, such as the PC-based development board by Atlanta Signal Processors and the development platform of Spectron Microsystems for PCs and Sun workstations.

Applications

Certain features of the 320C30 such as its high speed, versatile architecture, and rich instruction set, make it easy to implement very demanding algorithms. The large memory space makes the device suitable for application areas such as image processing in which memory addressing is one of the prime considerations. And the C compiler makes it easy to construct algorithms with complicated logic.

General DSP algorithms. Almost every DSP application needs to perform some kind of filtering, the first application considered for a DSP device. Digital filters are categorized as FIR (finite-length impulse response) and IIR (infinite impulse response) filters,[3,4] or, equivalently, as filters that have only zeros or both poles and zeros. Each of these categories can have either fixed or adaptive coefficients.

The 320C30 implements FIR filters very efficiently. For instance, let an FIR filter have an impulse response $h[0]$, $h[1], \ldots, h[N \times 1]$, and let $x[n]$ represent the input of the filter at time n. Then, the following equation gives the output $y[n]$ with the equation:

$$y[n] = h[0] \times x[n] + h[1] \times x[n-1] + \ldots + h[N-1] \times x[n-N+1]$$

```
;
; Typical Calling Sequence:
;
;       load    AR0
;       load    AR1
;       load    RC
;       load    BK
;       CALL    FIR
;
;
; Data Memory Organization:
;
;                Impulse                  Initial          Final
;               response                 input samples   input samples
;   Low    +-------------+  Oldest +-------------+  +-------------+
; address  |    h(N-1)    |  input  | x(n-(N-1)) |  |     x(n)    |---+
;          +-------------+         +-------------+  +-------------+   |
;          |    h(N-2)    |        | x(n-(N-2)) |  | x(n-(N-1)) |    |
;          +-------------+         +-------------+  +-------------+   |
;                .                       .                .        Circular
;                .                       .                .         queue
;          +-------------+         +-------------+  +-------------+   |
;          |     h(1)     |        |    x(n-1)    |  |    x(n-2)    |   |
;  High    +-------------+  Newest +-------------+  +-------------+   |
; address  |     h(0)     |  input  |     x(n)     |  |    x(n-1)    |---+
;          +-------------+         +-------------+  +-------------+
;
; The physical address for the start of the input samples must be on
; a boundary with the LSBs set to zero according to the length of the
; buffer. The pointer to the input sequence (x) is incremented and
; assumed to be moving from an older input to a newer input. At the
; end of the subroutine AR1 will be pointing to the position for the
; next input sample.
;
; Argument Assignments:
;
;   Argument  | Function
;   ----------+-----------------------
;   AR0       | Address of h(N-1)
;   AR1       | Address of x(N-1)
;   RC        | Length of filter - 2 (N-2)
;   BK        | Length of filter (N)
;
; Registers used as input: AR0, AR1, RC, BK
; Registers modified: R0, R2, AR0, AR1, RC
; Register containing result: R0
;
;
; Program size: 6 words
;
; Execution cycles: 11 + (N-1)
;
;=============================================================================
;
        .global     FIR
;                                       ; initialize R0:
FIR     MPYF3   *AR0++(1),*AR1++(1)%,R0 ; h(N-1) * x(n-(N-1)) -> R0
        LDF     0.0,R2                  ; initialize R2.
;
; filter ( 1 <= i < N)
;
        RPTS    RC                      ; setup the repeat single.
        MPYF3   *AR0++(1),*AR1++(1)%,R0 ; h(N-1-i) * x(n-(N-1-i)) -> R0
||      ADDF3   R0,R2,R2                ; multiply and add operation
;
        ADDF    R0,R2,R0                ; add last product
;
; return sequence
;
        RETS                            ; return
;
; end
;
    .end
```

Figure 7. FIR filter implementation on the 320C30.

```
;
; Typical Calling Sequence:
;
;       load    R2
;       load    AR0
;       load    AR1
;       load    IR0
;       load    IR1
;       load    BK
;       load    RC
;       CALL    IIR2
;
; Data Memory Organization:
;
;            Filter                    Initial delay     Final delay
;          coefficients                 node values       node values
;   Low   +------------+  Newest  +-------------+   +-------------+
; address |   a2(0)    |  delay   |   d(0,n)    |   |  d(0,n-1)   |----+
;         +------------+          +-------------+   +-------------+    |
;         |   b2(0)    |          |  d(0,n-1)   |   |  d(0,n-2)   | circular
;                                                                    queue
;         +------------+  oldest  +-------------+   +-------------+    |
;         |   a1(0)    |  delay   |  d(0,n-2)   |   |   d(0,n)    |----+
;         +------------+          +-------------+   +-------------+
;         |   b1(0)    |          |   Empty     |   |   Empty     |
;         +------------+          +-------------+   +-------------+
;         |   b0(0)    |                :
;         +------------+          +-------------+   +-------------+
;                                 |  d(N-1,n)   |   | d(N-1,n-1)  |----+
;               .                 +-------------+   +-------------+    |
;               .                 | d(N-1,n-1)  |   | d(N-1,n-2)  | circular
;               .                                                    queue
;         +------------+          +-------------+   +-------------+    |
;         |  a2(N-1)   |          | d(N-1,n-2)  |   |  d(N-1,n)   |----+
;         +------------+          +-------------+   +-------------+
;         |  b2(N-1)   |          |   Empty     |   |   Empty     |
;         +------------+          +-------------+   +-------------+
;         |  a1(N-1)   |
;         +------------+
;         |  b1(N-1)   |
;  High   +------------+
; address |  b0(N-1)   |
;         +------------+
;
; The physical address for the start of each circular queue of delay node
; values must be on a boundary with the LSBs set to zero according to the
; length of the buffer. The BK (block size) register must contain the
```

(Continued on page 26)

Figure 8. Implementation of N biquads on the 320C30.

Two features of the 320C30 facilitate the implementation of the FIR filters: parallel multiply/add operations and circular addressing. The first feature permits a multiplication and an addition to execute in one machine cycle, while the second makes a finite buffer of length N sufficient for the data $x[n]$. Figure 7 shows the arrangement of the data and the assembly code for an FIR filter. Note that the filter takes one cycle of execution per tap.

The transfer function of the IIR filters contains both poles and zeros, and its output depends on both the input and the past output. As a rule, these filters need less computation than a FIR filter of similar frequency response, but they have the drawback of being sensitive to coefficient quantization. Most often, the IIR filters are implemented as a cascade of second-order sections, called biquads. To implement an IIR filter consisting of N biquads, let $a1[i]$, $a2[i]$ be the numerator coefficients of the ith biquad and $b0[i]$, $b1[i]$, $b2[i]$ the denominator coefficients of the same biquad. Also, let $x[n]$ be the input and $y[n]$ be the output of the IIR filter. In canonic form, the following C code implements the N biquads:

```
y[0,n] = x[n];
for (i=0; i<N; i++){
 d[i,n] = a2[i]*d[i,n–2] + a1[i]*d[i,n–1] + y[i–1,n];
 y[i,n] = b2[i]*d[i,n–2] + b1[i]*d[i,n–1] +
          b0[i]*d[i,n];
}
y[n] = y[N–1,n];
```

Figure 8 shows the memory arrangement and the code for this implementation on the 320C30.

In addition to the fixed-coefficient filters, the 320C30 can also implement very effectively adaptive filters (with three cycles per updated tap).

Fourier transforms are another important tool often used in DSP systems. The purpose of the transform is to convert information from the time domain to the frequency do-

```
; value 3. The result y(n) is placed in R0.  At the end of the program,
; AR1 points to the new d(0,n-2) so that it is set when the new sample
; comes in.
;
; Argument Assignments:
;   Argument  | Function
;   ----------+-----------------------
;   R2        | Input sample x(n)
;   AR0       | Address of filter coefficients (a2(0))
;   AR1       | Address of delay node values (d(0,n-2))
;   BK        | BK = 3
;   IR0       | IR0 = 4
;   IR1       | IR1 = 4*N-4
;   RC        | Number of biquads (N) - 2
;
; Registers used as input: R2, AR0, AR1, IR0, IR1, BK, RC
; Registers modified: R0, R1, R2, AR0, AR1, RC
; Register containing result: R0
;
; Program size: 17 words
;
; Execution cycles: 23 + 6N
;
;===========================================================================
;
        .global IIR2
;
IIR2    MPYF3   *AR0, *AR1, R0              ; a2(0) * d(0,n-2) -> R0
        MPYF3   *++AR0(1), *AR1--(1)%, R1   ; b2(0) * d(0,n-2) -> R1
;
        MPYF3   *++AR0(1), *AR1, R0         ; a1(0) * d(0,n-1) -> R0
||      ADDF3   R0, R2, R2                  ; first sum term of d(0,n).
;
        MPYF3   *++AR0(1), *AR1--(1)%, R0   ; b1(0) * d(0,n-1) -> R0
||      ADDF3   R0, R2, R2                  ; second sum term of d(0,n).
;
        MPYF3   *++AR0(1), R2, R2           ; b0(0) * d(0,n) -> R2
||      STF     R2, *AR1--(1)%              ; store d(0,n); point to
;                                             d(0,n-2).
;
;
    RPTB    LOOP                            ; loop for 1 <= i < N
;
        MPYF3   *++AR0(1), *++AR1(IR0), R0  ; a2(i) * d(i,n-2) -> R0
||      ADDF3   R0,R2,R2                    ; first sum term of y(i-1,n)
;
        MPYF3   *++AR0(1), *AR1--(1)%, R1   ; b2(i) * d(i,n-2) -> R1
||      ADDF3   R1,R2,R2                    ; second sum term of y(i-1,n)
;
        MPYF3   *++AR0(1), *AR1, R0         ; a1(i) * d(i,n-1) -> R0
||      ADDF3   R0, R2, R2                  ; first sum term of d(i,n).
;
        MPYF3   *++AR0(1), *AR1--(1)%, R0   ; b1(i) * d(i,n-1) -> R0
||      ADDF3   R0, R2, R2                  ; second sum term of d(i,n).
;
        STF     R2, *AR1--(1)%              ; store d(i,n); point to
;                                             d(i,n-2).
LOOP    MPYF3   *++AR0(1), R2, R2           ; b0(i) * d(i,n) -> R2
;
; final summation
;
        ADDF    R0,R2                       ; first sum term of y(N-1,n)
        ADDF3   R1,R2,R0                    ; second sum term of y(N-1,n)
;
        NOP     *AR1--(IR1)                 ; return to first biquad
        NOP     *AR1--(1)%                  ; point to d(0,n-1)
;
; return sequence
;
        RETS                                ; return
;
; end
;
    .end
```

Figure 8 (cont'd.)

main. Computationally efficient implementation of Fourier transforms are known as the fast Fourier transform (FFT).[3-5] Table 3 shows the timing for different FFTs on the 320C30. The code for these FFTs, as well as the routines listed in Table 4, appear in the *TMS320C30 User's Guide*.[6]

The 320C30 has many features that make it well suited for FFTs, such as the high speed of the device, the floating-point capability, the block-repeat construct, and the bit-reversed addressing mode. For instance, the FFT shown in Figure 9 on the next page can be implemented in code that can be entirely contained in the 64-word cache of the 320C30.[7]

Telecommunications and speech. Telecommunications and speech applications have many requirements in common with other DSP applications, but they also have some special needs. For instance, telecommunications applications interfacing to T1 carriers sometimes need to convert between a linear signal and one compressed by μ-law or A-law formats. Such a conversion can be realized with hardware by adding a peripheral to the DSP peripheral bus. This is the approach taken in some members of the TMS320 first generation of devices. An alternative way is to do the same function with software.

In speech applications, digital filters are often implemented in lattice form. Depending on the application, both FIR and IIR filters are realized this way, although sometimes the terminology lattice filter and inverse lattice filter is used respectively.

Graphics and image processing. In graphics and image processing applications DSPs perform operations on two-dimensional signals, and matrix arithmetic takes on particular significance. In the 320C30 matrix arithmetic can be decomposed into a series of dot products, which can be very effectively implemented using constructs similar to the FIR filter implementation discussed earlier. Additionally, the large memory space of the 320C30 allows processing of large segments of data at a time.

Benchmarks. We have implemented several general-purpose and applications-oriented routines for the 320C30 and include these in the *User's Guide*.[6] Table 4 lists some of these routines with the necessary cycles and the memory requirements for the program.

The last five years have seen a tremendous growth in the utility of digital signal processors. This growth has been fueled, at least in part, by the ever-increasing level of performance and ease of use of general-purpose DSPs. The TMS320C30 represents the newest generation of DSPs. But, the end of this trend is not yet in sight. Rather, we expect the trend of higher levels of performance and greater ease of use to continue. For DSPs, the next five years look bright indeed.

Table 3.
Timing of an FFT on the 320C30.

Number of points	Radix-2 (complex)	Radix-4 (complex)	Radix-2 (real)
FFT timing (ms)			
64	0.167	0.123	0.075
128	0.367	—	0.162
256	0.801	0.624	0.354
512	1.740	—	0.771
1,024	3.750	3.040	1.670
Code size (Words)	55	176	86

The code size does not include the sine/cosine tables. The timing does not include bit reversal or data I/O.

Table 4.
Program memory and timing requirements for 320C30 routines.

Application	Words	Cycles (best case/ worst case)
Inverse of a floating-point number	31	31
Integer division	27	27/58
Double-precision integer multiplication	24	20/24
Square root	32	35
Dot product of two vectors	10	$8 + (N - 1)$
Matrix times vector operation	10	$2 + R(C + 9)$
FIR filter	5	$7 + (N - 1)$
IIR filter (one biquad)	7	7
IIR filter (N> 1 biquads)	16	$19+6N$
LMS adaptive filter	9	$8 + 3(N - 1)$
LPC lattice filter	11	$9 + 5(P - 1)$
Inverse LPC lattice filter	9	$9 + 3(P - 1)$
μ-law compression	16	16
μ-law expansion	13	11/16
A-law compression	18	18
A-law expansion	15	14/21

N = length of appropriate vector
P = length of lattice filter
R = number of rows of a matrix
C = number of columns of a matrix

```
;
;   GENERIC PROGRAM TO DO A LOOPED-CODE RADIX-2 FFT COMPUTATION IN 320C30.
;
; THE PROGRAM IS ADAPTED FROM THE FORTRAN PROGRAM IN PAGE 111 OF
;     REFERENCE [5]
;
; AUTHOR: PANOS E. PAPAMICHALIS
;         TEXAS INSTRUMENTS                          JULY 16, 1987
;
        .GLOBL  N               ; FFT SIZE
        .GLOBL  M               ; LOG2(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
        .BSS    INP,1024        ; MEMORY WITH INPUT/OUTPUT DATA

        .TEXT

;   INITIALIZE
          .WORD     FFT             ; STARTING LOCATION OF THE PROGRAM
          .SPACE    100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
FFTSIZ    .WORD     N
LOGFFT    .WORD     M
SINTAB    .WORD     SINE
INPUT     .WORD     INP

FFT:    LDP     FFTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=N/4, POINTER FOR SIN/COS TABLE
        LDI     0,AR6           ; AR6 HOLDS THE CURRENT STAGE NUMBER
        LDI     @FFTSIZ,IR0
        LSH     1,IR0           ; IR0=2*N1 (BECAUSE OF REAL/IMAG)
        LDI     @FFTSIZ,R7      ; R7=N2
        LDI     1,AR7           ; INITIALIZE REPEAT COUNTER OF FIRST LOOP
        LDI     1,AR5           ; INITIALIZE IE INDEX (AR5=IE)

;   OUTER LOOP
LOOP:   NOP     *++AR6(1)       ; CURRENT FFT STAGE
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        ADDI    R7,AR0,AR2      ; AR2 POINTS TO X(L)
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #

; BUTTERFLY WITHOUT TWIDDLE FACTORS
        RPTB    BLK1
        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*N1
; IF THIS IS THE LAST STAGE, YOU ARE DONE
        CMPI    @LOGFFT,AR6
        BZD     END

; MAIN INNER LOOP
        LDI     2,AR1                   ; INIT LOOP COUNTER FOR INNER LOOP
        LDI     @SINTAB,AR4             ; INITIALIZE IA INDEX (AR4=IA)
INLOP:  ADDI    AR5,AR4                 ; IA=IA+IE; AR4 POINTS TO COSINE
        LDI     AR1,AR0
        ADDI    2,AR1                   ; INCREMENT INNER LOOP COUNTER
        ADDI    @INPUT,AR0              ; (X(I),Y(I)) POINTER
        ADDI    R7,AR0,AR2              ; (X(L),Y(L)) POINTER
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDF     *AR4,R6                 ; R6=SIN

; GENERAL  BUTTERFLY
        RPTB    BLK2
        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 AND...
||      ADDF    *++AR2,*++AR0,R3        ; R3=Y(I)+Y(L)
        MPYF    R1,*++AR4(IR1),R3       ; R3=R1*COS AND...
```

Figure 9. Example of a radix-2, decimation-in-frequency FFT.

```
||      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, INCR AR2
AND...
||      STF     R4,*+AR2            ; Y(L)=R1*COS-R2*SIN

        CMPI    R7,AR1
        BNE     INLOP                 ; LOOP BACK TO THE INNER LOOP

        LSH     1,AR7               ; INCREMENT LOOP COUNTER FOR NEXT TIME
        LSH     1,AR5                 ; IE=2*IE
        LDI     R7,IR0                ; N1=N2
        LSH     -1,R7                 ; N2=N2/2
        BR      LOOP                  ; NEXT FFT STAGE
END     NOP
        .END
```

Figure 9 (cont'd.)

References

1. K.-S. Lin, G.A. Frantz, and R. Simar, "The TMS320 Family of Digital Signal Processors," *Proc. IEEE*, Vol. 75, No. 9, Sept. 1987, pp. 1143-1159.
2. R. Simar and A. Davis, "The Application of High-Level Languages to Single-Chip Digital Signal Processors," *Proc. 1988 Int'l. Conf. Acoustics, Speech, and Signal Processing*, Apr. 1988, pp. 1678-1681.
3. A. Oppenheim and R. Schafer, *Digital Signal Processing*, Prentice Hall, Englewood Cliffs, N.J., 1975, 585 pp.
4. L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, 1975, 762 pp.
5. C.S. Burrus and T.W. Parks, *DFT/FFT and Convolution Algorithms*, John Wiley & Sons, New York, 1985, 232 pp.
6. *TMS320C30 User's Guide*, Texas Instruments, Dallas, Tex., 1988.
7. P. Papamichalis, "FFT Implementation on the TMS320C30," *Proc. 1988 Int'l. Conf. on Acoustics, Speech, and Signal Processing*, Apr. 1988, pp. 1399-1402.

Panos Papamichalis is a senior member of the technical staff and a section manager in the Texas Instruments DSP Applications Group. He is also an adjunct professor for the Electrical and Computer Engineering Department at Rice University in Houston. Author of *Practical Approaches to Speech Coding*, his interests include digital signal processing with applications to speech processing and telecommunications.

Papamichalis received his engineering degree from the School of Mechanical and Electrical Engineering, National Technical University of Athens. His MS and PhD degrees in electrical engineering come from the Georgia Institute of Technology in Atlanta. He is a member of the Institute of Electrical and Electronics Engineers and Sigma Xi.

Ray Simar, Jr. is a group member of the TI Semiconductor technical staff and the principal architect and program manager of the TMS320C30. He has supported the TMS320 family of digital signal processors.

Simar holds a BS degree in bioengineering from Texas A&M University, College Station, and an MSEE from Rice University. He is a member of Tau Beta Pi, Phi Eta Sigma, and Phi Kappa Phi.

Questions concerning this article can be directed to Panos Papamichalis, Texas Instruments, Inc., PO Box 1443, M/S 701, Houston, TX 77251-1443.

Part II. Digital Signal Processing Routines

An Implementation of FFT, DCT, and Other Transforms on the TMS320C30

Panos Papamichalis

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

This report describes the implementation of several Fast Fourier Transforms (FFTs) and related algorithms on the TMS320C30. The TMS320C30 is the first device in the third generation of 32-bit floating-point Digital Signal Processors (DSPs) in the Texas Instruments TMS320 family. The algorithms considered here are the complex radix-2 FFT, the complex radix-4 FFT, the real-valued radix-2 FFT (both forward and inverse transforms), the Discrete Hartley Transform (DHT), and the Discrete Cosine Transform (DCT). These transforms have many applications, such as in image processing, sonar, and radar.

The introduction briefly describes transforms and their implementation on the TMS320 family of processors. Next, the different kinds of FFTs (including the real FFT), the closely-related Hartley transform, and the Cosine transform are described and compared. This is followed by a description of the TMS320C30 features that permit efficient implementations of these algorithms. Then, specific implementations, transforms, and TMS320C30 C Compiler facts are outlined. Finally, the report discusses some implementation issues, and the appendices list actual TMS320C30 code for performing transforms.

The powerful architecture and instruction set of the TMS320C30 permit flexible and compact coding of the algorithms in assembly language while preserving close correspondence to a high-level language implementation. The efficiency of the architecture and the speed of the device make faster realization of real and complex transforms possible. With the availability of a C compiler, these routines can be put in C-callable form and used as faster versions of FFT C functions.

Introduction

The Fast Fourier Transform (FFT) is an important tool used in Digital Signal Processing (DSP) applications. Its development by Cooley and Tuckey gave impetus to the establishment of DSP as an independent discipline. The well-structured form of the FFT has also made it one of the benchmarks in assessing the performance of number-crunching devices and systems.

In recent years, because of the popularity of this signal-processing tool, there have been efforts to improve its performance by advances both at the algorithmic level and in hardware implementation. Researchers have been developing efficient algorithms to increase the execution speed of FFTs while keeping requirements for memory size low. On the other hand, developers of VLSI systems are including features in their designs that improve system performance for applications requiring FFTs. In particular, single-chip programmable DSP devices, currently available or under development, can realize FFTs with speeds that allow the implementation of very complex systems in realtime.

The Texas Instruments TMS320 family consists of five generations of programmable digital signal processors. The TMS32010 introduced the first generation, which today encompasses more than twelve devices with various speeds, interfacing capabilities, and price/performance combinations. FFT implementations on the TMS32010 can be found in the appendix of the book by Burrus and Parks [1].

The second-generation TMS320 devices (the TMS32020, the TMS320C25, and their spinoffs) enhanced the architecture and speed capabilities of the first generation. Examples of FFT programs implemented on the TMS32020 can be found in an application report in the book *Digital Signal Processing Applications with the TMS320 Family* [2]. Such programs are easily extended to the TMS320C25 because of the code compatibility between devices.

The architectural and speed improvements on the processors from one generation to the next have made the FFT computation faster and the programming easier. These advantages have reached a new high level in the third generation. The TMS320C30 is the first device in the third generation, and this report examines implementation of the FFT algorithms on it. The fourth generation (TMS320C4x) is a new set of floating-point devices, while the fifth generation (TMS320C5x) is a continuation of the fixed-point devices. Since software compatibility is maintained within the fixed-point and the floating-point devices, the existing FFT implementations will also be applicable to these new generations.

The Fourier Transform of an analog signal $x(t)$, given as

$$X(\omega) = \int_{-\infty}^{\infty} x(t)\, e^{-j\omega t} dt \tag{1}$$

determines the frequency content of the signal $x(t)$. In other words, for every frequency, the Fourier transform $X(\omega)$ determines the contribution of a sinusoid of that frequency in the composition of the signal $x(t)$. For computations on a digital computer, the signal $x(t)$ is sampled at discrete-time instants. If the input signal is digitized, a sequence of numbers $x(n)$ is available instead of the continuous-time signal $x(t)$. Then, the Fourier transform takes the form

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)\, e^{-j\omega n} \tag{2}$$

The resulting transform $X(e^{j\omega})$ is a periodic function of ω, and it needs to be computed for only one period. The actual computation of the Fourier transform of a stream of data presents difficulties because $X(e^{j\omega})$ is a continuous function in ω. Since the transform must be computed at discrete points, the properties of the Fourier transform led to the definition of the *Discrete Fourier Transform* (DFT), given by

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-\frac{j2\pi kn}{N}} \tag{3}$$

When $x(n)$ consists of N points $x(0)$, $x(1)$, . . ., x(N-1), the frequency-domain representation is given by the set of N points $X(k)$, k=0,1, . . .,N-1. Equation (3) is often written in the form

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk} \tag{4}$$

where $W_N^{nk} = e^{-j\,2\pi nk/N}$. The factor W_N is sometimes referred to as the *twiddle factor*. A detailed description of the DFT can be found in references [1,3,4]. The computational requirements of the DFT increase rapidly with increasing block size N, having an impact on the real-time system performance. This problem was alleviated with the development of special fast algorithms, collectively known as Fast Fourier Transform (FFT). With an FFT, the computational burden increases much less rapidly with N, and for any given N, the FFT computational load, measured in terms of required multiplications and additions, is smaller than a brute-force computation of the DFT.

The definition of the FFT is identical to the DFT: only the method of computation differs. To achieve the efficiency of an FFT, it is important that N be a highly composite number. Typically, the length N of the FFT is a power of 2: $N = 2^M$, and the whole algorithm breaks down into a repeated application of an elementary transform known as a *butterfly*. If N is not a power of 2, the sequence $x(n)$ is appended with enough zeroes to make the total length a power of 2. Again, references [1,3,4] contain a detailed development of the FFT. Reference [2] also discusses the same topic.

Different Forms of the FFT

Over the years, researchers have developed different forms of FFT for more efficient computation. Special cases, such as those in which the input is a sequence of real numbers, have been investigated, and even more sophisticated algorithms have been developed. The general form of the FFT *butterfly* is given in Figure 1.

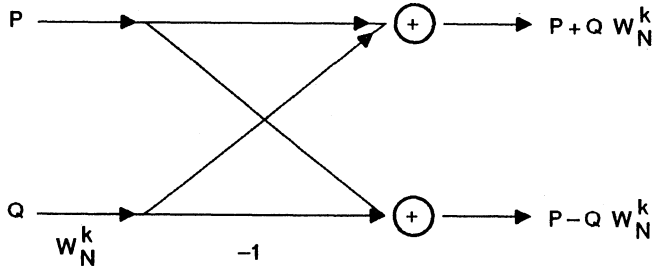


Figure 1. Radix-2 Butterfly for Decimation in Time

If the inputs to the butterfly are the two complex numbers P and Q, the outputs will be the complex numbers P' and Q', such that

$$P' = P + Q\,W_N^k \tag{5}$$

and

$$Q' = P - Q\,W_N^k \tag{6}$$

The quantities P, Q, and P', Q' represent different points in the array being transformed, and they may or may not occupy adjacent locations in that array. In an in-place computation, the result P' will overwrite P, and Q' will overwrite Q. W_N^k represents again the twiddle factor, and its exponent is determined by the location of the corresponding butterfly in the FFT algorithm.

Figure 2 shows an alternate form of the same FFT butterfly.

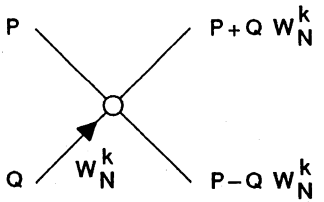


Figure 2. Alternate Form of Radix-2 Butterfly for Decimation in Time.

Although the notation is now less descriptive, it creates a clearer picture when several butterflies are put together to form an FFT. Using the first notation, Figure 3 is the flowgraph of an 8-point FFT example.

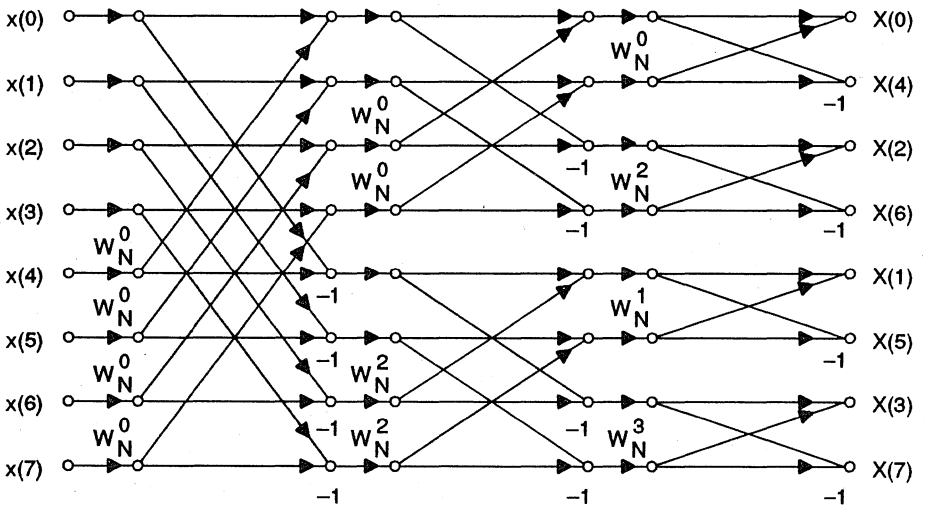


Figure 3. Example of 8-Point FFT with Decimation in Time.

Note that the input sequence *x(n)* is in the correct order, while the output *X(k)* is scrambled. Actually, this scrambling occurs in a very systematic way, called bit-reversed order: If you express the indices of a scrambled sequence in binary and you reverse this number, the result is the order that this particular point occupies. For instance, $X(3)$ occupies the sixth position in the output (when counting from the zero position). In binary form, $3_{10} = 011_2$, and if bit-reversed, you get $110_2 = 6_{10}$, which is the position that $X(3)$ occupies. It turns out that the third position is occupied by $X(6)$, and to restore the correct order at the output, you need only to swap these two numbers.

The same procedure can be repeated with all the scrambled numbers not occupying the position that their index suggests. If the input sequence $x(n)$ is rearranged to appear in bit-reversed form, the output $X(k)$ appears in the correct order, as shown in Figure 4.

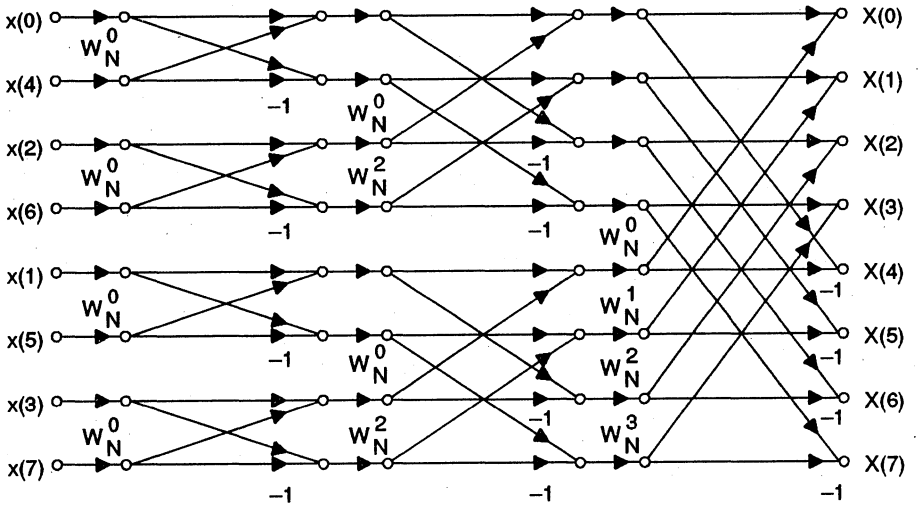


Figure 4. Alternate Form of 8-Point FFT with Decimation in Time. The Input Is in Bit-Reversed Order and the Output Is in the Correct Order.

Since the only difference between Figures 3 and 4 is a rearrangement of the butterflies, the computational load and the final results are identical. In terms of implementation, this rearrangement means that the nesting of the two innermost loops in the FFT routine is interchanged.

The butterflies and the FFT configurations presented thus far implement the FFT with a *decimation in time*. This terminology essentially describes a way of grouping the terms of the DFT definition; see Equation (3). An alternative way of grouping the DFT terms together is called *decimation in frequency*. Figures 5 and 6 show the same example of an 8-point FFT: Figure 5 with the input in correct order and the output in bit-reversed order, and Figure 6 vice-versa, and using the decimation in frequency (DIF).

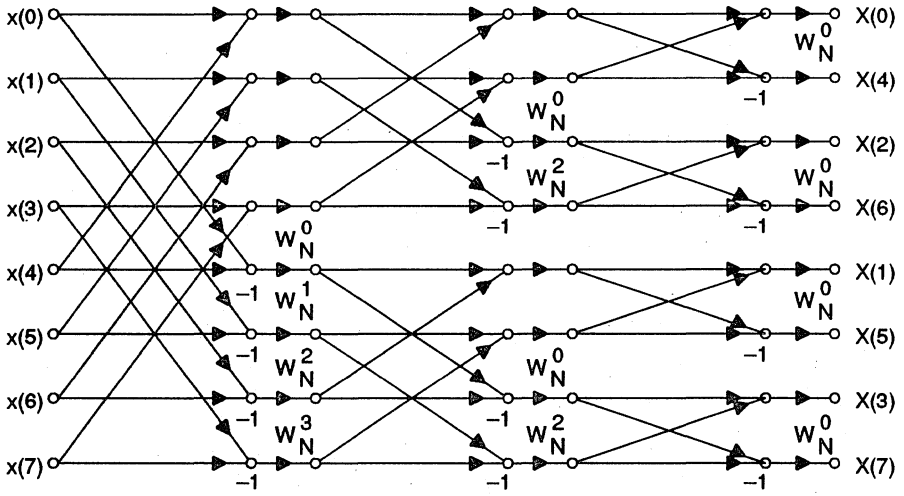


Figure 5. Example of an 8-Point FFT with Decimation in Frequency.

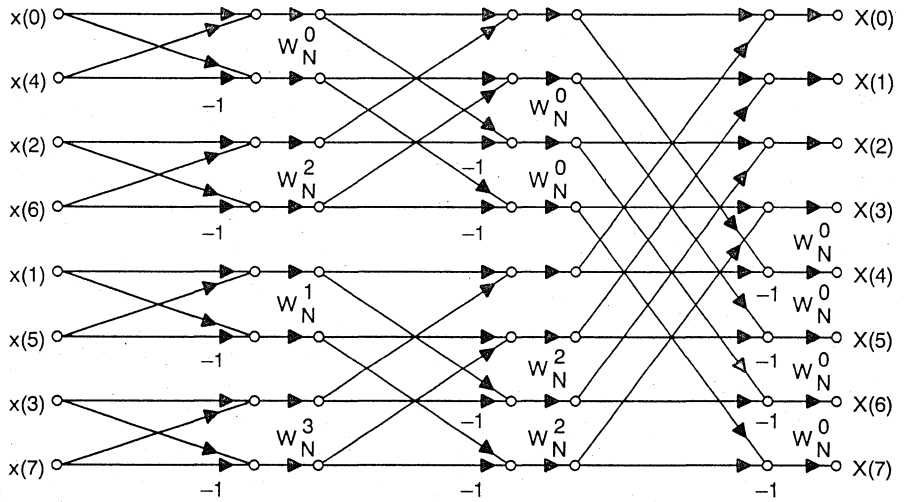


Figure 6. Alternate Form of 8-Point FFT with Decimation in Frequency. The Input Is in Bit-Reversed Order and the Output Is in the Correct Order

Pictorially, the difference between decimation in time and decimation in frequency is that the twiddle factor appears at the input of the butterfly in the first, and at the output in the second. Otherwise, the two methods are identical in terms of results. However, depending on what is the most convenient order of getting the twiddle factors and where the longest-span butterfly appears, you may prefer one method over the other.

The butterfly shown in Figure 1 (or Figure 2) is the smallest element in a radix-2 FFT. The radix of the FFT represents the number of inputs that are combined in a butterfly. The Fast Fourier Transform is usually explained around the radix-2 algorithm for conceptual simplicity. If, however, higher-order radices are used, more computational savings can be achieved. These savings increase with the radix, but there is very little improvement above radix 4. That's why the radix-2 and radix-4 FFTs are the most commonly used algorithms.

In radix-4 FFT, each butterfly has 4 inputs and 4 outputs, essentially combining two stages of a radix-2 algorithm in one. Figure 7 shows this combination graphically.

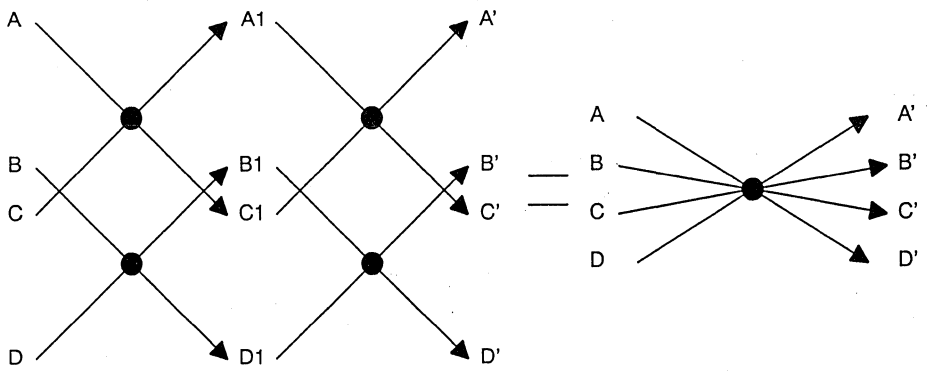


Figure 7. Butterfly for Radix-4, Decimation-in-Time FFT.

Although four radix-2 butterflies are combined into one radix-4 butterfly, the computational load of the latter is less than four times the load of a radix-2 butterfly. Examples of radix-4, 16-point FFTs are shown in Figures 8 and 9 for decimation in time and decimation in frequency, respectively.

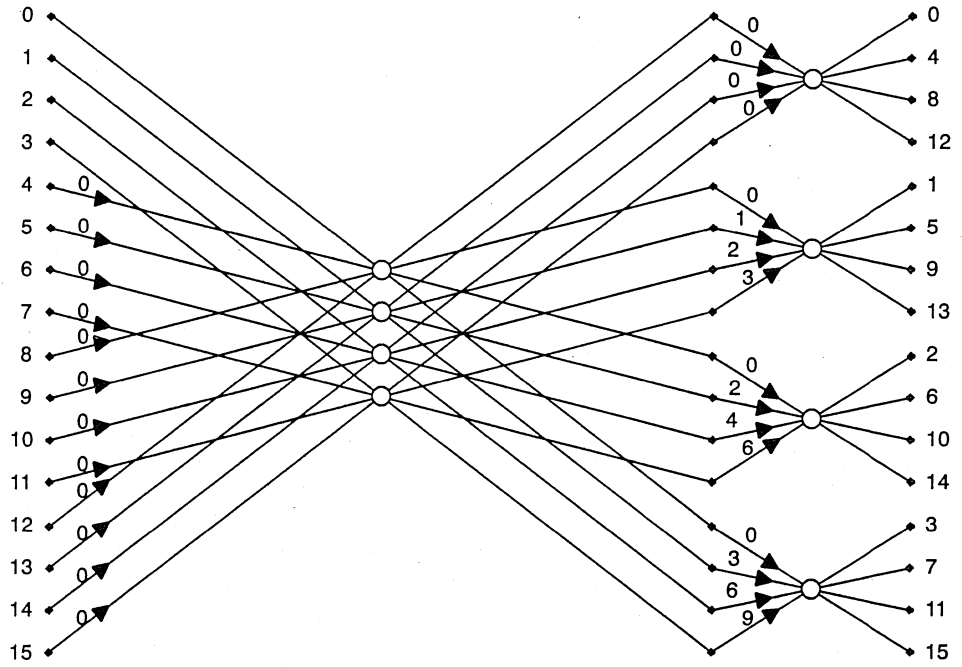


Figure 8. Example of a 16-Point, Radix-4, Decimation-in-Time FFT.

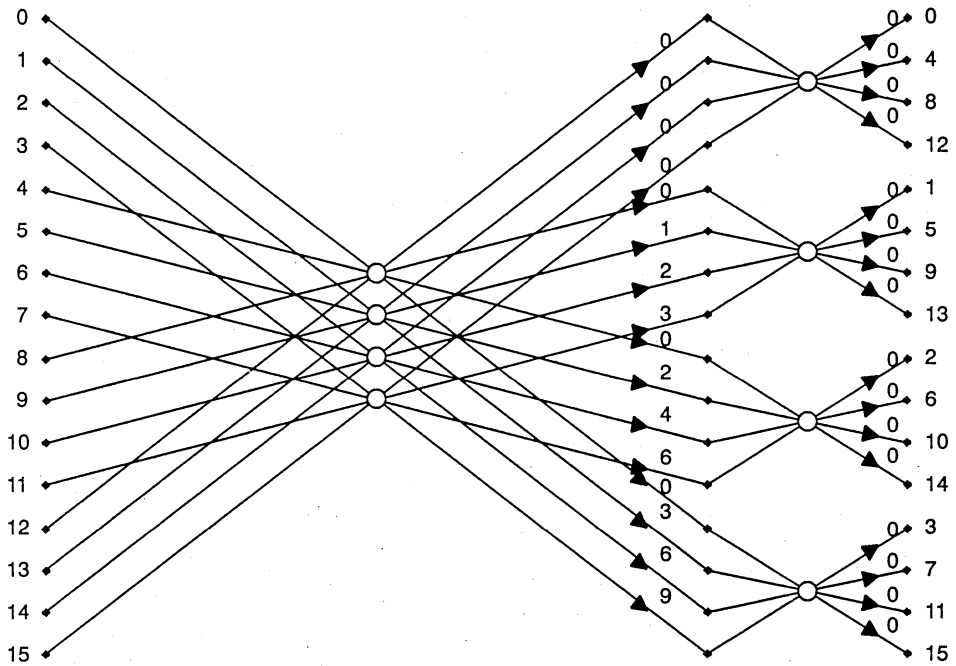


Figure 9. Example of a 16-Point, Radix-4, Decimation-in-Frequency FFT.

These configurations take the incoming sequence in order and produce the frequency-domain result in digit-reversed form. It is a simple matter to rearrange the FFT and have the input in digit-reversed form and the output in order.

Digit reversal is similar to bit reversal, except that the number whose digits are reversed is written in base 4 (equal to the radix) rather than base 2. For example, the output value $X(14)$ in a 16-point, radix-4 FFT occupies position eleven (again starting from zero) because $14_{10} = 32_4$ and, reversing the digits of the number, $23_4 = 11_{10}$. To restore the output to the correct order, the contents of locations with digit-reversed indices should be swapped. However, since the TMS320C30 has a special bit-reversed addressing mode, it is desirable to have the output of the radix-4 computation in bit-reversed rather than digit-reversed form. This is accomplished quite simply if, in each radix-4 butterfly, the two middle output legs are interchanged. That is, whenever the output of the butterfly is the four numbers *A'*, *B'*, *C'*, and *D'*, instead of storing them in that order, store them in the order *A'*, *C'*, *B'*, and *D'*, as shown in Figure 10.

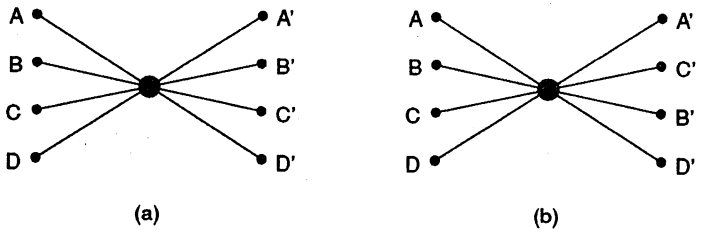


Figure 10. Radix-4 Butterflies. (a) Regularly-Ordered Output, (b) Bit-Reversed Output.

References [5, 6] explain why this simple rearrangement puts the result in bit-reversed order.

Features of the TMS320C30

The TMS320C30 is the first device introduced in the third generation of the TMS320 Digital Signal Processors [7,8]. It has many architectural features that permit very efficient implementation of algorithms. Some of those features pertinent to the FFT implementation are discussed in this section.

The two most salient characteristics of the TMS320C30 device are its high speed (60-ns cycle time) and floating-point arithmetic. The higher speed makes the implementation of real-time application easier than in earlier processors, even when the other architectural advantages are not considered. Each instruction executes in a single cycle under mild pipeline restrictions. The device automatically takes care of any potential conflicts. The pipeline should be observed closely (e.g., using the trace capability of the simulator) only if code optimization for speed is required.

The floating-point capability permits the handling of numbers of high dynamic range without concern for overflows. In FFT programs, in particular, the computed values tend to increase from one stage to the next, as discussed in reference [2]. Then, the fixed-point arithmetic will cause overflows if the incoming numbers are large enough and no provisions are made for scaling. All these considerations are eliminated with the floating-point capability of the TMS320C30. The TMS320C30 performs floating-point arithmetic with the same speed as any fixed point operation; no performance is sacrificed for this feature.

There are eight extended-precision registers, R0—R7, that can be used as accumulators or general-purpose registers, and eight auxiliary registers, AR0—AR7, for addressing and integer arithmetic. For many applications, these registers are sufficient for temporary storage of values, and there is no need to use memory locations. This is the case with the radix-2 FFT algorithm, where no locations are required other than those for the transformation of incoming data to be transformed. Also, arithmetic using these registers greatly increases the programming efficiency. The two index registers, IR0 and IR1, are used for indexing the contents of the auxiliary registers AR0—AR7, thus making the access of the butterfly legs and the twiddle factors easy.

A powerful structure in the TMS320C30 is the block-repeat capability that has the form

```
        RPTB    LABEL
        put instructions here
LABEL   last instruction
```

Whatever occurs after the RPTB instruction and up to the LABEL is repeated one time more than the number included in the repeat counter register, RC. The RC register must be initialized before entering the block-repeat construct. The net effect is that the repeated code behaves as if it were straight-line coded (no penalty for looping), with program size equal to the one in looped code. In this way, the FFT butterfly, being the core of the program, can be implemented in a block-repeat form, thereby saving execution time while preserving the clarity of the program and conserving program space.

A bit-reversed addressing mode is available to eliminate the need for swapping memory locations at the beginning or the end of the FFT (depending on the FFT type). When you use this addressing mode, you access a sequence of data points in bit-reversed order rather than sequentially, and you can recover the points in the correct order during retrieval of the data instead of spending extra cycles to accomplish it in software.

Implementation of Radix-2 and Radix-4 Complex FFTs

Because of the powerful architecture and the instruction set of the TMS320C30, the assembly language program follows closely the flow of a high-level language program; this makes it easy to read and debug. It also keeps the size of the program small and reduces the requirements for program memory. Appendix A presents an example of code for a Radix-2 complex FFT, while Appendix B is a radix-4 complex FFT. The program memory requirements for these programs (as well as others to be discussed later) are given in Table 1.

Table 1. Program Memory Requirements for the Core of the FFT and Hartley Transforms

Routine Type	Program Size
Radix-2, complex FFT	50 words
Radix-4, complex FFT	170 words
Radix-2, real FFT	68 words
Radix-2, real inverse FFT	76 words
Hartley transform	71 words

The numbers in the table correspond only to the core program and do not include the sine/cosine tables for the twiddle factors, any input/output, or any bit-reversing operations. Note also that they are independent of the FFT data size.

The data memory requirements are, of course, dependent on the FFT size. The maximum length of a complex, radix-2 FFT that can be implemented entirely on the internal memory of the TMS320C30 is 1024 points. In the present implementation, the 1024-point radix-4 FFT requires a few more locations (about 7) than are available on-chip.

The code (provided in the appendices) has been written to be independent of the FFT length. The length N, together with the sine/cosine tables for the twiddle factors, should be provided separately to maintain the generic nature of the core FFT program. An example of a file with the sine/cosine tables for a 64-point FFT is given in the Appendix F. Note that the FFT size and the number of stages are declared .global in both files (i.e., the main routine and the file with the table) so that the core program gets the actual values during linking.

To reduce the storage requirements of a sine/cosine table, a full sine and a cosine cycle are overlapped. The table stores 5/4 of a full sine wave, with the cosine table starting with a phase delay of 1/4 cycle from the sine table. This table size is larger than actually needed, and it is selected merely for testing convenience of the algorithms. The minimum table size for a radix-2 complex FFT includes 1/2 of a full sine wave, and 1/2 of a full cosine wave. If these two half waves are combined using the above quarter-cycle phase delay, the minimum table size for this kind of FFT is 3/4 of a full sine wave. For instance, for a 1024-point FFT, the table can be the first 768 points of a sine wave, where a full cycle would be 1024 points. In the case of a radix-4 complex FFT, the minimum table size should include 3/4 of a sine and 3/4 of a cosine wave. Overlapping these requirements, we get the minimum table size of a radix-4 algorithm to be one full sine wave.

An example of a linking file is also included in Appendix F to show how the different segments are assigned. For a complete description of the assembler and linker, consult the corresponding manual [6].

The timing of the FFT routines was done using the cycle-counting capability of the TMS320C30 simulator. For the conversion of the number of cycles into seconds, a cycle time of 60 ns was used. The timing refers only to the core FFT computation, ignoring read-in and write-out requirements, since such requirements are application-dependent. Also, no bit reversal is counted (although it may be included in the program), since it is performed as part of the read-in or read-out. Table 2 gives the timing for the different FFT routines and for the Hartley transform.

Table 2. FFT Timing in Milliseconds

Transform Size	Radix-2 Complex FFT	Radix-4 Complex FFT	Radix-2 Real FFT	Radix-2 Real Inverse FFT	Hartley Transform
64	0.165	0.123	0.077	0.085	0.081
128	0.370	–	0.174	0.193	0.181
256	0.816	0.624	0.387	0.434	0.403
512	1.784	–	0.857	0.964	1.132
1024	3.873	3.040	1.879	2.124	2.430
1024	2.366				

For the complex FFTs, the radix-4 algorithm reduces the execution time by 20-25% compared to radix-2, depending on the FFT size. The last entry in this table represents the timing of the radix-2, DIT routine generated at the University of Erlangen [18] and given in Appendix A. These numbers are typically used for benchmarking.

Implementation of Real FFT

The development of FFT algorithms is centered mostly around the assumption that the input sequence consists of complex numbers (as does the output). This assumption guarantees the generality of the algorithm. However, in a large number of actual applications, the input is a sequence of real numbers. If this condition is taken into consideration, additional computational savings can be achieved because the FFT of a real sequence demonstrates the following symmetries: Assuming that the FFT output $X(k)$ is complex,

$$X(k) = R(k) + j\, I(k) \tag{7}$$

and that the sequence has length N, R(k) and I(k) should satisfy the following relations:

$$R(k) = R(N-k),\ k = 1, \ldots, N/2-1 \tag{8}$$
$$I(k) = -I(N-k),\ k = 1, \ldots, N/2-1 \tag{9}$$
$$I(0) = I(N/2) = 0. \tag{10}$$

In other words, the real part of the transform is symmetric around zero frequency, while the imaginary part is antisymmetric. Similar conditions hold if the transform is expressed in terms of magnitude and phase.

The savings are due to the fact that not all points need to be computed. Since the not-computed points do not need to be saved either, there are also storage savings. An efficient algorithm for real-valued FFTs is described in [10]. This algorithm was implemented in the present study in such a way that, given the sequence of N *real* numbers $x(0), x(1), \ldots, x(N-1)$, the resulting FFT, consisting of complex numbers, is stored as $R(0), R(1), \ldots, R(N/2), I(N/2-1), I(N/2-2), \ldots, I(1)$. $R(k)$ and $I(k)$ represent the real and imaginary parts of the complex number $X(k)$. Figure 11 shows the memory arrangement for the FFT. Note that the input to the real FFT should be bit-reversed, but the bit reversal can be done as the data is brought in. With this arrangement, an N-point FFT uses exactly N memory locations. If the full array $X(k)$ is needed, the following relations should be used:

$$X(0) = R(0) \tag{11}$$

$$X(k) = R(k) + j\,I(k),\ K = 1, \ldots, N/2-1 \tag{12}$$

$$X(N/2) = R(N/2) \tag{13}$$

$$X(k) = R(N-k) - j\,I(N-k),\ k = N/2+1, \ldots, N-1 \tag{14}$$

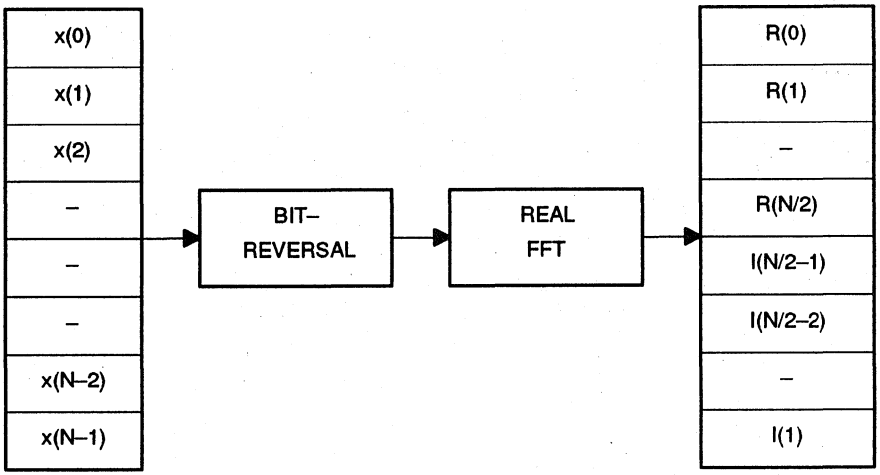


Figure 11. Memory Arrangement of a Real FFT.

It is expected that, in most signal processing applications, there will be no need to reconstruct the full $X(k)$ array and that the output shown in Figure 11 will be sufficient for any further processing.

Appendix C contains TMS320C30 routines implementing a radix-2 real FFT and its inverse. The implementation of the forward transformation is based on the FORTRAN programs contained in [10]. The inverse transformation assumes that the input data are given in the order presented at the output of the forward transformation and produces a time signal in the proper order (i.e., bit-reversing takes place at the end of the program). Viewed another way, the inverse real FFT operates as shown in Figure 11 but with the arrows reversed (and inverse FFT taking the place of the FFT).

The timing for the real-valued FFT (both forward and inverse) is included in Table 2, and the corresponding program sizes are shown in Table 1. As you can see, the real-valued FFT is considerably faster than the corresponding complex FFT because not all the computations need be performed. Furthermore, there are data storage savings because only half the values must be stored. As a result, the maximum length of real-valued FFT that can be implemented on the TMS320C30 without using any external memory is 2048 points. Of course, if all the values are needed, they can be recovered using the symmetry conditions mentioned earlier. To achieve the efficiencies of real FFT and not use any extra memory locations during the computation, the decimation-in-time method is applied [10]. Decimation in time requires the bit-reversal operation in the forward transform to be performed at the beginning of the program rather than at the end. The reverse is true for bit-reversing in the inverse transform.

The Discrete Hartley Transform

Another transform that has attracted attention recently is the Discrete Hartley Transform (DHT)[11, 12]. The DHT is applicable to real-valued signals and is closely related to the real-valued FFT. Comparison of references [10] and [12] describing the implementation of the two algorithms on FORTRAN programs shows that their implementation on the TMS320C30 should be similar. And indeed, this is the case.

The DHT pair is defined for a real-valued sequence $x(n)$, $n = 0, \ldots, N-1$, by the following equations:

$$H(k) = \sum_{n=0}^{N-1} x(n)\ cas(2\pi k\ n\ /\ N),\ k=0, \ldots, N-1 \tag{15}$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} H(k)\ cas(2\pi k\ n\ /\ N),\ k=0, \ldots, N-1 \tag{16}$$

where $cas(x) = cos(x) + sin(x)$. The DHT demonstrates a symmetry that is convenient for implementations: The same program can be used for both the forward and the inverse transforms, and the result is correct within a scale factor. Also, the real FFT and the DHT can be derived from each other [12].

A radix-2 Hartley transform was implemented on the TMS320C30, and the corresponding code is included in Appendix D. This code follows the structure of the real FFT in Appendix C. Tables 1 and 2 show the program memory requirements and the timing for the execution of Hartley transforms of different sizes. The sine/cosine table sizes are the same as in the case of a real FFT.

The Discrete Cosine Transform

The Discrete Cosine Transform (DCT), since its introduction in 1974 [13], has gained popularity in speech and image processing applications because of its near-optimal behavior. This discussion is based on the paper by Lee [14]. The DCT code was developed and implemented by Paul Wilhelm of the University of Washington.

If $x(n)$, $n=0, \ldots, N-1$ is a time-domain signal and $X(k)$ is the corresponding DCT, $x(n)$ and $X(k)$ are related by the following equations:

$$x(k) = \frac{2}{N} \sum_{n=0}^{N-1} e(k)\, x(n)\, \cos \frac{(2k+1)\pi n}{2N} \tag{17}$$

$$x(n) = \sum_{k=0}^{N-1} e(k)\, X(k)\, \cos \frac{(2k+1)\pi n}{2N} \tag{18}$$

$$e(0) = 1/\sqrt{2} \tag{19}$$

$$e(k) = 1, \quad \text{for } k \neq 0 \tag{20}$$

Appendix E shows an implementation of the DCT based on the paper by Lee [14]. The appendix contains the algorithms for both the forward and the inverse transformations and an example of a table for a 16-point DCT. Note that, because of the structure of the algorithm, the cosine table needed contains actually the inverses of the cosines (within a scale factor), and it is not stored in the natural order. Instead, it is generated by the following C pseudocode:

```
for [k=2, i=0; k=N/2; k*=2]
    for [j=k/2; j<N/2; j+=k]{
        cos_table[i++] = 1/[2*cos[j*pi/[2*N]]];
        cos_table[i++] = 1/[2*cos[[N-j]*pi/[2*N]]];
    }
cos_table[N-2] = cos[pi/4];
cos_table[N-1] = 2/N;
```

The last entry to the table is not part of the cosine itself; it is a constant that is used by the algorithm, and it is placed at the end of the cosine table for convenience.

Table 3 shows the timing of the forward and inverse transforms for different transform lengths. The difference in the timing between the forward and the inverse transforms is due to the fact that more time was expended to optimize the performance of the inverse transform. Since four of the smallest butterflies were done simultaneously in the center program loop, the minimum permissible array size to be transformed is 8.

Table 3. DCT Timing in Milliseconds

Transform Size	Forward Transform	Inverse Transform
16	0.023	0.020
64	0.105	0.088
128	0.230	0.193
256	0.502	0.416
512	1.094	0.905
1024	2.378	1.982

Other Related Transforms

In addition to the FFT types mentioned earlier (complex, real, decimation-in-time, decimation-in-frequency, etc.), newer forms of the FFT have been developed to reduce the computational load. One of the latest in the literature is the *Split-Radix* FFT. The Split-Radix FFT [16] has the lowest number of multiplies and adds of any known algorithm. It achieves this efficiency by combining certain radix-2 and radix-4 butterflies, but, as a result, the classical concept of FFT stages is lost. The new structure uses a rather complicated indexing scheme, which is the price paid for the reduced multiplies/adds. Since, on the TMS320C30, multiplies/adds are not more expensive computationally than any other operation, the indexing scheme wipes out the gains of the reduced arithmetic. Actually, an implementation of the split-radix FFT showed it to be slower than the radix-2 FFT, one of the main reasons being that the block-repeat structure could no longer be used effectively.

Very often, there is a question on what the different benchmark numbers mean. A useful comparison of execution times for different algorithms on different machines has been made [17]. Table 4 presents a small segment of the resulting information that is relevant to the present discussion: the timing in seconds for the radix-8, mix-radix, and split-radix algorithms that were implemented on various machines. Different operating systems and compilers have been used, as shown. The execution times of Table 4 should be compared with the 0.001879 s that it takes to implement a 1024-point, radix-2, real FFT on a TMS320C30. As can be seen, the TMS320C30 compares favorably to all the other machines investigated.

Table 4. Execution Times in Seconds for a 1024-Point Real FFT. The Numbers Should Be Compared with 0.001879 s of a 1024-Point Real FFT on the TMS320C30

Machine	Radix-8	Mix-radix	Split-radix
VAX 750 UNIX BSD4.2 f77	0.3634	0.3902	0.3021
VAX 750 UNIX BSD4.2 f77 –O	0.2376	0.2948	0.2089
VAX 750 UNIX BSD4.3 f77	0.2545	0.2600	0.2371
VAX 750 UNIX BSD4.3 f77 –O	0.1825	0.2127	0.1672
VAX 785 ULTRIX f77	0.1046	0.1107	0.1101
VAX 785 ULTRIX f77 –O	0.0796	0.0943	0.0811
VAX 785 VMS FOR/NOOPTM	0.0767	0.0871	0.0975
VAX 785 VMS FOR/OPTM	0.0539	0.0641	0.0633
VAX 8600 VMS FOR/OPTM	0.0217	0.0243	0.0235
MICROVAX VMS FOR/NOOPTM	0.1671	0.1846	0.1864
MICROVAX VMS FOR/OPTM	0.1299	0.1527	0.1419
DEC-10 TOPS-10 FOR/NOOPTM	0.0940	0.1184	0.0991
DEC-10 TOPS-10 FOR/OPTM	0.0885	0.1110	0.0845
CDC 855 FTN5,OPT=0	0.0277	0.0319	0.0338
CDC 855 FTN5,OPT=1	0.0277	0.0316	0.0337
CDC 855 FTN5,OPT=2	0.0182	0.0171	0.0151
CDC 855 FTN5,OPT=3	0.0180	0.0173	0.0150
SUN 3/50 UNIX BSD4.2 f77 –O -f68881	0.2518	0.3365	0.2103
SUN 3/50 UNIX BSD4.2 f77 –f68881	0.2806	0.3897	0.2802
SUN 3/50 UNIX BSD4.2 f77 –O	0.7586	1.047	0.6955
SUN 3/50 UNIX BSD4.2 f77	0.7476	1.029	0.7033
SUN 3/160 UNIX BSD4.2 f77	0.6037	0.6895	0.5660
SUN 3/160 UNIX BSD4.2 f77 –pfa	0.0983	0.1060	0.0946
SUN 3/260 UNIX BSD4.3 f77	0.3689	0.4126	0.3390
SUN 3/260 UNIX BSD4.3 f77 –O	0.3530	0.4142	0.3297
Pyramid 90X UNIX BSD4.2 f77 –O	0.2053	0.2244	0.1416
Pyramid 90X UNIX BSD4.2 f77	0.2206	0.2457	0.1326
HP-1000 21MX-E FTN7X	0.9400	1.248	0.9478
Apple MAC Microsoft FOR	2.6670	3.1600	2.8260
AST PC Microsoft FOR	1.5040	2.0800	1.4630

The TMS320C30 C Compiler

The C compiler for the TMS320C30 permits easy porting of high-level language programs to the DSP device. If the CPU loading of a particular application is not very high, the C compiler can create programs that run on the TMS320C30 in real time. If, however, the result is non-realtime, it may be necessary to use assembly language for more efficient coding.

In most cases, only a portion of the code needs to be written in assembly language. Typically, there are a few code segments where the device spends most of the time and which, when optimized in assembly language, yield the necessary performance improvement. By following the conventions outlined in the run-time environment of the C compiler [15], you can write these time-critical routines in assembly language and call them in a C program. This is also true for the FFT routines. In appendices A, B, and C, the radix-2, radix-4, and real FFT routines mentioned earlier are also put in a C-callable form by adding the necessary interface at the beginning and the end of the code. The tables with the sines and cosines are again assumed to be supplied during link time.

Issues in FFT Implementation

There are many ways of actually implementing the FFT code (and the other transformations), taking into consideration the different possibilities of program locations, the data locations, the ways of input and output, etc. Since it is impractical to cover every possible case, this report has concentrated on a configuration in which the use of external memory is minimized. With the source code and additional explanations provided, you should be able to customize the FFT implementation for a particular application.

Use of External Memory

In these implementations, only on-chip memory was used, and that's why the maximum transform size considered was 1024 points long (2048 for a real transform). Often, though, applications call for use of external memory for program or data or both. When external memory is used, the structure of the code does not change at all; it is only the timing that may be affected.

Fast external memory can be selected so that no wait states are necessary. But even when there are no wait states, accessing external memory may impose some limitations. For instance, you can make only one external memory access in a full cycle, but you can make two accesses of internal memory in each cycle. Also, because of mutliplexing of the busses, pipeline conflicts may arise if both program and data are placed on the same external port. Resolution of such conflicts causes extra cycles for the execution. The section on pipelining in the *TMS320C30 User's Guide* explains in detail what kind of potential conflicts may occur.

To minimize or avoid such conflicts, there are some simple steps that the designer can take. The TMS320C30 has three separate memory areas (one on-chip, one accessed by the primary bus, and one accessed by the expansion bus) that can be combined. For instance, the program can be placed on the expansion port and the data on the primary port. Or the data can first be brought into internal memory and then operated upon. Alternatively, the program may be relocated to internal memory. A related approach is to use the cache. All the transforms are implemented as loops that are executed many times. If you activate the on-chip cache after the first access of the code, the instructions execute from the cache instead of the external memory.

If there are additional conflicts, they can typically be resolved by some rearrangement of the code. For instance, consecutively writing to external memory takes two cycles per write. If, however, a write is followed by some internal operation, then the second cycle of the write is transparent, and the actual cost is one cycle.

Bit Reversal

The TMS320C30 has a special form of the indirect addressing mode for the bit-reversing operation that is required at the beginning or the end of an FFT. Through this addressing mode, the scrambled data are accessed in their proper order. This addressing mode works as follows:

Let ARn (n=0..7) be the auxiliary register pointing to the array with scrambled data. The index register IR0 contains a number equal to one-half the size of the FFT. Then, after every access of the data, ARn is incremented by IR0 using the construct

```
*ARn++[IR0]B
```

This causes the contents of ARn to be incremented by the contents of IR0, but if there is a carry in this incrementing, the carry propagates to the right instead of to the left. The result is the generation of the addresses in a bit-reversed order. The bit-reversed addressing mode works correctly if the array with the data is aligned in memory so that the first memory address is a multiple of the FFT size. This can be achieved if the first memory address has zeros for the last M bits, where $M = log_2 N$, with N being the FFT size. For example, in the case of a 1024-point FFT, the last 10 bits of the memory address of the first datum should be zeros.

In the implementation of the complex FFT, the output is complex even when the input is real. So, there is a need to consider both the real and the imaginary parts of the data array. The above description of the bit-reversed addressing mode assumed that the real and the imaginary parts are stored as separate arrays in the memory. In this case, each of the arrays (real or imaginary parts) can be accessed as described. However, in most cases (including this report), the real and imaginary points alternate in the same array.

In this arrangement, the following simple modification achieves the same goal: set IR0 equal to N instead of $N/2$, and access the N points of the transform. At every access, the auxiliary register is pointing to the real part of the FFT. The imaginary part is located in the next higher location, and it can be easily accessed.

With the bit-reversed addressing mode, the unscrambling of the data can take place when the FFT result is accessed for further processing or for I/O. It is possible, though, that certain applications demand the reordering of the data in the same array. Such a rearrangement can be done very simply for a complex FFT with the following code.

```
; DO THE BIT-REVERSING EXPLICITLY

        LDI   @FFTSIZ,RC      ; RC = FFT SIZE
        SUBI  1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI   @FFTSIZ,IR0     ; IR0 = FFT SIZE
        LDI   @INPUT,AR0
        LDI   @INPUT,AR1
*
        RPTB  BITRV
        CMPI  AR1,AR0         ; EXCHANGE LOCATIONS ONLY
        BGE   CONT            ;     IF AR0AR1
        LDF   *AR0,R0         ;
||      LDF   *AR1,R1         ;     EXCHANGE REAL PARTS
        STF   R0,*AR1         ;
||      STF   R1,*AR0         ;
        LDF   *+AR0,R0        ;
||      LDF   *+AR1,R1        ;     EXCHANGE IMAGINARY PARTS
        STF   R0,*+AR1        ;
||      STF   R1,*+AR0        ;
CONT    NOP   *AR0++[2]
BITRV   NOP   *AR1++[IR0]B
```

Note that AR1 is pointing to the bit-reversed version of the address contained in AR0. For real-valued FFT, or for FFTs that store the real and the imaginary parts in separate arrays, the real-FFT routine in Appendix C contains a modified example of the above code.

Use of DMA

If the signal to be transformed arrives as a continuous stream of data, the DMA could be used to collect the new data while the data already collected are processed. In this case, the data source address of the DMA points to the memory location corresponding to a serial port, or to another port associated with an external device. The destination is a memory space designated for storage.

There are two ways to use such buffers. One possibility is to designate one buffer as the temporary storage and the other buffer as the working area. When the storage buffer receives the necessary amount of data, the data is transferred to the working area, and the DMA starts refilling the storage buffer. Alternatively, the two buffers are considered equivalent: when the processor finishes processing and outputting the data from one and the DMA has filled the other, the two buffers switch functions; i.e., the DMA starts filling the first buffer while the CPU is processing the data in the buffer just filled.

Test Vector

For testing purposes, a vector with 64 (quasi-random) data points and the corresponding FFT values is given in Appendix F. In this way, if any of the routines is implemented, the test vectors can be used to verify the correct functionality of the routines. Together with the test vectors, Appendix C gives a sine/cosine table for a 64-point transform, and the linking file for such a transform.

Summary

This report examined implementations of fast transforms on the Texas Instruments TMS320C3x floating-point devices. The transforms considered were several forms of the FFT, the Discrete Hartley Transform, and the Discrete Cosine Transform. Because of the powerful architecture of the device, the implementation was done easily and efficiently. It was shown that a TMS320C30 executes the FFTs several times faster than large computers such as VAX and SUN workstations. With the availability of the C compiler, these routines can be put in C-callable form and be used to compute the corresponding transforms efficiently.

Appendices

Appendices A to F contain the TMS320C30 assembly language programs for the different algorithms considered. The contents of the appendices are as follows:

Appendix A: Radix-2 Complex FFT.
composed of
- A1: Generic Program to Do a Looped-Code Radix-2 FFT Computation on the TMS320C30.
- A2: fft_2 - Radix-2 Complex FFT to Be Called as a C Function.
- A3: Complex, Radix-2 DIT FFT - R2DIT.ASM.
- A4: Complex, Radix-2 DIT FFT - R2DITB.ASM.
- A5: TWID1KBR.ASM - Table with Twiddle Factors for a FFT up to a Length of 1024 Complex Points.

Appendix B: Radix-4 Complex FFT.
composed of
- B1: Generic Program to Do a Looped-Code Radix-4 FFT on the TMS320C30.
- B2: fft_4 - Radix-4 Complex FFT to Be Called as a C Function.

Appendix C: Radix-2 Real FFT.
composed of
- C1: Generic Program to Do a Radix-2 Real FFT Computation on the TMS320C30.
- C2: fft_rl - Radix-2 Real FFT to Be Called as a C Function.
- C3: Generic Program to Do a Radix-2 Real Inverse FFT Computation on the TMS320C30.

Appendix D: Discrete Hartley Transform.
composed of
- D1: Generic Program to Do a Radix-2 Hartley Transform on the TMS320C30.

Appendix E: Discrete Cosine Transform.
composed of
- E1: A Fast Cosine Transform.
- E2: A Fast Cosine Transform (Inverse Transform).
- E3: FCT Cosine Tables File.
- E4: Data File.

Appendix F: Test Vectors, 64-Point Sine Table, Link Command File. composed of

F1: Example of a 64-Point Vector to Test the FFT Routines.

F2: File to Be Linked with the Source Code for a 64-Point, Radix-4 FFT.

F3: Link Command File.

The first three appendices contain the code for the radix-2, complex radix-4, and real radix-2 FFT transformations. These routines are given in both the regular form and in a C-callable form. Furthermore, the contents of a file with the twiddle factors are given, as well as an example of a link command file for a 64-point FFT. Note that the source code of these routines can be downloaded from the TI DSP bulletin board (BBS) by calling (713) 274-2323. For questions regarding the BBS, call the TI DSP hotline at (713) 274-2320.

Acknowledgements

Mr. Raimund Meyer and Mr. Karl Schwarz (Lehrstuhl fur Nachrichtentechnik, University of Erlangen) provided the fast routines of Appendix A to do 1024-point, radix-2, DIT FFT. Mr. Paul Wilhelm of the University of Washington provided the routines for the Fast Cosine Transform (FCT) together with the related explanations and the test vector in Appendix E. Their contributions are gratefully acknowledged.

References

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

[2] Lin, K. -S., Ed. *Digital Signal Processing Applications with the TMS320 Family*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

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

[4] Rabiner, L.W., and Gold, B. *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[5] Burrus, C.S. "Unscrambling for Fast DSP Algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-36, No. 7, pp. 1086—1087, July 1988.

[6] Papamichalis, Panos E., and Burrus, C.S. "Conversion of Digit-Reversed to Bit-Reversed Order in FFT Algorithms," *Proceedings of 1989 IEEE International Conference on Acoustics, Speech, and Signal Processing*, May 1989.

[7] *Third-Generation TMS320 User's Guide*, Texas Instruments, Inc., Dallas, Texas, August 1988.

[8] Papamichalis, Panos E., and Simar, Ray Jr. "The TMS320C30 Floating-Point Digital Signal Processor," *IEEE Micro*, Vol. 8, No. 6, pp. 13—29, December1988.

[9] *TMS320C30 Assembly Language Tools User's Guide*, Texas Instruments Inc., Dallas, Texas, July 1987.

[10] Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burrus, C.S. "Real-Valued Fast Fourier Transform Algorithms", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-35, No. 6, pp. 849—863, June 1987.

[11] Bracewell, R.N. "The Fast Hartley Transform," *Proceedings of IEEE*, Vol. 72, No. 8, pp. 1010—1018, August 1984.

[12] Sorensen, H.V., Jones, D.L., Burrus, C.S., and Heideman, M.T. "On Computing the Discrete Hartley Transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-33, No. 4, pp. 1231—1238, October 1985.

[13] Ahmed, N., Natarajan, T., and Rao, K.R. "Discrete Cosine Transform," *IEEE Transactions on Computers*, Vol. C-23, pp. 90—93, January 1974.

[14] B. G. Lee, "FCT - A Fast Cosine Transform," *Proceedings of 1984 IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 28A.3.1—28A.3.4, March 1984.

[15] *TMS320C30 C Compiler Reference Guide*, Texas Instruments Inc., Dallas, Texas, December 1988.

[16] Sorensen, H.V., Heideman, M.T., and Burrus, C.S. "On Computing the Split-Radix FFT," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34, No. 1, pp. 152—156, February 1986.

[17] Sorensen, H.V.and Burrus, C.S. "Computer Dependency of FFT Algorithms", *Proceedings of ASILOMAR*, 1987.

[18] Schuessler, H.W., Meyer, R., and Schwarz, K. "FFT Implementation on DSP Chips—Theory and Practice," *Proposal for the 1990 IEEE International Conference on Acoustics, Speech, and Signal Processing.*

Appendix A. Radix-2 Complex FFT

Appendix A1. Generic Program to Do a Looped-Code Radix-2 FFT Computation on the TMS320C30

```
*   GENERIC PROGRAM TO DO A LOOPED-CODE RADIX-2 FFT COMPUTATION ON THE
*   TMS320C30.
*
*   THE PROGRAM IS TAKEN FROM THE BURRUS & 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.
*
*   AUTHOR: PANOS E. PAPAMICHALIS
*           TEXAS INSTRUMENTS          JULY 16, 1987
*
        .GLOBL  FFT             ; ENTRY POINT FOR EXECUTION
        .GLOBL  N               ; FFT SIZE
        .GLOBL  M               ; LOG2(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
*
INP     .USECT  "IN",1024       ; MEMORY WITH INPUT DATA
        .BSS    OUTP,1024       ; MEMORY WITH OUTPUT DATA
*
      .TEXT
*
*   INITIALIZE
*
        .WORD   FFT             ; STARTING LOCATION OF THE PROGRAM
*
        .SPACE  100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
*
FFTSIZ  .WORD   N
LOGFFT  .WORD   M
SINTAB  .WORD   SINE
INPUT   .WORD   INP
OUTPUT  .WORD   OUTP
*
FFT:    LDP     FFTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
*
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=N/4, POINTER FOR SIN/COS TABLE
        LDI     0,AR6           ; AR6 HOLDS THE CURRENT STAGE NUMBER
        LDI     @FFTSIZ,IR0
        LSH     1,IR0           ; IR0=2*N1 (BECAUSE OF REAL/IMAG)
        LDI     @FFTSIZ,R7      ; R7=N2
        LDI     1,AR7           ; INITIALIZE REPEAT COUNTER OF FIRST
*                               ; LOOP
        LDI     1,AR5           ; INITIALIZE IE INDEX (AR5=IE)
*
*   OUTER LOOP
*
LOOP:   NOP     *++AR6(1)       ; CURRENT FFT STAGE
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        ADDI    R7,AR0,AR2      ; AR2 POINTS TO X(L)
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
*   FIST LOOP
*
        RPTB    BLK1
        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*N1
*
*   IF THIS IS THE LAST STAGE, YOU ARE DONE
*
        CMPI    @LOGFFT,AR6
        BZD     END
*
*   MAIN INNER LOOP
*
        LDI     2,AR1           ; INIT LOOP COUNTER FOR INNER LOOP
        LDI     @SINTAB,AR4     ; INITIALIZE IA INDEX (AR4=IA)
INLOP:  ADDI    AR5,AR4         ; IA=IA+IE; AR4 POINTS TO COSINE
        LDI     AR1,AR0
        ADDI    2,AR1           ; INCREMENT INNER LOOP COUNTER
        ADDI    @INPUT,AR0      ; (X(I),Y(I)) POINTER
        ADDI    R7,AR0,AR2      ; (X(L),Y(L)) POINTER
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDF     *AR4,R6         ; R6=SIN
*
*   SECOND LOOP
*
        RPTB    BLK2
        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 AND...
||      ADDF    *++AR2,*++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    *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, INCR AR2 AND...
||      STF     R4,*++AR2       ;   Y(L)=R1*COS-R2*SIN
*
        CMPI    R7,AR1
        BNE     INLOP           ; LOOP BACK TO THE INNER LOOP
```

```
*
        LSH     1,AR7           ; INCREMENT LOOP COUNTER FOR NEXT TIME
*
        LSH     1,AR5           ; IE=2*IE
        LDI     R7,IR0          ; N1=N2
        LSH     -1,R7           ; N2=N2/2
        BR      LOOP            ; NEXT FFT STAGE
*
*   STORE RESULT OUT USING BIT-REVERSED ADDRESSING
*
END:    LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0     ; IR0=SIZE OF FFT=N
        LDI     2,IR1
        LDI     @INPUT,AR0
        LDI     @OUTPUT,AR1

        RPTB    BITRV
        LDF     *+AR0(1),R0
||      LDF     *AR0++(IR0)B,R1
BITRV   STF     R0,*+AR1(1)
||      STF     R1,*AR1++(IR1)
*
SELF    BR      SELF            ; BRANCH TO ITSELF AT THE END
        .END
```

Appendix A2. fft_2—Radix-2 Complex FFT to Be Called as a C Function

```
* NAME:
*    fft_2 --- RADIX-2 COMPLEX FFT TO BE CALLED AS A C FUNCTION.
*
* SYNOPSIS:
*    INT  fft_2(N, M, DATA)
*    INT  N      FFT SIZE: N=2**M
*    INT  M      NUMBER OF STAGES = LOG2(N)
*    FLOAT *DATA    ARRAY WITH INPUT AND OUTPUT DATA
*
* DESCRIPTION:
*    GENERIC FUNCTION TO DO A RADIX-2 FFT COMPUTATION ON THE 320C30.
*    THE DATA ARRAY IS 2*N-LONG, WITH REAL AND IMAGINARY VALUES ALTERNATING.
*    THE PROGRAM IS BASED ON THE FORTRAN PROGRAM IN THE BURRUS AND PARKS
*    BOOK, P. 111.
*
*    THE COMPUTATION IS DONE IN PLACE, AND THE ORIGINAL DATA IS DESTROYED.
*    BIT REVERSAL IS IMPLEMENTED AT THE END OF THE FUNCTION. IF THIS IS NOT
*    NECESSARY, THIS PART CAN BE COMMENTED OUT.
*
*    THE SINE/COSINE TABLE FOR THE TWIDDLE FACTORS IS EXPECTED TO BE SUPPLIED
*    DURING LINK TIME, AND IT SHOULD HAVE THE FOLLOWING FORMAT:
*
*        .GLOBAL   _sine
*        .DATA
*        _sine     .FLOAT  VALUE1  = sin(0*2*pi/N)
*        .FLOAT    VALUE  = sin(1*2*pi/N)
*        ......
*        .FLOAT    VALUE(5N/4)  = sin((5*N/4-1)*2*pi/N)
*
*    THE VALUES VALUE1, VALUE2, ETC., ARE THE SAME WAVE VALUES. FOR AN
*    N-POINT FFT, THERE ARE N+N/4 VALUES FOR A FULL AND A QUARTER PERIOD OF
*    THE SINE WAVE. IN THIS WAY, A FULL SINE AND COSINE PERIOD ARE AVAILABLE
*    (SUPERIMPOSED).
*
*    STACK STRUCTURE UPON THE CALL:
*                  +--------------+
*        -FP(4)    |    DATA      |
*        -FP(3)    |    M         |
*        -FP(2)    |    N         |
*        -FP(1)    | RETURN ADDR  |
*        -FP(0)    |    OLD FP    |
*                  +--------------+
*
*    REGISTERS USED: R0, R1, R2, R3, R4, R5, R6, R7, AR0, AR1, AR2, AR4, AR5
*                    AR6, AR7, IR0, IR1, RS, RE, RC
*
*    AUTHOR: PANOS E. PAPAMICHALIS
*            TEXAS INSTRUMENTS          OCTOBER 13, 1987
*
*****************************************************************************
*
FP      .set    AR3
*
        .GLOBL  _fft_2          ; ENTRY POINT FOR EXECUTION
        .GLOBL  _sine           ; ADDRESS OF SINE TABLE
*
        .BSS    FFTSIZ,1
        .BSS    LOGFFT,1
        .BSS    INPUT,1
*
        .TEXT
*
SINTAB  .word   _sine
*
*   INITIALIZE C FUNCTION
*
_fft_2: PUSH    FP              ; SAVE DEDICATED REGISTERS
        LDI     SP,FP
        PUSH    R4
        PUSH    R5
        PUSHF   R6
        PUSHF   R7
        PUSH    AR4
        PUSH    AR5
        PUSH    AR6
        PUSH    AR7
*
        LDI     *-FP(2),R0      ; MOVE ARGUMENTS TO LOCATIONS MATCHING
        STI     R0,@FFTSIZ      ; THE NAMES IN THE PROGRAM
        LDI     *-FP(3),R0
        STI     R0,@LOGFFT
        LDI     *-FP(4),R0
        STI     R0,@INPUT
*
*   INITIALIZE FFT ROUTINE
*
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=N/4, POINTER FOR SIN/COS TABLE
        LDI     0,AR6           ; AR6 HOLDS THE CURRENT STAGE NUMBER
        LDI     @FFTSIZ,IR0
        LSH     1,IR0           ; IR0=2*N1 (BECAUSE OF REAL/IMAG)
        LDI     @FFTSIZ,R7      ; R7=N2
        LDI     1,AR7           ; INITIALIZE REPEAT COUNTER OF FIRST
*                               ;   LOOP
        LDI     1,AR5           ; INITIALIZE IE INDEX (AR5=IE)
*
*   OUTER LOOP
*
LOOP:   NOP     *++AR6(1)       ; CURRENT FFT STAGE
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        ADDI    R7,AR0,AR2      ; AR2 POINTS TO X(L)
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
```

```
*   FIST LOOP
*
        RPTB    BLK1
        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*N1
*
*   IF THIS IS THE LAST STAGE, YOU ARE DONE
*
        CMPI    @LOGFFT,AR6
        BZD     END
*
*   MAIN INNER LOOP
*
        LDI     2,AR1               ; INIT LOOP COUNTER FOR INNER LOOP
        LDI     @SINTAB,AR4         ; INITIALIZE IA INDEX (AR4=IA)
INLOP:  ADDI    AR5,AR4             ; IA=IA+IE; AR4 POINTS TO COSINE
        LDI     AR1,AR0
        ADDI    2,AR1               ; INCREMENT INNER LOOP COUNTER
        ADDI    @INPUT,AR0          ; (X(I),Y(I)) POINTER
        ADDI    R7,AR0,AR2          ; (X(L),Y(L)) POINTER
        LDI     AR7,RC
        SUBI    1,RC                ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDF     *AR4,R6             ; R6=SIN
*
*   SECOND LOOP
*
        RPTB    BLK2
        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 AND...
||      ADDF    *++AR2,*++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    *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, INCR AR2 AND...
||      STF     R4,*+AR2            ;   Y(L)=R1*COS-R2*SIN
*
        CMPI    R7,AR1
        BNE     INLOP               ; LOOP BACK TO THE INNER LOOP
*
        LSH     1,AR7               ; INCREMENT LOOP COUNTER FOR NEXT TIME
*
        LSH     1,AR5               ; IE=2*IE
        LDI     R7,IR0              ; N1=N2
        LSH     -1,R7               ; N2=N2/2
        BR      LOOP                ; NEXT FFT STAGE
*
*   DO THE BIT-REVERSING OF THE OUTPUT
*
END:    LDI     @FFTSIZ,RC          ; RC=N
        SUBI    1,RC                ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0         ; IR0=SIZE OF FFT=N
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
*
        RPTB    BITRV
        CMPI    AR0,AR1
        BGE     CONT
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
        LDF     *+AR0(1),R0
||      LDF     *+AR1(1),R1
        STF     R0,*+AR1(1)
||      STF     R1,*+AR0(1)
CONT    NOP     *++AR0(2)
BITRV   NOP     *AR1++(IR0)B
*
*   RESTORE THE REGISTER VALUES AND RETURN
*
        POP     AR7
        POP     AR6
        POP     AR5
        POP     AR4
        POPF    R7
        POPF    R6
        POP     R5
        POP     R4
        POP     FP
        RETS
```

Appendix A3. Complex, Radix-2 DIT FFT – R2DIT.ASM

```
*                COMPLEX, RADIX-2 DIT FFT  :  R2DIT.ASM                *
*               ----------------------------------------               *
*                                                                      *
*   GENERIC PROGRAM FOR A FAST LOOPED-CODE RADIX-2 DIT FFT COMPUTATION *
*                          ON THE TMS320C30                            *
*                                                                      *
*   WRITTEN BY: RAIMUND MEYER, KARL SCHWARZ                19.07.89    *
*               LEHRSTUHL FUER NACHRICHTENTECHNIK                      *
*               UNIVERSITAET ERLANGEN-NUERNBERG                        *
*               CAUERSTRASSE 7, D-8520 ERLANGEN, FRG                   *
*                                                                      *
*   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.                           *
*                                                                      *
*   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.                                     *
*                                                                      *
**************************************************************************
*                                                                      *
*   EXAMPLE FOR A 1024-POINT FFT (EXCLUDING BIT REVERSAL):             *
*                                                                      *
*   MEMORY SIZE:                                                       *
*        PROGRAM                    =   229 WORDS                      *
*        DATA (TWIDDLE FACTORS)     =   512 WORDS                      *
*                                                                      *
*   CYCLES PER BUTTERFLY:                                              *
*        STAGES 1 AND 2             =     4                            *
*        STAGES 3 TO  8             =     8                            *
*        STAGE 9                    =     8.25                         *
*        STAGE 10                   =     8.5                          *
*                                                                      *
*   AVERAGE CYCLES/BUTTERFLY         =    7.275                        *
*   TOTAL BUTTERFLYCYCLES            = 37248                           *
*   INITIALIZATION OVERHEAD          =  2181  =  5.55 % OF TOTAL TIME  *
*   TOTAL NUMBER OF INSTRUCTION CYCLES = 39429                         *
*   TOTAL TIME FOR A 1024 POINT FFT  =    2.36 ms (EXCLUDING BIT       *
*                                                          REVERSAL)   *
*                                                                      *
**************************************************************************
```

```
**************************************************************************
*                                                                      *
*   THIS PROGRAM INCUDES FOLLOWING FILES:                              *
*   -------------------------------------                              *
*   THE FILE 'TWIDIKBR.ASM' CONSISTS OF TWIDDLE FACTORS                *
*                                                                      *
*   THE TWIDDLE FACTORS ARE STORED IN BITREVERSED ORDER AND WITH A TABLE *
*   LENGTH OF N/2 (N = FFTLENGTH).                                     *
*   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      *
*   AVAILABLE FFT OF LESS OR EQUAL LENGTH.                             *
*                                                                      *
*   THE MISSING TWIDDLE FACTORS (WN(),WN(),....) ARE GENERATED BY USING *
*   THE SYMMETRY WN(N/4+n) = -j*WN(n). THIS CAN BE EASILY REALIZED BY  *
*   CHANGING REAL- AND IMAGINARY PART OF THE TWIDDLE FACTORS AND BY    *
*   NEGATING THE NEW REAL PART.                                        *
*                                                                      *
*   TO CHANGE THE FFT LENGTH, ONLY THE PARAMETERS IN THE HEADER OF     *
*   TWIDIKBR.ASM AND THE INPUT AND OUTPUT VECTOR LENGTHS NEED TO BE    *
*   ALTERED.                                                           *
**************************************************************************
*                                                                      *
*                                        +                             *
*  AR + j AI ------------------------------------------- AR' + j AI'   *
*                                    \       / +                       *
*                                     \     /                          *
*                                      \ /                             *
*                                      / \                             *
*                                     /   \                            *
*                                    /       \ +                       *
*  BR + j BI ---- ( COS - j SIN ) ------------------------ BR' + j BI'  *
*                                        -                             *
*                                                                      *
*   TR = BR * COS + BI * SIN                                           *
*   TI = BR * SIN - BI * COS                                           *
*   AR'= AR + TR                                                       *
*   AI'= AI - TI                                                       *
*   BR'= AR - TR                                                       *
*   BI'= AI + TI                                                       *
*                                                                      *
**************************************************************************
```

```
*
        .global  FFT
        .global  N
        .global  NHALB
        .global  NVIERT
        .global  NATCHEL
        .global  M
        .global  SINE
*
        .BSS     INP,2048          ; INPUT VECTOR LENGTH = 2N (DEPENDS
*                                  ;   ON N)
        .BSS     OUTP,2048         ; OUTPUT VECTOR LENGTH = 2N (DEPENDS
*                                  ;   ON N)
*
        .text
*
FFTSIZ  .word    N
FG4M2   .word    NVIERT-2
FG4M3   .word    NVIERT-3
FG8M2   .word    NATCHEL-2
FG2     .word    NHALB
FG2M3   .word    NHALB-3
LOGFFT  .word    M
SINTAB  .word    SINE
SINTM1  .word    SINE-1
SINTP2  .word    SINE+2
INPUT   .word    INP
INPUTP2 .word    INP+2
OUTPUT  .word    OUTP

*   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
*
FFT:    LDP      FFTSIZ           ; LOAD PAGE POINTER
        LDI      @FG2,IR0         ; IR0 = N/2 = OFFSET BETWEEN INPUTS
        LDI      @SINTAB,AR7      ; AR7 POINTS TO TWIDDLE FACTOR 1
        LDI      @INPUT,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            ; ADDRESS OFFSET
        LSH      -1,IR0           ; IR0 = N/4 = NUMBER OF R4-BUTTERFLIES
        LDI      IR0,RC
        SUBI     2,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     *AR3++,*AR7,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++
        ADDF     R0,R2,R4         ; AI' = R4 = R2 + R0
*
*   RADIX-4 BUTTERFLY LOOP
*
        RPTB     BLK1
        MPYF     *AR2--,*AR7,R0   ; R0 = CR , (BI' = R2 = R2 - R0)
||      SUBF     R0,R2,R2
        MPYF     *AR1++,*AR7,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     *AR3++,*AR7,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 OF STAGE 2
*

        LDI     @FG2,IR1
        LDI     IR0,AR5
        SUBI    1,AR5
        LDI     1,AR6

STUFE   LDI     @SINTAB,AR7     ; POINTER TO TWIDDLE FACTOR
        LDI     0,AR4           ; GROUP COUNTER
        LDI     @INPUT,AR0      ; UPPER REAL BUTTERFLY INPUT
        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
        MPYF    *AR1++,*AR7--,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++
        LDI     AR5,RC
*****************************************************************
*                                                               *
*   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                                            *
*                                                               *
*****************************************************************
*
        RPTB    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
        MPYF    *AR1++,*AR7++,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++
        LDI     AR5,RC
```

```
*****************************************************************
*                                                               *
*   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                                            *
*                                                               *
*****************************************************************
*
        RPTB    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 BUTTERFLY GROUP
*
        CMPI    4,IR0               ; JUMP OUT AFTER LD(N)-3 STAGE
        BNZ     STUFE
*
*
*    SECOND TO LAST STAGE
*
        LDI     @INPUT,AR0          ; UPPER INPUT
        LDI     AR0,AR2             ; UPPER OUTPUT
        ADDI    IR0,AR0,AR1         ; LOWER INPUT
        LDI     AR1,AR3             ; LOWER OUTPUT
        LDI     @SINTP2,AR7         ; POINTER TO TWIDDLE FACTOR
        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)
        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:
*
        RPTB    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     @INPUT,AR0      ; UPPER INPUT
        LDI     AR0,AR2         ; UPPER OUTPUT
        LDI     @INPUTP2,AR1    ; LOWER INPUT
        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
*
        DDF     *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)
        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:
*
        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
*
*   BIT REVERSAL
*
        LDI     @FFTSIZ,IR0
        LDI     2,IR1
        LDI     @INPUT,AR0
        LDI     @OUTPUT,AR1
        LDI     @FFTSIZ,RC
        SUBI    2,RC
*
        LDF     *++AR0(1),R0
        RPTB    BITRV
        LDF     *AR0++(IR0)b,R1
||      STF     R0,*++AR1(1)
BITRV   LDF     *++AR0(1),R0
||      STF     R1,*AR1++(IR1)
        LDF     *AR0++(IR0)b,R1
||      STF     R0,*++AR1(1)
        STF     R1,*AR1
*
END:    NOP
        NOP
        NOP
        NOP
*
SELF    BR      SELF
        .end
*
```

Appendix A4. Complex, Radix-2 DIT FFT – R2DITB.ASM

```
*                                                                     *
*   APPENDIX A4                                                       *
*                                                                     *
*              COMPLEX, RADIX-2 DIT FFT  :  R2DITB.ASM                *
*              ---------------------------------------                *
*                                                                     *
*   GENERIC PROGRAM FOR A FAST LOOPED-CODE RADIX-2 DIT FFT COMPUTATION*
*                        ON THE TMS320C30                           *
*                                                                     *
*   WRITTEN BY: RAIMUND MEYER, KARL SCHWARZ                24.07.89   *
*               LEHRSTUHL FUER NACHRICHTENTECHNIK                     *
*               UNIVERSITAET ERLANGEN-NUERNBERG                       *
*               CAUERSTRASSE 7, D-8520 ERLANGEN, FRG                  *
*                                                                     *
*   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.                          *
*                                                                     *
*   FOR THIS PROGRAM THE MINIMUM FFT LENGTH 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.                                    *
*                                                                     *
*********************************************************************
*                                                                     *
*   EXAMPLE FOR A 1024-POINT FFT (WITH BIT REVERSAL) :                *
*                                                                     *
*   MEMORY SIZE :                                                     *
*        PROG                        =   231 WORDS                  *
*        DATA                        =   512 WORDS                  *
*                                                                     *
*   CYCLES PER BUTTERFLY :                                            *
*        STAGES 1 AND 2              =     4                          *
*        STAGES 3 TO  8              =     8                          *
*        STAGE 9                     =     8.25                       *
*        STAGE 10                    =    10.5 (DUE TO EXT. MEMORY WAITS) *
*                                                                     *
*   AVERAGE CYCLES/BUTTERFLY         =     7.475                      *
*   TOTAL BUTTERFLYCYCLES            = 38272                          *
*   INITIALIZATION OVERHEAD          =  2185  =  5.4 % OF TOTAL TIME  *
*   TOTAL NUMBER OF INSTRUCTION CYCLES = 40457                        *
*   TOTAL TIME FOR A 1024 POINT FFT  =     2.42 ms (INCLUDING BIT     *
*                                                          REVERSAL)  *
*                                                                     *
*********************************************************************
```

```
*                                                                     *
*   THIS PROGRAM INCUDES FOLLOWING FILES:                             *
*   -------------------------------------                             *
*   THE FILE 'TWIDIKBR.ASM' CONSISTS OF TWIDDLE FACTORS               *
*                                                                     *
*   THE TWIDDLE FACTORS ARE STORED IN BIT REVERSED ORDER AND WITH A TABLE *
*   LENGTH OF N/2 (N = FFTLENGTH).                                    *
*                                                                     *
*   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     *
*   AVAILABLE FFT OF LESS OR EQUAL LENGTH.                            *
*                                                                     *
*   THE MISSING TWIDDLE FACTORS (WN(),WN(),....) ARE GENERATED BY USING *
*   THE SYMMETRY WN(N/4+n) = -j*WN(n). THIS CAN BE EASILY REALIZED, BY *
*   CHANGING REAL- AND IMAGINARY PART OF THE TWIDDLE FACTORS AND BY   *
*   NEGATING THE NEW REAL PART.                                       *
*                                                                     *
*   TO CHANGE THE FFT LENGTH ONLY THE PARAMETERS IN THE HEADER OF     *
*   TWIDIKBR.ASM AND THE INPUT AND OUTPUT VECTOR LENGTHS NEED TO BE   *
*   ALTERED.                                                          *
*********************************************************************
*                                                                     *
*                                        +                            *
*  AR + j AI ------------------------------------------- AR' + j AI'  *
*                                   \       / +                       *
*                                    \     /                          *
*                                     \ /                             *
*                                     / \                             *
*                                    /     \                          *
*                                   /       \ +                       *
*  BR + j BI ---- ( COS - j SIN ) ------------------------ BR' + j BI' *
*                                        -                            *
*  TR = BR * COS + BI * SIN                                           *
*  TI = BR * SIN - BI * COS                                           *
*  AR'= AR + TR                                                       *
*  AI'= AI - TI                                                       *
*  BR'= AR - TR                                                       *
*  BI'= AI + TI                                                       *
*                                                                     *
*********************************************************************
```

```
*
        .global  FFT
        .global  N
        .global  NHALB
        .global  NVIERT
        .global  NACHTEL
        .global  M
        .global  SINE

        .bss     INP,2048          ; INPUT VECTOR LENGTH = 2N (DEPENDS
*                                  ;   ON N)
        .bss     OUTP,2048         ; OUTPUT VECTOR LENGTH = 2N (DEPENDS
*                                  ;   ON N)
*
        .text
*
FFTSIZ  .word    N
FG4M2   .word    NVIERT-2
FG4M3   .word    NVIERT-3
FG8M2   .word    NACHTEL-2
FG2     .word    NHALB
FG2M3   .word    NHALB-3
LOGFFT  .word    M
SINTAB  .word    SINE
SINTM1  .word    SINE-1
SINTP2  .word    SINE+2
INPUT   .word    INP
INPUTP2 .word    INP+2
OUTPUT  .word    OUTP
OUTP1   .word    OUTP+1
*
*   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
*
FFT:    LDP      FFTSIZ           ; LOAD PAGE POINTER
        LDI      @FG2,IR0         ; IR0 = N/2 = OFFSET BETWEEN INPUTS
        LDI      @SINTAB,AR7      ; AR7 POINTS TO TWIDDLE FACTOR 1
        LDI      @INPUT,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            ; ADDRESS OFFSET
        LSH      -1,IR0           ; IR0 = N/4 = NUMBER OF R4-BUTTERFLIES
        LDI      IR0,RC
        SUBI     2,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     *AR3++,*AR7,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++
        ADDF     R0,R2,R4         ; AI' = R4 = R2 + R0
*
*   RADIX-4 BUTTERFLY LOOP
*
        RPTB     BLK1
        MPYF     *AR2--,*AR7,R0   ; R0 = CR , (BI' = R2 = R2 - R0)
||      SUBF     R0,R2,R2
        MPYF     *AR1++,*AR7,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     *AR3++,*AR7,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
        LDI     IR0,AR5
        SUBI    1,AR5
        LDI     1,AR6
*
STUFE   LDI     @SINTAB,AR7     ; POINTER TO TWIDDLE FACTOR
        LDI     0,AR4           ; GROUP COUNTER
        LDI     @INPUT,AR0      ; UPPER REAL BUTTERFLY INPUT
        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
        MPYF    *AR1++,*AR7--,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++

        LDI     AR5,RC
*
*   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*
*
        RPTB    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
        MPYF    *AR1++,*AR7++,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++
        LDI     AR5,RC
*
*   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
*
        RPTB    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 BUTTERFLY GROUP
*
        CMPI    4,IR0               ; JUMP OUT AFTER LD(N)-3 STAGE
        BNZ     STUFE
*
*   SECOND TO LAST STAGE
*
        LDI     @INPUT,AR0          ; UPPER INPUT
        LDI     AR0,AR2             ; UPPER OUTPUT
        ADDI    IR0,AR0,AR1         ; LOWER INPUT
        LDI     AR1,AR3             ; LOWER OUTPUT
        LDI     @SINTP2,AR7         ; POINTER TO TWIDDLE FACTOR
        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^N/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^N/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)
        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:
*
        RPTB    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 WITH INTEGRATED BIT REVERSAL
*
        LDI     @INPUT,AR0          ; UPPER INPUT
        LDI     @OUTPUT,AR2         ; REAL OUTPUT !!!
        LDI     @INPUTP2,AR1        ; LOWER INPUT
        LDI     @OUTP1,AR3          ; IMAGINARY OUTPUT !!!
        LDI     @SINTP2,AR7         ; POINTER TO TWIDDLE FACTORS
        LDI     @FFTSIZ,IR0         ; BIT REVERSAL
        LDI     3,IR1               ; 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
        SUBF    *AR1,*AR0,R4        ; BI' = R4 = AI - BI
        ADDF    *AR1++(IR1),*AR0++(IR1),R5   ; AI' = R5 = AI + BI
*
*   2. BUTTERFLY: w^M/4
*
        UBF     *+AR1,*AR0,R3       ; BR' = R3 = AR - BI
        LDF     *-AR7,R1            ; R1 = 0 (FOR INNER LOOP)
||      LDF     *AR1++,R0           ; R0 = BR (FOR INNER LOOP)
        ADDF    *AR1++(IR1),*AR0++,R2   ; AR' = R2 = AR + BI
        STF     R6,*AR2++(IR0)b     ; (AR' = R6)
||      STF     R5,*AR3++(IR0)b     ; (AI' = R5)
        STF     R7,*AR2++(IR0)b     ; (BR' = R7)
*
*   3. TO M. BUTTERFLY:
*
        PTB     BFLEND
*
*   17 CYCLES IF FFT SIZE <1024 DUE TO THE USE OF INTERNAL MEMORY FOR BIT
*   REVERSAL, 21 CYCLES IF FFT SIZE = 1024 DUE TO THE USE OF EXTERNAL MEMORY
*   FOR BIT REVERSAL
*
        LDF     *AR7++,R7           ; R7 = COS , ((BI' = R4))
||      STF     R4,*AR3++(IR0)B
        LDF     *AR7++,R6           ; R6 = SIN , (AR' = R2)
||      STF     R2,*AR2++(IR0)B
        MPYF    *+AR1,R6,R5         ; R5 = BI * SIN , (BR' = R3)
||      STF     R3,*AR2++(IR0)B
        ADDF    R1,R0,R2            ; (R2 = TI = R0 + R1)
        MPYF    *AR1,R7,R0          ; R0 = BR * COS , (AI' = R3 = AI - TI)
||      SUBF    R2,*AR0,R3
        ADDF    R2,*AR0++(IR1),R4   ; (BI' = R4 = AI + TI , AI' = R3)
||      STF     R3,*AR3++(IR0)B
        ADDF    R0,R5,R3            ; R3 = TR = R0 + R5
        MPYF    *AR1++,R6,R0        ; R0 = BR * SIN , AR' = R2 = AR + TR
```

```
||      ADDF    R3,*AR0,R2
        MPYF    *AR1++(IR1),R7,R1   ; R1 = BI * COS , (BI' = R4)
||      STF     R4,*AR3++(IR0)B
        SUBF    R3,*AR0++,R3        ; BR' = R3 = AR - TR , AR' = R2
||      STF     R2,*AR2++(IR0)B
*
        MPYF    *+AR1,R7,R5         ; R5 = BI * COS , (BR' = R3)
||      STF     R3,*AR2++(IR0)B
        SUBF    R1,R0,R2            ; (R2 = TI = R0 - R1)
        MPYF    *AR1,R6,R0          ; R0 = BR * SIN , (AI' = R3 = AI - TI)
||      SUBF    R2,*AR0,R3
        ADDF    R2,*AR0++(IR1),R4   ; (BI' = R4 = AI + TI , AI' = R3)
||      STF     R3,*AR3++(IR0)B
        SUBF    R0,R5,R3            ; R3 = TR = R0 - R5
        MPYF    *AR1++,R7,R0        ; R0 = BR * COS , AR' = R2 = AR + TR
||      ADDF    R3,*AR0,R2
BFLEND  MPYF    *AR1++(IR1),R6,R1   ; R1 = BI * SIN , BR' = R3 = AR - TR
||      SUBF    R3,*AR0++,R3
*
*   CLEAR PIPELINE
*
        STF     R2,*AR2++(IR0)B     ; AR' = R2 , (BI' = R4)
||      STF     R4,*AR3++(IR0)B
        ADDF    R1,R0,R2            ; R2 = TI = R0 + R1
        SUBF    R2,*AR0,R3          ; AI' = R3 = AI - TI , BR' = R3
||      STF     R3,*AR2
        ADDF    R2,*AR0,R4          ; BI' = R4 = AI + TI , AI' = R3
||      STF     R3,*AR3++(IR0)B
        STF     R4,*AR3             ; BI' = R4
*
*   END OF FFT
*
END:    NOP
        NOP
        NOP
        NOP
*
SELF    BR      SELF
        .end
*
```

Appendix A5. TWID1KBR.ASM – Table With Twiddle Factors for a FFT up to a Length of 1024 Complex Points.

```
*************************************************************
*                                                           *
*   APPENDIX A5                                             *
*                                                           *
*   TITLE: TWID1KBR.ASM                                     *
*                                                           *
*   TABLE WITH TWIDDLE FACTORS FOR A FFT UP TO A LENGTH OF 1024 COMPLEX   *
*   POINTS.                                                 *
*                                                           *
*   FILE TO BE LINKED WITH THE SOURCE CODE : R2DIT.ASM OR R2DITB.ASM      *
*                                                           *
*   WRITTEN BY : RAIMUND MEYER AND KARL SCHWARZ          14.07.89         *
*                LEHRSTUHL FUER NACHRICHTENTECHNIK                        *
*                UNIVERSITAET ERLANGEN-NUERNBERG                          *
*                                                           *
*   LENGTH OF TWIDDLE FACTOR TABLE : 512 REAL VALUES (=1024 FFT)          *
*                                                           *
*************************************************************
*
        .global  sine
        .global  n
        .global  nhalb
        .global  nviert
        .global  nachtel
        .global  m
*
n       .set     1024            ; FFT-LENGTH n
nhalb   .set     512             ; n/2
nviert  .set     256             ; n/4
nachtel .set     128             ; n/8
m       .set     10              ; NUMBER OF STAGES = ld(n)
*
* ANOTHER EXAMPLE OF FFT-LENGTH n = 32:
* ONLY THE FIRST 16 VALUES OF THE TABLE ARE NEEDED
*
*n       .set     2
*nhalb   .set     16
*nviert  .set     8
*nachtel .set     4
*m       .set     5
*
        .data
*
sine
        .float   1.00000000000000e+000
        .float   0.00000000000000e+000
        .float   7.07106781186548e-001
        .float   7.07106781186548e-001
        .float   9.23879532511287e-001
        .float   3.82683432365090e-001
        .float   3.82683432365090e-001
        .float   9.23879532511287e-001
        .float   9.80785280403230e-001
        .
        .
        .
        .float   7.11432195745216e-001
        .float   7.02754744457225e-001
        .float   6.13588464915452e-003
        .float   9.99981175282601e-001
```

Appendix B. Radix-4 Complex FFT

Appendix B1. Generic Program to Do a Looped-Code Radix-4 FFT on the TMS320C30

```
*************************************************************************
*
*    APPENDIX B1
*
*   GENERIC PROGRAM TO DO A LOOPED-CODE RADIX-4 FFT COMPUTATION ON THE
*   TMS320C30.
*
*   THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 117. THE COMPLEX
*   DATA RESIDE IN INTERNAL MEMORY, AND THE COMPUTATION IS DONE IN-PLACE.
*
*   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 LOG4(N) ARE
*   DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED DURING LINKING.
*
*   IN ORDER TO HAVE THE FINAL RESULT IN BIT-REVERSED ORDER, THE TWO MIDDLE
*   BRANCHES OF THE RADIX-4 BUTTERFLY ARE INTERCHANGED DURING STORAGE. NOTE
*   THIS DIFFERENCE WHEN COMPARING WITH THE PROGRAM IN P. 117 OF THE BURRUS
*   AND PARKS BOOK.
*
*   AUTHOR: PANOS E. PAPAMICHALIS
*           TEXAS INSTRUMENTS                          AUGUST 23, 1987
*
*************************************************************************
*
        .GLOBL  FFT             ; ENTRY POINT FOR EXECUTION
        .GLOBL  N               ; FFT SIZE
        .GLOBL  M               ; LOG4(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
*
INP     .USECT  "IN",1024       ; MEMORY WITH INPUT DATA
*
        .TEXT
*
*   INITIALIZE
*
        .WORD   FFT             ; STARTING LOCATION OF THE PROGRAM
*
        .SPACE  100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
*
TEMP    .WORD   $+2
STORE   .WORD   FFTSIZ          ; BEGINNING OF TEMP STORAGE AREA
        .WORD   N
        .WORD   M
        .WORD   SINE
        .WORD   INP
*
        .BSS    FFTSIZ,1        ; FFT SIZE
        .BSS    LOGFFT,1        ; LOG4(FFTSIZ)
        .BSS    SINTAB,1        ; SINE/COSINE TABLE BASE
        .BSS    INPUT,1         ; AREA WITH INPUT DATA TO PROCESS
        .BSS    STAGE,1         ; FFT STAGE #
        .BSS    RPTCNT,1        ; REPEAT COUNTER
        .BSS    IEINDX,1        ; IE INDEX FOR SINE/COSINE
        .BSS    LPCNT,1         ; SECOND-LOOP COUNT
        .BSS    JT,1            ; JT COUNTER IN PROGRAM, P. 117
        .BSS    IA1,1           ; IA1 INDEX IN PROGRAM, P. 117
*
FFT:
*                               ; INITIALIZE DATA LOCATIONS
        LDP     TEMP            ; COMMAND TO LOAD DATA PAGE POINTER
        LDI     @TEMP,AR0
        LDI     @STORE,AR1
        LDI     *AR0++,R0       ; XFER DATA FROM ONE MEMORY TO THE
*                               ;   OTHER
        STI     R0,*AR1++
        LDI     *AR0++,R0
        STI     R0,*AR1++
        LDI     *AR0++,R0
        STI     R0,*AR1++
        LDI     *AR0,R0
        STI     R0,*AR1
*
        LDP     FFTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
        LDI     @FFTSIZ,R0
        LDI     @FFTSIZ,IR0
        LDI     @FFTSIZ,IR1
        LDI     0,AR7
        STI     AR7,@STAGE      ; @STAGE HOLDS THE CURRENT STAGE
*                               ;   NUMBER
        LSH     1,IR0           ; IR0=2*N1 (BECAUSE OF REAL/IMAG)
        LSH     -2,IR1          ; IR1=N/4, POINTER FOR SIN/COS TABLE
        LDI     1,AR7
        STI     AR7,@RPTCNT     ; INITIALIZE REPEAT COUNTER OF FIRST
*                               ;   LOOP
        LSH     -2,R0
        STI     AR7,@IEINDX     ; INITIALIZE IE INDEX
        ADDI    2,R0
        STI     R0,@JT          ; JT=R0/2+2
        SUBI    2,R0
        LSH     1,R0            ; R0=N2
*
*   OUTER LOOP
*
LOOP:
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        ADDI    R0,AR0,AR1      ; AR1 POINTS TO X(I1)
        ADDI    R0,AR1,AR2      ; AR2 POINTS TO X(I2)
        ADDI    R0,AR2,AR3      ; AR3 POINTS TO X(I3)
        LDI     @RPTCNT,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
*   FIST LOOP
*
        RPTB    BLK1
        ADDF    *+AR0,*+AR2,R1  ; R1=Y(I)+Y(I2)
        ADDF    *+AR3,*+AR1,R3  ; R3=Y(I1)+Y(I3)
        ADDF    R3,R1,R6        ; R6=R1+R3
```

```
        SUBF    *++AR2,*++AR0,R4    ; R4=Y(I)-Y(I2)
        STF     R6,*++AR0           ; Y(I)=R1+R3
        SUBF    R3,R1               ; R1=R1-R3
        LDF     *AR2,R5             ; R5=X(I2)
||      LDF     *++AR1,R7           ; R7=Y(I1)
        ADDF    *AR3,*AR1,R3        ; R3=X(I1)+X(I3)
        ADDF    R5,*AR0,R1          ; R1=X(I)+X(I2)
||      STF     R1,*++AR1           ; Y(I1)=R1-R3
        ADDF    R3,R1,R6            ; R6=R1+R3
        SUBF    R5,*AR0,R2          ; R2=X(I)-X(I2)
||      STF     R6,*AR0++(IR0)      ; X(I)=R1+R3
        SUBF    R3,R1               ; R1=R1-R3
        SUBF    *AR3,*AR1,R6        ; R6=X(I1)-X(I3)
        SUBF    R7,*++AR3,R3        ; -R3=Y(I1)-Y(I3)  !!!
||      STF     R1,*AR1++(IR0)      ; X(I1)=R1-R3
        SUBF    R6,R4,R5            ; R5=R4-R6
        ADDF    R6,R4               ; R4=R4+R6
        STF     R5,*++AR2           ; Y(I2)=R4-R6
||      STF     R4,*++AR3           ; Y(I3)=R4+R6
        SUBF    R3,R2,R5            ; R5=R2-R3  !!!
        ADDF    R3,R2               ; R2=R2+R3  !!!
BLK1    STF     R5,*AR2++(IR0)      ; X(I2)=R2-R3  !!!
||      STF     R2,*AR3++(IR0)      ; X(I3)=R2+R3  !!!
*
*   IF THIS IS THE LAST STAGE, YOU ARE DONE
*
        LDI     @STAGE,AR7
        ADDI    1,AR7
        CMPI    @LOGFFT,AR7
        BZD     END
        STI     AR7,@STAGE          ; CURRENT FFT STAGE
*
*   MAIN INNER LOOP
*
        LDI     1,AR7
        STI     AR7,@IA1            ; INIT IA1 INDEX
        LDI     2,AR7
        STI     AR7,@LPCNT          ; INIT LOOP COUNTER FOR INNER LOOP
INLOP:
        LDI     2,AR6               ; INCREMENT INNER LOOP COUNTER
        ADDI    @LPCNT,AR6
        LDI     @LPCNT,AR0
        LDI     @IA1,AR7
        ADDI    @IEINDX,AR7         ; IA1=IA1+IE
        ADDI    @INPUT,AR0          ; (X(I),Y(I)) POINTER
        STI     AR7,@IA1
        ADDI    R0,AR0,AR1          ; (X(I1),Y(I1)) POINTER
        STI     AR6,@LPCNT
        ADDI    R0,AR1,AR2          ; (X(I2),Y(I2)) POINTER
        ADDI    R0,AR2,AR3          ; (X(I3),Y(I3)) POINTER
        LDI     @RPTCNT,RC
        SUBI    1,RC                ; RC SHOULD BE ONE LESS THAN DESIRED #
        CMPI    @JT,AR6             ; IF LPCNT=JT, GO TO
        BZD     SPCL                ;   SPECIAL BUTTERFLY
        LDI     @IA1,AR7
        LDI     @IA1,AR4
        ADDI    @SINTAB,AR4         ; CREATE COSINE INDEX AR4
        ADDI    AR4,AR7,AR5
        SUBI    1,AR5               ; IA2=IA1+IA1-1
        ADDI    AR7,AR5,AR6
        SUBI    1,AR6               ; IA3=IA2+IA1-1
*
*   SECOND LOOP
*
        RPTB    BLK2
        ADDF    *++AR2,*++AR0,R3    ; R3=Y(I)+Y(I2)
        ADDF    *++AR3,*++AR1,R5    ; R5=Y(I1)+Y(I3)
        ADDF    R5,R3,R6            ; R6=R3+R5
        SUBF    *++AR2,*++AR0,R4    ; R4=Y(I)-Y(I2)
        SUBF    R5,R3               ; R3=R3-R5
        ADDF    *AR2,*AR0,R1        ; R1=X(I)+X(I2)
        ADDF    *AR3,*AR1,R5        ; R5=X(I1)+X(I3)
        MPYF    R3,*++AR5(IR1),R6   ; R6=R3*CO2
||      STF     R6,*++AR0           ; Y(I)=R3+R5
        ADDF    R5,R1,R7            ; R7=R1+R5
        SUBF    *AR2,*AR0,R2        ; R2=X(I)-X(I2)
        SUBF    R5,R1               ; R1=R1-R5
        MPYF    R1,*AR5,R7          ; R7=R1*SI2
||      STF     R7,*AR0++(IR0)      ; X(I)=R1+R5
        SUBF    R7,R6               ; R6=R3*CO2-R1*SI2
        SUBF    *++AR3,*++AR1,R5    ; R5=Y(I1)-Y(I3)
        MPYF    R1,*++AR5(IR1),R7   ; R7=R1*CO2
||      STF     R6,*++AR1           ; Y(I1)=R3*CO2-R1*SI2
        MPYF    R3,*AR5,R6          ; R6=R3*SI2
        ADDF    R7,R6               ; R6=R1*CO2+R3*SI2
        ADDF    R5,R2,R1            ; R1=R2+R5
        SUBF    R5,R2               ; R2=R2-R5
        SUBF    *AR3,*AR1,R5        ; R5=X(I1)-X(I3)
        SUBF    R5,R4,R3            ; R3=R4-R5
        ADDF    R5,R4               ; R4=R4+R5
        MPYF    R3,*++AR4(IR1),R6   ; R6=R3*CO1
||      STF     R6,*AR1++(IR0)      ; X(I1)=R1*CO2+R3*SI2
        MPYF    R1,*AR4,R7          ; R7=R1*SI1
        SUBF    R7,R6               ; R6=R3*CO1-R1*SI1
        MPYF    R1,*++AR4(IR1),R6   ; R6=R1*CO1
||      STF     R6,*++AR2           ; Y(I2)=R3*CO1-R1*SI1
        MPYF    R3,*AR4,R7          ; R7=R3*SI1
        ADDF    R7,R6               ; R6=R1*CO1+R3*SI1
        MPYF    R4,*++AR6(IR1),R6   ; R6=R4*CO3
||      STF     R6,*AR2++(IR0)      ; X(I2)=R1*CO1+R3*SI1
        MPYF    R2,*AR6,R7          ; R7=R2*SI3
        SUBF    R7,R6               ; R6=R4*CO3-R2*SI3
        MPYF    R2,*++AR6(IR1),R6   ; R6=R2*CO3
||      STF     R6,*++AR3           ; Y(I3)=R4*CO3-R2*SI3
        MPYF    R4,*AR6,R7          ; R7=R4*SI3
        ADDF    R7,R6               ; R6=R2*CO3+R4*SI3
BLK2    STF     R6,*AR3++(IR0)      ; X(I3)=R2*CO3+R4*SI3
*
```

```
        CMPI    @LPCNT,R0
        BP      INLOP           ; LOOP BACK TO THE INNER LOOP
        BR      CONT
*
*   SPECIAL BUTTERFLY FOR W=J
*
SPCL    LDI     IR1,AR4
        LSH     -1,AR4          ; POINT TO SIN(45)
        ADDI    @SINTAB,AR4     ; CREATE COSINE INDEX AR4=C021
*
        RPTB    BLK3
        ADDF    *AR2,*AR0,R1    ; R1=X(I)+X(I2)
        SUBF    *AR2,*AR0,R2    ; R2=X(I)-X(I2)
        ADDF    *+AR2,*+AR0,R3  ; R3=Y(I)+Y(I2)
        SUBF    *+AR2,*+AR0,R4  ; R4=Y(I)-Y(I2)
        ADDF    *AR3,*AR1,R5    ; R5=X(I1)+X(I3)
        SUBF    R1,R5,R6        ; R6=R5-R1
        ADDF    R5,R1           ; R1=R1+R5
        ADDF    *+AR3,*+AR1,R5  ; R5=Y(I1)+Y(I3)
        SUBF    R5,R3,R7        ; R7=R3-R5
        ADDF    R5,R3           ; R3=R3+R5
        STF     R3,*+AR0        ; Y(I)=R3+R5
||      STF     R1,*AR0++(IR0)  ; X(I)=R1+R5
        SUBF    *AR3,*AR1,R1    ; R1=X(I1)-X(I3)
        SUBF    *+AR3,*+AR1,R3  ; R3=Y(I1)-Y(I3)
        STF     R6,*+AR1        ; Y(I1)=R5-R1
||      STF     R7,*AR1++(IR0)  ; X(I1)=R3-R5
        ADDF    R3,R2,R5        ; R5=R2+R3
        SUBF    R2,R3,R2        ; R2=-R2+R3 !!!
        SUBF    R1,R4,R3        ; R3=R4-R1
        ADDF    R1,R4           ; R4=R4+R1
        SUBF    R5,R3,R1        ; R1=R3-R5
        MPYF    *AR4,R1         ; R1=R1*C021
        ADDF    R5,R3           ; R3=R3+R5
        MPYF    *AR4,R3         ; R3=R3*C021
||      STF     R1,*+AR2        ; Y(I2)=(R3-R5)*C021
        SUBF    R4,R2,R1        ; R1=R2-R4  !!!
        MPYF    *AR4,R1         ; R1=R1*C021
||      STF     R3,*AR2++(IR0)  ; X(I2)=(R3+R5)*C021
        ADDF    R4,R2           ; R2=R2+R4  !!!
        MPYF    *AR4,R2         ; R2=R2*C021  !!!
BLK3    STF     R1,*+AR3        ; Y(I3)=-(R4-R2)*C021  !!!
||      STF     R2,*AR3++(IR0)  ; X(I3)=(R4+R2)*C021  !!!
*
*
        CMPI    @LPCNT,R0
        BPD     INLOP           ; LOOP BACK TO THE INNER LOOP
*
CONT    LDI     @RPTCNT,AR7
        LDI     @IEINDX,AR6
        LSH     2,AR7           ; INCREMENT REPEAT COUNTER FOR NEXT
*                               ;   TIME
        STI     AR7,@RPTCNT
        LSH     2,AR6           ; IE=4*IE
        STI     AR6,@IEINDX
        LDI     R0,IR0          ; N1=N2
        LSH     -3,R0
        ADDI    2,R0
        STI     R0,@JT          ; JT=N2/2+2
        SUBI    2,R0
        LSH     1,R0            ; N2=N2/4
        BR      LOOP            ; NEXT FFT STAGE
*
*   STORE RESULT OUT USING BIT-REVERSED ADDRESSING
*
END:    LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0     ; IR0=SIZE OF FFT=N
        LDI     2,IR1
        LDI     @INPUT,AR0
        LDP     STORE
        LDI     @STORE,AR1
*
        RPTB    BITRV
        LDF     *+AR0(1),R0
||      LDF     *AR0++(IR0)B,R1
BITRV   STF     R0,*+AR1(1)
||      STF     R1,*AR1++(IR1)
*
SELF    BR      SELF            ; BRANCH TO ITSELF AT THE END
        .END
```

Appendix B2. fft_4—Radix-4 Complex FFT to Be Called as a C Function

```
*
*   APPENDIX B2
*
*   NAME: fft_4 --- RADIX-4 COMPLEX FFT TO BE CALLED AS A C FUNCTION.
*
*   SYNOPSIS:
*       int    fft_4(N, M, DATA)
*       int    N        FFT SIZE: N=4**M
*       int    M        NUMBER OF STAGES = LOG4(N)
*       float  *data    ARRAY WITH INPUT AND OUTPUT DATA
*
*   DESCRIPTION:
*       GENERIC FUNCTION TO DO A RADIX-4 FFT COMPUTATION ON THE TMS320C30.
*       THE DATA ARRAY IS 2*N-LONG, WITH REAL AND IMAGINARY VALUES ALTER-
*       NATING. THE PROGRAM IS BASED ON THE FORTRAN PROGRAM IN THE BURRUS
*       AND PARKS BOOK, P. 117.
*
*       IN ORDER TO HAVE THE FINAL RESULT IN BIT-REVERSED ORDER, THE TWO
*       MIDDLE BRANCHES OF THE RADIX-4 BUTTERFLY ARE INTERCHANGED DURING
*       STORAGE. NOTE THIS DIFFERENCE WHEN COMPARING WITH THE PROGRAM ON
*       P. 117. THE COMPUTATION IS DONE IN-PLACE, AND THE ORIGINAL DATA IS
*       DESTROYED. BIT REVERSAL IS IMPLEMENTED AT THE END OF THE FUNCTION.
*       IF THIS IS NOT NECESSARY, THIS PART CAN BE COOOENTED OUT. THE
*       SINE/COSINE TABLE FOR THE TWIDDLE FACTORS IS EXPECTED TO BE SUPPLIED
*       DURING LINK TIME, AND IT SHOULD HAVE THE FOLLOWING FORMAT:
*
*               .global   _sine
*               .data
*       _sine   .float    value1  = sin(0*2*pi/N)
*               .float    value2  = sin(1*2*pi/N)
*       ......
*               .float    value(5N/4) = sin((5*N/4-1)*2*pi/N)
*
*       THE VALUES value1, value2, ETC., ARE THE SINE WAVE VALUES. FOR AN
*       N-POINT FFT, THERE ARE N+N/4 VALUES FOR A FULL AND A QUARTER PERIOD
*       OF THE SINE WAVE. IN THIS WAY, A FULL SINE AND COSINE PERIOD ARE
*       AVAILABLE (SUPERIMPOSED).
*
*   STACK STRUCTURE UPON THE CALL:
*                 +--------------+
*       -FP(4)    |    DATA      |
*       -FP(3)    |    M         |
*       -FP(2)    |    N         |
*       -FP(1)    | RETURN ADDR  |
*       -FP(0)    |   OLD FP     |
*                 +--------------+
*
*   REGISTERS USED: R0, R1, R2, R3, R4, R5, R6, R7, AR0, AR1, AR2, AR3, AR4,
*                   AR5, AR6, AR7, IR0, IR1, RS, RE, RC
*
*   AUTHOR: PANOS E. PAPAMICHALIS
*         TEXAS INSTRUMENTS                                  OCTOBER 13, 1987
*
*********************************************************************************

*
FP      .SET    AR3
*
        .GLOBL  _FFT_4          ; ENTRY POINT FOR EXECUTION
        .GLOBL  _SINE           ; ADDRESS OF SINE TABLE
*
        .BSS    FFTSIZ,1
        .BSS    LOGFFT,1
        .BSS    INPUT,1
*
        .TEXT
*
SINTAB  .word   _SINE
*
*   INITIALIZE C FUNCTION
*
_fft_4: PUSH    FP              ; SAVE DEDICATED REGISTERS
        LDI     SP,FP
        PUSH    R4
        PUSH    R5
        PUSHF   R6
        PUSHF   R7
        PUSH    AR4
        PUSH    AR5
        PUSH    AR6
        PUSH    AR7
*
        LDI     *-FP(2),R0      ; MOVE ARGUMENTS TO LOCATIONS MATCHING
        STI     R0,@FFTSIZ      ; THE NAMES IN THE PROGRAM
        LDI     *-FP(3),R0
        STI     R0,@LOGFFT
        LDI     *-FP(4),R0
        STI     R0,@INPUT
*
*   INITIALIZE FFT ROUTINE
*
        .BSS    STAGE,1         ; FFT STAGE #
        .BSS    RPTCNT,1        ; REPEAT COUNTER
        .BSS    IEINDX,1        ; IE INDEX FOR SINE/COSINE
        .BSS    LPCNT,1         ; SECOND-LOOP COUNT
        .BSS    JT,1            ; JT COUNTER IN PROGRAM, P. 117
        .BSS    IA1,1           ; IA1 INDEX IN PROGRAM, P. 117
*
        LDI     @FFTSIZ,R0
        LDI     @FFTSIZ,IR0
        LDI     @FFTSIZ,IR1
        LDI     0,AR7
        STI     AR7,@STAGE      ; @STAGE HOLDS THE CURRENT STAGE
*                               ;   NUMBER
        LSH     1,IR0           ; IR0=2*N1 (BECAUSE OF REAL/IMAG)
        LSH     -2,IR1          ; IR1=N/4, POINTER FOR SIN/COS TABLE
        LDI     1,AR7
        STI     AR7,@RPTCNT     ; INITIALIZE REPEAT COUNTER OF FIRST
*                               ;   LOOP
```

```
        LSH     -2,R0
        STI     AR7,@IEINDX         ; INITIALIZE IE INDEX
        ADDI    2,R0
        STI     R0,@JT              ; JT=R0/2+2
        SUBI    2,R0
        LSH     1,R0                ; R0=N2
*
*   OUTER LOOP
*
LOOP:
        LDI     @INPUT,AR0          ; AR0 POINTS TO X(I)
        ADDI    R0,AR0,AR1          ; AR1 POINTS TO X(I1)
        ADDI    R0,AR1,AR2          ; AR2 POINTS TO X(I2)
        ADDI    R0,AR2,AR3          ; AR3 POINTS TO X(I3)
        LDI     @RPTCNT,RC
        SUBI    1,RC                ; RC SHOULD BE ONE LESS THAN DESIRED #
*
*   FIST LOOP
*
        RPTB    BLK1
        ADDF    *++AR0,*++AR2,R1    ; R1=Y(I)+Y(I2)
        ADDF    *++AR3,*++AR1,R3    ; R3=Y(I1)+Y(I3)
        ADDF    R3,R1,R6            ; R6=R1+R3
        SUBF    *++AR2,*++AR0,R4    ; R4=Y(I)-Y(I2)
        STF     R6,*++AR0           ; Y(I)=R1+R3
        SUBF    R3,R1               ; R1=R1-R3
        LDF     *AR2,R5             ; R5=X(I2)
||      LDF     *++AR1,R7           ; R7=Y(I1)
        ADDF    *AR3,*AR1,R3        ; R3=X(I1)+X(I3)
        ADDF    R5,*AR0,R1          ; R1=X(I)+X(I2)
||      STF     R1,*++AR1           ; Y(I1)=R1-R3
        ADDF    R3,R1,R6            ; R6=R1+R3
        SUBF    R5,*AR0,R2          ; R2=X(I)-X(I2)
||      STF     R6,*AR0++(IR0)      ; X(I)=R1+R3
        SUBF    R3,R1               ; R1=R1-R3
        SUBF    *AR3,*AR1,R6        ; R6=X(I1)-X(I3)
        SUBF    R7,*++AR3,R3        ; -R3=Y(I1)-Y(I3)  !!!
||      STF     R1,*AR1++(IR0)      ; X(I1)=R1-R3
        SUBF    R6,R4,R5            ; R5=R4-R6
        ADDF    R6,R4               ; R4=R4+R6
        STF     R5,*++AR2           ; Y(I2)=R4-R6
||      STF     R4,*++AR3           ; Y(I3)=R4+R6
        SUBF    R3,R2,R5            ; R5=R2-R3  !!!
        ADDF    R3,R2               ; R2=R2+R3  !!!
BLK1    STF     R5,*AR2++(IR0)      ; X(I2)=R2-R3  !!!
||      STF     R2,*AR3++(IR0)      ; X(I3)=R2+R3  !!!
*
*   IF THIS IS THE LAST STAGE, YOU ARE DONE
*
        LDI     @STAGE,AR7
        ADDI    1,AR7
        CMPI    @LOGFFT,AR7
        BZD     END
        STI     AR7,@STAGE          ; CURRENT FFT STAGE
*
*   MAIN INNER LOOP
*
        LDI     1,AR7
        STI     AR7,@IA1            ; INIT IA1 INDEX
        LDI     2,AR7
        STI     AR7,@LPCNT          ; INIT LOOP COUNTER FOR INNER LOOP
INLOP:
        LDI     2,AR6               ; INCREMENT INNER LOOP COUNTER
        ADDI    @LPCNT,AR6
        LDI     @LPCNT,AR0
        LDI     @IA1,AR7
        ADDI    @IEINDX,AR7         ; IA1=IA1+IE
        ADDI    @INPUT,AR0          ; (X(I),Y(I)) POINTER
        STI     AR7,@IA1
        ADDI    R0,AR0,AR1          ; (X(I1),Y(I1)) POINTER
        STI     AR6,@LPCNT
        ADDI    R0,AR1,AR2          ; (X(I2),Y(I2)) POINTER
        ADDI    R0,AR2,AR3          ; (X(I3),Y(I3)) POINTER
        LDI     @RPTCNT,RC
        SUBI    1,RC                ; RC SHOULD BE ONE LESS THAN DESIRED #
        CMPI    @JT,AR6             ; IF LPCNT=JT, GO TO
        BZD     SPCL                ;   SPECIAL BUTTERFLY
        LDI     @IA1,AR7
        LDI     @IA1,AR4
        ADDI    @SINTAB,AR4         ; CREATE COSINE INDEX AR4
        ADDI    AR4,AR7,AR5
        SUBI    1,AR5               ; IA2=IA1+IA1-1
        ADDI    AR7,AR5,AR6
        SUBI    1,AR6               ; IA3=IA2+IA1-1
*
*   SECOND LOOP
*
        RPTB    BLK2
        ADDF    *++AR2,*++AR0,R3    ; R3=Y(I)+Y(I2)
        ADDF    *++AR3,*++AR1,R5    ; R5=Y(I1)+Y(I3)
        ADDF    R5,R3,R6            ; R6=R3+R5
        SUBF    *++AR2,*++AR0,R4    ; R4=Y(I)-Y(I2)
        SUBF    R5,R3               ; R3=R3-R5
        ADDF    *AR2,*AR0,R1        ; R1=X(I)+X(I2)
        ADDF    *AR3,*AR1,R5        ; R5=X(I1)+X(I3)
        MPYF    R3,*+AR5(IR1),R6    ; R6=R3*CO2
||      STF     R6,*++AR0           ; Y(I)=R3+R5
        ADDF    R5,R1,R7            ; R7=R1+R5
        SUBF    *AR2,*AR0,R2        ; R2=X(I)-X(I2)
        SUBF    R5,R1               ; R1=R1-R5
        MPYF    R1,*AR5,R7          ; R7=R1*SI2
||      STF     R7,*AR0++(IR0)      ; X(I)=R1+R5
        SUBF    R7,R6               ; R6=R3*CO2-R1*SI2
        SUBF    *++AR3,*++AR1,R5    ; R5=Y(I1)-Y(I3)
        MPYF    R1,*+AR5(IR1),R7    ; R7=R1*CO2
||      STF     R6,*++AR1           ; Y(I1)=R3*CO2-R1*SI2
        MPYF    R3,*AR5,R6          ; R6=R3*SI2
        ADDF    R7,R6               ; R6=R1*CO2+R3*SI2
```

```
        ADDF    R5,R2,R1        ; R1=R2+R5
        SUBF    R5,R2           ; R2=R2-R5
        SUBF    *AR3,*AR1,R5    ; R5=X(I1)-X(I3)
        SUBF    R5,R4,R3        ; R3=R4-R5
        ADDF    R5,R4           ; R4=R4+R5
        MPYF    R3,*+AR4(IR1),R6 ; R6=R3*C01
||      STF     R6,*AR1++(IR0)  ; X(I1)=R1*C02+R3*SI2
        MPYF    R1,*AR4,R7      ; R7=R1*SI1
        SUBF    R7,R6           ; R6=R3*C01-R1*SI1
        MPYF    R1,*+AR4(IR1),R6 ; R6=R1*C01
||      STF     R6,*+AR2        ; Y(I2)=R3*C01-R1*SI1
        MPYF    R3,*AR4,R7      ; R7=R3*SI1
        ADDF    R7,R6           ; R6=R1*C01+R3*SI1
        MPYF    R4,*+AR6(IR1),R6 ; R6=R4*C03
||      STF     R6,*AR2++(IR0)  ; X(I2)=R1*C01+R3*SI1
        MPYF    R2,*AR6,R7      ; R7=R2*SI3
        SUBF    R7,R6           ; R6=R4*C03-R2*SI3
        MPYF    R2,*+AR6(IR1),R6 ; R6=R2*C03
||      STF     R6,*+AR3        ; Y(I3)=R4*C03-R2*SI3
        MPYF    R4,*AR6,R7      ; R7=R4*SI3
        ADDF    R7,R6           ; R6=R2*C03+R4*SI3
BLK2    STF     R6,*AR3++(IR0)  ; X(I3)=R2*C03+R4*SI3
*
        CMPI    @LPCNT,R0
        BP      INLOP           ; LOOP BACK TO THE INNER LOOP
        BR      CONT
*
*   SPECIAL BUTTERFLY FOR W=J
*
SPCL    LDI     IR1,AR4
        LSH     -1,AR4          ; POINT TO SIN(45)
        ADDI    @SINTAB,AR4     ; CREATE COSINE INDEX AR4=C021
*
        RPTB    BLK3
        ADDF    *AR2,*AR0,R1    ; R1=X(I)+X(I2)
        SUBF    *AR2,*AR0,R2    ; R2=X(I)-X(I2)
        ADDF    *++AR2,*++AR0,R3 ; R3=Y(I)+Y(I2)
        SUBF    *++AR2,*++AR0,R4 ; R4=Y(I)-Y(I2)
        ADDF    *AR3,*AR1,R5    ; R5=X(I1)+X(I3)
        SUBF    R1,R5,R6        ; R6=R5-R1
        ADDF    R5,R1           ; R1=R1+R5
        ADDF    *++AR3,*++AR1,R5 ; R5=Y(I1)+Y(I3)
        SUBF    R5,R3,R7        ; R7=R3-R5
        ADDF    R5,R3           ; R3=R3+R5
        STF     R3,*++AR0       ; Y(I)=R3+R5
||      STF     R1,*AR0++(IR0)  ; X(I)=R1+R5
        SUBF    *AR3,*AR1,R1    ; R1=X(I1)-X(I3)
        SUBF    *++AR3,*++AR1,R3 ; R3=Y(I1)-Y(I3)
        STF     R6,*++AR1       ; Y(I1)=R5-R1
||      STF     R7,*AR1++(IR0)  ; X(I1)=R3-R5
        ADDF    R3,R2,R5        ; R5=R2+R3
        SUBF    R2,R3,R2        ; R2=-R2+R3 !!!
        SUBF    R1,R4,R3        ; R3=R4-R1
        ADDF    R1,R4           ; R4=R4+R1
        SUBF    R5,R3,R1        ; R1=R3-R5
        MPYF    *AR4,R1         ; R1=R1*C021
        ADDF    R5,R3           ; R3=R3+R5
        MPYF    *AR4,R3         ; R3=R3*C021
||      STF     R1,*+AR2        ; Y(I2)=(R3-R5)*C021
        SUBF    R4,R2,R1        ; R1=R2-R4  !!!
        MPYF    *AR4,R1         ; R1=R1*C021
||      STF     R3,*AR2++(IR0)  ; X(I2)=(R3+R5)*C021
        ADDF    R4,R2           ; R2=R2+R4  !!!
        MPYF    *AR4,R2         ; R2=R2*C021  !!!
BLK3    STF     R1,*+AR3        ; Y(I3)=-(R4-R2)*C021  !!!
||      STF     R2,*AR3++(IR0)  ; X(I3)=(R4+R2)*C021  !!!
*
        CMPI    @LPCNT,R0
        BPD     INLOP           ; LOOP BACK TO THE INNER LOOP
*
CONT    LDI     @RPTCNT,AR7
        LDI     @IEINDX,AR6
        LSH     2,AR7           ; INCREMENT REPEAT COUNTER FOR NEXT
*                               ;   TIME
        STI     AR7,@RPTCNT
        LSH     2,AR6           ; IE=4*IE
        STI     AR6,@IEINDX
        LDI     R0,IR0          ; N1=N2
        LSH     -3,R0
        ADDI    2,R0
        STI     R0,@JT          ; JT=N2/2+2
        SUBI    2,R0
        LSH     1,R0            ; N2=N2/4
        BR      LOOP            ; NEXT FFT STAGE
*
*   DO THE BIT-REVERSING OF THE OUTPUT
*
END:    LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0     ; IR0=SIZE OF FFT=N
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
*
        RPTB    BITRV
        CMPI    AR0,AR1
        BGE     CONT
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
        LDF     *++AR0(1),R0
||      LDF     *++AR1(1),R1
        STF     R0,*++AR1(1)
||      STF     R1,*++AR0(1)
CONT    NOP     *++AR0(2)
BITRV   NOP     *AR1++(IR0)B
*
*   RESTORE THE REGISTER VALUES AND RETURN
```

```
        POP     AR7
        POP     AR6
        POP     AR5
        POP     AR4
        POPF    R7
        POPF    R6
        POP     R5
        POP     R4
        POP     FP

        RETS
```

Appendix C.Radix-2 Real FFT

Appendix C1. Generic Program to Do a Radix-2 Real FFT Computation on the TMS320C30

```
*
*   APPENDIX C1
*
*   GENERIC PROGRAM TO DO A RADIX-2 REAL FFT COMPUTATION ON THE TMS320C30
*
*   THE PROGRAM IS TAKEN FROM THE PAPER BY SORENSEN ET AL., JUNE 1987 ISSUE
*   OF THE TRANSACTIONS ON ASSP.
*
*   THE (REAL) DATA RESIDE IN INTERNAL MEMORY. THE COMPUTATION IS DONE
*   IN-PLACE. THE BIT REVERSAL IS DONE AT THE BEGINNING OF THE PROGRAM.
*
*   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. THE LENGTH OF
*   THE TABLE IS N/4 + N/4 = N/2.
*
*   AUTHOR: PANOS E. PAPAMICHALIS
*           TEXAS INSTRUMENTS                              SEPTEMBER 8, 1987
*
        .GLOBL  FFT             ; ENTRY POINT FOR EXECUTION
        .GLOBL  N               ; FFT SIZE
        .GLOBL  M               ; LOG2(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
*
INP     .USECT  "IN",1024       ; MEMORY WITH INPUT DATA
        .BSS    OUTP,1024       ; MEMORY WITH OUTPUT DATA
*
        .TEXT
*
*   INITIALIZE
*
        .WORD   FFT             ; STARTING LOCATION OF THE PROGRAM
*
        .SPACE  100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
*
FFTSIZ  .WORD   N
LOGFFT  .WORD   M
SINTAB  .WORD   SINE
INPUT   .WORD   INP
OUTPUT  .WORD   OUTP
*
FFT:    LDP     FFTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
*
*   DO THE BIT-REVERSING AT THE BEGINNING
*
        LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0
        LSH     -1,IR0          ; IR0=HALF THE SIZE OF FFT=N/2
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
*
        RPTB    BITRV
        CMPI    AR1,AR0         ; XCHANGE LOCATIONS ONLY
        BGE     CONT            ; IF AR0<AR1
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
CONT    NOP     *AR0++
BITRV   NOP     *AR1++(IR0)B
*
*   LENGTH-TWO BUTTERFLIES
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        LDI     IR0,RC          ; REPEAT N/2 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        RPTB    BLK1
        ADDF    *+AR0,*AR0++,R0 ; R0=X(I)+X(I+1)
        SUBF    *AR0,*-AR0,R1   ; R1=X(I)-X(I+1)
BLK1    STF     R0,*-AR0        ; X(I)=X(I)+X(I+1)
||      STF     R1,*AR0++       ; X(I+1)=X(I)-X(I+1)
*
*   FIRST PASS OF THE DO-20 LOOP (STAGE K=2 IN DO-10 LOOP)
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        LDI     2,IR0           ; IR0=2=N2
        LDI     @FFTSIZ,RC
        LSH     -2,RC           ; REPEAT N/4 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        RPTB    BLK2
        ADDF    *+AR0(IR0),*AR0++(IR0),R0   ; R0=X(I)+X(I+2)
        SUBF    *AR0,*-AR0(IR0),R1  ; R1=X(I)-X(I+2)
        NEGF    *+AR0,R0        ; R0=-X(I+3)
||      STF     R0,*-AR0(IR0)   ; X(I)=X(I)+X(I+2)
BLK2    STF     R1,*AR0++(IR0)  ; X(I+2)=X(I)-X(I+2)
||      STF     R0,*+AR0        ; X(I+3)=-X(I+3)
*
*   MAIN LOOP (FFT STAGES)
*
        LDI     @FFTSIZ,IR0
        LSH     -2,IR0          ; IR0=INDEX FOR E
        LDI     3,R5            ; R5 HOLDS THE CURRENT STAGE NUMBER
        LDI     1,R4            ; R4=N4
        LDI     2,R3            ; R3=N2
LOOP    LSH     -1,IR0          ; E=E/2
        LSH     1,R4            ; N4=2*N4
        LSH     1,R3            ; N2=2*N2
*
*   INNER LOOP (DO-20 LOOP IN THE PROGRAM)
*
        LDI     @INPUT,AR5      ; AR5 POINTS TO X(I)
INLOP   LDI     IR0,AR0
        ADDI    @SINTAB,AR0     ; AR0 POINTS TO SIN/COS TABLE
        LDI     R4,IR1          ; IR1=N4
```

```
*
        LDI     AR5,AR1
        ADDI    1,AR1               ; AR1 POINTS TO X(I1)=X(I+J)
        LDI     AR1,AR3
        ADDI    R3,AR3              ; AR3 POINTS TO X(I3)=X(I+J+N2)
        LDI     AR3,AR2
        SUBI    2,AR2               ; AR2 POINTS TO X(I2)=X(I-J+N2)
        ADDI    R3,AR2,AR4          ; AR4 POINTS TO X(I4)=X(I-J+N1)
*
        LDF     *AR5++(IR1),R0      ; R0=X(I)
        ADDF    *++AR5(IR1),R0,R1   ; R1=X(I)+X(I+N2)
        SUBF    R0,*++AR5(IR1),R0   ; R0=-X(I)+X(I+N2)
||      STF     R1,*-AR5(IR1)       ; X(I)=X(I)+X(I+N2)
        NEGF    R0                  ; R0=X(I)-X(I+N2)
        NEGF    *++AR5(IR1),R1      ; R1=-X(I+N4+N2)
||      STF     R0,*AR5             ; X(I+N2)=X(I)-X(I+N2)
        STF     R1,*AR5             ; X(I+N4+N2)=-X(I+N4+N2)
*
*   INNERMOST LOOP
*
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1              ; IR1=SEPARATION BETWEEN SIN/COS TBLS
        LDI     R4,RC
        SUBI    2,RC                ; REPEAT N4-1 TIMES
*
        RPTB    BLK3
        MPYF    *AR3,*+AR0(IR1),R0  ; R0=X(I3)*COS
        MPYF    *AR4,*AR0,R1        ; R1=X(I4)*SIN
        MPYF    *AR4,*+AR0(IR1),R1  ; R1=X(I4)*COS
||      ADDF    R0,R1,R2            ; R2=X(I3)*COS+X(I4)*SIN
        MPYF    *AR3,*AR0++(IR0),R0 ; R0=X(I3)*SIN
        SUBF    R0,R1,R0            ; R0=-X(I3)*SIN+X(I4)*COS !!!
        SUBF    *AR2,R0,R1          ; R1=-X(I2)+R0 !!!
        ADDF    *AR2,R0,R1          ; R1=X(I2)+R0 !!!
||      STF     R1,*AR3++           ; X(I3)=-X(I2)+R0 !!!
        ADDF    *AR1,R2,R1          ; R1=X(I1)+R2
||      STF     R1,*AR4--           ; X(I4)=X(I2)+R0 !!!
        SUBF    R2,*AR1,R1          ; R1=X(I1)-R2
||      STF     R1,*AR1++           ; X(I1)=X(I1)+R2
BLK3    STF     R1,*AR2--           ; X(I2)=X(I1)-R2
*
        SUBI    @INPUT,AR5
        ADDI    R4,AR5              ; AR5=I+N1
        CMPI    @FFTSIZ,AR5
        BLTD    INLOP               ; LOOP BACK TO THE INNER LOOP
        ADDI    @INPUT,AR5
        NOP
        NOP
*
        ADDI    1,R5
        CMPI    @LOGFFT,R5
        BLE     LOOP
        NOP
        NOP
        NOP
        NOP
*
END     BR      END                 ;BRANCH TO ITSELF AT THE END
        .END
```

Appendix C2. fft_rl – Radix-2 Real FFT to Be Called as a C Function

```
*
*   APPENDIX C2
*
*   NAME:
*        fft_rl --- RADIX-2 REAL FFT TO BE CALLED AS A C FUNCTION.
*
*   SYNOPSIS:
*        int fft_rl(N, M, data)
*        int N     FFT SIZE: N=2**M
*        int M     NUMBER OF STAGES = LOG2(N)
*        float *data    ARRAY WITH INPUT AND OUTPUT DATA
*
*   DESCRIPTION:
*        GENERIC FUNCTION TO DO A RADIX-2 FFT COMPUTATION ON THE TMS320C30.
*        THE DATA ARRAY IS N-LONG, WITH ONLY REAL DATA. THE OUTPUT IS STORED
*        IN THE SAME LOCATIONS WITH REAL AND IMAGINARY POINTS R AND I AS
*        FOLLOWS: R(0), R(1),..., R(N/2), I(N/2-1),..., I(1)
*
*        THE PROGRAM IS BASED ON THE FORTRAN PROGRAM IN THE PAPER BY SORENSEN
*        ET AL., JUNE 1987 ISSUE OF TRANS. ON ASSP. THE COMPUTATION IS DONE
*        IN-PLACE, AND THE ORIGINAL DATA IS DESTROYED. BIT REVERSAL IS
*        IMPLEMENTED AT THE BEGINNING OF THE FUNCTION. IF THIS IS NOT
*        NECESSARY, THIS PART CAN BE COMMENTED OUT.
*
*        THE SINE/COSINE TABLE FOR THE TWIDDLE FACTORS IS EXPECTED TO BE
*        SUPPLIED DURING LINK TIME, AND IT SHOULD HAVE THE FOLLOWING FORMAT:
*
*                .global   _sine
*                .data
*        _sine   .float  value1  = sin(0*2*pi/N)
*                .float  value2  = sin(1*2*pi/N)
*       ......
*                .float  value(N/2)    = cos((N/4)*2*pi/N)
*
*        THE VALUES value1 TO value(N/4) ARE THE FIRST QUARTER OF THE SINE
*        PERIOD ANDd value(N/4+1) TO value(N/2) ARE THE FIRST QUARTER OF THE
*        COSINE PERIOD.
*
*        STACK STRUCTURE UPON THE CALL:
*                  +--------------+
*        -FP(4)    |    DATA      |
*        -FP(3)    |    M         |
*        -FP(2)    |    N         |
*        -FP(1)    | RETURN ADDR  |
*        -FP(0)    |   OLD FP     |
*                  +--------------+
*
*   REGISTERS USED: R0, R1, R2, R3, R4, R5, AR0, AR1, AR2, AR4, AR5, IR0,
*                   IR1, RS, RE, RC
*
*   AUTHOR: PANOS E. PAPAMICHALIS
*           TEXAS INSTRUMENTS                              OCTOBER 13, 1987
*
*********************************************************************
*
FP      .SET    AR3
*
        .GLOBL  _FFT_RL         ; ENTRY POINT FOR EXECUTION
        .GLOBL  _SINE           ; ADDRESS OF SINE TABLE
*
        .BSS    FFTSIZ,1
        .BSS    LOGFFT,1
        .BSS    INPUT,1
*
        .TEXT
*
SINTAB  .word   _SINE
*
*   INITIALIZE C FUNCTION
*
_FFT_RL: PUSH   FP              ; SAVE DEDICATED REGISTERS
        LDI     SP,FP
        PUSH    R4
        PUSH    R5
        PUSH    AR4
        PUSH    AR5
*
        LDI     *-FP(2),R0      ; MOVE ARGUMENTS TO LOCATIONS MATCHING
        STI     R0,@FFTSIZ      ;   THE NAMES IN THE PROGRAM
        LDI     *-FP(3),R0
        STI     R0,@LOGFFT
        LDI     *-FP(4),R0
        STI     R0,@INPUT
*
*   DO THE BIT REVERSING AT THE BEGINNING
*
        LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0
        LSH     -1,IR0          ; IR0=HALF THE SIZE OF FFT=N/2
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
*
        RPTB    BITRV
        CMPI    AR1,AR0         ; XCHANGE LOCATIONS ONLY
        BGE     CONT            ; IF AR0<AR1
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
CONT    NOP     *AR0++
BITRV   NOP     *AR1++(IR0)B
*
*   LENGTH-TWO BUTTERFLIES
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        LDI     IR0,RC          ; REPEAT N/2 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
```

```
*
        RPTB    BLK1
        ADDF    *+AR0,*AR0++,R0     ; R0=X(I)+X(I+1)
        SUBF    *AR0,*-AR0,R1       ; R1=X(I)-X(I+1)
BLK1    STF     R0,*-AR0            ; X(I)=X(I)+X(I+1)
||      STF     R1,*AR0++           ; X(I+1)=X(I)-X(I+1)
*
*   FIRST PASS OF THE DO-20 LOOP (STAGE K=2 IN DO-10 LOOP)
*
        LDI     @INPUT,AR0          ; AR0 POINTS TO X(I)
        LDI     2,IR0               ; IR0=2=N2
        LDI     @FFTSIZ,RC
        LSH     -2,RC               ; REPEAT N/4 TIMES
        SUBI    1,RC                ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        RPTB    BLK2
        ADDF    *+AR0(IR0),*AR0++(IR0),R0   ; R0=X(I)+X(I+2)
        SUBF    *AR0,*-AR0(IR0),R1  ; R1=X(I)-X(I+2)
        NEGF    *+AR0,R0            ; R0=-X(I+3)
||      STF     R0,*-AR0(IR0)       ; X(I)=X(I)+X(I+2)
BLK2    STF     R1,*AR0++(IR0)      ; X(I+2)=X(I)-X(I+2)
||      STF     R0,*+AR0            ; X(I+3)=-X(I+3)
*
*   MAIN LOOP (FFT STAGES)
*
        LDI     @FFTSIZ,IR0
        LSH     -2,IR0              ; IR0=INDEX FOR E
        LDI     3,R5                ; R5 HOLDS THE CURRENT STAGE NUMBER
        LDI     1,R4                ; R4=N4
        LDI     2,R3                ; R3=N2
LOOP    LSH     -1,IR0              ; E=E/2
        LSH     1,R4                ; N4=2*N4
        LSH     1,R3                ; N2=2*N2
*
*   INNER LOOP (DO-20 LOOP IN THE PROGRAM)
*
        LDI     @INPUT,AR5          ; AR5 POINTS TO X(I)
INLOP   LDI     IR0,AR0
        ADDI    @SINTAB,AR0         ; AR0 POINTS TO SIN/COS TABLE
        LDI     R4,IR1              ; IR1=N4
*
        LDI     AR5,AR1
        ADDI    1,AR1               ; AR1 POINTS TO X(I1)=X(I+J)
        LDI     AR1,AR3
        ADDI    R3,AR3              ; AR3 POINTS TO X(I3)=X(I+J+N2)
        LDI     AR3,AR2
        SUBI    2,AR2               ; AR2 POINTS TO X(I2)=X(I-J+N2)
        ADDI    R3,AR2,AR4          ; AR4 POINTS TO X(I4)=X(I-J+N1)
*
        LDF     *AR5++(IR1),R0      ; R0=X(I)
        ADDF    *+AR5(IR1),R0,R1    ; R1=X(I)+X(I+N2)
        SUBF    R0,*++AR5(IR1),R0   ; R0=-X(I)+X(I+N2)
||      STF     R1,*-AR5(IR1)       ; X(I)=X(I)+X(I+N2)
        NEGF    R0                  ; R0=X(I)-X(I+N2)
        NEGF    *++AR5(IR1),R1      ; R1=-X(I+N4+N2)
||      STF     R0,*AR5             ; X(I+N2)=X(I)-X(I+N2)
        STF     R1,*AR5             ; X(I+N4+N2)=-X(I+N4+N2)
*
*   INNERMOST LOOP
*
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1              ; IR1=SEPARATION BETWEEN SIN/COS TBLS
        LDI     R4,RC
        SUBI    2,RC                ; REPEAT N4-1 TIMES
*
        RPTB    BLK3
        MPYF    *AR3,*+AR0(IR1),R0  ; R0=X(I3)*COS
        MPYF    *AR4,*AR0,R1        ; R1=X(I4)*SIN
        MPYF    *AR4,*+AR0(IR1),R1  ; R1=X(I4)*COS
||      ADDF    R0,R1,R2            ; R2=X(I3)*COS+X(I4)*SIN
        MPYF    *AR3,*AR0++(IR0),R0 ; R0=X(I3)*SIN
        SUBF    R0,R1,R0            ; R0=-X(I3)*SIN+X(I4)*COS !!!
        SUBF    *AR2,R0,R1          ; R1=-X(I2)+R0 !!!
        ADDF    *AR2,R0,R1          ; R1=X(I2)+R0 !!!
||      STF     R1,*AR3++           ; X(I3)=-X(I2)+R0 !!!
        ADDF    *AR1,R2,R1          ; R1=X(I1)+R2
||      STF     R1,*AR4--           ; X(I4)=X(I2)+R0 !!!
        SUBF    R2,*AR1,R1          ; R1=X(I1)-R2
||      STF     R1,*AR1++           ; X(I1)=X(I1)+R2
BLK3    STF     R1,*AR2--           ; X(I2)=X(I1)-R2
*
        SUBI    @INPUT,AR5
        ADDI    R3,AR5              ; AR5=I+N1
        CMPI    @FFTSIZ,AR5
        BLED    INLOP               ; LOOP BACK TO THE INNER LOOP
        ADDI    @INPUT,AR5
        NOP
        NOP
*
        ADDI    1,R5
        CMPI    @LOGFFT,R5
        BLE     LOOP
*
*   RESTORE THE REGISTER VALUES AND RETURN
*
        POP     AR5
        POP     AR4
        POP     R5
        POP     R4
        POP     FP
        RETS
```

Appendix C3. Generic Program to Do a Radix-2 Real Inverse FFT Computation on the TMS320C30

```
*
*   APPENDIX C3
*
*   GENERIC PROGRAM TO DO A RADIX-2 REAL INVERSE FFT COMPUTATION ON THE
*   TMS320C30.
*
*   THE (REAL) DATA RESIDE IN INTERNAL MEMORY. THE COMPUTATION IS DONE
*   IN-PLACE. THE BIT REVERSAL IS DONE AT THE BEGINNING OF THE PROGRAM. THE
*   INPUT DATA ARE STORED IN THE FOLLOWING ORDER:
*
*   RE(0), RE(1),..., RE(N/2), IM(N/2-1),..., IM(1)
*
*   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. THE LENGTH OF
*   THE TABLE IS N/4 + N/4 = N/2.
*
*   AUTHOR: PANOS PAPAMICHALIS                          DECEMBER 21, 1988
*           TEXAS INSTRUMENTS
*
        .GLOBL  IFFT            ; ENTRY POINT FOR EXECUTION
        .GLOBL  N               ; FFT SIZE
        .GLOBL  M               ; LOG2(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
*
        .BSS    INP,1024        ; MEMORY WITH INPUT DATA
*
        .TEXT
*
*   INITIALIZE
*
        .WORD   IFFT            ; STARTING LOCATION OF THE PROGRAM
*
        .SPACE  100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
*
FFTSIZ  .WORD   N
LOGFFT  .WORD   M
SINTAB  .WORD   SINE
INPUT   .WORD   INP
*
IFFT:   LDP     FFTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
*
*   MAIN LOOP (FFT STAGES)
*
        LDI     1,IR0           ; IR0=INDEX FOR E
        LDI     3,R5            ; R5 HOLDS THE CURRENT STAGE NUMBER
        LDI     @FFTSIZ,R3
        LSH     -1,R3           ; R3=N1/2=N2
        LDI     @FFTSIZ,R4
        LSH     -2,R4           ; R4=N1/4=N4
*
*   INNER LOOP
*
LOOP    LDI     @INPUT,AR5      ; AR5 POINTS TO X(I)
        LDI     IR0,AR0
        ADDI    @SINTAB,AR0     ; AR0 POINTS TO SIN/COS TABLE
INLOP   LDI     R4,IR1          ; IR1=N4
*
        LDI     AR5,AR1
        ADDI    1,AR1           ; AR1 POINTS TO X(I1)=X(I+J)
        LDI     AR1,AR3
        ADDI    R3,AR3          ; AR3 POINTS TO X(I3)=X(I+J+N2)
        LDI     AR3,AR2
        SUBI    2,AR2           ; AR2 POINTS TO X(I2)=X(I-J+N2)
        ADDI    R3,AR2,AR4      ; AR4 POINTS TO X(I4)=X(I-J+N1)
*
        NOP     *++AR5(IR1)     ; POINT TO X(I+N4)
        ADDF    *-AR5(IR1),*++AR5(IR1),R0
        SUBF    *++AR5(IR1),*-AR5(IR1),R1
        STF     R0,*-AR5(IR1)   ; X(I)=X(I)+X(I+N2)
        STF     R1,*++AR5(IR1)  ; X(I+N2)=X(I)-X(I+N2)
||      LDF     *AR5,R0
        MPYF    2.0,R0
        STF     R0,*-AR5(IR1)   ; X(I+N4)=2*X(I+N4)
||      LDF     *++AR5(IR1),R1
        MPYF    -2.0,R1
        STF     R1,*AR5++(IR1)  ; X(I+N4+N2)=-X(I+N4+N2)*2
*
*   INNERMOST LOOP
*
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=SEPARATION BETWEEN SIN/COS TBLS
        LDI     R4,RC
        SUBI    2,RC            ; REPEAT N4-1 TIMES
*
        RPTB    BLK3
        SUBF    *AR2,*AR1,R1    ; R1=T1=X(I1)-X(I2)
        ADDF    *AR2,*AR1,R0
        MPYF    R1,*+AR0(IR1),R0 ; R0=T1*COS
||      STF     R0,*AR1++       ; X(I1)=X(I1)+X(I2)
        ADDF    *AR3,*AR4,R2    ; R2=T2=X(I3)+X(I4)
        SUBF    *AR3,*AR4,R6
        MPYF    R2,*AR0,R6      ; R6=T2*SIN
||      STF     R6,*AR2--       ; X(I2)=X(I4)-X(I3)
        SUBF    R6,R0
        MPYF    R2,*+AR0(IR1),R6 ; R6=T2*COS
||      STF     R0,*AR3++       ; X(I3)=T1*COS-T2*SIN
        MPYF    R1,*+AR0++(IR0),R0 ; R0=T1*SIN
        ADDF    R6,R0
BLK3    STF     R0,*AR4--       ; X(I4)=T1*SIN+T2*COS
*
        SUBI    @INPUT,AR5
        CMPI    @FFTSIZ,AR5
        BLTD    INLOP           ; LOOP BACK TO THE INNER LOOP
        ADDI    @INPUT,AR5
        LDI     IR0,AR0
        ADDI    @SINTAB,AR0     ; AR0 POINTS TO SIN/COS TABLE
```

```
*
        ADDI    1,R5
        CMPI    @LOGFFT,R5
        BLED    LOOP
        LSH     1,IR0           ; E=E*2
        LSH     -1,R4           ; N4=N4/2
        LSH     -1,R3           ; N2=N2/2
*
*   LAST PASS OF THE MAIN LOOP
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        LDI     2,IR0           ; IR0=2=N2
        LDI     @FFTSIZ,RC
        LSH     -2,RC           ; REPEAT N/4 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        LDF     *++AR0(IR0),R0  ; R0=X(I+2)
        RPTB    BLK2
        ADDF    R0,*AR0++(IR0),R1   ; R1=X(I)+X(I+2)
        SUBF    R0,*-AR0(IR0),R1    ; R1=X(I)-X(I+2)
||      STF     R1,*-AR0(IR0)       ; X(I)=X(I)+X(I+2)
        STF     R1,*AR0++           ; X(I+2)=X(I)-X(I+2)
||      LDF     *-AR0,R1
        MPYF    2.0,R1          ; R1=2.0*X(I+1)
        STF     R1,*-AR0(IR0)   ; X(I+1)=2.0*X(I+1)
||      LDF     *AR0++,R1
        MPYF    -2.0,R1         ; R1=-2.0*X(I+3)
BLK2    STF     R1,*-AR0        ; X(I+3)=-2.0*X(I+3)
||      LDF     *++AR0(IR0),R0  ; R0=X(I+4+2)
*
*   LENGTH-TWO BUTTERFLIES
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        LDI     @FFTSIZ,RC
        LSH     -1,RC           ; REPEAT N/2 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        RPTB    BLK1
        ADDF    *++AR0,*AR0++,R0    ; R0=X(I)+X(I+1)
        SUBF    *AR0,*-AR0,R1       ; R1=X(I)-X(I+1)
BLK1    STF     R0,*-AR0        ; X(I)=X(I)+X(I+1)
||      STF     R1,*AR0++       ; X(I+1)=X(I)-X(I+1)
*
*   DO THE BIT REVERSING AT THE END
*
        LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FFTSIZ,IR0
        LSH     -1,IR0          ; IR0=HALF THE SIZE OF FFT=N/2
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
*
        RPTB    BITRV
        CMPI    AR1,AR0         ; XCHANGE LOCATIONS ONLY
        BGE     CONT            ; IF AR0<AR1
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
CONT    NOP     *AR0++
BITRV   NOP     *AR1++(IR0)B
*
END     BR      END             ; BRANCH TO ITSELF AT THE END
        .END
```

Appendix D. Discrete Hartley Transform

Appendix D1. Generic Program to Do a Radix-2 Hartley Transform on the TMS320C30

```
*   APPENDIX D1
*
*   GENERIC PROGRAM TO DO A RADIX-2 HARTLEY TRANSFORM ON THE TMS320C30.
*
*   THE PROGRAM IS TAKEN FROM THE PAPER BY SORENSEN ET AL., OCT 1985 ISSUE
*   OF THE TRANSACTIONS ON ASSP.
*
*   THE (REAL) DATA RESIDE IN INTERNAL MEMORY. THE COMPUTATION IS DONE
*   IN-PLACE. THE BIT-REVERSAL IS DONE AT THE BEGINNING OF THE PROGRAM.
*
*   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 FHT N AND LOG2(N) ARE
*   DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED DURING LINKING. THE LENGTH OF
*   THE TABLE IS N/4 + N/4 = N/2.
*
*   AUTHOR: PANOS PAPAMICHALIS                          DECEMBER 14, 1988
*           TEXAS INSTRUMENTS
*
        .GLOBL  FHT             ; ENTRY POINT FOR EXECUTION
        .GLOBL  N               ; FHT SIZE
        .GLOBL  M               ; LOG2(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
*
        .BSS    INP,1024        ; MEMORY WITH INPUT DATA
*
        .TEXT
*
*   INITIALIZE
*
        .WORD   FHT             ; STARTING LOCATION OF THE PROGRAM
*
        .SPACE  100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
*
FHTSIZ  .WORD   N
LOGFHT  .WORD   M
SINTAB  .WORD   SINE
INPUT   .WORD   INP
*
FHT:    LDP     FHTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
*
*   DO THE BIT REVERSING AT THE BEGINNING
*
        LDI     @FHTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDI     @FHTSIZ,IR0
        LSH     -1,IR0          ; IR0=HALF THE SIZE OF FHT=N/2
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
*
        RPTB    BITRV
        CMPI    AR1,AR0         ; XCHANGE LOCATIONS ONLY
        BGE     CONT            ;   IF AR0<AR1
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
CONT    NOP     *AR0++
BITRV   NOP     *AR1++(IR0)B
*
*   LENGTH-TWO BUTTERFLIES
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        LDI     IR0,RC          ; REPEAT N/2 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        RPTB    BLK1
        ADDF    *+AR0,*AR0++,R0 ; R0=X(I)+X(I+1)
        SUBF    *AR0,*-AR0,R1   ; R1=X(I)-X(I+1)
BLK1    STF     R0,*-AR0        ; X(I)=X(I)+X(I+1)
||      STF     R1,*AR0++       ; X(I+1)=X(I)-X(I+1)
*
*   FIRST PASS OF THE DO-30 LOOP (STAGE K=2 IN DO-20 LOOP)
*
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(J)
        LDI     2,IR0           ; IR0=2=N2
        LDI     @FHTSIZ,RC
        LSH     -2,RC           ; REPEAT N/4 TIMES
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
*
        RPTB    BLK2
        ADDF    *+AR0(IR0),*AR0++(IR0),R0   ; R0=X(J)+X(L2)
        SUBF    *AR0,*-AR0(IR0),R1  ; R1=X(J)-X(L2)
        STF     R0,*-AR0(IR0)       ; X(J)=X(J)+X(L2)
||      LDF     *+AR0,R0            ; R0=X(L4)
        ADDF    R0,*-AR0,R1         ; R1=X(L3)+X(L4)
||      STF     R1,*AR0++           ; X(L2)=X(J)-X(L2)
        SUBF    R0,*-AR0(IR0),R1    ; R1=X(L3)-X(L4)
||      STF     R1,*-AR0(IR0)       ; X(L3)=X(L3)+X(L4)
BLK2    STF     R1,*AR0++           ; X(L4)=X(L3)-X(L4)
*
*   MAIN LOOP (FHT STAGES)
*
        LDI     @FHTSIZ,IR0
        LSH     -2,IR0          ; IR0=INDEX FOR E
        LDI     3,R5            ; R5 HOLDS THE CURRENT STAGE NUMBER
        LDI     1,R4            ; R4=N4
        LDI     2,R3            ; R3=N2
LOOP    LSH     -1,IR0          ; E=E/2
        LSH     1,R4            ; N4=2*N4
        LSH     1,R3            ; N2=2*N2
*
*   INNER LOOP (DO-30 LOOP IN THE PROGRAM)
*
        LDI     @INPUT,AR5      ; AR5 POINTS TO X(J)
INLOP   LDI     IR0,AR0
        ADDI    @SINTAB,AR0     ; AR0 POINTS TO SIN/COS TABLE
        LDI     R4,IR1          ; IR1=N4
```

```
*
        LDI     AR5,AR1
        ADDI    1,AR1           ; AR1 POINTS TO X(L1)=X(J+I-1)
        LDI     AR1,AR3
        ADDI    R3,AR3          ; AR3 POINTS TO X(L3)=X(L1+N2)
        LDI     AR3,AR2
        SUBI    2,AR2           ; AR2 POINTS TO X(L2)=X(J-I+1+N2)
        ADDI    R3,AR2,AR4      ; AR4 POINTS TO X(L4)=X(L2+N2)
*
        LDF     *AR5++(IR1),R0  ; R0=X(J)
        ADDF    **AR5(IR1),R0,R1 ; R1=X(J)+X(L2)
        SUBF    R0,**++AR5(IR1),R0 ; R0=-X(J)+X(L2)
||      STF     R1,*-AR5(IR1)   ; X(J)=X(J)+X(L2)
        NEGF    R0              ; R0=X(J)-X(L2)
        STF     R0,*AR5         ; X(L2)=X(J)-X(L2)
||      LDF     **AR5(IR1),R0   ; R0=X(L4)
        ADDF    R0,*-AR5(IR1),R1 ; R1=X(L3)+X(L4)
        SUBF    R0,*-AR5(IR1),R1 ; R1=X(L3)-X(L4)
||      STF     R1,*-AR5(IR1)   ; X(L3)=X(L3)+X(L4)
        STF     R1,**++AR5(IR1) ; X(L4)=X(L3)-X(L4)
*
*   INNERMOST LOOP
*
        LDI     @FHTSIZ,IR1
        LSH     -2,IR1          ; IR1=SEPARATION BETWEEN SIN/COS TBLS
        LDI     R4,RC
        SUBI    2,RC            ; REPEAT N4-1 TIMES
*
        RPTB    BLK3
        MPYF    *AR3,**++AR0(IR1),R0 ; R0=X(L3)*COS
        MPYF    *AR4,**AR0,R1   ; R1=X(L4)*SIN
        MPYF    *AR4,**++AR0(IR1),R1 ; R1=X(L4)*COS
||      ADDF    R0,R1,R2        ; R2=X(L3)*COS+X(L4)*SIN=T1
        MPYF    *AR3,*AR0++(IR0),R0 ; R0=X(L3)*SIN
        SUBF    R1,R0,R0        ; R0=X(L3)*SIN-X(I4)*COS=T2
        SUBF    R0,*AR2,R1      ; R1=X(L2)-T2
        ADDF    *AR2,R0,R1      ; R1=X(L2)+T2
||      STF     R1,*AR4--       ; X(L4)=X(L2)-T2
        ADDF    *AR1,R2,R1      ; R1=X(L1)+T1
||      STF     R1,*AR2--       ; X(L2)=X(L2)+T2
        SUBF    R2,*AR1,R1      ; R1=X(L1)-T1
||      STF     R1,*AR1++       ; X(L1)=X(L1)+T1
BLK3    STF     R1,*AR3++       ; X(L3)=X(L1)-T1
*
        SUBI    @INPUT,AR5
        ADDI    R3,AR5          ; AR5=I+N1
        CMPI    @FHTSIZ,AR5
        BLTD    INLOP           ; LOOP BACK TO THE INNER LOOP
        ADDI    @INPUT,AR5
        NOP
        NOP
*
        ADDI    1,R5
        CMPI    @LOGFHT,R5
        BLE     LOOP
*
END     BR      END             ;BRANCH TO ITSELF AT THE END
        .END
```

Appendix E. Discrete Cosine Transform

Appendix E1. A Fast Cosine Transform

```
*
*   APPENDIX E1
*
*   A FAST COSINE TRANSFORM
*
*   BASED ON THE ALGORITHM OUTLINED BY BYEONG GI LEE IN HIS ARTICLE, FCT - A
*   FAST COSINE TRANSFORM, PUBLISHED IN THE PROCEEDINGS OF THE IEEE INTER-
*   NATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, SAN
*   DIEGO, CA, 19-21 MARCH 1984, P 28A.3/1-4 VOL. 2, (CH1954-5/84/0000-0299).
*
*   LEE'S ALGORITHM HAS BEEN MODIFIED TO ALLOW NATURAL ORDER TIME DOMAIN
*   COEFFICIENTS RATHER THAN THE LESS ORDERED INPUT SUGGESTED IN HIS ARTICLE.
*
*   THE FREQUENCY DOMAIN COEFFICIENTS ARE IN BIT REVERSE ORDER. THIS IS AN IN
*   PLACE CALCULATION.
*
*   AUTHOR: PAUL WILHELM
*
*
        .global  FCT              ; FAST COSINE TRANSFORM ENTRY POINT.
        .global  M                ; LENGTH OF DATA ENTRY.
        .global  COS_TAB          ; TABLE OF COSINE COEFFICIENTS.
        .global  COEFF            ; TABLE OF INPUT DATA.
*
        .text
*
FCTSIZE .word    M
_COS    .word    COS_TAB
_DATA   .word    COEFF
*
FCT:
        LDI      @FCTSIZE,AR0     ; LOAD DATA LENGTH.
        LDI      @FCTSIZE,BK      ; SET BLOCK SIZE FOR CIRCULAR
*                                 ;   ADDRESSING.
        LDI      @_DATA,AR6       ; LOAD DATA POINTER.
        LDI      @_COS,AR7        ; LOAD COSINE TABLE POINTER.
        LDI      AR0,IR1          ; INITIALIZE INDEX REGISTERS FOR FIRST
        LDI      -1,IR0           ;   BUTTERFLY SERIES.
        LDI      AR6,AR1          ; INITIALIZE DATA POINTERS.
        ADDI3    AR6,AR0,AR2
        SUBI     1,AR2
        LSH3     IR0,AR0,AR3
        LDI      1,AR5            ; INITIALIZE 2'S POWER COUNTER.
        ADDI     AR6,AR3          ; FINISH DATA POINTER INITIALIZATION.
        ADDI3    IR0,AR3,AR4
        ADDI3    IR0,AR5,RC       ; RC SHOULD BE ONE LESS THAN COUNT
*                                 ;   DESIRED.
*
*   FIRST LOOP SERIES
*
*   THIS LOOP SERIES DOES ALL THE BUTTERFLY STAGES EXCEPT THE FINAL ONE.
*
        RPTB     END_CENTER_LOOP
*
OUTSIDE_LOOP:                     ; TWO BUTTERFLIES ARE CALCULATED AT
MIDDLE_LOOP:                      ;   THE SAME TIME.
*
        LDF      *AR2,R2          ; GET LOWER HALF OF EACH BUTTERFLY.
||      LDF      *AR3,R3          ;   (THIS ALLOWS FOR MORE PARALLEL
*                                 ;   COMMANDS LATER)
        SUBF3    *AR3,*AR4,R1     ; SUBTRACT SECOND BUTTERFLY DATA.
        SUBF3    *AR2,*AR1,R0     ; SUBTRACT FIRST BUTTERFLY DATA.
        MPYF3    R1,*++AR7,R1     ; MULTIPLY 2ND SUBTRACTION RESULT BY
||      ADDF3    R3,*AR4,R3       ;   COSINE COEFFICIENT. ADD SECOND
*                                 ;   BUTTERFLY DATA.
        MPYF3    R0,*--AR7,R0     ; MULTIPLY 1ST SUBTRACTION RESULT BY
||      ADDF3    R2,*AR1,R2       ;   COSINE COEFFICIENT. ADD FIRST
*                                 ;   BUTTERFLY DATA.
        STF      R1,*AR2++(IR1)%  ; SAVE 2ND MULTIPLY RESULT IN LOWER
||      STF      R3,*AR4++(IR1)%  ;   HALF IF BUTTERFLY. SAVE 2ND
*                                 ;   ADDITION IN UPPER 2ND BUTTERFLY.
END_CENTER_LOOP:
*
        STF      R0,*AR3++(IR1)%  ; SAVE 1ST MULTIPLY IN LOWER HALF OF
||      STF      R2,*AR1++(IR1)%  ;   2ND BUTTERFLY. SAVE 1ST ADDITION
*                                 ;   IN UPPER 1ST BUTTERFLY.
*
*   END OF CENTER LOOP OF FIRST LOOP SERIES.
*
        ADDI3    IR0,AR5,RC       ; UPDATE REPEAT COUNTER FOR NEXT BLOCK
*                                 ;   REPEAT.
        ADDF3    *AR3++,*AR2--,R0 ; UPDATE DATA POINTERS.
        CMPI     AR3,AR2          ; HAVE BUTTERFLIES BEEN COMPLETED?
        BGTD     MIDDLE_LOOP      ; DELAYED BRANCH, IF NOT.
        ADDF3    *AR1++,*AR4--,R0 ; UPDATE FINAL TWO POINTERS FOR NEXT
*                                 ;   REPEAT.
        ADDI     2,AR7            ; UPDATE COSINE COEFFICIENT POINTER.
        OR       0100H,ST         ; SET REPEAT MODE. (FASTER THAN USING
*                                 ;   RPTB WHEN START AND END ADDRESS
*                                 ;   ARE STILL GOOD)
*
*   DELAY BRANCH FROM HERE TO MIDDLE_LOOP.
*
        LSH      -1,IR1           ; UPDATE INDEX REGISTER. (DIVIDE BY 2)
        LDI      AR6,AR1          ; REINITIALIZE DATA POINTERS.
        ADDI     IR0,AR6,AR2
        ADDI     IR1,AR2
        CMPI     2,IR1            ; IS FIRST BUTTERFLY SERIES COMPLETE?
        BGTD     OUTSIDE_LOOP     ; DELAY BRANCH, IF NOT.
        LSH      1,AR5            ; MULTIPLY 2'S POWER COUNTER BY 2.
        SUBI3    IR0,AR4,AR3      ; CONTINUE REINITIALIZING DATA
*                                 ;   POINTERS.
        ADDI3    IR0,AR5,RC       ; SET REPEAT COUNTER FOR REPEAT BLOCK.
*
*   END OF FIRST LOOP SERIES.
*
*   FINAL BUTTERFLY STAGE LOOP.
*
```

```
*    INCLUDES LAST BUTTERFLIES AND FIRST STAGE OF BIT REVERSE ADDITIONS.
*
        LDI     4,IR1               ; INITIALIZE INDEX REGISTER.
        ADDI    1,AR3               ; SET UP DATA POINTERS.
        LSH     -1,AR5
        ADDI    3,AR4
        ADDI3   IR0,AR5,RC          ; INITIALIZE REPEAT COUNTER.
        MPYF3   *AR7,*++AR7,R4      ; CALCULATE (2/M)*COS(PI/4).
*                                   ;   (I.E.-> (SQRT(2))/M THIS VALUE IS
*                                   ;   CALLED, S, BELOW.)
        RPTB    END_2ND_LOOP        ; TWO BUTTERFLIES ARE CALCULATED PER
*                                   ;   LOOP.
*
        SUBF3   *AR2,*AR1,R0        ; SUBTRACT 1ST BUTTERFLY DATA.
        SUBF3   *AR4,*AR3,R1        ; SUBTRACT 2ND BUTTERFLY DATA.
        MPYF3   R0,R4,R0            ; MULTIPLY 1ST SUBTRACTION RESULT
||      ADDF3   *AR3++(IR1),*AR4++(IR1),R3   ; BY S. ADD 2ND BUTTERFLY
*                                   ;   DATA.
        MPYF3   R1,R4,R1            ; MULTIPLY 2ND SUBTRACTION RESULT
||      ADDF3   *AR1++(IR1),*AR2++(IR1),R2   ;   BY S. ADD 1ST BUTTERFLY
*                                   ;   DATA.
        MPYF3   R3,*++AR7,R3        ; MULTIPLY 2ND ADDITION RESULT BY
||      STF     R0,*-AR2(IR1)       ;   7071. SAVE 1ST SUBTRACTION IN
*                                   ;   LOWER 1/2 OF 1ST BUTTERFLY.
        MPYF3   R2,*++AR7,R2        ; MULTIPLY 1ST ADDITION RESULT BY
||      STF     R1,*-AR4(IR1)       ;   .7071 SAVE 2ND SUBTRACTION IN
*                                   ;   LOWER 1/2 OF 2ND BUTTERFLY.
        ADDF3   R3,R1,R3            ; ADD 2ND SUBTRACTION MULTIPLY TO 2ND
*                                   ;   ADDITION MULTIPLY.
        STF     R2,*-AR1(IR1)       ; SAVE 1ST ADDITION MULTIPLY IN UPPER
*                                   ;   1/2 OF BUTTERFLY.
*
END_2ND_LOOP:
*
        STF     R3,*-AR3(IR1)       ; SAVE 2ND ADDITION MULTPLY IN UPPER
*                                   ;   1/2 OF UPPER BUTTERFLY.
*
*   END OF FINAL BUTTERFLY STAGE LOOP.
*
*   BIT REVERSE ADDITION LOOP SERIES.
*
*   THIS LOOP SERIES DOES ALL OF THE BIT REVERSE ADDITIONS AT THE END OF FAST
*   COSINE TRANSFORM.
*
        LDI     2,IR0               ; INITIALIZE INDEX REGISTERS AND DATA
        LDI     AR6,AR1             ;   POINTERS FOR FINAL ADDITION
        ADDI    4,AR1               ;   SERIES.
        LDI     AR1,AR2
        LDI     8,IR1
*
LAST_OUTSIDE_LOOP:
*
        LDI     AR2,AR4             ; UPDATE POINTERS AND COUNTERS.
        LSH     -1,AR5
        LDI     AR5,RC              ; SET UP REPEAT COUNTER.
        ADDF3   *AR2++(IR0)B,*AR4++(IR0)B,R0    ; DATA POINTER UPDATE.
        LDI     AR1,R4              ; USE INITIAL AR1 VALUE AS INNER LOOP
*                                   ;   CONTROL.
        SUBI    1,RC
        NOP     *AR4++(IR0)B        ; CONTINUE UPDATING POINTERS.
        LDI     AR2,AR3
*
        RPTB    END_INSIDE          ; TWO ADDITIONS ARE DONE IN EACH LOOP.
*
LAST_INSIDE_LOOP:
*
        ADDF3   *AR1,*AR2++(IR1)%,R0   ; ADD FIRST TWO DATA.
        ADDF3   *AR3,*AR4++(IR1)%,R1   ; ADD SECOND TWO DATA.
        STF     R0,*AR1++(IR1)%     ; SAVE FIRST ADDITION.
*
END_INSIDE:
*
        STF     R1,*AR3++(IR1)%     ; SAVE SECOND ADDITION.
*
*   END OF INSIDE LOOP FOR LAST LOOP SERIES.
*
        ADDF3   *AR1++(IR0)B,*AR2++(IR0)B,R0    ; UPDATE DATA POINTERS.
        ADDF3   *AR3++(IR0)B,*AR4++(IR0)B,R0
        ADDF3   *AR3++(IR0)B,*AR4++(IR0)B,R0
        ADDF3   *AR1++(IR0)B,*AR2++(IR0)B,R0
        CMPI    R4,AR4              ; IS THIS LOOP COMPLETE?
        BNED    LAST_INSIDE_LOOP    ; DELAYED BRANCH, IF NOT.
        LDI     AR5,RC              ; SET UP REPEAT COUNTER.
        SUBI    1,RC
        OR      0100H,ST            ; SET REPEAT MODE.
*
*   BRANCH DELAYED TO LAST_INSIDE_LOOP.
*
        RPTB    LAST_BLOCK          ; SINCE THERE ARE AN ODD NUMBER OF
        ADDF3   *AR1,*AR2++(IR1)%,R0    ;   ADDITIONS, THE FINAL ONES ARE
*                                   ;   DONE NOW.
*
LAST_BLOCK:
*
        STF     R0,*AR1++(IR1)%     ; SAVE ADDITION.
*
*   END OF LAST REPEAT BLOCK.
*
        LSH     1,IR0               ; MULTIPLY IR0 BY 2.
        ADDI    IR0,R4              ; UPDTEE INNER LOOP CONTROL REGISTER.
        CMPI    1,AR5               ; ARE CALCULATIONS COMPLETE ?
        BGTD    LAST_OUTSIDE_LOOP   ; DELAYED BRANCH, IF NOT.
        LDI     R4,AR2              ; UPDATE DATA POINTERS.
        LDI     R4,AR1
        LSH     1,IR1               ; MULTIPLY IR1 BY 2.
*
*   DELAYED BRANCH TO LAST_OUTSIDE_LOOP.
*
```

```
*    END OF LAST LOOP SERIES.
*
*    MULTIPLY COEFFICIENT ZERO BY .5, IF NOT ZERO.
*
         LDF       *AR6,R0             ; SET ZERO FLAG IF *AR6 = 0.
         BEQD      DONT_STORE          ; IF COEFFICIENT IS ZERO, DON'T DO
*                                      ;   THIS.
         LSH       24,AR5              ; USE INTEGER MATH FOR FLOAT DIVIDE
*                                      ;   BY 2.
         SUBI3     AR5,*AR6,AR1
         NOP
*
*    DELAYED BRANCH FROM HERE IF VALUE IS NOT TO BE STORED.
*
         STI       AR1,*AR6            ; STORE, IF EXPONENT WASN'T -128.
*
DONT_STORE:
*
         RETS
```

```
*
*   APPENDIX E2
*
*   A FAST COSINE TRANSFORM (INVERSE TRANSFORM)
*
*   BASED ON THE ALGORITHM OUTLINED BY BYEONG GI LEE IN HIS ARTICLE, FCT - A
*   FAST COSINE TRANSFORM, PUBLISHED IN THE PROCEEDINGS OF THE IEEE Inter-
*   NATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, SAN
*   DIEGO, CA, 19-21 MARCH 1984, P 28A.3/1-4 VOL 2., (CH1954-5/84/0000-0299).
*
*   LEE'S ALGORITHM HAS BEEN MODIFIED TO ALLOW NATURAL ORDER TIME DOMAIN
*   COEFFICIENTS.
*
*   THE FREQUENCY DOMAIN COEFFICIENTS ARE IN BIT REVERSE ORDER. THIS IS AN IN
*   PLACE CALCULATION.
*
*   AUTHOR: PAUL WILHELM
*
        .global  IFCT              ; INVERSE FAST COSINE TRANSFORM ENTRY
*                                  ;   POINT.
        .global  M                 ; LENGTH OF ARRAY TO BE TRANSFORMED.
        .global  COEFF             ; TABLE OF COSINE COEFFICIENTS.
        .global  COS_TAB           ; TABLE OF ARRAY DATA TO BE
*                                  ;   TRANSFORMED.
*
        .text
*
FCTSIZE .word    M
_DATA   .word    COEFF
_COS    .word    COS_TAB
*
IFCT:
        LDI      @FCTSIZE,AR0      ; LOAD ARRAY SIZE.
        LDI      @FCTSIZE,BK       ; LOAD BLOCK SIZE FOR CIRCULAR
*                                  ;   ADDRESSING
        LDI      @_DATA,AR6        ; LOAD POINTER TO DATA TABLE.
        LDI      @_COS,AR7         ; LOAD POINTER TO COSINE TABLE.
        ADDI     AR0,AR7           ; POINT TO LAST COSINE VALUE IN TABLE.
        SUBI     2,AR7
        LDI      AR0,IR0           ; INITIALIZE INDEX REGISTERS FOR BIT
        LSH      -2,IR0            ;   REVERSED ADDITION SEQUENCE.
        LDI      AR0,IR1
        LDI      AR6,AR1           ; INITIALIZE DATA POINTERS.
        ADDI     IR0,AR1
*
*   START OF BIT REVERSED ADDITION LOOP SERIES.
*
OUTSIDE:                           ; TOP OF OUTSIDE LOOP FOR BIT REVERSED
*                                  ;   ADDITIONS.
        ADDI     IR0,AR1           ; UPDATE DATA POINTERS AND REPEAT
*                                  ;   COUNTER.
        LDI      AR1,AR2
        LDI      IR0,RC
        SUBI     2,RC
        NOP      *AR2++(IR0)B
        ADDF3    *AR1++(IR0)B,*AR2++(IR0)B,R0   ; FIND FIRST SUM. (MAKES
        LDI      AR1,AR3           ;   MIDDLE LOOP MORE EFFICIENT)
        LDI      AR2,AR4
        LDI      AR1,AR5
        ADDF3    *AR3++(IR0)B,*AR4++(IR0)B,R1   ; DUMMY ADD TO UPDATE
*                                  ;   POINTERS.
        LSH      -1,IR0            ; UPDATE INDEX REGISTER.
*
        RPTB     END_CENTER        ; TOP OF INNER MOST LOOP.
*
MIDDLE:                            ; TOP OF MIDDLE LOOP.
*
        LDF      *AR3,R3           ; GET UPPER HALF OF SECOND ADDITION.
        ADDF3    *AR1,*AR2++(IR0)B,R1   ; DO FIRST ADDITION.
        STF      R0,*AR1++(IR0)B   ; STORE ADDITION DONE THE LAST LOOP OR
*                                  ;   WHEN INITIALIZATION WAS DONE ABOVE
END_CENTER:
*
        ADDF3    R3,*AR4++(IR0)B,R0  ; DO SECOND ADDITION.
||      STF      R1,*AR3++(IR0)B   ; STORE FIRST ADDITION.
*
*   END OF INNER MOST LOOP.
*
        ADDF3    *AR3++(IR1)%,*AR4++(IR1)%,R2   ; DUMMY ADD TO UPDATE
*                                  ;   POINTERS.
        LDF      *AR3++(IR0)B,R3   ; GET VALUE FOR LAST ADDITION.
||      LDF      *AR2++(IR0)B,R2   ; DUMMY ADD TO UPDATE POINTER.
        ADDF3    R3,*AR4++(IR0)B,R0  ; DO LAST ADDITION.
||      STF      R0,*AR1++(IR0)B   ; STORE NEXT TO LAST ADDITION.
        ADDF3    *AR1++(IR1)%,*AR2++(IR1)%,R2   ; DUMMY ADD TO UPDATE
*                                  ;   POINTERS.
        LDI      IR0,RC            ; UPDATE REPEAT COUNTER.
        CMPI     AR1,AR5           ; IS MIDDLE LOOP COMPLETE ?
        BNED     MIDDLE            ; IF NOT, DO DELAYED BRANCH.
        LSH      1,RC
        SUBI     2,RC
        OR       0100H,ST          ; SET REPEAT MODE.
*                                  ;   (START/STOP ADDRESSES STILL OK)
*
*   DELAY BRANCH FROM HERE TO MIDDLE.
*
        CMPI     1,IR0             ; IS OUTSIDE LOOP COMPLETE ?
        BGTD     OUTSIDE           ; IF NOT, DO DELAYED BRANCH.
        LDI      AR6,AR1           ; PREPARE TO UPDATE POINTERS AT TOP OF
*                                  ;   LOOP.
        ADDI     IR0,AR1
        LSH      -1,IR1            ; UPDATE INDEX REGISTER.
*
*   DELAY BRANCH FROM HERE TO OUTSIDE.
*
*   END OF BIT REVERSED ADDITION LOOP SERIES.
*
*   START OF CENTER BUTTERFLY LOOP.
```

```
*
*   THIS LOOP INCLUDES THE LAST BIT REVERSED ADDITION STAGE, THE FIRST
*   BUTTERFLY, AND THE COSINE MULTIPLICATIONS FOR THE SECOND BUTTERFLY
*   SERIES.
*
        SUBI    3,AR2           ; UPDATE DATA POINTER FOR THIS LOOP.
        LDI     8,IR1           ; INITIALIZE INDEX REGISTER.
        LDI     AR0,RC          ; INITIALIZE REPEAT COUNTER.
        LSH     -3,RC
        LDF     *AR7--,R7       ; GET COSINE PI/4.
        SUBI    1,RC
        LDI     RC,AR5          ; SAVE REPEAT COUNTER FOR LATER USE.
*
        RPTB    END_CENTER_LOOP ; FOUR BUTTERFLIES ARE DONE EACH CYCLE
*                               ;   THROUGH THIS LOOP.
*
        ADDF3   *++AR2,*AR2,R4  ; BIT REVERSED ADDITION FOR 2ND
*                               ;   BUTTERFLY.
        MPYF3   *AR1,R7,R5      ; COSINE  PI/4 TIMES LOWER HALF OF 1ST
*                               ;   BUTTERFLY.
        MPYF3   R7,R4,R0        ; COSINE PI/4 TIMES LOWER HALF OF 2ND
*                               ;   BUTTERFLY.
||      ADDF3   *AR4,*-AR4,R3   ; BIT REVERSED ADDITION FOR 4TH
*                               ;   BUTTERFLY.
        ADDF3   R5,*-AR1,R4     ; ADD UPPER HALF OF 1ST BUTTERFLY.
        MPYF3   *++AR7,R3,R1    ; COSINE PI/4 TIMES LOWER HALF OF 4TH
*                               ;   BUTTERFLY.
||      ADDF3   R0,*AR2,R2      ; ADD UPPER HALF OF 2ND BUTTERFLY.
        SUBF3   R5,*-AR1,R5     ; SUBTRACT LOWER HALF OF 1ST
*                               ;   BUTTERFLY.
        MPYF3   *-AR7,R2,R0     ; MULTIPLY UPPER HALF OF 2ND BUTTERFLY
*                               ;   BY COSINE COEFFIEIENT.
||      SUBF3   R0,*AR2,R2      ; SUBTRACT LOWER HALF OF 2ND
*                               ;   BUTTERFLY.
        STF     R4,*-AR1        ; STORE UPPER HALF OF 1ST BUTTERFLY.
||      STF     R5,*AR1++(IR1)% ; STORE LOWER HALF OF 1ST BUTTERFLY.
        STF     R0,*++AR2       ; STORE LOWER HALF OF 2ND BUTTERFLY.
        MPYF3   *AR3,R7,R4      ; COSINE PI/4 TIMES LOWER HALF OF 3RD
*                               ;   BUTTERFLY.
        MPYF3   *AR7,R2,R0      ; MULTIPLY LOWER HALF OF 2ND BUTTERFLY
*                               ;   BY COSINE COEFFICIENT
||      SUBF3   R1,*-AR4,R3     ; SUBTRACT LOWER HALF OF 4TH
*                               ;   BUTTERFLY.
        ADDF3   R4,*-AR3,R5     ; ADD UPPER HALF OF 3RD BUTTERFLY.
        MPYF3   *AR7,R3,R1      ; MULTIPLY ;OWER HALF OF 4TH BUTTERFLY
*                               ;   BY COSINE COEFFICIENT
||      ADDF3   R1,*-AR4,R3     ; ADD UPPER HALF OF 4TH BUTTERFLY.
        SUBF3   R4,*-AR3,R4     ; SUBTRACT LOWER HALF OF 3RD
*                               ;   BUTTERFLY.
        MPYF3   *-AR7,R3,R1     ; MULTIPLY UPPER HALF OF 4TH BUTTERFLY
*                               ;   BY COSINE COEFFICIENT.
||      STF     R1,*-AR4        ; STORE UPPER HALF OF 4TH BUTTERFLY.
        STF     R0,*AR2++(IR1)% ; STORE UPPER HALF OF 2ND BUTTERFLY.
||      STF     R5,*-AR3        ; STORE UPPER HALF OF 3RD BUTTERFLY.
*
END_CENTER_LOOP:
*
        STF     R1,*AR4++(IR1)% ; STORE LOWER HALF OF 4TH BUTTERFLY.
||      STF     R4,*AR3++(IR1)% ; STORE LOWER HALF OF 3RD BUTTERFLY.
*
*   END OF CENTER BUTTERFLY LOOP.
*
*   START NEXT TO LAST LOOP SERIES.
*
*   THIS SERIES OF LOOPS DOES ALL BUT THE LAST BUTTERFLY STAGE. ALL THE
*   COSINE COEFFICIENT MULTIPLICATIONS ARE DONE, INCLUDING THE MULTI-
*   PLICATIONS FOR THE LAST BUTTERFLY STAGE. (THIS PROGRAM FLOW ALLOWS FOR
*   FAST EXECUTION.)
*
        SUBI    2,AR7           ; UPDATE COSINE COEFFICIENT POINTER.
        SUBI    1,AR4           ; UPDATE DATA POINTER.
        LDI     AR5,RC          ; RELOAD REPEAT COUNTER.
        LDF     *AR7--,R5       ; GET COSINE COEFFICIENTS.
        LDF     *AR7--,R4
*
        RPTB    END_NTL         ; TWO BUTTERFLIES ARE CALCULATED PER
*                               ;   CYCLE THROUGH THE INNER LOOP.
*
NTL_LOOP:
*
        SUBF3   *AR4,*AR3,R6    ; SUBTRACT LOWER HALF OF 2ND
*                               ;   BUTTERFLY.
        ADDF3   *AR4,*AR3,R7    ; ADD UPPER HALF OF 2ND BUTTERFLY.
        MPYF3   R5,R6,R0        ; MULTIPLY UPPER HALF OF 2ND BUTTERFLY
*                               ;   BY COSINE COEFFICIENT.
||      ADDF3   *AR2,*AR1,R2    ; ADD UPPER HALF OF 1ST BUTTERFLY.
        MPYF3   R4,R7,R1        ; MULTIPLY LOWER HALF OF 2ND BUTTERFLY
*                               ;   BY COSINE COEFFIEICENT.
||      SUBF3   *AR2,*AR1,R3    ; SUBTRACT LOWER HALF OF 1ST
*                               ;   BUTTERFLY.
        STF     R0,*AR3++(IR1)% ; STORE UPPER HALF OF 2ND BUTTERFLY.
||      STF     R2,*AR1++(IR1)% ; STORE UPPER HALF OF 1ST BUTTERFLY.
*
END_NTL:
*
        STF     R1,*AR4++(IR1)% ; STORE LOWER HALF OF 1ST BUTTERFLY.
||      STF     R3,*AR2++(IR1)% ; STORE LOWER HALF OF 2ND BUTTERFLY.
*
*   END OF CENTER LOOP OF NEXT TO LAST SERIES.
*
        LDI     AR5,RC          ; RELOAD REPEAT COUNTER.
        LDF     *AR7--,R5       ; GET NEW COSINE COEFFICIENTS. (FYI-
        LDF     *AR7--,R4       ;   THE LAST TIME, THIS WILL FETCH
*                               ;   FROM MEMORY BELOW THE COSINE
*                               ;   TABLE.)
        CMPI    AR1,AR6         ; HAS MIDDLE LOOP BEEN COMPLETED ?
        BNED    NTL_LOOP        ; IF NOT, BRANCH DELAYED.
        ADDF3   *AR4++,*AR3--,R0 ; DUMMY ADDS TO UPDATE DATA POINTERS.
```

```
        ADDF3   *AR2++,*AR1--,R0
        OR      0100H,ST            ; SET REPEAT MODE. (START/STOP
*                                   ;   ADDRESSES ARE STILL GOOD.)
*
*   BRANCH DELAY FROM HERE TO NTL_LOOP.
*
        LDI     AR3,AR1             ; UPDATE DATA POINTERS.
        ADDI3   IR1,AR1,AR3
        LSH     1,IR1               ; UPDATE INDEX REGISTER.
        CMPI    IR1,AR0             ; IS THIS LOOP SERIES COMPLETE ?
        BGED    NTL_LOOP            ; IF NOT, BRANCH DELAYED.
        ADDI3   IR0,AR3,AR4         ; UPDATE DATA POINTER.
        LSH     -1,AR5              ; UPDATE REPEAT COUNTER.
        LDI     AR5,RC
*
*   DELAYED BRANCH FROM HERE TO NTL_LOOP.
*
*   END OF NEXT TO LAST LOOP SERIES.
*
*   START OF THE LAST LOOP.
*
*   THE LAST LOOP IS THE LAST BUTTERFLY STAGE WITHOUT THE COSINE COEFFICIENT
*   MULTIPLICATIONS, WHICH HAVE ALREADY BEEN DONE.
*
        LDI     2,IR1               ; INITIALIZE INDEX REGISTER.
        ADDI3   IR0,AR2,AR4         ; INITIALIZE DATA POINTERS.
        SUBI3   IR0,AR1,AR3
        LDI     AR0,RC              ; INITIALIZE REPEAT COUNTER.
        LSH     -2,RC
        SUBI    1,RC
*
        RPTB    END_LAST_LOOP       ; TWO BUTTERFLIES ARE DONE FOR EACH
*                                   ;   CYCLE THROUGH THE LOOP.
*
        LDF     *AR4,R0             ; GET VALUE FOR LOWER HALF OF 2ND
*                                   ;   BUTTERFLY.
        ADDF3   *AR2,*AR1,R1        ; ADD UPPER HALF OF 1ST BUTTERFLY.
        SUBF3   *AR2,*AR1,R2        ; SUBTRACT LOWER HALF OF 1ST
*                                   ;   BUTTERFLY.
        ADDF3   R0,*AR3,R3          ; ADD UPPER HALF OF 2ND BUTTERFLY.
||      STF     R1,*AR1--(IR1)      ; STORE UPPER HALF OF 1ST BUTTERFLY.
        SUBF3   R0,*AR3,R4          ; SUBTRACT LOWER HALF OF 2ND
*                                   ;   BUTTERFLY.
||      STF     R2,*AR2++(IR1)      ; STORE LOWER HALF OF 1ST BUTTERFLY.
        STF     R3,*AR3--(IR1)      ; STORE UPPER HALF OF 2ND BUTTERFLY.
*
END_LAST_LOOP:
*
        STF     R4,*AR4++(IR1)      ; STORE LOWER HALF OF 2ND BUTTERFLY.
*
*   END OF LAST LOOP, AND INVERSE COSINE TRANSFORM.
*
        RETS
        .end
```

Appendix E3. FCT Cosine Tables File

```
*
*    APPENDIX E3
*
*    FCT COSINE TABLES FILE
*
*    TO BE LINKED WITH FCT SOURCE CODE FOR 32 POINT FCT.
*
*    COEFFICIENTS ARE  1/(2 * COS(N*PI/2M)), WHERE N IS A NUMBER FROM 1 to
*    M-1. M IS THE ORDER OF THE TRANSFORM.
*
*    FOR A 32 POINT FCT, N IS IN THE FOLLOWING ORDER:
*         1, 15, 3, 13, 5, 11, 7, 9,
*         2, 14, 6, 10,
*         4, 12,
*         8
*
*    THE LAST VALUE IN THE TABLE IS 2/M.
*
*
          .global   COS_TAB
          .global   M
*
M         .set      16
*
          .data
*
COS_TAB
          .float    0.5024193
          .float    5.1011487
          .float    0.5224986
          .float    1.7224471
          .float    0.5669440
          .float    1.0606777
          .float    0.6468218
          .float    0.7881546
          .float    0.5097956
          .float    2.5629154
          .float    0.6013449
          .float    0.8999762
          .float    0.5411961
          .float    1.3065630
          .float    0.7071068
          .float    0.1250000
          .end
```

Appendix E4. Data File

```
*
*    APPENDIX E4
*
*    DATA FILE
*
          .global   COEFF
*
          .data
*
COEFF
          .float    137.0
          .float    249.0
          .float    105.0
          .float    217.0
          .float    73.0
          .float    185.0
          .float    41.0
          .float    153.0
          .float    9.0
          .float    121.0
          .float    233.0
          .float    89.0
          .float    201.0
          .float    57.0
          .float    169.0
          .float    25.0
          .end
```

Appendix F. Test Vectors, 64-Point Sine Table, Link Command File

Appendix F1. Example of a 64-Point Vector to Test the FFT Routines

```
*
*   APPENDIX F1
*
*   EXAMPLE OF A 64-POINT VECTOR TO TEST THE FFT ROUTINES

      X =

            0.2113
            0.0824
            0.7599
            0.0087
            0.8096
            0.8474
            0.4524
            0.8075
            0.4832
            0.6135
            0.2749
            0.8807
            0.6538
            0.4899
            0.7741
            0.9626
            0.9933
            0.8360
            0.7469
            0.0378
            0.4237
            0.2613
            0.2403
            0.3405
            0.1167
            0.6250
            0.5510
            0.3550
            0.4943
            0.0365
            0.2260
            0.8159
            0.2284
            0.8553
            0.0621
            0.7075
            0.2408
            0.6907
            0.1062
            0.2640
            0.7034
            0.4021
            0.6553
            0.9700
            0.0380
            0.0988
            0.2560
            0.5598
            0.9166
            0.1402
            0.7054
            0.0178
            0.2611
            0.1358
            0.0503
            0.5782
            0.2432
            0.9448
            0.5876
            0.7256
            0.2849
            0.6767
            0.8642
            0.1943
*
*   64-POINT FFT CORRESPONDING TO VECTOR X
*
      Y =

           30.3774
            1.7780 - 2.5584i
           -1.0376 - 2.3999i
           -1.0123 + 2.4889i
            0.6594 + 2.3639i
           -1.5228 - 0.7527i
           -3.8171 - 0.2050i
           -2.7096 + 1.2841i
            2.1622 - 1.6863i
            0.2879 + 1.8671i
           -1.5479 + 1.6298i
           -0.6366 - 0.1176i
            2.2902 + 1.5549i
           -2.4837 - 0.5842i
           -1.7338 + 0.0738i
           -0.2180 - 0.4726i
           -0.2104 + 0.4897i
           -1.7473 - 1.0213i
            0.1233 - 2.3915i
           -0.6415 - 1.1144i
           -2.7719 - 0.4802i
           -0.0063 - 0.3885i
           -0.7163 + 1.5682i
            0.3218 - 1.3316i
           -0.7823 + 1.0607i
           -0.2553 + 2.8270i
           -1.0813 - 2.7861i
            3.4869 + 1.9485i
            3.0352 + 1.3855i
            3.2099 + 2.3564i
           -1.9511 - 0.7714i
            1.8755 + 0.2867i
```

```
-1.5474
 1.8755 - 0.2867i
-1.9511 + 0.7714i
 3.2099 - 2.3564i
 3.0352 - 1.3855i
 3.4869 - 1.9485i
-1.0813 + 2.7861i
-0.2553 - 2.8270i
-0.7823 - 1.0607i
 0.3218 + 1.3316i
-0.7163 - 1.5682i
-0.0063 + 0.3885i
-2.7719 + 0.4802i
-0.6415 + 1.1144i
 0.1233 + 2.3915i
-1.7473 + 1.0213i
-0.2104 - 0.4897i
-0.2180 + 0.4726i
-1.7338 - 0.0738i
-2.4837 + 0.5842i
 2.2902 - 1.5549i
-0.6366 + 0.1176i
-1.5479 - 1.6298i
 0.2879 - 1.8671i
 2.1622 + 1.6863i
-2.7096 - 1.2841i
-3.8171 + 0.2050i
-1.5228 + 0.7527i
 0.6594 - 2.3639i
-1.0123 - 2.4889i
-1.0376 + 2.3999i
 1.7780 + 2.5584i
```

Appendix F2. File to Be Linked with the Source Code for a 64-Point, Radix-4 FFT.

```
*
*   APPENDIX F2
*
*   FILE TO BE LINKED WITH THE SOURCE CODE FOR A 64-POINT, RADIX-4 FFT.
*
        .globl   SINE
        .globl   N
        .globl   M
*
N       .set     64
M       .set     6
*
        .data
*
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
COSINE
        .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   -0.923880
        .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
```

Appendix F3. Link Command File

```
*
*    APPENDIX F3
*
*
*    LINK COMMAND FILE
*
*    DO NOT TYPE IN THESE FIRST SEVEN LINES
-o l2opt64.out
l2fopt.obj
sin64.obj

SECTIONS
{
    .text : {}
    .data : {}
    IN 809800h : { l2fopt.obj(IN) }
    .bss 809C00h: {}
}
```

Doublelength Floating-Point Arithmetic on the TMS320C30

Al Lovrich

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

In the past, extended-precision arithmetic has been implemented only on fixed-point processors. The introduction of the TMS320C30 Digital Signal Processor (DSP), a floating-point 33-MFLOP device, enables us to represent multilength floating-point math in terms of singlelength floating-point math. Extended-precision arithmetic allows designers to have more accuracy in their applications. Some of these applications include digital filtering, FFTs, image processing, control, etc.

This application report describes how to extend the available precision of floating-point arithmetic on the TMS320C30. Our emphasis is on implementing an efficient extension of the available precision while minimizing both the execution time and the memory usage.

The structure of this report is as follows: The first section describes the TMS320C30 DSP floating-point number representation. The second section discusses doublelength arithmetic and some basic definitions. The third section discusses the algorithms used along with the TMS320C30 implementation. An analysis of the error introduced by the algorithm is presented in the fourth section. The last section provides an insight into generating C-callable functions from assembly language routines. Finally, the appendix provides the source listings for the extended-precision arithmetic.

Floating Point Format

The TMS320C30 supports three floating-point formats [1].

- Short floating-point format, used to represent immediate operands, consisting of a 4-bit exponent and a 12-bit mantissa.
- Single-precision format, used for regular floating-point value representation, consisting of an 8-bit exponent and a 24-bit mantissa.
- The extended-precision format, used with the extended-precision registers, consisting of an 8-bit exponent and a 32-bit mantissa.

For the extended-precision algorithms to work properly on the DSP, it is important to start from the highest-precision floating-point format available in the system that is used for basic floating-point operations. The single-precision format is of particular interest in developing the TMS320C30 code for extended-precision floating-point operations. Therefore, a working knowledge of the properties of this format is essential for the concepts presented in this application report.

In the single-precision format, the floating-point number is represented by an 8-bit exponent field (e) in two's complement notation, and a two's complement 24-bit mantissa field (f) with an implied most-significant nonsign bit. Bit 23 of the mantissa indicates the sign (s), as shown in Figure 1.

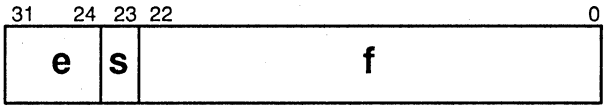


Figure 1. Single-Precision Floating-Point Format of the TMS320C30

Operations are performed with an implied binary point between bits 23 and 22. When the implied most-significant nonsign bit is made explicit, it is located to the immediate left of the binary point after the sign bit. We show the implied bit explicitly throughout this application report for clarity. The floating-point number x is expressed as follows:

$$x = \begin{cases} 01.f \times 2^e & \text{if } s = 0; \\ 10.f \times 2^e & \text{if } s = 1; \\ 0 & \text{if } e = -128,\ s = 0, \text{ and } f = 0 \end{cases}$$

The range and precision available with the TMS320C30 single-precision floating-point format are illustrated by the following values:

Most Positive:	$x = +3.4028234 \times 10^{+38}$
Least Positive:	$x = +5.8774717 \times 10^{-39}$
Least Negative:	$x = -5.8774724 \times 10^{-39}$
Most Negative:	$x = -3.4028236 \times 10^{+38}$

Doublelength Floating-Point – The Basics

The techniques used to develop doublelength results in this application report require a singlelength floating-point system and arithmetic that satisfy certain conditions. The TMS320C30 implementation takes the singlelength system as the highest floating-point precision system available. The algorithms presented do not require a doublelength accumulator with respect to the singlelength system used. The extended-precision formats available are used to control the truncation or rounding of the single-precision results.

The doublelength arithmetic presented here increases precision of a given floating-point operation without the need for a doublelength accumulator. Using this method, the result of the floating-point operations on two single-precision numbers can be determined exactly. If x and y are two such numbers and the desired operation is addition, the result can be represented as a pair of floating-point numbers z and zz. The z value represents

the most significant portion of the floating-point operation, while zz represents the least significant portion of the floating-point operation.

As an example, consider the result of the exact addition of two floating-point numbers x and y that are expressed in the single-precision format of the TMS320C30:

x = 217FFFFFh (decimal: $1.71798682 \times 10^{10}$)
y = 0C7FFFFFh (decimal: 8.19199951×10^{3})

The values are represented in the TMS320C30 binary equivalent as follows:

$x = 2^{33} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1111b$
$y = 2^{12} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1111b$

Addition of two floating-point numbers requires aligning the two variables x and y [1]:

$x = 2^{33} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1111b$
$y = 2^{33} \times 00.000\ 0000\ 0000\ 0000\ 0000\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000b$

As can be seen in this example, most of the precision available for y will not be available to carry out the addition. Maintaining full precision for floating-point addition requires extra mantissa bits beyond the 24 bits available on the DSP. Since the need for such precision is rare, software methods are used to represent the result of the operation as a floating-point number pair (z,zz). In our example, the exact result is represented as follows:

$z = 2^{34} \times 01.000\ 0000\ 0000\ 0000\ 0000\ 0011b$
$zz = 2^{09} \times 01.111\ 1111\ 1111\ 1111\ 1111\ 1000b$

The corresponding hexadecimal representation of (z,zz) is shown below:

z = 22000003h (decimal: $1.71798753 \times 10^{10}$)
zz = 097FFFF8h (decimal: 1.0239995×10^{3})

Some definitions are basic to the development of concepts in this report. First is the definition of the floating-point operations over a system *R*. The system contains all the possible floating-point numbers that the single-precision format of the TMS320C30 can represent. All the floating-point arithmetic is carried out in base 2. Therefore, *R* can be represented as follows on the TMS320C30:

$R = \{x | x = m(x)2^{e(x)},\ |m(x)| < 2^{24},\ -128 < e(x) < 127\}$

A floating-point operation is *faithful* if the result of the operation fl(x *y) equals either:

The largest element of *R* that is smaller than or equal to (x * y) or

The smallest element of *R* that is larger than or equal to (x * y)

where * represents one of the following floating-point operations: +, −, ×, ÷. In other words, faithful refers to truncating the floating-point operation result. The floating-point

multiplier on the TMS320C30 saves the upper 40 bits of the mantissa in one of the extended-precision registers [1] and drops the least significant byte of the result. By this definition, the floating-point multiplication on the TMS320C30 is *faithful*. Since the algorithms require the floating-point result to be in single-precision format, the floating-point multiplication on the DSP must therefore be followed by a second truncation step. Saving the contents of the extended-precision register to a memory location or masking off the low 8 bits results in truncation.

A floating-point operation is *optimal* if for all x and y, the result of fl(x * y) is an element of *R* nearest to (x * y). In other words, the round-off error should not exceed one-half of the last remaining bit position. This is commonly referred to as *rounding*.

The results of floating-point operations on the TMS320C30 are stored in the extended-precision registers [1]. The extended-precision register adds 8 bits of precision to the floating-point arithmetic result. Execution of the RND (round) instruction forces the result of the floating-point arithmetic to be *optimal*. When you round the result of the addition or subtraction operations on the TMS320C30, these floating-point operations become *optimal*.

Implementing Doublelength Floating-Point Arithmetic

This section presents the algorithms used in implementing doublelength arithmetic in pseudo-code for a number of fundamental floating-point operations. The basic idea of doublelength arithmetic can be extended to multiplelength precision, given that the start of the implementation is based on the highest precision available on the system. Therefore, to achieve quadruplelength results, the same algorithm can be applied to doublelength values, and so on. The implementation is based on the theoretical results presented in Reference [2].

Exact Singlelength Addition

In this discussion of the algorithm used to carry out *exact* addition and its implementation on the TMS320C30 DSP, the term *exact* refers to performing an operation on two floating-point numbers, x and y, and obtaining a doublelength floating-point number pair (z,zz) to represent the result. In this implementation, we have not accounted for floating-point exponent overflow or underflow. For this algorithm to produce a correct result, the floating-point addition and subtraction must be *optimal*.

The purpose of *exact* addition is to find a term, zz, that satisfies Equation (2).

$$z + zz = x + y \tag{2}$$

Equation (2) can be rewritten as

$$zz = y - (z - x) \tag{3}$$

Equation (3) can be expanded into Equation (4).

$$w = z - x \tag{4}$$
$$zz = y - w$$

In particular, $|x| > |y|$ must be valid for Equation (4) to be valid. Implementation of Equation (4) on the TMS320C30 always generates the exact correction term zz if the result of floating-point addition operation is made *optimal*. This requirement guarantees that the result of single-precision floating-point add and subtract belongs to system R. By swapping the x and y values when $|x| < |y|$, the condition for obtaining an *exact* result is met.

The algorithm requires that x and y be normalized. Normalization guarantees that the floating-point number has only one sign bit, and that sign bit is followed by nonsign bits [1]. Floating-point addition on the TMS320C30 assumes that the operands are normalized.

The TMS320C30 assembly code for obtaining the doublelength sum of two singlelength floating-point numbers x and y is shown in Appendix A. First, the values for x and y are interchanged when $|x| < |y|$. When you add x and y values, the number with the smaller exponent, y, is shifted repeatedly until the exponents of x and y are equal and their mantissas are aligned. We have now calculated the singlelength number, z, that satisfies Equation (2). Since the floating-point addition on the TMS320C30 is made optimal by rounding, the extra precision is, in effect, dropped. The extra precision value, zz, is obtained by implementing Equation (4). Figure 2 is a graphical representation of the implemented algorithm. The figure also shows the relationship between doublelength number pair (z,zz) and singlelength floating-point numbers and their representation on the TMS320C30.

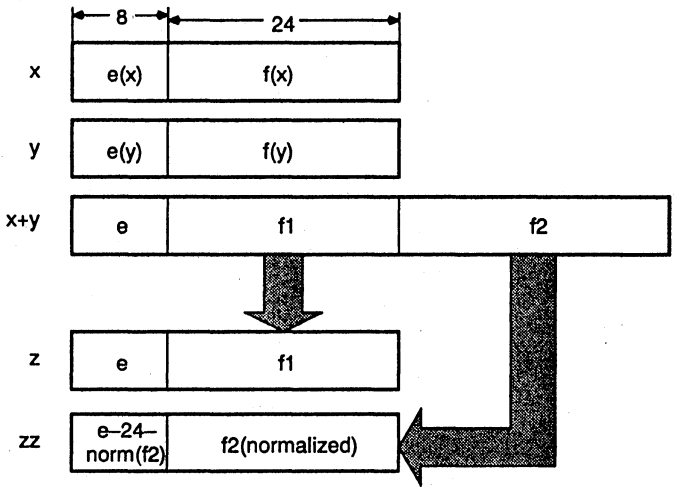


Figure 2. Exact Singlelength Addition

The same algorithm can be used to implement exact floating-point subtraction on the DSP. This is accomplished by negating the second operand and performing an exact addition.

Doublelength Addition

A natural extension of exact singlelength addition and subtraction is its application to doublelength arithmetic. Figure 3 shows an algorithm for implementing doublelength addition on the DSP. Using this algorithm, you can add two doublelength numbers (x,xx) and (y,yy) and represent the result as a doublelength number (z,zz).

The algorithm requires forming a doublelength number (r,rr) that represents an exact addition of x and y. Generating a second number, s = ((rr + yy) + xx), results in a number pair (r,s) that approximates the addition of (x,xx) and (y,yy). Finally, an exact addition of r and s generates a doublelength number (z,zz) that has the same value as (x,xx) + (y,yy).

To obtain exact results for addition and subtraction, subtraction and addition must be optimal; this is guaranteed by following each subtraction or addition instruction on the DSP with a round instruction.

```
; Calculate the doublelength sum of (x,xx) and (y,yy),
; the result being (z,zz)
;
        r = x + y;
        if (abs(x)>abs(y))
              s = x - r + y + yy + xx;
        else
              s = y - r + x + xx + yy;
        z = r + s;
        zz = r - z + s;
```

Figure 3. Doublelength Addition

Exact Singlelength Multiplication

The exact singlelength multiplication is shown in Figure 4. The algorithm requires breaking the x and y mantissas into half-length numbers, referred to as head (hx,hy) and tail (tx,ty) sections [2]. This algorithm requires addition and subtraction to be optimal and multiplication faithful. The TMS320C30 DSP multiplication result is faithful if the contents of the extended-precision register are truncated.

To split x and y into two half-length numbers, a constant value is needed that is dependent on the number of available digits. The TMS320C30 device has t = 24 bits of mantissa in the single-precision format. Equation (5) shows that head section hx is chosen to be as near to the value of x as possible.

$$hx = round(m(x)2^{-t1})2^{e(x)+t1} \tag{5}$$

Also, t1 is chosen to be approximately one-half of the available precision, or 12, on the processor. This effectively breaks the mantissa into half-length values. Equation (5) shows that hx is obtained by rounding and is defined to be an element of R{t1}. The tail section tx is easily obtained by subtracting hx from x. Since floating-point subtraction can be made optimal on the TMS320C30, it follows that tx is an element of R{t1 − 1}. Setting the constant equal to 2^{12} does not always satisfy Equation (5) when t is even. When the constant is set to $2^{12} + 1$, the definition of Equation (5) is satisfied. The proof for the above is given in Reference [2].

```
; Calculate the exact product of x and y, the result being
; a doublelength number (z,zz). This algorithm uses the
; following syntax when called from a user program as shown
; mult12 (x,y,z,zz);
;
        p = x × constant;
        hx = x − p + p;
        tx = x − hx;

        p = y × constant;
        hy = y − p + p;
        ty = y − hy;

        p = hx × hy;
        q = hx × ty + tx × hy;
        z = p + q;
        zz = p − z + q + tx × ty;
```

Figure 4. Exact Singlelength Product

Doublelength Multiplication

The doublelength multiplication algorithm, shown in Figure 5, relies on the singlelength algorithm discussed earlier. The algorithm generates a nearly doublelength approximation of the output result (c,cc). Note that the exact singlelength multiplication routine is used for this approximation. Exact addition is used to generate a doublelength floating-point number that is the closest approximation to the actual result.

The doublelength product program implementation uses the TMS320C30 stack capabilities to save some intermediate variables. These programs are written to be used as callable functions or macros in your program. In either case, the stack pointer must be set to a valid memory segment for proper code execution.

```
; Calculate the doublelength product of (x,xx) and (y,yy)
; the result being a nearly doublelength number (z,zz).
; Program uses exact singlelength multiplication, mult12 (.).
;
        mult12 (x, y, c, cc);
        cc = x × yy + xx × y + cc;
        z = c + cc;
        zz = c − z + cc;
```

Figure 5. Exact Doublelength Product

Doublelength Quotient and Square Root

Figures 6 and 7 show the algorithm used in calculating the doublelength quotient and doublelength square root routines. Singlelength multiplication is used to generate a doublelength approximation of the quotient or square root values. As with doublelength multiplication, exact addition is used to generate a doublelength floating-point result.

```
;
; Calculates the doublelength quotient of (x,xx) and (y,yy)
; the result being (z,zz)
;
        c = x / y;
        mult12(c, y, u, uu);
        cc = (x − u − uu + xx − c × yy) / y;
        z = c + cc;
        zz = c − z + cc;
```

Figure 6. Doublelength Quotient

```
; Calculate the doublelength square root of (x,xx), the
; result being (z,zz)
;
        if (x>0) {
                c = sqrt (x);
                mult12 (c, c, u, uu);
                cc = (x − u − uu + xx) × 0.5 / c;
                z = c + cc;
                zz = c − z + cc;}
        else {
                z = zz = 0.};
```

Figure 7. Doublelength Square Root

Error Analysis

This section discusses and determines an upper bound for the error generated in forming a doublelength result. The value of the doublelength number (z,zz) is equal to z + zz. Singlelength addition, subtraction, and multiplication results are always exact. In doublelength addition, any error introduced in the end result is generated by calculating the zz term. An upper bound error magnitude has been calculated in Reference [2] and is shown in Equation (6) as follows:

$$|E^+| \leq \{|x+xx| + |y+yy|\} \times 2^{2-2t} = |Z| \times 2^{2-2t} \quad (6)$$

where t = 24 for this system. This gives an upper bound of $|Z| \times 2^{-46}$, or approximately $|Z| \times 1.42 \times 10^{-14}$. This translates to a theorical accuracy greater than 13 decimal places. Table 1 shows an example of doublelength addition using the exact addition algorithm previously described. The numbers in the left column represent TMS320C30 hexadecimal notation for the floating-point results, and (z,zz) is the decimal equivalent of the doublelength output result. Appendix B shows a listing of C programs (exact) that convert from TMS320C30 hexadecimal notation to decimal notation.

Table 1. Exact Singlelength Arithmetic Examples

Singlelength Addition		
x = 217FFFFFh		
y = 0C7FFFFFh		
z = 22000003h	(z,zz) =	17179876351.9995117 (Exact)
zz = 097FFFF8h		17179876351.9995117 (DSP)
x = FC7C8923h		
y = 0A29A7E5h		
z = 0A29ABD8h	(z,zz) =	1357.37010409682989 (Exact)
zz = EFA46000h		1357.37010409682989 (DSP)
Singlelength Multiplication		
x = 0F7FFFFFh		
y = 21FFFFFFh		
z = 30800000h	(z,zz) =	−562949986975740 (Exact)
zz = 18800002h		−562949986975740 (DSP)
x = FC7CB923h		
y = 0A29A7E5h		
z = 07277BF7h	(z,zz) =	167.484236862815123 (Exact)
zz = EBA714F0h		167.484236862815123 (DSP)

The doublelength product, quotient, and square-root algorithms all have a small relative error. The upperbound error magnitude for each is given in Equations (7) through (9).

$$|E^{\times}| = (|x + xx| \times |y + yy|) \times 11 \times 2^{-48} \tag{7}$$

$$|E^{\div}| = (|x + xx| \div |y \times yy|) \times 21.1 \times 2^{-48} \tag{8}$$

$$|E^{\sqrt{\ }}| = \text{sqrt}(|x + xx|) \times 12.7 \times 2^{-48} \tag{9}$$

Equation (7) establishes an upperbound of $|Z| \times 3.9 \times 10^{-14}$, or approximately 13 decimal digits of accuracy for doublelength multiplication. Similarly, an upperbound of $|Z| \times 7.5 \times 10^{-14}$, or greater than 13 decimal digits for the doublelength square-root algorithm, is established. Table 2 shows examples for each algorithm discussed, along with the algorithm output and expected theorical output.

Table 2. Exact Doublelength Arithmetic Examples

Doublelength Multiplication		
x = 22000000h		
xx = 097FFFFEh		
y = 21000001h		
yy = 097FFFFEh		
z = 43000002h	(z,zz) =	$1.47573996570139475 \times 10^{20}$ (Exact)
zz = 2A7FFFFCh		$1.47573996570139427 \times 10^{20}$ (DSP)
x = 22000003h		
xx = 097FFFF8h		
y = 0A29ABD8h		
yy = EFA46000h		
z = 2C29ABDDh	(z,zz) =	23319450552284.2434 (Exact)
zz = 13907DC2h		23319450552284.1250 (DSP)
Doublelength Quotient		
x = 43000002h		
xx = 2A7FFFFCh		
y = 2C29ABDDh		
yy = 13907DC2h		
z = 1641205Ah	(z,zz) =	6328365.08044074177 (Exact)
zz = FC24BE20h		6328365.08044075966 (DSP)
x = 22000000h		
xx = 097FFFFEh		
y = 21000001h		
yy = 097FFFFEh		
z = 007FFFFDh	(z,zz) =	1.99999964237223082 (Exact)
zz = D3400000h		1.99999964237217398 (DSP)
Doublelength Square Root		
x = 2C2BDD00h		
xx = 3907DC2h		
z = 61451A4h	(z,zz) =	4860114.04539400958 (Exact)
zz = FB39EF11h		4860114.04539400712 (DSP)
x = 21000001h		
xx = 097FFFFEh		
z = 103504F5h	(z,zz) =	92681.9110722252960 (Exact)
zz = F7BC0784h		92681.9110722253099 (DSP)

Note that the results were obtained using the programs shown in Appendix B. The C programs were created and compiled on a 80386-based microcomputer running under MS-DOS 3.3.

How to Generate C-Callable Functions

The source listings for the extended-precision arithmetic presented in Appendix A are optimized for execution speed and code size. These routines are designed to be used as macros in a user program environment or, with a few adjustments, as a C function.

This section provides an overview of TMS320C30 C compiler calling conventions necessary to create functions that can be added to the C compiler library. You need a working knowledge of C language to understand the terminology in this section [4, 5, 6].

The C compiler uses the processor stack to pass arguments to functions, store local variables, and save temporary values. The C compiler uses two registers of the TMS320C30 to manage the stack pointer (SP) and the frame pointer (AR3).

When a C program calls a function, it must

1. Push the arguments onto the stack,
2. Call the function, and
3. Pop the arguments off the stack,

in that order.

On the other hand, the called C function must perform the following tasks:

1. Set up a local frame by saving the old frame pointer on the stack.
2. Assign the new frame pointer to the current value of stack pointer.
3. Allocate the frame.
4. Save any dedicated registers that the function modifies.
5. Execute function code.
6. Store a scalar value in R0.
7. Deallocate the frame.
8. Lastly, restore the old frame pointer [4].

The following code segment shows the singlelength addition routine modified to be in C-callable form. Note that registers R4 through R7 and AR4 through AR7 are dedicated registers used by the compiler. These registers must be saved as floating-point values.

```
single    .set    0FFh
fp        .set    ar3
x         .set    r0
y         .set    r1
z         .set    r2
zz        .set    r3
```

```
w       .set    r4
x1      .set    r2
y1      .set    r3
        .global __add12:
        .width  96
        .text
__add12:
        push    fp              ; Save old fp
        pushf   r4
        push    r4
        ldi     sp,fp           ; Point to top of stack
        ldi     *-fp[2],r0      ; Load x into r0
        ldi     *-fp[3],r1      ; Load y into r1
        absf    x,x1
        absf    y,y1
        cmpf    y1,x1           ; |x| > |y|
        ldflt   x,x1
        ldflt   y,x
        dflt    x1,y
;
        addf3   x,y,z           ; z = x + y
        rnd     z
        subf3   x,z,w           ; Form w = z - x
        rnd     w
        subf3   w,y,zz          ; zz = y - (y - w)
        rnd     zz
        pop     r4
        popf    r4
        pop     fp              ; Restore fp
        retsu
        .end
```

Conclusion

This report presented an implementation of extended-precision arithmetic routines for the TMS320C30 DSP. The programs presented include singlelength floating-point addition, subtraction, and multiplication, which produce exact doublelength results. Doublelength floating-point addition, subtraction, multiplication, division, and square root were also presented. The doublelength floating-point routines all had a small relative error that appeared in the correction term zz. However, it has been shown that the accuracy of the doublelength floating-point result is at least 13 decimal digits. Table 3 is a summary of information about the routines contained in Appendices A and B. Execution times shown

in the table are given only for the routines in Appendix A. These times do not include the call and return if the routine is implemented as a called function. They also do not include any context saves and restores that may be required.

Table 3. Summary Information

Routine	Mnemonic	Appendix	Code Size (Words)	Execution (Cycles)
Singlelength Add	_add12	A1	12	12
Doublelength Add	_dbladd	A2	25	25
Singlelength Multiply	_mult12	A3	35	35
Doublelength Multiply	_mult2	A4	51	51
Doublelength Divide	_div2	A5	115	115
Doublelength Square Root	_sqrt2	A6	163	163
Change Two Single-Precision TMS320C30 Numbers to One Double-Precision Result	C30DBL	B1		
Change Two Double-Precision TMS320C30 Numbers to a Double-Precision Result	C30DBL2	B2		

References

[1.] *Third-Generation TMS320 User's Guide* (literature number SPRU031), Texas Instruments, Inc., 1988.

[2.] Dekker, T.J.,"A Floating-Point Technique for Extending the Available Precision", Numer. Math. 18, 1971, pp 224-242.

[3.] Linnainmaa, S.,"Software for Doubled-Precision Floating-Point Computations", *ACM Transactions on Mathematical Software*, Vol. 7, No. 3, Sept. 1981, pp 272-283.

[4.] *TMS320C30 C Compiler* (literature number SPRU034), Texas Instruments, Inc., 1988.

[5.] Kernigan, B.W. and Ritchie, D.M., *The C Programming Language*, 2nd Revision, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[6.] Kochan, S.G., *Programming in C*, Second Edition, Howard K. Sams, Indianapolis, Indiana, 1988.

Appendix A

Appendix A1. Single Length Add

```
*****************************************************
*   FUNCTION DEF : _add12
*
*   AUTHOR: Al Lovrich 2/21/89
*     Texas Instruments, Inc.
*
*   Entry Conditions:
*   Upon entry (r0,r1) contains (x,y)
*   Exit Conditions:
*   Upon exit  (r2,r3) contains (z,zz).
*   Registers Affected:
*   r0, r1, r2, r3, r4
*
*   Revision: Original
*   Execution Time:  12 cycles
*****************************************************
single      .set        0ffh
            .global     _add12
x           .set        r0
y           .set        r1
z           .set        r2
zz          .set        r3
w           .set        r4
x1          .set        r2
y1          .set        r3
            .text
_add12:
            absf        x,x1
            absf        y,y1
            cmpf        y1,x1           ; |x| > |y| ?
            ldf1t       x,x1            ; if not, exchange x & y
            ldf1t       y,x
            ldf1t       x1,y
*
            addf3       x,y,z           ; z = x + y
            rnd         z
            subf3       x,z,w           ; form w = z - x
            rnd         w
subf3       w,y,zz          ; zz = y - w
rnd         zz
retsu
.end
```

Appendix A2. Double Length Add

```
*********************************************************
* FUNCTION DEF : _dbladd
*
* AUTHOR: Al Lovrich   2/21/89
*         Texas Instruments, Inc.
*
* Entry Conditions:
*      Upon entry (r0,r1) contains (x,xx) and
*      (r2,r3) contain (y,yy).
* Exit Conditions:
*      Upon exit (r4,r5) contains (z,zz).
* Registers Affected:
*      r0, r1, r2, r3, r4, r5, r6, r7
*
* Revision: Original
* Execution time:  25 cycles
*********************************************************
        .global _dbladd
x       .set    r0
xx      .set    r1
y       .set    r2
yy      .set    r3
z       .set    r4
zz      .set    r5
x1      .set    r6
y1      .set    r7
r       .set    r6
s       .set    r7

        .text
_dbladd:
        absf    x,x1
        absf    y,y1
        cmpf    y1,x1                   ; check for |x| > |y|
        ldflt   x,x1                    ; if not, exchange (x,xx)
        ldflt   xx,y1                   ; and (y,yy)
        ldflt   y,x
        ldflt   yy,xx
        ldflt   x1,y
        ldflt   y1,yy
;
        addf3   x,y,r                   ; r = x + y
        rnd     r
*
        subf3   r,x,s                   ; s = x - r
        rnd     s
        addf3   y,s,s                   ; s = x - r + y
        rnd     s
        addf    yy,s                    ; s = x - r + y + yy
        rnd     s
        addf    xx,s                    ; s = x - r + y + yy + xx
        rnd     s
*
        addf3   s,r,z                   ; z = r + s
        rnd     z
*
        subf3   z,r,zz                  ; zz = r - z
        rnd     zz
        addf3   s,zz,zz                 ; zz = r - z + s
        rnd     zz
        retsu
         .end
```

Appendix A3. Single Length Multiply

```
*******************************************************
*  FUNCTION DEF : _mult12
*
*  AUTHOR: Al Lovrich  2/21/89
*    Texas Instruments, Inc.
*
*  Entry Conditions:
*       Upon entry (r0,r1) contains (x,y)
*  Exit Conditions:
*       Upon exit  (r0,r1) contains (z,zz).
*  Registers Affected:
*       r0, r1, r2, r3, r4, r5, r6, r7
*
*  Revision: Original
*  Execution Time: 35 Cycles
*******************************************************
          .global   _mult12
single    .set      0ffh
x         .set      r0
y         .set      r1
p         .set      r2
x         .set      r3
tx        .set      r4
q         .set      r5
hy        .set      r5
ty        .set      r6
z         .set      r0
zz        .set      r1
.temp     .set      r7
          .text
_mult12:
          ldf       @constant,temp
          mpyf3     temp,x,p        ; p = x * constant
          andn      single,p        ; fl(*) is faithful
*
          subf3     p,x,hx          ; hx = x - p
          rnd       hx


          addf3     hx,p,hx         ; hx = x - p + p
          rnd       hx
*
          subf3     hx,x,tx         ; tx = x - hx
          rnd       tx
*
          mpyf3     temp,y,p        ; p = y * constant
          andn      single,p        ; fl(*) is faithful
*
          subf3     p,y,hy          ; hy = y - p
          rnd       hy
          addf3     hy,p,hy         ; hy = y - p + p
          rnd       hy
*
          subf3     hy,y,ty         ; ty = y - hy
          rnd       ty
*
          mpyf3     hx,hy,p         ; p = hx * hy
          andn      single,p        ; fl(*) is faithful
*
          mpyf3     hx,ty,temp      ; temp = hx * ty
          andn      single,temp     ; fl(*) is faithful
          mpyf3     tx,hy,q         ; q = tx * hy
          andn      single,q        ; fl(*) is faithful
          addf3     q,temp,q        ; q = hx * ty + tx * hy
          rnd       q
*
          addf3     p,q,z           ; z = p + q
          rnd       z
*
          subf3     z,p,zz          ; zz = p - z
          rnd       zz
          addf      q,zz            ; zz = p - z + q
          rnd       zz
          mpyf3     tx,ty,temp      ; temp = tx * ty
          andn      single,temp     ; fl(*) is faithful
          addf3     zz,temp,zz      ; zz = p - z + q + tx * ty
          rnd       zz
*
          retsu
          .data

constant:
          .float    4097            ; constant = 2^(24-24/2)+1
          .end
```

Appendix A4. Double Length Multiply

```
*************************************************
*   FUNCTION DEF : _mult2
*
*   AUTHOR: Al Lovrich    2/21/89
*           Texas Instruments, Inc.
*   Entry Conditions:
*         Upon entry (r0,r1) contains (x,y),
*         and (r2,r3) contains (xx,yy).
*   Exit Conditions:
*         Upon exit  (r0,r1) contains (z,zz).
*   Registers Affected:
*         r0, r1, r2, r3, r4, r5, r6, r7
*
*   Algorithm used:
*         mult12(x, y, c, cc);
*         cc = x * yy + xx * y + cc;
*         z  = c + cc;
*         zz = c - z + cc;
*
*   Revision: Original
*   Execution Time: 51 cycles
*************************************************
            .global     _mult2
single      .set        0ffh
x           .set        r0
y           .set        r1
p           .set        r2
hx          .set        r3
tx          .set        r4
q           .set        r5
hy          .set        r5
ty          .set        r6
z           .set        r0
zz          .set        r1
xx          .set        r2
yy          .set        r3
c           .set        r4
cc          .set        r6
temp0       .set        r6
temp        .set        r7
            .text
_mult2:
            mpyf3       x,yy,temp0      ; temp0 = x*yy
            andn        single,temp0
            mpyf3       y,xx,temp       ; temp = y*xx
            andn        single,temp
            addf        temp0,temp      ; temp = x*yy + y*xx
            rnd         temp
            pushf       temp            ; (x*yy + y*xx)
*
*  mult12(x, y, c, cc)
*
            ldf         @constant,temp
            mpyf3       temp,x,p        ; p = x * constant
            andn        single,p
*
            subf3       p,x,hx          ; hx = x - p
            rnd         hx
            addf3       hx,p,hx         ; hx = x - p + p
            rnd         hx
*
            subf3       hx,x,tx         ; tx = x - hx
            rnd         tx
*
            mpyf3       temp,y,p        ; p = y * constant
            andn        single,p
*
            subf3       p,y,hy          ; hy = y - p
            rnd         hy
            addf3       hy,p,hy         ; hy = y - p + p
            rnd         hy
*
            subf3       hy,y,ty         ; ty = y - hy
            rnd         ty
*
            mpyf3       hx,hy,p         ; p = hx * hy
            andn        single,p
*
            mpyf3       hx,ty,temp      ; temp = hx * ty
            andn        single,temp
            mpyf3       tx,hy,q         ; q = tx * hy
            andn        single,q
            addf3       q,temp,q        ; q = hx * ty + tx * hy
            rnd         q
*
            mpyf3       tx,ty,temp      ; temp = tx * ty
            andn        single,temp
            addf3       p,q,c           ; c = p + q
            rnd         c
*
            subf3       c,p,cc          ; cc = p - c
            rnd         cc
            addf        q,cc            ; cc = p - c + q
            rnd         cc
            addf        temp,cc         ; cc = p - c + q + tx * ty
            rnd         cc
*
*  restore variables
*
break:
            popf        temp            ; x*yy + y*xx
*
*  cc = x * yy + xx * y + cc
*
            addf        temp,cc         ; cc = x * yy + xx * y + cc
            rnd         cc
*
*  z = c + cc
*
            addf3       cc,c,z          ; z = c + cc
            rnd         z
*
*  zz = c - z + cc
*
            subf3       z,c,zz          ; zz = c - z
            rnd         zz
            addf3       zz,cc,zz        ; zz = c - z + cc
            rnd         zz
*
            retsu
            .data
constant:
            .float      4097            ; constant = 2^(24-24/2)+1
            .end
```

Appendix A5. Double Length Divide

```
*************************************************
*   FUNCTION DEF : _div2
*
*   AUTHOR: Al Lovrich 2/21/89
*     Texas Instruments, Inc.
*
*   Entry Conditions:
*   Upon entry (r0,r1) contains (x,y),
*   and (r2,r3) contains (xx,yy).
*   Exit Conditions:
*   Upon exit  (r0,r1) contains (z,zz).
*   Registers Affected:
*   r0, r1, r2, r3, r4, r5, r6, r7
*
*   Algorithm used:
*   c = x / y;
*   multi2(c, y, u, uu);
*   cc = ( x - u - uu + xx - c * yy ) / y;
*   z  = c + cc;
*   zz = c - z + cc;
*
*   Revision: Original
*   Execution Time: 115 Cycles
*************************************************
          .global   _div2
single    .set      0ffh
x         .set      r0
y         .set      r1
p         .set      r2
hx        .set      r3
tx        .set      r4
yl        .set      r4
q         .set      r5
hy        .set      r5
ty        .set      r6
z         .set      r0
zz        .set      r1
xx        .set      r2
yy        .set      r3
temp      .set      r7
temp1     .set      r3
temp2     .set      r1
c         .set      r2
cc        .set      r3
u         .set      z
uu        .set      zz
          .text
*
_div2:
          pushf     yy              ; save yy
          pushf     xx              ; save xx
          pushf     x               ; save x
          pushf     y               ; save y
*
*  c = x / y;
*
*  The floating-point number v is stored in R1. After the computation is
*  completed, 1/v is also stored in R4.
*
*  Register used as input: R1
*  Registers modified: R0, R1, R2, R3
*  Register containing result: R4
*
inv_f:    ldf       r1,r3           ; v is saved for later.
          absf      r1              ; The algorithm uses v = |v|.
*
*  Extract the exponent of v.
*
          pushf     r1
          pop       r0
          ash       -24,r0          ; The 8 LSBs of R1 contain the exponent
*                                   ; of v.
*
*  A few comments on boundary conditions. If e = -128, then v = 0. The
*  following x[0] calculation yields R1 =  -128 - 1 = 127 and the algoritm
*  overflow and saturate since x[0] is large. This seems reasonable. If 127,
*  the R1 = -127 - 1 = -128. Thus x[0] = 0 and this will cause the algorithm
*  to yield zero. Since the mantissa of v is always between 1 this is also
*  reasonable. As a result, boundary conditions are handled automatically in
*  a reasonable fashion.
*
*  x[0] formation given the exponent of v.
*
          negi      r0
          subi      1,r0            ; Now we have -e-1, the exponent of x[0].
          ash       24,r0
          push      r0
          popf      r0              ; Now R1 = x[0] = 1.0 * 2**(-e-1).
*
*  Now the iterations begin.
*
          mpyf3     r0,r1,r2        ; R2 = v * x[0]
          andn      single,r2
          subrf     2.0,r2          ; R2 = 2.0 - v * x[0]
          rnd       r2
          mpyf      r2,r0           ; R1 = x[1] = x[0] * (2.0 - v * x[0])
          andn      single,r0
*
          mpyf      r0,r1,r2        ; R2 = v * x[1]
          andn      single,r2
          subrf     2.0,r2          ; R2 = 2.0 - v * x[1]
          rnd       r2
          mpyf      r2,r0           ; R1 = x[2] = x[1] * (2.0 - v * x[1])
          andn      single,r0
```

```
*
        mpyf    r0,r1,r2    ; R2 = v * x[2]
        andn    single,r2
        subr    2.0,r2      ; R2 = 2.0 - v * x[2]
        rnd     r2
        mpyf    r2,r0       ; R1 = x[3] = x[2] * (2.0 - v * x[2])
        andn    single,r0
*
        mpyf    r0,r1,r2    ; R2 = v * x[3]
        andn    single,r0
        subrf   2.0,r2      ; R2 = 2.0 - v * x[3]
        rnd     r2
        mpyf    r2,r0       ; R1 = x[4] = x[3] * (2.0 - v * x[3])
*
        andn    single,r0   ; This minimizes error in the LSBs.
*
* For the last iteration we use the formulation:
* x[5] = (x[4] * (1.0 - (v * x[4]))) + x[4]
*
        mpyf    r0,r1,r2    ; R2 = v * x[4] = 1.0..01.. => 1
        andn    single,r2
        subrf   1.0,r2      ; R2 = 1.0 - v*x[4] = 0.0..01... => 0
        rnd     r2
        mpyf    r0,r2       ; R2 = x[4] * (1.0 - v * x[4])
        andn    single,r2
        addf    r2,r0       ; R2 = x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
*
        rnd     r0,r1       ; Round since this is follow by a MPYF.
*
* Now the case of v < 0 is handled.
*
        negf    r1,r2
        ldf     r3,r3       ; This sets condition flags.
        ldfn    r2,r1       ; If v < 0, then R1 = -R1
*
        ldf     r1,r4       ; save 1/y
*
* restore variables
*
        popf    y           ; restore y
        popf    x           ; restore x
        pushf   x           ; save x
*
        mpyf    y1,x        ; c = x * ( 1/y)
        andn    single,x
*
* save variables
*
        pushf   x           ; save c
        pushf   y1          ; save 1/y
*
* multi2(c, y, u, uu)
*
        ldf     @constant,temp
        mpyf3   temp,x,p    ; p = x * constant
        andn    single,p
*
        subf3   p,x,hx      ; hx = x - p
        rnd     hx
        addf3   hx,p,hx     ; hx = x - p + p
        rnd     hx
*
        subf3   hx,x,tx     ; tx = x - hx
        rnd     tx
*
        mpyf3   temp,y,p    ; p = y * constant
        andn    single,p
*
        subf3   p,y,hy      ; hy = y - p
        rnd     hy
        addf3   hy,p,hy     ; hy = y - p + p
        rnd     hy
*
        subf3   hy,y,ty     ; ty = y - hy
        rnd     ty
*
        mpyf3   hx,hy,p     ; p = hx * hy
        andn    single,p
*
        mpyf3   hx,ty,temp  ; temp = hx * ty
        andn    single,temp
        mpyf3   tx,hy,q     ; q = tx * hy
        andn    single,q
        addf3   q,temp,q    ; q = hx * ty + tx * hy
        rnd     q
*
* perform tx * ty operation and store the result in temp. This is to
* optimize use of registers on the device.
*
        mpyf3   tx,ty,temp  ; temp = tx * ty
        andn    single,temp
        addf3   p,q,u       ; u = p + q
        rnd     u
*
        subf3   u,p,uu      ; uu = p - u
        rnd     uu
        addf    q,uu        ; uu = p - u + q
        rnd     uu
```

```
        popf    y1          ; restore 1/y
        popf    c           ; restore c
        popf    temp        ; restore x
        subf3   u,temp,cc   ; cc = x - u
        rnd     cc
        subf    uu,cc       ; cc = x - u - uu
        rnd     cc
        popf    temp        ; restore xx
        addf    temp,cc     ; cc = x - u - uu + xx
        rnd     cc
        popf    temp        ; restore yy
        mpyf    c,temp      ; c * yy
        andn    single,temp
        subf    temp,cc     ; cc = x - u - uu + xx - c * yy
        rnd     cc
        mpyf    y1,cc       ; cc = ( x - u - uu + xx - c * yy ) / y
        andn    single,cc
*
*  z = c + cc
*
        addf3   c,cc,z      ; z = c + cc
        rnd     z
*
*  zz = c - z + cc
*
        subf    z,c,zz      ; zz = c - z
        rnd     zz
        addf    cc,zz       ; zz = c - z + cc
        rnd     zz
*
        retsu
        .data
constant:
        .float  4097        ; constant = 2^(24-24/2)+1
        .end
```

Appendix A6. Double Length Square Root

```
*****************************************************
*  FUNCTION DEF : _sqrt2
*
*  AUTHOR: Al Lovrich    2/21/89
*          Texas Instruments, Inc.
*
*  Entry Conditions:
*       Upon entry (r0,r1) contains (x,xx).
*  Exit Conditions:
*       Upon exit  (r0,r1) contains (z,zz).
*  Registers Affected:
*       r0, r1, r2, r3, r4, r5, r6, r7
*
*  Algorithm used:
*       c = sqrt(x);
*       mult12(c, c, u, uu);
*       cc = ( x - u - uu + xx ) * 0.5 / c;
*       z  = c + cc;
*       zz = c - z + cc;
*
*  Revision: Original
*  Execution Time: 163 Cycles
*****************************************************
          .global   _sqrt2
single    .set      0ffh
x         .set      r0
y         .set      r1
p         .set      r2
hx        .set      r3
tx        .set      r4
q         .set      r5
hy        .set      r5
ty        .set      r6
z         .set      r0
zz        .set      r1
xx        .set      r1
temp      .set      r7
c         .set      r2
cc        .set      r3
u         .set      z
uu        .set      zz
c1        .set      r0
          .text
_sqrt2:
*
*  c = sqrt(x)
*
*  Extract the exponent of v.
*
          ldf       r0,r3       ; save v
          retsle                ; return if number non-positive
          pushf     xx          ; save xx
          pushf     x           ; save x
          mpyf      2.0,r0      ; add a rounding bit in the exponent
          andn      single,r0
          pushf     r0
          pop       r1
          ash       -25,r1      ; The 8 LSBs of R1 contain 1/2 the expon
*
*  x[0] formation given the exponent of v.
*
          negi      r1
          ash       24,r1
          push      r1
          popf      r1          ; Now r1 = x[0] = 1.0 * 2**(-e/2).
*
*  Generate v/2.
*
          mpyf      0.25,r0     ; v/2 and take rounding bit out.
          andn      single,r0
*
*  Now the iterations begin.
*
          mpyf      r1,r1,r2    ; r2 = x[0] * x[0]
          andn      single,r2
          mpyf      r0,r2       ; r2 = (v/2) * x[0] * x[0]
          andn      single,r2
          subrf     1.5,r2      ; r2 = 1.5 - (v/2) * x[0] * x[0]
          rnd       r2
          mpyf      r2,r1       ; r1 = x[1] = x[0] * (1.5 - (v/2)*x[0]*x
          andn      single,r1
*
          mpyf      r1,r1,r2    ; r2 = x[1] * x[1]
          andn      single,r2
          mpyf      r0,r2       ; r2 = (v/2) * x[1] * x[1]
          andn      single,r2
          subrf     1.5,r2      ; r2 = 1.5 - (v/2) * x[1] * x[1]
          rnd       r2
          mpyf      r2,r1       ; r1 = x[2] = x[1] * (1.5 - (v/2)*x[1]*x
          andn      single,r1
*
          mpyf      r1,r1,r2    ; r2 = x[2] * x[2]
          andn      single,r2
          mpyf      r0,r2       ; r2 = (v/2) * x[2] * x[2]
          andn      single,r2
          subrf     1.5,r2      ; r2 = 1.5 - (v/2) * x[2] * x[2]
          rnd       r2
          mpyf      r2,r1       ; r1 = x[3] = x[2] * (1.5 - (v/2)*x[2]*x
          andn      single,r1
*
          mpyf      r1,r1,r2    ; r2 = x[3] * x[3]
          andn      single,r2
          mpyf      r0,r2       ; r2 = (v/2) * x[3] * x[3]
          andn      single,r2
```

```
        subrf   1.5,r2          ; r2 = 1.5 - (v/2) * x[3] * x[3]
        rnd     r2
        mpyf    r2,r1           ; r1 = x[4] = x[3] * (1.5 - (v/2)*x[3]*x
        andn    single,r1
*
        mpyf    r1,r1,r2        ; r2 = x[4] * x[4]
        andn    single,r2
        mpyf    r0,r2           ; r2 = (v/2) * x[4] * x[4]
        andn    single,r2
        subrf   1.5,r2          ; r2 = 1.5 - (v/2) * x[4] * x[4]
        rnd     r2
        mpyf    r2,r1           ; r1 = x[5] = x[4] * (1.5 - (v/2)*x[4]*x
*
        andn    single,r1
        ldf     r1,r0
*
        mpyf    r3,r0           ; sqrt(v) from sqrt(v**(-1))
        andn    single,r0
*
* Save variables
*
        pushf   x               ; save c = sqrt(x)
        ldf     x,y             ; get ready for multiplication
*
* mult12(c, c, u, uu)
*
        ldf     @constant,temp
        mpyf3   temp,x,p        ; p = x * constant
        andn    single,p
*
        subf3   p,x,hx          ; hx = x - p
        rnd     hx
        addf    p,hx            ; hx = x - p + p
        rnd     hx
*
        subf3   hx,x,tx         ; tx = x - hx
        rnd     tx
*
        mpyf3   temp,y,p        ; p = y * constant
        andn    single,p
*
        subf3   p,y,hy          ; hy = y - p
        rnd     hy
        addf3   hy,p,hy         ; hy = y - p + p
        rnd     hy
*
        subf3   hy,y,ty         ; ty = y - hy
        rnd     ty
*
        mpyf3   hx,hy,p         ; p = hx * hy
        andn    single,p
*
        mpyf3   hx,ty,temp      ; temp = hx * ty
        andn    single,temp
        mpyf3   tx,hy,q         ; q = tx * hy
        andn    single,q
        addf    temp,q          ; q = hx * ty + tx * hy
        rnd     q
*
* perform tx * ty operation and store the result in temp.
* This is to optimize use of registers on the device.
*
        mpyf3   tx,ty,temp      ; temp = tx * ty
        andn    single,temp
        addf3   p,q,u           ; u = p + q
        rnd     u
*
        subf3   u,p,uu          ; uu = p - u
        rnd     uu
        addf    q,uu            ; uu = p - u + q
        rnd     uu
        addf    temp,uu         ; uu = p - u + q + tx * ty
        rnd     uu
*
* cc = ( x - u - uu + xx ) * 0.5 / c
*
        popf    c               ; restore c
        popf    temp            ; restore x
        subf3   u,temp,cc       ; cc = x - u
        rnd     cc
        subf    uu,cc           ; cc = x - u - uu
        rnd     cc
        popf    temp            ; restore xx
        addf    temp,cc         ; cc = x - u - uu + xx
        rnd     cc
*
        pushf   cc              ; save cc
        pushf   c               ; save c
*
* The floating-point number v is stored in R1. After the computation is
* completed, 1/v is also stored in R4.
*
* Register used as input: R2
* Registers modified: R0, R1, R2, R3
* Register containing result: R2
*
        ldf     r2,r3           ; v is saved for later.
        absf    r2              ; The algorithm uses v = |v|.
*
```

```
*  Extract the exponent of v.
*
           pushf    r2
           pop      r1
           ash      -24,r1          ; The 8 LSBs of R0 contain the exponent
*                                   ; of v
*
*  x[0] formation given the exponent of v.
*
           negi     r1
           subi     1,r1            ; Now we have -e-1, the exponent of x[0]
           ash      24,r1
           push     r1
           popf     r1              ; Now R0 = x[0] = 1.0 * 2**(-e-1).
*
*  Now the iterations begin.
*
           mpyf3    r1,r2,r0        ; R1 = v * x[0]
           andn     single,r0
           subrf    2.0,r0          ; R1 = 2.0 - v * x[0]
           rnd      r0
           mpyf     r0,r1           ; R0 = x[1] = x[0] * (2.0 - v * x[0])
           andn     single,r1
*
           mpyf     r1,r2,r0        ; R1 = v * x[1]
           andn     single,r0
           subrf    2.0,r0          ; R1 = 2.0 - v * x[1]
           rnd      r0
           mpyf     r0,r1           ; R0 = x[2] = x[1] * (2.0 - v * x[1])
           andn     single,r1
*
           mpyf     r1,r2,r0        ; R1 = v * x[2]
           andn     single,r0
           subrf    2.0,r0          ; R1 = 2.0 - v * x[2]
           rnd      r0
           mpyf     r0,r1           ; R0 = x[3] = x[2] * (2.0 - v * x[2])
           andn     single,r1
*
           mpyf     r1,r2,r0        ; R1 = v * x[3]
           andn     single,r0
           subrf    2.0,r0          ; R1 = 2.0 - v * x[3]
           rnd      r0
           mpyf     r0,r1           ; R0 = x[4] = x[3] * (2.0 - v * x[3])
*
           andn     single,r1
*
*  For the last iteration we use the formulation:
*  x[5] = (x[4] * (1.0 - (v * x[4]))) + x[4]
*
           mpyf     r1,r2,r0        ; R1 = v * x[4] = 1.0..01.. => 1
           andn     single,r0
           subrf    1.0,r0          ; R1 = 1.0 - v * x[4] = 0.0..01... => 0
           rnd      r0
           mpyf     r1,r0           ; R1 = x[4] * (1.0 - v * x[4])
           andn     single,r0
           addf     r0,r1           ; R0 = x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
*
           rnd      r1,r2           ; Round since this is followed by a MPYF
*
*  Now the case of v < 0 is handled.
*
           negf     r2,r0
           ldf      r3,r3           ; This sets condition flags.
           ldfn     r0,r2           ; If v < 0, then R2 = -R2
*
*  restore variables
*
           popf     temp            ; restore c
           popf     cc              ; restore cc
           mpyf     0.5,cc          ; cc = ( x - u - uu + xx ) * 0.5
           andn     single,cc
           mpyf     r2,cc           ; cc = ( x - u - uu + xx ) * 0.5 / c
           andn     single,cc
*
*  z = c + cc
*
           addf3    temp,cc,z       ; z = c + cc
           rnd      z
*
*  zz = c - z + cc
*
           subf     z,temp,zz       ; zz = c - z
           rnd      zz
           addf     cc,zz           ; zz = c - z + cc
           rnd      zz
*
           retsu
           .data
constant:
           .float   4097            ; constant = 2^(24-24/2)+1
           .end
```

Appendix B

Appendix B1. Change Two Single-Precision Numbers to One Double-Precision Result

```
3/* C30DBL  -- Program to operate on two single-precision numbers
  in C30 format and produce a double-precision result */
#include <math.h>
#include <stdio.h>

main()
{
 long double x, y, z;
 long int x1, y1;
 int  i, operation;
 long int c30toe(long int);

 i=1;
 do{
  printf("Type two C30 hex numbers:\n");
  printf("x = ");
  scanf("%X",&x1);
  printf("y = ");
  scanf("%X",&y1);
  x1 = c30toe(x1);
  x = (long double)(*(float *)(&x1));
  y1 = c30toe(y1);
  y = (long double)(*(float *)(&y1));
  do{
   printf("Add(1), Sub(2), Mpy(3), Div(4), Sqrt(5): ");
   scanf("%d", &operation);
  } while (operation<1 || operation>5);

  if (operation == 1) z = x + y;
  if (operation == 2) z = x - y;
  if (operation == 3) z = x * y;
  if (operation == 4) z = x / y;
  if (operation == 5) z = sqrt(x);
  printf("\nz = %.18Lg", z);

  printf("\n\nType in C30 hex result:\n");
  printf("z = ");
  scanf("%X",&x1);
  printf("zz = ");
  scanf("%X",&y1);
  x1 = c30toe(x1);
  x = (long double)(*(float *)(&x1));
  y1 = c30toe(y1);
  y = (long double)(*(float *)(&y1));
  z = x + y;
  printf("\nz = %.18Lg", z);
  printf("\n\nType 0 to exit, else continue : ");
  scanf ("%d", &i);
 } while (i != 0);
}
/* C30TOE -- routine to convert from a c30 floating point number to a
  number in ieee format.  Both input and output in hex. */

long int c30toe(long int x)
{
 long int mantissa, sign;
 long int exp;

 sign = x & 0x00800000;
 exp = x >> 24;

/* exp=-128 corresponds to 0.  exp=-127 is denormalized in ieee:
 represent it as 0. */

 if (exp <= -127) return(0);

/* add implied bit and sign-extend mantissa */

 mantissa = x & 0x007fffff;
 if (sign)
  mantissa |= 0xff000000;
 else
  mantissa |= 0x00800000;

/* convert mantissa to sign-magnitude */

 if (sign) mantissa = -mantissa;

/* adjust mantissa if it was -2.0 */

 if (mantissa == 0x01000000){
  exp++;
  mantissa = 0x00800000;
 }
 if (exp > 127) return(0);  /* too large number; return error */

/* make exponent 127-excess and return ieee number */

 exp += 127;
 mantissa = (mantissa & 0x007fffff) | (sign << 8) | (exp << 23);

 return(mantissa);
}
```

Appendix B2. Change Two Double-Precision Numbers to One Double-Precision Result

```
/* C30DBL2  — Program to operate on two double-precision numbers
  in C30 format and produce a double-precision result */
#include <math.h>
#include <stdio.h>

main()
{
 long double x, y, z;
 long int x1, y1, xx1, yy1;
 int  i, operation;
 long int c30toe(long int);

i=1;
do{
 printf("Type two C30 hex numbers:\n");
 printf("x = ");
 scanf("%X",&x1);
 printf("xx = ");
 scanf("%X",&xx1);
 printf("y = ");
 scanf("%X",&y1);
 printf("yy = ");
 scanf("%X",&yy1);
 x1 = c30toe(x1);
 xx1 = c30toe(xx1);
 y1 = c30toe(y1);
 yy1 = c30toe(yy1);
 x = (long double)(*(float *)(&x1)) +
   (long double)(*(float *)(&xx1));
 y = (long double)(*(float *)(&y1)) +
   (long double)(*(float *)(&yy1));
 do{
  printf("Add(1), Sub(2), Mpy(3), Div(4), Sqrt(5): ");
  scanf("%d", &operation);
 } while (operation<1 || operation>5);

 if (operation == 1) z = x + y;
 if (operation == 2) z = x - y;
 if (operation == 3) z = x * y;
 if (operation == 4) z = x / y;
 if (operation == 5) z = sqrt(x);
 printf("\nz = %.18Lg", z);

 printf("\n\nType in C30 hex result:\n");
 printf("z = ");
 scanf("%X",&x1);
 printf("zz = ");
 scanf("%X",&y1);
 x1 = c30toe(x1);
 x = (long double)(*(float *)(&x1));
 y1 = c30toe(y1);
 y = (long double)(*(float *)(&y1));
 z = x + y;
 printf("\nz = %.18Lg", z);
 printf("\n\nType 0 to exit, else continue : ");
 scanf ("%d", &i);
 } while (i != 0);
}
/* C30TOE — routine to convert from a c30 floating point number to a
  number in ieee format.  Both input and output in hex. */

long int c30toe(long int x)
{
 long int mantissa, sign;
 long int exp;

 sign = x & 0x00800000;
 exp = x >> 24;

/* exp=-128 corresponds to 0.  exp=-127 is denormalized in ieee:
 represent it as 0. */

 if (exp <= -127) return(0);

/* add implied bit and sign-extend mantissa */

 mantissa = x & 0x007fffff;
 if (sign)
  mantissa |= 0xff000000;
 else
  mantissa |= 0x00800000;

/* convert mantissa to sign-magnitude */

 if (sign) mantissa = -mantissa;

/* adjust mantissa if it was -2.0 */

 if (mantissa == 0x01000000){
  exp++;
  mantissa = 0x00800000;
 }
 if (exp > 127) return(0);  /* too large number; return error */

/* make exponent 127-excess and return ieee number */

 exp += 127;
 mantissa = (mantissa & 0x007fffff) | (sign << 8) | (exp << 23);

 return(mantissa);
}
```

8×8 Discrete Cosine Transform Implementation on the TMS320C25 or the TMS320C30

William Hohl

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

Introduction

In the general class of orthogonal transforms, there exists one in particular, the discrete cosine transform (DCT), that has recently gained wide popularity in signal processing. The DCT has found applications in such areas as data compression, pattern recognition, and Weiner filtering, primarily because of its close comparison to the Karhunen-Loeve Transform (KLT) with respect to rate distortion criteria [1]. Although the KLT is considered to be optimal, there is no fast algorithm to compute it. Since there is no fast KLT algorithm, the DCT is an attractive alternative.

For image coding, the DCT works well because of the high correlation among adjacent data samples (pixel values). Because of this correlation, the DCT provides near optimal reduction while retaining high image quality. In a comparative study [2], the DCT was shown to outperform the Fourier, Hartley, and cas-cas transforms for image compression, providing even more motivation for finding fast implementations.

A number of algorithms have been developed, most notably those of Hou [3] and Lee [4], which generate higher-order DCTs from lower-order ones. This paper presents two 8×8 DCT routines, one for the TMS320C25 and another for the TMS320C30, based upon the routine in [3].

The DCT Algorithm

For a given real data sequence $x_0, x_1, \ldots, x_{N-1}$, the discrete cosine transform is given in [1] as

$$z_k = \sqrt{\frac{2}{N}}\,\alpha(k)\sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi\,(2n+1)k}{2N}\right) \quad k = 0, 1, \ldots, N-1 \tag{1a}$$

and its inverse is

$$x_n = \sqrt{\frac{2}{N}}\sum_{k=0}^{N-1}\alpha(k)z_k \cos\left(\frac{\pi\,(2n+1)k}{2N}\right) \quad k = 0, 1, \ldots, N-1 \tag{1b}$$

where $\alpha\,(k) = \dfrac{1}{\sqrt{2}}$ for $k = 0$; otherwise, the transform is unitary. If z_0 is scaled up by 2, the DCT can also be written in matrix form as

$$\mathbf{z} = \sqrt{\frac{2}{N}}\,T(N)\,\mathbf{x}, \tag{2}$$

where **x** and **z** are column vectors denoting the input and output data sequences, and $T(N)$ is the DCT matrix of order N. Actually, expanding the matrix (neglecting the factor of $\sqrt{\frac{2}{N}}$ for the moment), a 4-point DCT appears as

$$\begin{bmatrix} z_0 \\ z_2 \\ z_1 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \alpha & -\alpha & \alpha & -\alpha \\ \beta & -\delta & -\beta & \delta \\ \delta & \beta & -\delta & -\beta \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \\ x_3 \\ x_1 \end{bmatrix}, \tag{3}$$

where $\alpha = \frac{1}{\sqrt{2}}$, $\beta = \cos\left(\frac{\pi}{8}\right)$, and $\delta = \sin\left(\frac{\pi}{8}\right)$. Similarly, the 8-pt DCT can be expressed as

$$\begin{bmatrix} z_0 \\ z_4 \\ z_2 \\ z_6 \\ z_1 \\ z_5 \\ z_3 \\ z_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \alpha & -\alpha & \alpha & -\alpha & \alpha & -\alpha & \alpha & -\alpha \\ \beta & -\delta & -\beta & \delta & \beta & -\delta & -\beta & \delta \\ \delta & \beta & -\delta & -\beta & \delta & \beta & -\delta & -\beta \\ \lambda & \mu & -\nu & -\gamma & -\lambda & -\mu & \nu & \gamma \\ \mu & \nu & -\gamma & \lambda & -\mu & -\nu & \gamma & -\lambda \\ \gamma & -\lambda & \mu & \nu & -\gamma & \lambda & -\mu & -\nu \\ \nu & \gamma & \lambda & \mu & -\nu & -\gamma & -\lambda & -\mu \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ x_7 \\ x_5 \\ x_3 \\ x_1 \end{bmatrix}, \quad (4)$$

where $\lambda = \cos\left(\frac{\pi}{16}\right)$, $\gamma = \cos\left(\frac{3\pi}{16}\right)$, $\mu = \sin\left(\frac{3\pi}{16}\right)$, and $\nu = \sin\left(\frac{\pi}{16}\right)$. Note that the input is no longer in natural order but has been rearranged according to the permutation matrix P and the relation

$$\tilde{x} = Px, \quad (5)$$

where

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Upon examination, the matrix $\hat{T}(N)$ in (4), which is the matrix $T(N)$ with the rows and columns rearranged, can be described more compactly as

$$\hat{T}(N) = \begin{bmatrix} \hat{T}\left(\frac{N}{2}\right) & \hat{T}\left(\frac{N}{2}\right) \\ \hat{D}\left(\frac{N}{2}\right) & -\hat{D}\left(\frac{N}{2}\right) \end{bmatrix}, \tag{6}$$

since the upper half of the 8-point DCT is exactly the 4-point DCT matrix previously generated. Using the results obtained in [3], the relationship between $\hat{D}\left(\frac{N}{2}\right)$ and $T\left(\frac{N}{2}\right)$ is a given as

$$\hat{D}\left(\frac{N}{2}\right) = K\hat{T}\left(\frac{N}{2}\right)Q, \tag{7}$$

where

$$K = RLR^t,$$

R being the matrix that performs a bit reversal on the input data; L is the lower triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 2 & 0 & 1 & 0 & 0 & 0 \\ -1 & 2 & -2 & 2 & 0 & 0 & 0 & 0 \\ 1 & -2 & 2 & -2 & 2 & 0 & 0 & 0 \\ -1 & 2 & -2 & 2 & -2 & 2 & 0 & 0 \\ 1 & -2 & 2 & -2 & 2 & -2 & 2 & 0 \\ -1 & 2 & -2 & 2 & -2 & 2 & -2 & 2 \end{bmatrix},$$

and $Q = \text{diag}\left[\cos\left(n + \frac{1}{4}\right)\left(\frac{2\pi}{N}\right)\right]$, for $n = 0,1, \ldots, 7$. The output vector $\mathbf{z}$ is now in bit-reversed order. Signal flow graphs for 2-point, 4-point, and 8-point DCTs are shown in Figure 1, with the multipliers defined as in (4).

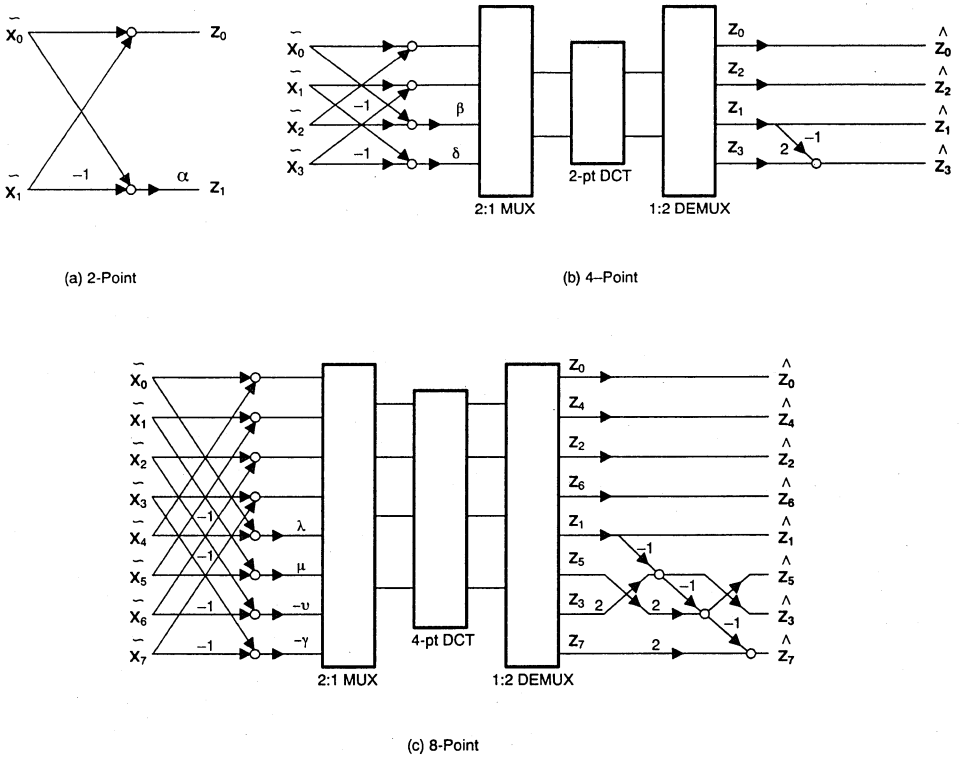


Figure 1. Signal Flow Graphs for 2-Point, 4-Point, and 8-Point DCTs

The structure of the algorithm looks very much like that of a Fast Fourier Transform (FFT), since the most fundamental computation is a 2-point butterfly. This routine is actually a generalized case of the Cooley-Tukey FFT algorithm with the addition of the recursion at the end. If the equations for the signal flow graph are written explicitly, the recursive nature of the DCT becomes clear; for a 4-point DCT, we have

$$\hat{z}_0 = z_0,$$
$$\hat{z}_2 = z_2,$$
$$\hat{z}_1 = z_1,$$
$$\hat{z}_3 = 2z_3 - \hat{z}_1,$$

and for the 8-point DCT,

$$\hat{z}_0 = z_0,$$
$$\hat{z}_4 = z_4,$$
$$\hat{z}_2 = z_2,$$
$$\hat{z}_6 = z_6,$$
$$\hat{z}_1 = z_1,$$
$$\hat{z}_3 = 2z_3 - \hat{z}_1,$$
$$\hat{z}_5 = 2z_5 - \hat{z}_3,$$
$$\hat{z}_7 = 2z_7 - \hat{z}_5.$$

To create a unitary transform, each element in the vector should be multiplied by the scaling factor $\sqrt{\frac{2}{N}}$ for both the forward and inverse transforms. The inverse transform is obtained by completely reversing the direction of the signal flow graph; i.e., performing the bit-reversal first, then the recursions and the butterflies, and finally, the data permutation.

For the two-dimensional case of interest, the DCT can be described in the form

$$z(k,l) = \frac{2}{N}\,\alpha(k)\,\alpha(l)\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)\cos\left(\frac{\pi\,(2m+1)k}{2N}\right)\cos\left(\frac{\pi\,(2n+1)l}{2N}\right) \quad (8a)$$

$$x(m,n) = \frac{2}{N}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1}\alpha(k)\,\alpha(l)z(k,l)\cos\left(\frac{\pi\,(2m+1)k}{2N}\right)\cos\left(\frac{\pi\,(2n+1)l}{2N}\right) \quad (8b)$$

where $\alpha\,(k) = \frac{1}{\sqrt{2}}$ for $k = 0$, unity otherwise. Like the FFT, the DCT kernel is separable, allowing the transform to be performed in two steps, first along the rows and then the columns.

Implementation on the TMS320C25

The DCT algorithm may be carried out in one of two ways, either using

1. A matrix formulation, where the DCT coefficients are simply multiplied by the data, or
2. The signal flow graph.

This routine uses a matrix formulation, which requires the sixty-four cosine coefficients to be stored in an array in memory. The matrix formulation is based on the following equation:

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \lambda & \gamma & \mu & \nu & -\nu & -\mu & -\gamma & -\lambda \\ \beta & \delta & -\delta & -\beta & -\beta & -\delta & \delta & \beta \\ \gamma & -\nu & -\lambda & -\mu & \mu & \lambda & \nu & -\gamma \\ \alpha & -\alpha & -\alpha & \alpha & \alpha & -\alpha & -\alpha & \alpha \\ \mu & -\lambda & \nu & \gamma & -\gamma & -\nu & \lambda & -\mu \\ \delta & -\beta & \beta & -\delta & -\delta & \beta & -\beta & \delta \\ \nu & -\mu & \gamma & -\lambda & \lambda & -\gamma & \mu & -\nu \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}, \quad (7)$$

where $\lambda = \cos\left(\frac{\pi}{16}\right)$, $\gamma = \cos\left(\frac{3\pi}{16}\right)$, $\mu = \sin\left(\frac{3\pi}{16}\right)$, and $\nu = \sin\left(\frac{\pi}{16}\right)$.

The algorithm described above has been shown to be numerically stable for fixed-point processors; however, to prevent serious data errors, truncation and roundoff must be accounted for. A roundoff technique similar to the one in [6], is used to prescale the matrix coefficients by $(2^{15} - 1)$. This product is then loaded into the accumulator with a one-bit left shift, effectively dividing it by 2^{15}. After a multiplication is performed, the 32-bit value in the accumulator must be rounded to sixteen bits, where bits 13,14, and 15 are used to determine the value of the sixteenth bit. The TMS320C25 performs this operation in a single instruction by adding 3000h to the accumulator product with a one-bit left shift, as outlined in the code shown in Figure 2.

```
*
*       INITIALIZE MATRIX COEFFICIENTS AND ROUNDOFF VALUES INTO
          INTERNAL BLOCK 0
*
DCTINI    LDPK    RNDOFF
          RSXM                    ;  SIGN-EXTENSION MODE
          SPM     1               ;  LEFT SHIFT 1 BIT
          LRLK    AR1,COEFF       ;  COEFFICIENTS
          RPTK    EDATA-IDATA
          BLKP    IDATA,*+
          LRLK    AR1,RNDOFF      ;  VARIABLES
          RPTK    10
          BLKP    EDATA,*+
          .
          .
          .
*
*       SECOND SET OF COEFFICIENTS
*
          LAR     AR1,DST         ;  AR1 IS NOW DESTINATION
                                     POINTER
          MAR     *+,AR2          ;  WORK ON SECOND COLUMN
          LAR     AR2,SRC
          LARK    AR3,7
          LT      *+,AR2
          MPY     C10
T2        ZAC
          RPTK    6
          MAC     C11,*+
*
          LTA     *+,AR1
          MPY     C10
          ADD     RNDOFF
          SACH    *0+,AR3
          BANZ    t2,*-,AR2
```

Figure 2. TMS320C25 Code for Roundoff Routine

After the multiplications are computed, the results are stored in another array area in transposed order; thus, a separate routine for transposing the matrix is not needed. Once the rows are transformed, the pointers for the input and output matrices are exchanged. When the procedure is repeated, the output is stored as rows, completing the transform. Appendix A contains a complete program listing for the forward transform on the TMS320C25. To perform an inverse DCT, the table of cosine coefficients should be replaced with those used for an inverse transform.

Implementation on the TMS320C30

The TMS320C30's increased speed and flexible addressing modes can reduce execution time substantially. In using the FFT-like structure, extraneous multiplications are removed, and because of the TMS320C30's ability to perform parallel multiplication/additions, two butterflies can be computed at once. After an initial subtraction is done, the coefficient multiplication can be executed in parallel with the addition of the data. The TMS320C30's floating-point capability eliminates not only the problems of roundoff error associated with fixed point processors but also the need for any truncation routines.

Because the DCT size is fixed to eight points, there are only four locations that need exchanging; this allows for a fast bit-reversal of the data. When using the TMS320C30's extended-precision registers for temporary storage, the transfers can be done in-place. These data transfers are also done in parallel, since two load or store operations can be performed simultaneously. The code for performing the bit reversal is shown in Figure 3 below.

```
*       CORRECT ORDER FROM BIT REVERSED TO NATURAL
*
BITREV  LDF     *AR0,R0         ;   ONLY FOUR LOCATIONS ARE
||      LDF     *-AR2,R1        ;     ACTUALLY SWITCHED
        STF     R1,*AR0
||      STF     R0,*-AR2
        LDF     *AR1,R0
||      LDF     *-AR3,R1
        STF     R1,*AR1
||      STF     R0,*-AR3
```

Figure 3. TMS320C30 Code for Bit Reversal

Because of the amount of data shuffling that occurs, an eight-word scratch-pad vector has been created with four permanent pointers set up at every other memory location. This allows access to each element in the vector (by predecrement or preincrement addressing) without requiring constant alteration of one or two pointer locations. Although there is no overhead for looping on the TMS320C30, straight-line coding is used as much as possible to increase performance.

You can transpose the DCT matrix in the same way as in the TMS320C25 implementation: namely, store the transformed row vector as a column vector in another matrix and interchange the input and output pointers.

The complete routines for the forward and inverse transforms are given in Appendix B.

Results

The execution times and memory requirements for the two routines are given in Table 1. For the TMS320C30 implementation, the forward transform contains the scale factor of $\frac{2}{N}$, so the transform is not unitary. When the signal flow is reversed, instructions accumulate and the time required to perform the inverse transform actually increases (see Table 1). This increase occurs because certain multiplications cannot be performed in parallel with another instruction. The two times are identical on a TMS320C25 because it uses a matrix routine to compute the transform.

Table 1. Execution Times and Memory Requirements

Device	Memory Required		Time Required (μs)
	Program	Data	
TMS320C25	232 words*	203 words	257.3 (forward)
	232 words	203 words	257.3 (inverse)
TMS320C30	148 words**	136 words	99.4 (forward)
	155 words	136 words	107.9 (inverse)

* TMS320C25 wordlengths are 16 bits

** TMS320C30 wordlengths are 32 bits

Summary

Two routines for a two-dimensional Discrete Cosine Transform are presented: one for the TMS320C25 and one for the TMS320C30, with a development of the algorithm given for clarification. This report also discussed the similarities of the DCT to the Cooley-Tukey FFT algorithm and arithmetic shortcuts which can reduce the DCT's execution time. Although these implementations use the most recent formulation, there is still room for investigation into more efficient methods. Another approach that might prove fruitful is to deal with the entire 8×8 array all at once, as suggested by Haque [7], rather than transforming the array by rows and columns. However, both routines given in the appendices provide fast, numerically stable solutions for applications requiring the DCT.

Acknowledgements

The author thanks Steve Ford for supplying the original code for the TMS320C25 implementation. Francois Charlot helped in modifying the code for the TMS320C25, as well as in preparing this manuscript. Daniel Chen improved the performance of the code for both the TMS320C25 and the TMS320C30.

References

[1] Ahmed, N., Natarajan, T., and Rao, K.R. "Discrete Cosine Transform," *IEEE Transactions on Computing,* vol. C-23, pp. 90-93, January 1974.

[2] Perkins, M. "A Comparison of the Hartley, Cas-Cas, Fourier, and Discrete Cosine Transforms for Image Coding," *IEEE Transactions on Computing,* vol. 36, pp. 758-760, June 1988.

[3] Hou, H.S. "A Fast Recursive Algorithm for Computing the Discrete Cosine Transform," *IEEE Transactions on ASSP,* vol. ASSP-35, No. 10, pp. 1455-1461, October 1987.

[4] Lee, B.G. "FCT - A Fast Cosine Transform," *Proceedings of 1984 Conference on ASSP,* pp. 28.A.3.1-28.A.3.4, March 1984.

[5] Jayant, N.S., and Noll, P. *Digital Coding of Waveforms,* New York, Prentice-Hall, 1984.

[6] Srinivasan, S., Jain, A.K., and Chin, T.M. "Cosine Transform Block Codec for Images Using the TMS32010," *Proceedings of IEEE ISCAS '86,* Cat. No. 86CH2255-8, vol. 1, pp. 299-302.

[7] Haque, M.A. "A Two-Dimensional Fast Cosine Tranform," *IEEE Transactions on ASSP,* vol. ASSP-33, pp. 1532-1539, December 1985.

```
*****************************************************************
*
*   8 X 8 2D-DCT ALGORITHM FOR THE TMS320C25
*
*   THIS PROGRAM WILL PERFORM A TWO-DIMENSIONAL DCT ON EIGHT-BIT IMAGE DATA
*   AND NORMALIZE THE DATA TO MIMIMIZE TRUNCATION AND ROUNDOFF.
*
*
*****************************************************************
*
        .title   '8x8 DCT'

*
*   RESET: BRANCH TO DCT, AND SET ARP TO 0
*
        .sect    "RESET"
        B        DCTINI,*,AR1
        .text
*
*   INITIALIZE MATRIX COEFFICIENTS AND ROUNDOFF VALUES INTO INTERNAL BLOCK B0
*
DCTINI  LDPK     RNDOFF
        RSXM                    ; SIGN-EXTENSION MODE
        SPM      1              ; LEFT SHIFT 1 BIT
        LRLK     AR1,COEFF      ; COEFFICIENTS
        RPTK     EDATA-IDATA
        BLKP     IDATA,*+
        LRLK     AR1,RNDOFF     ; VARIABLES
        RPTK     10
        BLKP     EDATA,*+
*
*   HERE IS THE DCT FUNCTION
*
DCT     LARK     AR7,1          ; AR7: DIMENSION-1
        LARK     AR0,8          ; POINTER INCREMENT FOR DATA TRANSPOSITION
        CNFP                    ; "MAC" NEEDS 1 OPERAND IN PROGRAM MEMORY
*
*   LOOP FOR DIMENSIONS
*
DIMS    .equ     $
*
*   FIRST SET OF COEFFICIENTS
*
        LARK     AR3,7          ; COUNT FOR 8 1-D DCTs
        LAR      AR1,SRC        ; SOURCE ADDRESS
        LAR      AR2,DST        ; DESTINATION ADDRESS (FIRST COLUMN)
        LT       *+             ; TREG = X0
        MPY      C_00           ; ACC = 0 , PREG= x0 * c00
T1      ZAC
        RPTK     6
        MAC      C01,*+         ; ACC = 0 ,PREG= X0 * C00
*
        LTA      *+,AR2         ; INCLUDE LAST PRODUCT AND LOAD PREG
        MPY      C_00
        ADD      RNDOFF
        SACH     *0+,AR3        ; STORE RESULT AND TRANSPOSE
        BANZ     T1,*-,AR1
*
*   SECOND SET OF COEFFICIENTS
*
        LAR      AR1,DST        ; AR1 IS NOW DESTINATION POINTER
        MAR      *+,AR2         ; WORK ON SECOND COLUMN
        LAR      AR2,SRC
        LARK     AR3,7
        LT       *+,AR2
        MPY      C_10
T2      ZAC
        RPTK     6
        MAC      C11,*+
*
        LTAS     *+,AR1
        MPY      C_10
        ADD      RNDOFF
        SACH     *0+,AR3
        BANZ     T2,*-,AR2
*
*   THIRD SET OF COEFFICIENTS
*
        LAR      AR1,SRC        ; AR1 NOW SOURCE POINTER
        LAR      AR2,DST
        ADRK     2              ; THIRD COLUMN
        LARP     1              ; ACTIVATE AR1
        LARK     AR3,7
        LT       *+
        MPY      C_20
T3      ZAC
        RPTK     6
        MAC      C21,*+
        LTA      *+,AR2
        MPY      C_20
        ADD      RNDOFF
        SACH     *0+,AR3
        BANZ     T3,*-,AR1
*
*   FOURTH SET OF COEFFICIENTS
*
        LAR      AR1,DST
        ADRK     3
        LARP     2
        LAR      AR2,SRC
        LARK     AR3,7
        LT       *+
        MPY      C_30
T4      ZAC
```

```
        RPTK    6
        MAC     C31,*+
        LTA     *+,AR1
        MPY     C_30
        ADD     RNDOFF
        SACH    *0+,AR3
        BANZ    T4,*-,AR2
*
*   FIFTH SET OF COEFFICIENTS
*
        LAR     AR1,SRC
        LAR     AR2,DST
        ADRK    4
        LARP    1
        LARK    AR3,7
        LT      *+
        MPY     C_40
T5      ZAC
        RPTK    6
        MAC     C41,*+
        LTA     *+,AR2
        MPY     C_40
        ADD     RNDOFF
        SACH    *0+,AR3
        BANZ    T5,*-,AR1
*
*   SIXTH SET OF COEFFICIENTS
*
        LAR     AR1,DST
        ADRK    5
        LARP    2
        LAR     AR2,SRC
        LARK    AR3,7
        LT      *+
        MPY     C_50
T6      ZAC
        RPTK    6
        MAC     C51,*+
        LTA     *+,AR1
        MPY     C_50
        ADD     RNDOFF
        SACH    *0+,AR3
        BANZ    T6,*-,AR2
*
*   SEVENTH SET OF COEFFICIENTS
*
        LAR     AR1,SRC
        LAR     AR2,DST
        ADRK    6
        LARP    1
        LARK    AR3,7
        LT      *+
        MPY     C_60
T7      ZAC
        RPTK    6
        MAC     C61,*+
        LTA     *+,AR2
        MPY     C_60
        ADD     RNDOFF
        SACH    *0+,AR3
        BANZ    T7,*-,AR1
*
*   EIGHTH SET OF COEFFICIENTS
*
        LAR     AR1,DST
        ADRK    7
        LARP    2
        LAR     AR2,SRC
        LARK    AR3,7
        LT      *+
        MPY     C_70
T8      ZAC
        RPTK    6
        MAC     C71,*+
        LTA     *+,AR1
        MPY     C_70
        ADD     RNDOFF
        SACH    *0+,AR3
        BANZ    T8,*-,AR2
*
*   LOOP FOR NEXT DIMENSION
*
        LAC     DST             ; CHANGE SOURCE AND DESTINATION POINTERS,
        DMOV    SRC             ; SO RESULT OF FIRST PASS BECOMES OPERAND
        SACL    SRC             ; OF SECOND PASS. FINAL RESULT WILL BE IN
*                               ; PICT
        LARP    AR7             ; AR7 : DIMENSION COUNTER
        BANZ    DIMS,*-,AR1     ; LOOP FOR NEXT DIMENSION
*
STOP:   CNFD
        B       $               ; STOP HERE
        .page
*
*   DATAS - TABLES AND DECLARATIONS
*
        .asect  "RCOEF",0FF00h  ; THIS IS TO SET UP THE LABELS FOR A CNFP
        .label  IDATA           ; DCT COEFFICIENTS
C00     .word   5792            ; FIRST ROW OF COEFFICIENTS
C01     .word   5792            ; 5792 = (1/4) * 2**(-1/2) IN Q15 FORMAT
C02     .word   5792
C03     .word   5792
C04     .word   5792
C05     .word   5792
C06     .word   5792
C07     .word   5792
C10     .word   8034            ; SECOND ROW OF COEFFICIENTS
C11     .word   6811
C12     .word   4551
```

```
C13     .word   1598
C14     .word   -1598       ; 1598 = (1/4) * SIN(PI/16) IN Q15 FORMAT
C15     .word   -4551       ; 4551 = (1/4) * SIN(3PI/16) IN Q15 FORMAT
C16     .word   -6811       ; 6811 = (1/4) * COS(3PI/16) IN Q15 FORMAT
C17     .word   -8034       ; 8034 = (1/4) * COS(PI/16) IN Q15 FORMAT
C20     .word   7568        ; third row of coefficients
C21     .word   3134        ; 3134 = (1/4) * SIN(PI/8) IN Q15 FORMAT
C22     .word   -3134       ; 7568 = (1/4) * COS(PI/8) IN Q15 FORMAT
C23     .word   -7568
C24     .word   -7568
C25     .word   -3134
C26     .word   3134
C27     .word   7568
C30     .word   6811        ; FOURTH ROW OF COEFFICIENTS
C31     .word   -1598
C32     .word   -8034
C33     .word   -4551
C34     .word   4551
C35     .word   8034
C36     .word   1598
C37     .word   -6811
C40     .word   5792        ; FIFTH ROW OF COEFFICIENTS
C41     .word   -5792
C42     .word   -5792
C43     .word   5792
C44     .word   5792
C45     .word   -5792
C46     .word   -5792
C47     .word   5792
C50     .word   4551        ; SIXTH ROW OF COEFFICIENTS
C51     .word   -8034
C52     .word   1598
C53     .word   6811
C54     .word   -6811
C55     .word   -1598
C56     .word   8034
C57     .word   -4551
C60     .word   3134        ; SEVENTH ROW OF COEFFICIENTS
C61     .word   -7568
C62     .word   7568
C63     .word   -3134
C64     .word   -3134
C65     .word   7568
C66     .word   -7568
C67     .word   3134
C70     .word   1598        ; EIGHTH ROW OF COEFFICIENTS
C71     .word   -4551
C72     .word   6811
C73     .word   -8034
C74     .word   8034
C75     .word   -6811
C76     .word   4551
C77     .word   -1598
        .label  EDATA       ; END OF COEFFICIENTS TABLE

        .word   12288       ; ROUNDOFF FACTOR
        .word   PICT        ; ADDRESS OF PICTURE
        .word   RESULT      ; ADDRESS OF RESULT
        .word   5792        ; C00 COEFFICIENT
        .word   8034        ; C10 COEFFICIENT
        .word   7568        ; C20 COEFFICIENT
        .word   6811        ; C30 COEFFICIENT
        .word   5792        ; C40 COEFFICIENT
        .word   4551        ; C50 COEFFICIENT
        .word   3134        ; C60 COEFFICIENT
        .word   1598        ; C70 COEFFICIENT
*
*   DATA DEFINITIONS
*
COEFF   .usect  "COEFFS",64 ; DCT COEFFICIENTS (GOES INTO B0)
        .BSS    PICT,64     ; PICTURE
        .BSS    RESULT,64   ; RESULT, AFTER DCT
        .BSS    RNDOFF,1    ; ROUNDOFF FACTOR
        .BSS    SRC,1       ; SOURCE ADDRESS FOR CURRENT DCT LOOP
        .BSS    DST,1       ; DESTINATION ADDRESS
        .BSS    C_00,1      ; C00 COEFFICIENT
        .BSS    C_10,1      ; C10 COEFFICIENT
        .BSS    C_20,1      ; C20 COEFFICIENT
        .BSS    C_30,1      ; C30 COEFFICIENT
        .BSS    C_40,1      ; C40 COEFFICIENT
        .BSS    C_50,1      ; C50 COEFFICIENT
        .BSS    C_60,1      ; C60 COEFFICIENT
        .BSS    C_70,1      ; C70 COEFFICIENT
*
        .end
```

```
*****************************************************************
*
*   TITLE: 2-D DISCRETE COSINE TRANSFORM, (8x8) VERSION 1.0
*
*   AUTHOR: WILLIAM HOHL
*
*
*   THIS PROGRAM IS BASED ON A RECENT ALGORITHM PROPOSED BY H.S. HOU
*   (TRANSACTIONS ON ASSP, VOL. ASSP-35, NO. 10, OCTOBER 1987, PP. 1455-
*   1461).
*
*   INPUT MATRIX IS STORED IN RAM, AND THE RESULTS ARE STORED IN THE SAME
*   LOCATION.
*
*****************************************************************
*
        .BSS    OUT,64
        .BSS    INP,64
        .BSS    SCR,8           ; SCRATCHPAD MEMORY
        .global COSTAB
        .global START
        .data
*
_COS    .word   COSTAB
INPUT   .word   INP
OUTPUT  .word   OUT
SCRATCH .word   SCR
SCRLAST .word   SCR+7
RTN1    .word   TRANS1
RTN2    .word   TRANS2
        .text
*
START   LDI     7,RC
        LDI     2,IR0
        LDI     8,IR1
        LDI     8,BK            ; SET BUFFER LENGTH=8
        LDP     @SCRATCH
        LDI     @SCRATCH,AR4
        LDI     @OUTPUT,AR6     ; VARIABLE LOCATIONS
        LDI     @INPUT,AR5      ; HOLDS INPUT MATRIX
        LDF     0.25,R6         ; CONSTANT 0.25
        LDF     2.0,R7          ; CONSTANT 2.0
*
        LDI     @RTN1,R4        ; RETURN ADDRESS OF SUBROUTINE
        RPTB    BLK1
        BRD     DCT
        LDI     AR5,AR0         ; POINTS TO INPUT
        LDI     AR5,AR1
        ADDI    1,AR1
*
TRANS1: LDF     *AR4++(1)%,R1   ; TRANSPOSE THE ROWS
        STF     R1,*AR6++(IR1)  ; INTO COLUMNS
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR5++(IR1),R5
*
BLK1    SUBI    63,AR6
*
        LDI     @SCRATCH,AR4
        LDI     @OUTPUT,AR5     ; DO DCT ON COLUMN
        LDI     @INPUT,AR6      ; VECTORS
        LDI     7,RC
*
        LDI     @RTN2,R4        ; RETURN ADDRESS OF SUBROUTINE
        RPTB    BLK3
        BRD     DCT
        LDI     AR5,AR0         ; POINTS TO INPUT
        LDI     AR5,AR1
        ADDI    1,AR1
*
TRANS2: LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR4++(1)%,R1
        STF     R1,*AR6++(IR1)
||      LDF     *AR5++(IR1),R5
*
BLK3    SUBI    63,AR6          ; INCREMENT POINTERS
*
END     BR      END             ; END
*
```

```
*   SHUFFLE THE DATA ACCORDING TO PERMUTATION MATRIX P
*
DCT     LDI     AR4,AR2         ; POINTS TO OUTPUT
        LDI     @SCRLAST,AR3
        LDI     @_COS,AR7       ; TABLE POINTER
*
        LDF     *AR0++(IR0),R0
||      LDF     *AR1++(IR0),R1
        STF     R0,*AR2++(1)    ; GOING DOWN
||      STF     R1,*AR3--(1)    ; GOING UP
        LDF     *AR0++(IR0),R0
||      LDF     *AR1++(IR0),R1
        STF     R0,*AR2++(1)
||      STF     R1,*AR3--(1)
        LDF     *AR0++(IR0),R0
||      LDF     *AR1++(IR0),R1
        STF     R0,*AR2++(1)
||      STF     R1,*AR3--(1)
        LDF     *AR0++(IR0),R0
||      LDF     *AR1++(IR0),R1
        STF     R0,*AR2++(1)
||      STF     R1,*AR3--(1)
*
*   MODIFIED FFT ALGORITHM
*
        LDI     AR4,AR0         ; POINT TO OUTPUT
        ADDI    1,AR0
        LDI     AR0,AR1
        ADDI    2,AR1           ; SET UP POINTERS
        LDI     AR1,AR2
        ADDI    2,AR2
        LDI     AR2,AR3
        ADDI    2,AR3
*
        LDF     *-AR2,R2        ; THESE SECTIONS PERFORM
||      LDF     *AR2,R3         ; TWO BUTTERFLIES AT ONCE
        SUBF3   *-AR2,*-AR0,R1
        SUBF3   *AR2,*AR0,R0    ; POINTERS ARE SET AS FOLLOWS:
        MPYF3   R1,*AR7++(1),R1 ;
||      ADDF3   R3,*AR0,R3      ; X(0)
        MPYF3   R0,*AR7++(1),R0 ; X(1)  AR0
||      ADDF3   R2,*-AR0,R2     ; X(2)
        STF     R1,*-AR2        ; X(3)  AR1
||      STF     R2,*-AR0        ; X(4)
        STF     R0,*AR2         ; X(5)  AR2
||      STF     R3,*AR0         ; X(6)
        LDF     *-AR3,R2        ; X(7)  AR3
||      LDF     *AR3,R3
        SUBF3   *-AR3,*-AR1,R1
        SUBF3   *AR3,*AR1,R0
        MPYF3   R1,*AR7++(1),R1
||      ADDF3   R3,*AR1,R3
        MPYF3   R0,*AR7++(1),R0
||      ADDF3   R2,*-AR1,R2
        STF     R1,*-AR3
||      STF     R2,*-AR1
        STF     R0,*AR3
||      STF     R3,*AR1
*
*   SECOND GROUP OF BUTTERFLIES
*
        LDF     *-AR1,R2        ; THIS IS THE SAME AS ABOVE EXCEPT THE
||      LDF     *AR1,R3         ;   POINTERS CHANGE
        SUBF3   *-AR1,*-AR0,R1
        SUBF3   *AR1,*AR0,R0
        MPYF3   R1,*AR7++(1),R1
||      ADDF3   R3,*AR0,R3
        MPYF3   R0,*AR7--(1),R0
||      ADDF3   R2,*-AR0,R2
        STF     R1,*-AR1
||      STF     R2,*-AR0
        STF     R0,*AR1
||      STF     R3,*AR0
        LDF     *-AR3,R2
||      LDF     *AR3,R3
        SUBF3   *-AR3,*-AR2,R1
        SUBF3   *AR3,*AR2,R0
        MPYF3   R1,*AR7++(1),R1
||      ADDF3   R3,*AR2,R3
        MPYF3   R0,*AR7++(1),R0
||      ADDF3   R2,*-AR2,R2
        STF     R1,*-AR3
||      STF     R2,*-AR2
        STF     R0,*AR3
||      STF     R3,*AR2
*
*   LAST SET OF BUTTERFLIES
*
        LDF     *AR0,R2
||      LDF     *AR1,R3
        SUBF3   *AR0,*-AR0,R1
        SUBF3   *AR1,*-AR1,R0
        MPYF3   R1,*AR7,R1
||      ADDF3   R3,*-AR1,R3
        MPYF3   R0,*AR7,R0
||      ADDF3   R2,*-AR0,R2
        STF     R1,*AR0
||      STF     R2,*-AR0
        STF     R3,*-AR1
||      STF     R0,*AR1
        LDF     *AR2,R2
||      LDF     *AR3,R3
        SUBF3   *AR2,*-AR2,R1
        SUBF3   *AR3,*-AR3,R0
        MPYF3   R1,*AR7,R1
||      ADDF3   R3,*-AR3,R3
        MPYF3   R0,*AR7,R0
||      ADDF3   R2,*-AR2,R2
```

```
        STF     R1,*AR2
||      STF     R2,*-AR2
        STF     R3,*-AR3
||      STF     R0,*AR3
*
*   CORRECT ORDER FROM BIT-REVERSED TO NATURAL
*
BITREV  LDF     *AR0,R0         ; ONLY TWO LOCATIONS ARE ACTUALLY SWITCHED
||      LDF     *-AR2,R1
        STF     R1,*AR0
||      STF     R0,*-AR2
        LDF     *AR1,R0
||      LDF     *-AR3,R1
        STF     R1,*AR1
||      STF     R0,*-AR3
*
*   CONTINUE WITH RECURSIVE ALGORITHM
*
RECURSE MPYF3   R7,*-AR3,R2
        MPYF3   R7,*AR3,R1
||      SUBF3   *-AR1,R2,R2     ; 2X(7)-X(3)
        SUBF3   *AR1,R1,R1      ; 2X(8)-X(4)
        STF     R1,*AR3
||      STF     R2,*-AR3
*
LASTLOOP MPYF3  R7,*AR1,R0      ; X(4)=2*X(4)
        MPYF3   R7,*AR2,R1      ; X(6)=2*X(6)
||      SUBF3   *AR0,R0,R2      ; R2=2X(4)-X(2)
        MPYF3   R7,*AR3,R3      ; R3=2*X(8)
||      STF     R2,*AR1
        SUBF3   *AR1,R1,R1      ; R1=2X(6)-X(4)
        SUBF3   R1,R3,R3        ; R3=2X(8)-X(6)
        STF     R1,*AR2
||      STF     R3,*AR3
*
*   SCALE FACTOR OF (2/N)=0.25
*
        MPYF3   R6,*AR3,R0
        STF     R0,*AR3--(1)
||      MPYF3   R6,*-AR3,R1
        STF     R1,*AR3--(1)
||      MPYF3   R6,*-AR3,R0
        STF     R0,*AR3--(1)
||      MPYF3   R6,*-AR3,R1
        STF     R1,*AR3--(1)
||      MPYF3   R6,*-AR3,R0
        STF     R0,*AR3--(1)    ; OK TO MOVE AR3
||      MPYF3   R6,*-AR3,R1
        STF     R1,*AR3--(1)
||      MPYF3   R6,*-AR3,R0
        STF     R0,*AR3--(1)
||      MPYF3   R6,*-AR3,R1
        STF     R1,*AR3
*
*   CORRECT X(0) IF NONZERO
*
EXIT    BUD     R4              ; RETURN
        LDF     *-AR0,R0
        MPYF3   *AR7,R0,R0      ; MULT BY 1/SQRT(2)
        STF     R0,*-AR0        ; STORE THE RESULT
        .end
*
COSTAB  .float  0.980785280403 ; LAMBDA
        .float  0.555570233019 ; MU
        .float  -0.195090322016 ; -NU
        .float  -0.831469612303 ; -GAMMA
        .float  0.923879532511 ; BETA
        .float  -0.382683432365 ; -DELTA
        .float  0.707106781188 ; ALPHA
        .end
```

```
*************************************************************************
*
*    TITLE: 2-D INVERSE DISCRETE COSINE TRANSFORM, (8x8) VERSION 1.0
*
*    AUTHOR: WILLIAM HOHL
*
*
*    THIS PROGRAM IS BASED ON A RECENT ALGORITHM PROPOSED BY H.S. HOU
*    (TRANSACTIONS ON ASSP, VOL. ASSP-35, NO. 10, OCTOBER 1987, PP. 1455-
*    1461).
*
*    INPUT MATRIX IS STORED IN RAM, AND THE RESULTS ARE STORED IN THE SAME
*    LOCATION.
*
*************************************************************************
*
          .BSS      OUT,64
          .BSS      INP,64
          .BSS      SCR,8
          .global   COS_TAB
          .global   START
          .data
*
_COS      .word     COS_TAB
INPUT     .word     INP
OUTPUT    .word     OUT
SCRATCH   .word     SCR
RTN1      .word     TRANS1
RTN2      .word     TRANS2
          .text
*
START     LDI       7,RC
          LDI       2,IR0
          LDI       8,IR1
          LDF       2.0,R7          ; MULTIPLIER
          LDI       8,BK            ; SET BUFFER LENGTH=64
          LDP       @OUTPUT
          LDI       @OUTPUT,AR6     ; VARIABLE LOCATIONS
          LDI       @SCRATCH,AR4
          LDI       @INPUT,AR5      ; HOLDS INPUT MATRIX
*
          LDI       @RTN1,R4        ; RETURN ADDRESS OF SUBROUTINE
          RPTB      BLK1
          BRD       IDCT
          LDI       AR5,AR0         ; POINT TO INPUT
          LDI       @_COS,AR7       ; TABLE POINTER
          ADDI      1,AR0
*
TRANS1:   LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR5++(IR1),R5
*
BLK1      SUBI      63,AR6
*
          LDI       @INPUT,AR6      ; REALIGN POINTERS
          LDI       @OUTPUT,AR5
          LDI       @SCRATCH,AR4
          LDI       7,RC
*
          LDI       @RTN2,R4        ; RETURN ADDRESS OF SUBROUTINE
          RPTB      BLK6
          BRD       IDCT
          LDI       AR5,AR0         ; POINT TO INPUT
          LDI       @_COS,AR7       ; TABLE POINTER
          ADDI      1,AR0
*
TRANS2:   LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR4++(1)%,R1
          STF       R1,*AR6++(IR1)
||        LDF       *AR5++(IR1)R5
*
BLK6      SUBI      63,AR6
*
END       BR        END             ; END
*
*    CORRECT X(0) IF NONZERO
*
```

```
IDCT    LDI     AR0,AR1
        ADDI    2,AR1
        LDI     AR1,AR2
        ADDI    2,AR2
        LDI     AR2,AR3
        ADDI    2,AR3
*
        LDF     *-AR0,R0
        MPYF3   *AR7,R0,R0      ; MULT BY 1/SQRT(2)
        STF     R0,*-AR0        ; STORE THE RESULT
*
*   BEGIN WITH RECURSION
*
        SUBF3   *AR3,*AR2,R2    ; X(6)-X(8)
        SUBF3   R2,*AR1,R3      ; X(4)-X(6)
        MPYF3   *AR3,R7,R0      ; 2X(8)->R0
||      STF     R2,*AR2
        SUBF3   R3,*AR0,R2      ; X(2)-X(4)
        MPYF3   *AR2,R7,R1      ; 2*X(6)->R1
||      STF     R3,*AR1
        STF     R0,*AR3
||      STF     R1,*AR2
        MPYF3   *AR1,R7,R0
        STF     R0,*AR1
||      STF     R2,*AR0
SECLOOP SUBF3   *-AR3,*-AR1,R2 ; X(3)-X(7)
        SUBF3   *AR3,*AR1,R3    ; X(4)-X(8)
        MPYF3   R7,*-AR3,R0     ; 2*X(7)
||      STF     R2,*-AR1
        MPYF3   R7,*AR3,R1      ; 2*X(8)
||      STF     R3,*AR1
        STF     R0,*-AR3
||      STF     R1,*AR3
*
*   CORRECT ORDER FROM NATURAL TO BIT-REVERSED
*
BITREV  LDF     *AR0,R0         ; ONLY TWO LOCATIONS ARE ACTUALLY SWITCHED
||      LDF     *-AR2,R1
        STF     R1,*AR0
||      STF     R0,*-AR2
        LDF     *AR1,R0
||      LDF     *-AR3,R1
        STF     R1,*AR1
||      STF     R0,*-AR3
*
*   FIRST SET OF BUTTERFLIES
*
        LDF     *AR0,R0
||      LDF     *AR1,R1
        LDF     *AR2,R2
||      LDF     *AR3,R3
        MPYF3   *AR7,R1,R1      ; PERFORM THE ALPHA MULT'S
        MPYF3   *AR7,R0,R0
        MPYF3   *AR7,R2,R2
        MPYF3   *AR7++(1),R3,R3 ; SKIP TO NEXT COEFF
        STF     R1,*AR1
||      STF     R0,*AR0
        STF     R2,*AR2
||      STF     R3,*AR3
        LDF     *AR0,R2         ; THESE SECTIONS PERFORM
||      LDF     *AR1,R3         ; TWO BUTTERFLIES AT ONCE
        SUBF3   *AR0,*-AR0,R0
        SUBF3   *AR1,*-AR1,R1
        STF     R0,*AR0
        MPYF3   R1,*++AR7,R1    ; -DELTA
        ADDF3   R3,*-AR1,R0
        MPYF3   R0,*AR7,R0      ; BETA
||      ADDF3   R2,*-AR0,R2
        STF     R2,*-AR0
||      STF     R0,*-AR1
        STF     R1,*AR1
*
        LDF     *AR2,R2
||      LDF     *AR3,R3
        SUBF3   *AR2,*-AR2,R0
        SUBF3   *AR3,*-AR3,R1
        STF     R0,*AR2
        MPYF3   R1,*++AR7,R1    ; -DELTA ON NEXT GROUP
        ADDF3   R3,*-AR3,R0
        MPYF3   R0,*AR7++(IR0),R0  ; BETA ON NEXT GROUP
||      ADDF3   R2,*-AR2,R2
        STF     R2,*-AR2
||      STF     R0,*-AR3
        STF     R1,*AR3
*
*   SECOND GROUP OF BUTTERFLIES
*
        LDF     *-AR1,R2        ; THIS IS THE SAME AS ABOVE, EXCEPT THE
||      LDF     *AR1,R3         ; POINTERS CHANGE
        SUBF3   *-AR1,*-AR0,R1
        SUBF3   *AR1,*AR0,R0
        ADDF3   R3,*AR0,R3
        ADDF3   R2,*-AR0,R2
        STF     R1,*-AR1
||      STF     R2,*-AR0
        STF     R0,*AR1
||      STF     R3,*AR0
*
        LDF     *-AR3,R2
||      LDF     *AR3,R3
        SUBF3   *-AR3,*-AR2,R1
        SUBF3   *AR3,*AR2,R0
        MPYF3   *AR7++(1),R1,R1 ; -NU
||      ADDF3   R3,*AR2,R3
        MPYF3   *AR7++(1),R0,R0 ; -GAMMA
||      ADDF3   R2,*-AR2,R2
        MPYF3   R2,*AR7++(1),R2 ; LAMBDA
        MPYF3   R3,*AR7,R3      ; MU
```

```
        STF     R1,*-AR3
||      STF     R2,*-AR2
        STF     R0,*AR3
||      STF     R3,*AR2
*
*   LAST SET OF BUTTERFLIES
*
        LDF     *-AR2,R2
||      LDF     *AR2,R3
        SUBF3   *-AR2,*-AR0,R1
        SUBF3   *AR2,*AR0,R0    ; POINTERS ARE SET AS FOLLOWS:
        ADDF3   R3,*AR0,R3      ; X(0)
        ADDF3   R2,*-AR0,R2     ; X(1)  AR0
        STF     R1,*-AR2        ; X(2)
||      STF     R2,*-AR0        ; X(3)  AR1
        STF     R0,*AR2         ; X(4)
||      STF     R3,*AR0         ; X(5)  AR2
        LDF     *-AR3,R2        ; X(6)
||      LDF     *AR3,R3         ; X(7)  AR3
        SUBF3   *-AR3,*-AR1,R1
        SUBF3   *AR3,*AR1,R0
        ADDF3   R3,*AR1,R3
        ADDF3   R2,*-AR1,R2
        STF     R1,*-AR3
||      STF     R2,*-AR1
        STF     R0,*AR3
||      STF     R3,*AR1
*
*   SHUFFLE THE DATA ACCORDING TO PERMUTATION MATRIX P
*
        LDI     AR4,AR0         ; POINTS TO SCRATCH
        LDI     AR4,AR1
        ADDI    1,AR1
        LDI     AR5,AR2         ; POINTS TO INPUT
        LDI     7,AR3           ; VECTOR
        ADDI    AR2,AR3,AR3
        LDF     *AR2++(1),R0    ; GOING UP
||      LDF     *AR3--(1),R1    ; GOING DOWN
        STF     R0,*AR0++(IR0)
||      STF     R1,*AR1++(IR0)
        LDF     *AR2++(1),R0
||      LDF     *AR3--(1),R1
        STF     R0,*AR0++(IR0)
||      STF     R1,*AR1++(IR0)
        LDF     *AR2++(1),R0
||      LDF     *AR3--(1),R1
        BUD     R4              ; RETURN HOME
        STF     R0,*AR0++(IR0)
||      STF     R1,*AR1++(IR0)
        LDF     *AR2++(1),R0
||      LDF     *AR3--(1),R1
        STF     R0,*AR0++(IR0)
||      STF     R1,*AR1++(IR0)
        .end
```

```
        .global  COS_TAB
        .data
COS_TAB .float   0.707106781188 ; ALPHA
        .float   0.923879532511 ; BETA
        .float   -0.382683432365 ; -DELTA
        .float   -0.195090322016 ; -NU
        .float   -0.831469612303 ; -GAMMA
        .float   0.980785280403 ; LAMBDA
        .float   0.555570233019 ; MU
        .end
```

An Implementation of Adaptive Filters with the TMS320C25 or the TMS320C30

Sen Kuo
Northern Illinois University

Chein Chen
Digital Signal Processor Products—Semiconductor Group
Texas Instruments

Introduction

A filter selects or controls the characteristics of the signal it produces by conditioning the incoming signal. The coefficients of the filter determine its characteristics and output *a priori* in many cases. Often, a specific output is desired, but the coefficients of the filter cannot be determined at the outset. An example is an echo canceller; the desired output cancels the echo signal (an output result of zero when there is no other input signal). In this case, the coefficients cannot be determined initially since they depend on changing line or transmission conditions. For applications such as this, it is necessary to rely on adaptive filtering techniques.

An adaptive filter is a filter containing coefficients that are updated by an adaptive algorithm to optimize the filter's response to a desired performance criterion. In general, adaptive filters consist of two distinct parts: a filter, whose structure is designed to perform a desired processing function; and an adaptive algorithm, for adjusting the coefficients of that filter to improve its performance, as illustrated in Figure 1. The incoming signal, x(n), is weighted in a digital filter to produce an output, y(n). The adaptive algorithm adjusts the weights in the filter to minimize the error, e(n), between the filter output, y(n), and the desired response of the filter, d(n). Because of their robust performance in the unknown and time-variant environment, adaptive filters have been widely used from telecommunications to control.

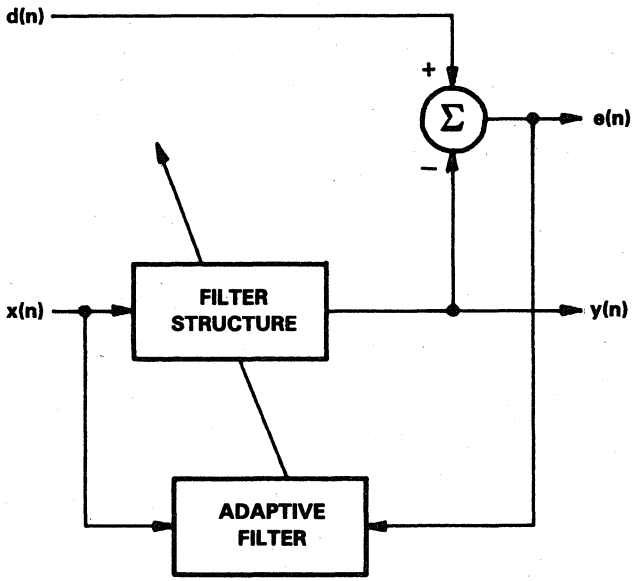


Figure 1. General Form of an Adaptive Filter

Adaptive filters can be used in various applications with different input and output configurations. In many applications requiring real-time operation, such as adaptive prediction, channel equalization, echo cancellation, and noise cancellation, an adaptive filter implementation based on a programmable digital signal processor (DSP) has many advantages over other approaches such as a hard-wired adaptive filter. Not only are power, space, and manufacturing requirements greatly reduced, but also programmability provides flexibility for system upgrade and software improvement.

The early research on adaptive filters was concerned with adaptive antennas [1] and adaptive equalization of digital transmission systems [2]. Much of the reported research on the adaptive filter has been based on Widrow's well-known Least Mean Square (LMS) algorithm, because the LMS algorithm is relatively simple to design and implement, and it is well-understood and well-suited for many applications. All the filter structures and update algorithms discussed in this application report are Finite Impulse Response (FIR) filter structures and LMS-type algorithms. However, for a particular application, adaptive filters can be implemented in a variety of structures and adaptation algorithms [1, 3 through 9]. These structures and algorithms generally trade increased complexity for improved performance. An interactive software package to evaluate the performance of adaptive filters has also been developed [10].

The complexity of an adaptive filter implementation is usually measured in terms of its multiplication rate and storage requirement. However, the data flow and data manipulation capabilities of a DSP are also major factors in implementing adaptive filter systems. Parallel hardware multiplier, pipeline architecture, and fast on-chip memory size are major features of most DSPs [11, 12] and can make filter implementation more efficient.

Two such devices, the TMS320C25 and TMS320C30 from Texas Instruments [13, 14], have been chosen as the processors for fixed-point and floating-point arithmetic. They combine the power, high speed, flexibility, and an architecture optimized for adaptive signal processing. The instruction execution time is 80 ns for the TMS320C25 and only 60 ns for the TMS320C30. Most instructions execute in a single cycle, and the architectures of both processors make it possible to execute more than one operation per instruction. For example, in one instruction, the TMS320C25 processor can generate an instruction address and fetch that instruction, decode the instruction, perform one or two data moves (if the second data is from program memory), update one address pointer, and perform one or two computations (multiplication and accumulation). These processors are designed for real-time tasks in telecommunications, speech processing, image processing, and high-speed control, etc.

To direct the present research toward realistic real-time applications, three adaptive structures were implemented:

1. Transversal
2. Symmetric transversal
3. Lattice

Each structure utilizes five different update algorithms:

1. LMS
2. Normalized LMS
3. Leaky LMS
4. Sign-error LMS
5. Sign-sign LMS

Each structure with its adaptation algorithms is implemented using the TMS320C25 with fixed-point arithmetic and the TMS320C30 with floating-point arithmetic. The processor assembly code is included in the Appendix for each implementation. The assembly code for each structure and adaptation strategy can be readily modified by the reader to fit his/her applications and could be incorporated into a C function library as callable routines.

In this application report, the applications of adaptive filters, such as adaptive prediction, adaptive equalization, adaptive echo cancellation, and adaptive noise cancellation are presented first. Next, the implementation of the three filter structures and five adaptive algorithms with the TMS320C25 and TMS320C30 is described. This is followed by the practical considerations on the implementation of these adaptive filters. The remainder of the application report covers coding options, such as the routine libraries that support both assembly and C languages.

Applications of Adaptive Filters

The most important feature of an adaptive filter is the ability to operate effectively in an unknown environment and track time-varying characteristics of the input signal. The adaptive filter has been successfully applied to communications, radar, sonar, control, and image processing. Figure 1 illustrates a general form of an adaptive filter with input signals, x(n) and d(n), output signal, y(n), and error signal, e(n), which is the difference between the desired signal, d(n), and output signal, y(n). The adaptive filter can be used in different applications with different input/output configurations. In this section we briefly discuss several potential applications for the adaptive filters [15].

Adaptive Prediction

Adaptive prediction [16 through 18] is illustrated in Figure 2. In the general application of adaptive prediction, the signals are x(n) – delayed version of original signal, d(n) – original input signal, y(n) – predicted signal, and e(n) – prediction error or residual.

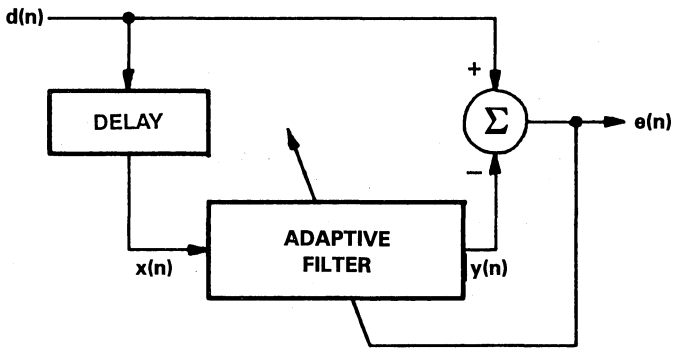


Figure 2. Block Diagram of an Adaptive Predictor

A major application of the adaptive prediction is the waveform coding of a speech signal. The adaptive filter is designed to exploit the correlation between adjacent samples of the speech signal so that the prediction error is much smaller than the input signal on the average. This prediction error signal is quantized and sent to the receiver in order to reduce the number of bits required for the transmission. This type of waveform coding is called Adaptive Differential Pulse-Code Modulation (ADPCM) [17] and provides data rate compression of the speech at 32 kb/s with toll quality. More recently, in certain on-line applications, time recursive modeling algorithms have been proposed to facilitate speech modeling and analysis.

The coefficients of the adaptive predictor can be used as the autoregressive (AR) parameters of the nonstationary model. The equation of the AR process is

$$u(n) = a_1 * u(n-1) + a_2 * u(n-2) + \ldots\ldots + a_m * u(n-m) + v(n)$$

where $a_1, a_2, \ldots, a_m$ are the AR parameters. Thus, the present value of the process u(n) equals a finite linear combination of past values of the process plus an error term v(n). This adaptive AR model provides a practical means to measure the instantaneous frequency of input signal. The adaptive predictor can also be used to detect and enhance a narrow band signal embedded in broad band noise. This Adaptive Line Enhancer (ALE) provides at its output y(n) a sinusoid with an enhanced signal-to-noise ratio, while the sinusoidal components are reduced at the error output e(n).

Adaptive Equalization

Figure 3 shows another model known as adaptive equalization [2, 9, 15]. The signals in the adaptive equalization model are defined as x(n) − received signal (filtered version of transmitted signal) plus channel noise, d(n) − detected data signal (data mode) or pseudo random number (training mode), y(n) − equalized signal used to detect received data, and e(n) − residual intersymbol interference plus noise.

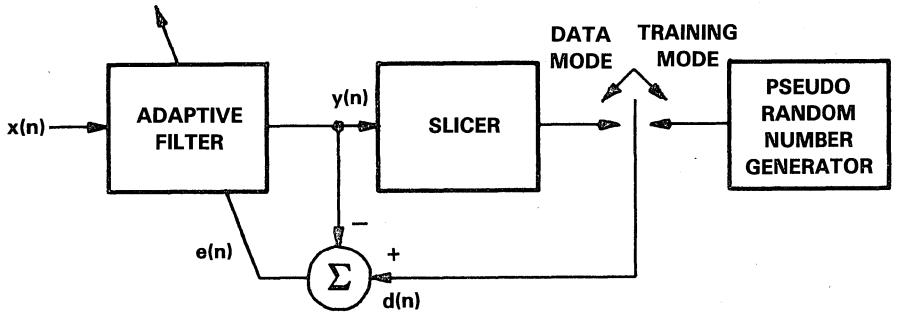


Figure 3. Block Diagram of an Adaptive Equalizer

The use of adaptive equalization to eliminate the amplitude and phase distortion introduced by the communication channel was one of the first applications of adaptive filtering in telecommunications [19]. The effect of each symbol transmitted over a time-dispersive channel extends beyond the time interval used to represent that symbol, resulting in an overlay of received symbols. Since most channels are time-varying and unknown in advance, the adaptive channel equalizer is designed to deal with this intersymbol interference and is widely used for bandwidth-efficient transmission over telephone and radio channels.

Adaptive Echo Cancellation

Another application, known as adaptive echo cancellation [20, 21] is shown in Figure 4. In this application, the signals are identified as x(n) – far-end signal, d(n) – echo of far-end signal plus near-end signal, y(n) – estimated echo of far-end signal, and e(n) – near-end signal plus residual echo.

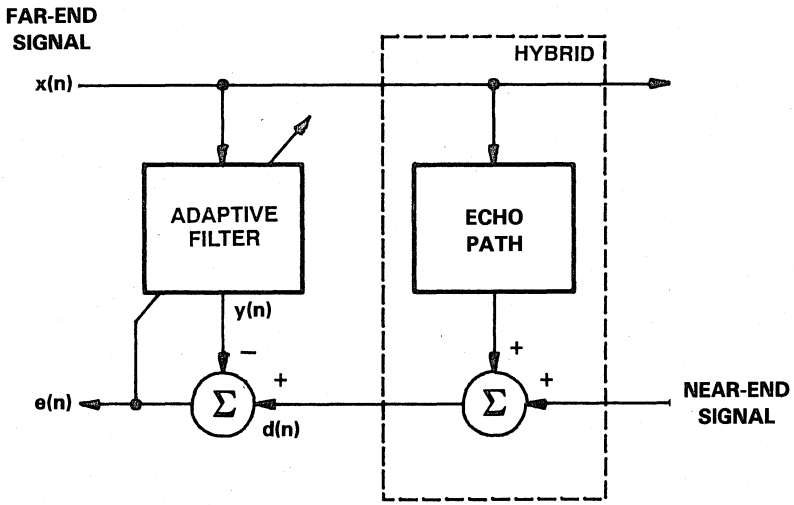


Figure 4. Block Diagram of an Echo Canceller

The adaptive echo cancellers are used in practical applications of cancelling echoes for long-distance telephone voice communication, full-duplex voiceband data modems, and high-performance audio-conferencing systems. To overcome the echo problem, echo cancellers are installed at both ends of the network. The cancellation is achieved by estimating the echo and subtracting it from the return signal.

Adaptive Noise Cancellation

One of the simplest and most effective adaptive signal processing techniques is adaptive noise cancelling [1, 22]. As shown in Figure 5, the primary input d(n) contains both signal and noise, where x(n) is the noise reference input. An adaptive filter is used to estimate the noise in d(n) and the noise estimate y(n) is then subtracted from the primary channel. The noise cancellation output is then the error signal e(n).

The applications of noise cancellation include the cancellation of various forms of interference in electrocardiography, noise in speech signals, noise in fighter cockpit environments, antennas sidelobe interference, and the elimination of 60-Hz hum. In the majority of these noise cancellation applications, the LMS algorithm has been utilized.

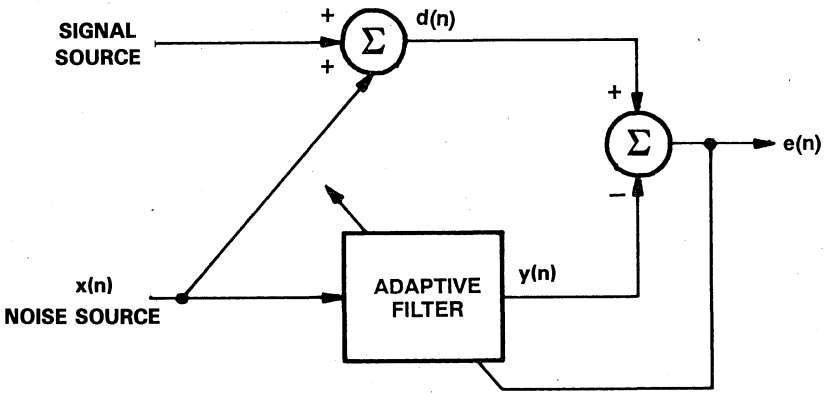


Figure 5. General Form of a Noise Canceller

Application Summary

The above list of applications is not exhaustive and is limited primarily to applications within the field of telecommunications. Adaptive filtering has been used extensively in the context of many other fields including, but not limited to, instantaneous frequency tracking, intrusion detection, acoustic Doppler extraction, on-line system identification, geophysical signal processing, biomedical signal processing, the elimination of radar clutter, beamforming, sonar processing, active sound cancellation, and adaptive control.

Implementation of Adaptive Structures and Algorithms

Several types of filter structures can be implemented in the design of the adaptive filters such as Infinite Impulse Response (IIR) or Finite Impulse Response (FIR). An adaptive IIR filter [1, 5], with poles as well as zeros, makes it possible to offer the same filter characteristics as the FIR filter with lower filter complexity. However, the major problem with adaptive IIR filter is the possible instability of the filter if the poles move outside the unit circle during the adaptive process. In this application report, only FIR structure is implemented to guarantee filter stability.

An adaptive FIR filter can be realized using transversal, symmetric transversal, and lattice structures. In this section, the adaptive transversal filter with the LMS algorithm is introduced and implemented first to provide a working knowledge of adaptive filters.

Transversal Structure with LMS Algorithm

Transversal Structure Filter

The most common implementation of the adaptive filter is the transversal structure (tapped delay line) illustrated in Figure 6. The filter output signal y(n) is

$$y(n) = \underline{w}^T(n)\underline{x}(n) = \sum_{i=0}^{N-1} w_i(n)\, x(n-i) \qquad (1)$$

where $\underline{x}(n)=[x(n)\ x(n-1)\ \ldots\ x(n-N+1)]^T$ is the input vector, $\underline{w}(n)=[w_0(n)\ w_1(n)\ \ldots\ w_{N-1}(n)]^T$ is the weight vector, T denotes transpose, n is the time index, and N is the order of filter. This example is in the form of a finite impulse response filter as well as the convolution (inner product) of two vectors $\underline{x}(n)$ and $\underline{w}(n)$. The implementation of Equation (1) is illustrated using the following C program:

```
y[n] = 0.;
for (i = 0; i < N; i++) {
      y[n] += wn[i]*xn[i];
      {
```

where wn [i] denotes wi(n) and xn[i] represents x(n−i).

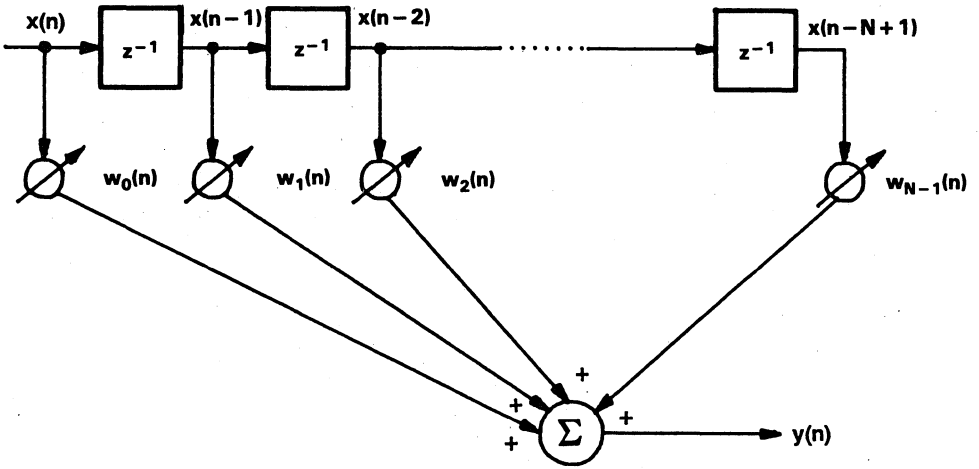


Figure 6. Transversal Filter Structure

TMS320C25 Implementation

The architecture of TMS320C25 [13] is optimized to implement the FIR filter. After execution of the CNFP (Configure Block B0 as Program Memory) instruction, the filter coefficients $w_i(n)$ from RAM block B0 (via program bus) and data $x(n-i)$ from RAM block B1 (via data bus) are available simultaneously for the parallel multiplier (see Figure 7).

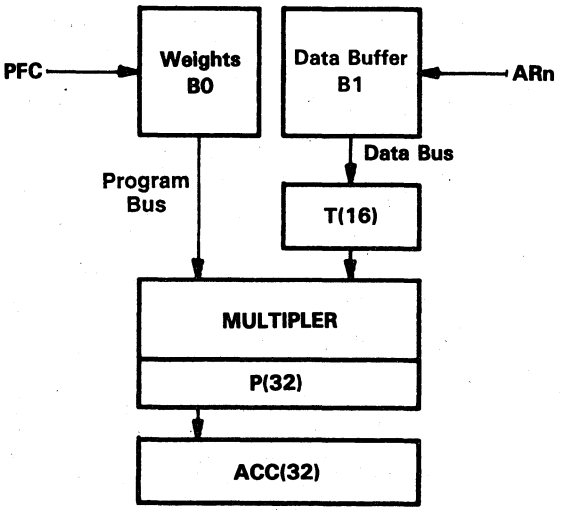
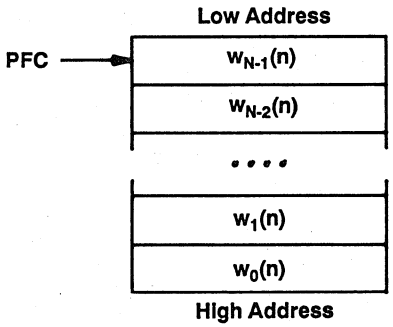


Figure 7. TMS320C25 Arithmetic Unit (after execute CNFP instruction)

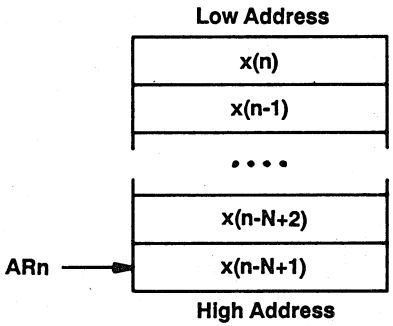
The MACD instruction enables complete multiply/accumulate, data move, and pointer update operations to be completed in a single instruction cycle (80 ns) if filter coefficients are stored in on-chip RAM or ROM or in off-chip program memory with zero wait states. Since the adaptive weights $w_i(n)$ need to be updated in every iteration, the filter coefficients must be stored in RAM. The implementation of the inner product in Equation (1) can be made even more efficient with a repeat instruction, RPTK. An N-weight transversal filter can be implemented as follows [23]:

```
LARP    ARn
LRLK    ARn,LASTAP
RPTK    N-1
MACD    COEFFP,*-                                   (A)
```

Where ARn is an auxiliary address register that points to $x(n-N+1)$, and the Prefetch Counter (PFC) points to the last weight $w_{N-1}(n)$ indicated by COEFFP. When the MACD instruction is repeated, the coefficient address is transferred to the PFC and is incremented by one during its operation. Therefore, the components of weight vector $\underline{w}(n)$ are stored in B0 as



The MACD in repeat mode will also copy data pointed to by ARn, to the next higher on-chip RAM location. The buffer memories of transversal filter are therefore stored as



In general, roundoff noise occurs after each multiplication. However, the TMS320C25 has a 16×16-bit multiplier and a 32-bit accumulator, so there is no roundoff during the summing of a set of product terms in Program (A). All multiplication products are represented in full precision, and rounding is performed after they are summed. Thus y(n) is obtained from the accumulator with only one roundoff, which minimizes the roundoff noise in the output y(n). Since both the tapped delay line and the adaptive weights are stored in data RAM to achieve the fastest throughput, the highest transversal filter order for efficient implementation on the TMS320C25 is 256. However, if necessary, higher order filters can be implemented by using external data RAM.

TMS320C30 Implementation

The architecture of TMS320C30 [14] is quite different from TI's second generation processors. Instead of using program/data memory, it provides two data address buses to do the data memory manipulations. This feature allows two data memory addresses to be generated at the same time. Hence, parallel data store, load, or one data store with one data load can be done simultaneously. Such capabilities make the programming much easier and more flexible. Since the hardware multiplier and arithmetic logic unit (ALU) of TMS320C30 are separated, with proper operand arrangement, the processor can do one multiplication and one addition or subtraction at the same time. With these two combined features, the TMS320C30 can execute several other parallel instructions. These parallel instructions can be found in Section 11 of the *Third-Generation TMS320 User's Guide* [14]. Associating with single repeat instruction RPTS, an inner product in Equation (1) can be implemented as follows:

```
    MPYF3   *AR0++(1)%,*AR1++(1)%,R1    ; w[0].x[0]
    RPTS    N-2                         ; Repeat N-1 times
    MPYF3   *AR0++(1)%,*AR1++(1)%,R1    ; y[] = w[].x[]
||  ADDF3   R1,R2,R2
    ADDF3   R1,R2,R2                    ; Include last product
```

where auxiliary registers AR0 and AR1 point to x and w arrays. The addition in the parallel instruction sums the previous values of R1 and R2. Therefore, R1 is initialized with the first product prior to the repeat instruction RPTS.

Note that the implementation above does not move the data in the x array like MACD does in TMS320C25. For filter delay taps, the TMS320C30 uses a circular buffer method to implement the delay line. This method reserves a certain size of memory for the buffer and uses a pointer to indicate the beginning of the buffer. Instead of moving data to next memory location, the pointer is updated to point to the previous memory location. Therefore, from the new beginning of the buffer, it has the effect of the tapped delay line. When the value of the pointer exceeds the end of the buffer, it will be circled around to the other end of the buffer. It works just like joining two ends of the buffer together as a necklace. Thus, new data is within the circular queue, pointed to by AR0, replacing

the oldest value. However, from an adaptive filter point of view, data doesn't have to be moved at this point yet.

TMS320C30 has a 32-bit floating point multiplier and the result from the multiplier is put and accumulated into a 40-bit extended precision register. If the input from A/D converter is equal to or less than 16 bits, there is no roundoff noise after multiplication. Theoretically, the TMS320C30 can implement a very high order of adaptive filter. However, for the most efficient implementation, the limitation of filter order is 2K because the TMS320C30 external data write requires at least two cycles. If the filter coefficients are put in somewhere other than internal data RAM, the instruction cycles will be increased.

LMS Adaptation Algorithm

The adaptation algorithm uses the error signal

$$e(n) = d(n) - y(n), \tag{2}$$

where d(n) is the desired signal and y(n) is the filter output. The input vector $\underline{x}(n)$ and e(n) are used to update the adaptive filter coefficients according to a criterion that is to be minimized. The criterion employed in this section is the mean-square error (MSE)ϵ:

$$\epsilon = E[e^2(n)] \tag{3}$$

where E [.] denotes the expectation operator. If y(n) from Equation (1) is substituted into Equation (2), then Equation (3) can be expressed as

$$\epsilon = E[d^2(n)] + \underline{w}^T(n)R\underline{w}(n) - 2\,\underline{w}^T(n)\underline{p} \tag{4}$$

where $R = E[x(n)x^T(n)]$ is the N x N autocorrelation matrix, which indicates the sample-to-sample correlation within a signal, and $\underline{p} = E[d(n)\,\underline{x}(n)]$ is the N x 1 cross-correlation vector, which indicates the correlation between the desired signal d(n) and the input signal vector $\underline{x}(n)$.

The optimum solution $w^* = [w_0^*\ w_1^*\ \ldots\ w_{N-1}^*]^T$, which minimizes MSE, is derived by solving the equation

$$\frac{\delta\epsilon}{\delta\underline{w}(n)} = 0 \tag{5}$$

This leads to the normal equation

$$R\,\underline{w}^* = \underline{p} \tag{6}$$

If the R matrix has full rank (i.e., R^{-1} exists), the optimum weights are obtained by

$$\underline{w}^* = R^{-1} \underline{p} \tag{7}$$

In Linear Predictive Coding (LPC) of a speech signal, the input speech is divided into short segments, the quantities of R and $\underline{p}$ are estimated, and the optimal weights corresponding to each segment are computed. This procedure is called a block-by-block data-adaptive algorithm [24].

A widely used LMS algorithm is an alternative algorithm that adapts the weights on a sample-by-sample basis. Since this method can avoid the complicated computation of R^{-1} and $\underline{p}$, this algorithm is a practical method for finding close approximate solutions to Equation (7) in real time. The LMS algorithm is the steepest descent method in which the next weight vector w(n+1) is increased by a change proportional to the negative gradient of mean-square-error performance surface in Equation (7)

$$\underline{w}(n+1) = \underline{w}(n) - u\underline{\nabla}(n) \tag{8}$$

where u is the adaptation step size that controls the stability and the convergence rate. For the LMS algorithm, the gradient at the nth iteration, $\underline{\nabla}(n)$, is estimated by assuming squared error $e^2(n)$ as an estimate of the MSE in Equation (3). Thus, the expression for the gradient estimate can be simplified to

$$\underline{\nabla}(n) = \frac{\delta[e^2(n)]}{\delta \underline{w}(n)} = -2\, e(n)\, \underline{x}(n) \tag{9}$$

Substitution of this instantaneous gradient estimate into Equation (8) yields the Widrow-Hoff LMS algorithm

$$\underline{w}(n+1) = \underline{w}(n) + 2\, u\, e(n)\, \underline{x}(n) \tag{10}$$

where 2 u in Equation (10) is usually replaced by u in practical implementation.

Starting with an arbitrary initial weight vector $\underline{w}(0)$, the weight vector $\underline{w}(n)$ will converge to its optimal solution $\underline{w}^*$, provided u is selected such that [1]

$$0 < u < \frac{1}{\lambda_{max}} \tag{11}$$

where λ_{max} is the largest eigenvalue of the matrix R. λ_{max} can be bounded by

$$\lambda_{max} < \text{Tr}\,[R] = \sum_{i=0}^{N-1} r\,(0) = N\,r(0) \tag{12}$$

where Tr [.] denotes the trace of a matrix and $r(0) = E\,[x^2(n)]$ is average input power.

For adaptive signal processing applications, the most important practical consideration is the speed of convergence, which determines the ability of the filter to track nonstationary signals. Generally speaking, weight vector convergence is attained only when the slowest weight has converged. The time constant of the slowest mode is [1]

$$t = \frac{1}{u\lambda_{min}} \tag{13}$$

This indicates that the time constant for weight convergence is inversely proportional to u and also depends on the eigenvalues of the autocorrelation matrix of the input. With the disparate eigenvalues, i.e., $\lambda_{max} >> \lambda_{min}$, the setting time is limited by the slowest mode, λ_{min}. Figure 8 shows the relaxation of the mean square error from its initial value ϵ_0 toward the optimal value ϵ_{min}.

Adaptation based on a gradient estimate results in noise in the weight vector, therefore a loss in performance. This noise in the adaptive process causes the steady state weight vector to vary randomly about the optimum weight vector. The accuracy of weight vector in steady state is measured by excess mean square error (excess MSE = $E\,[\epsilon - \epsilon_{min}]$). The excess MSE in the LMS algorithm [1] is

$$\text{excess MSE} = u\ \text{Tr}[R]\ \epsilon_{min} \tag{14}$$

where ϵ_{min} is minimum MSE in the steady state.

Equations (13) and (14) yield the basic trade-off of the LMS algorithm: to obtain high accuracy (low excess MSE) in the steady state, a small value of u is required, but this will slow down the convergence rate. Further discussions of the characteristics and properties of the LMS algorithm are presented in [1, 3 through 9]. The implementations of LMS algorithm with the TMS320C25 and TMS320C30 are presented next.

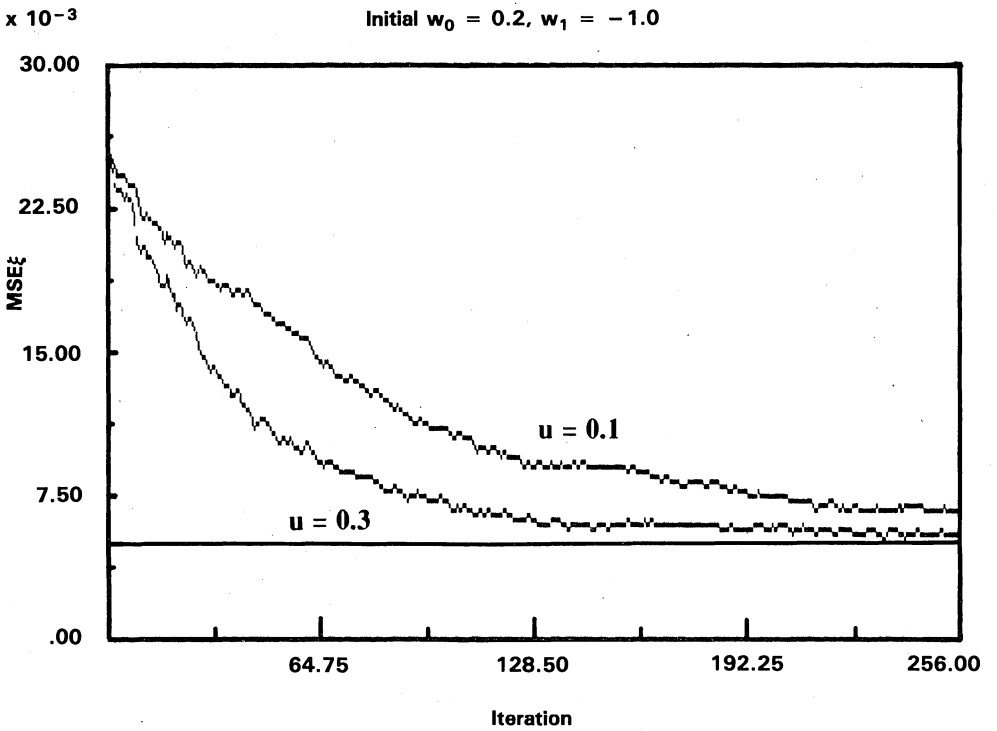


Figure 8. Learning Curve of an Adaptive Transversal Filter and an LMS Algorithm with Different Step Sizes

Since u*e(n) is constant for N weights update, the error signal e(n) is first multiplied by u to get ue(n). This constant can be computed first and then multiplied by x(n) to update w(n). An implementation method of the LMS algorithm in Equation (10) is illustrated as

```
ue(n) = u*e[n];
for (i=0; i<N; i++) {
      wn[i] += uen * xn[i];
}
```

TMS320C25 Implementation

The TMS320C25 provides two powerful instructions (ZALR and MPYA) to perform the update example in Equation (10).

- ZALR loads a data memory value into the high-order half of the accumulator while rounding the value by setting bit 15 of the accumulator to one and setting bits 0-14 of the accumulator to zero. The rounding is necessary because it can reduce the roundoff noise from multiplication.
- MPYA accumulates the previous product in the P register and multiplies the operand with the data in T register.

Assuming that ue(n) is stored in T and the address pointer is pointing to AR3, the adaptation of each weight is shown in the following instruction sequence:

```
        LRLK  AR1,N-1          ; Initialize loop counter
        LRLK  AR2,COEFFD       ; Point to w_{N-1}(n)
        LRLK  AR3,LASTAP+1     ; Point to x(n-N+1), since MACD in (A)
                               ; Already moved elements of current
                               ; x(n) to the next higher location
        MPY   *-,AR2           ; P=ue(n) * x(n-N+1)
ADAP    ZALR  *,AR3            ; Load w_i(n) and round
        MPYA  *-,AR2           ; ACC=P+w_i(n) and P=ue(n) * x(n-i)
        SACH  *+,0,AR1         ; Store w_i(n+1)
        BANZ  ADAP,*-,AR2      ; Test loop counter, if counter not
                               ; Equal to 0, decrement counter,
                               ; Branch to ADAP and select AR2 as
                               ; Next pointer.
```

For each iteration, N instruction cycles are needed to perform Equation (1), 6N instruction cycles are needed to perform weight updates in Equation (10), and the total number of instruction cycles needed is 7N+28. An example of a TMS320C25 program implementing a LMS transversal filter is presented in Appendix A1. Note that BANZ needs three instruction cycles to execute. This can be avoided by using straight line code, which requires 4N+33 instruction cycles [25].

TMS320C30 Implementation

Although the TMS320C30 doesn't provide any specific instruction for adaptive filter coefficients update, it still can achieve the weight updating in two instructions because of its powerful architecture. The TMS320C30 has a repeat block instruction RPTB, which allows a block of instructions to be repeated a number of times without any penalty for looping. A single repeat mode, RM, in the status register, ST, and three registers – repeat start address (RS), repeat end address (RE), and repeat counter (RC) – control the block repeat. When RM is set, the PC repeats the instructions between RS and RE a number of times, which is determined by the value of RC. The repeat modes repeat a block of code at least once in a typical operation. The repeat counter should be loaded with one less than the desired number of repetitions. Assuming the error signal e(n) in Equation (10) is stored in R7, the adaptation of filter coefficients is shown as follows:

```
        MPYF3   *AR0++(1)%,R7,R1  ; R1 = u*e(n)*x(n)
        LDI     order-3,RC        ; Initialize repeat counter
        RPTB    LMS               ; Do i = 0, N-3
        MPYF3   *AR0++(1)%,R7,R1   ; Compute u*e(n)*x(n-i-1)
      ||ADDF3   *AR1,R1,R2        ; Compute wi(n) + u*e(n)*x(n-i)
LMS     STF     R2,*AR1++(1)%     ; Store wi(n+1)

        MPYF3   *AR0,R7,R1        ; For i = N-2
      ||ADDF3   *AR1,R1,R2
        STF     R2,*AR1++(1)%     ; Store wN-2(n+1)
        ADDF3   *AR1,R1,R2        ; Include last w
        STF     R2,*AR1++(1)%     ; Store wN-1(n+1)
```

where auxiliary register AR0 and AR1 point to x and w arrays. R1 is updated before loop since the accumulation in the parallel instruction uses the previous value in R1. In order to update x array pointer to the new beginning of the data buffer for next iteration (i.e., perform the data move), one of the loop instruction set has been taken out of loop and modified by eliminating the incrementation of AR0.

To perform an N–weight adaptive LMS transversal filter on TMS320C30 requires 3N+15 instruction cycles. There are N and 2N instruction cycles to perform Equations (1) and (10), respectively. The TMS320C30 example program is given in Appendix A2.

The LMS algorithm considerably reduces the computational requirements by using a simplified mean square error estimator (an estimate of the gradient). This algorithm has proved useful and effective in many applications. However, it has several limitations in performance such as the slow initial convergence, the undesirable dependence of its convergence rate on input signal statistics, and an excess mean square error still in existence after convergence.

Symmetric Transversal Structure [5]

A transversal filter with symmetric impulse response (weight values) about the center weight has a linear phase response. In applications such as speech processing, linear phase filters are preferred since they avoid phase distortion by causing all the components in the filter input to be delayed by the same amount. The adaptive symmetric transversal structure is shown in Figure 9.

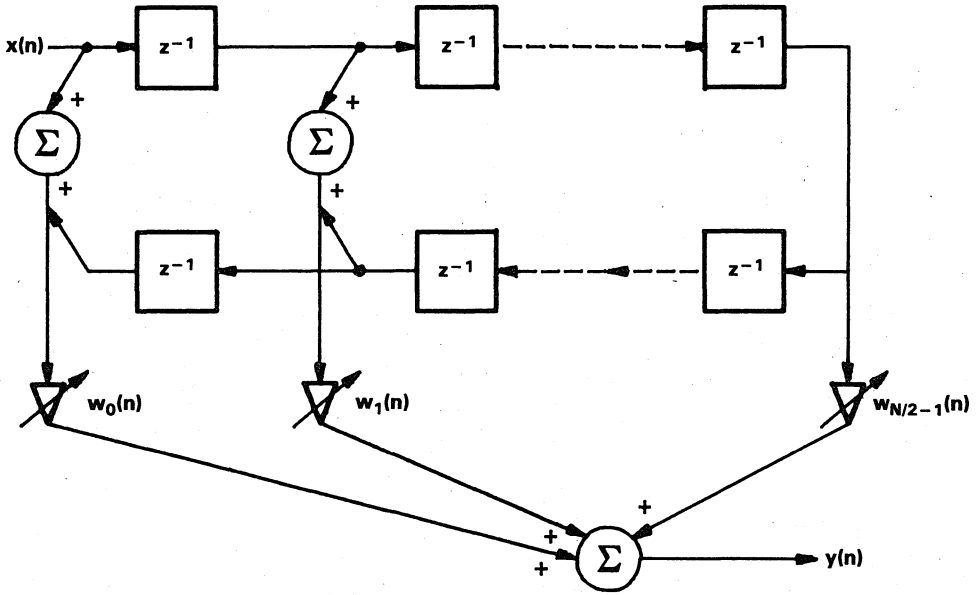


Figure 9. Symmetric Transversal Structure (even order)

This filter is actually an FIR filter with an impulse response that is symmetric about the center tap. The output of the filter is obtained as

$$y(n) = \sum_{i=0}^{N/2-1} w_i(n)\,[x(n-i) + x(n-N+i+1)] \qquad (15a)$$

where N is an even number. Note that, for fixed-point processors, the addition in the brackets may introduce overflow because the input signals $x(n-i)$ and $x(n-N+i+1)$ are in the range of -1 and $1-2^{-15}$. This problem can be solved by shifting x(n) to the right one bit. The update of the weight vector is

$$w_i(n+1) = w_i(n) + ue(n)[x(n-1) + x(n-N+i+1)] \qquad (15b)$$

for $i=0,1,\ldots,(N/2^{-1})$, which requires N/2 multiplications and N additions. Theoretically, this symmetric structure can also reduce computational complexity since such filters require only half the multiplications of the general transversal filter. However, it is true only for the TMS320C30 processor. When a filter is implemented on the TMS320C25, the transversal structure is more efficient than the symmetric transversal structure due to the pipeline multiplication and accumulation instruction MACD, which is optimized to implement convolution in Equation (1).

TMS320C25 Implementation

For TMS320C25, in order to implement the instructions MAC, ZALR, and MPYA, we can trade memory requirements for computation saving by defining

$$z(n-i) = x(n-i) + x(n-N+i+1) \ , \ i=0,1,\ldots,N/2^{-1} \tag{16a}$$

Now, Equation (15) can be expressed as

$$y(n) = \sum_{i=0}^{N/2-1} w_i(n)\, z(n-i) \tag{16b}$$

$$w_i(n+1) = w_i(n) + u\, e(n)\, z(n-i) \ , \ i=0,1,\ldots,N/2^{-1} \tag{16c}$$

Equation (16a) can be implemented using the TMS320C25 as

```
        LARK    AR1, N/2-1     ; Counter = N/2 -1
        LRLK    AR2,LAST_X     ; Point to x(n-N+1)
        LRLK    AR3,FIRST_X    ; Point to x(n)
        LRLK    AR4,FIRST_Z    ; Point to z(n)
        LARP    AR3
SYM     LAC     *+,0,AR2
        ADD     *-,0,AR4
        SACL    *+,0,AR1
        BANZ    SYM,*-,AR3
```

The instruction sequence to implement the LMS algorithm in Equations (1) and (10) can be used to implement Equations (16b) and (16c), except using MAC instead of MACD in Program (A). Therefore, N instruction cycles are needed to shift data in x(n), 3N instruction cycles are needed to implement Equation (16a), N/2 for Equation (16b), and 3N for Equation (16c). The total number of instruction cycles required to implement the symmetric transversal filter with the LMS algorithm is 7.5N+38. Where 7.5N is an integer because N is chosen as an even number. The 0.5N instruction cycles come from Equation (15a) since symmetric transversal structure folds the filter taps into half of the order N (see Figure 9). The maximum filter length for most efficient code, 256, is the

same as for the FIR filter. The use of the additional data memory can be obtained from the reduced data memory requirement for weights of the symmetric transversal filter. The complete TMS320C25 program is given in Appendix B1.

Note that instead of storing buffer locations x(n) contiguously, then using DMOV to shift data in the buffer memory (requiring N cycles) at the end of each iteration, we can use a circular buffer with pointers pointing to x(n) and x(n−N+1). Since pointer updating requires several instruction cycles, compared with N cycles using DMOV to update the buffer memory contents, the circular buffer technique is more efficient if N is large.

TMS320C30 Implementation

As mentioned above, the TMS320C30 uses a circular buffer instead of data move technique. Therefore, it does not have to implement tapped delay line separately as TMS320C25. Equations (1) and (16a) can be combined and implemented in the same loop. The advantage of this is that a parallel instruction reduces the number of the instruction cycles. The implementation is shown as follows:

```
        LDF     0.0,R2                          ; Clear R2
        LDI     order/2-2,RC                    ; Set up loop counter
        RPTB    INNER                           ; Do i = 0, N/2 -2
        ADDF3   *AR4++(1)%,*AR5--(1)%,R1        ; z(i) = x(n-i) + x(n+N-i)
        MPYF3   R1,*AR1++(1),R3                 ; R3 = w[] * z[]
||      STF     R1,*AR2++(1)                    ; Store z(i)
INNER   ADDF3   R3,R2,R2                        ; Accumulate the result for y

        ADDF3   *AR4++(1)%,*AR5--(1)%,R1        ; For i = N/2 -1
        MPYF3   R1,*AR1--(IR0),R3
||      STF     R1,*AR2--(IR0)
        ADDF3   R3,R2,R2                        ; Include last product
```

where AR4 and AR5 point to x[0] and x[N−1]. AR1 and AR2 point to w and z array, respectively. IR0 contains value of N/2 −1. The same instruction codes of weight update of transversal filter can be used in symmetric transversal structure by changing the x array pointer to the z array pointer. Appendix B2 presents an example program. The total number of instructions needed is 2.5N+15, which is less than that of the transversal structure.

Lattice Structure [6]

An alternative FIR filter realization is the lattice structure [26]. A discussion of the transversal filter with the LMS algorithm shows that the convergence rate of the transversal structure is restricted by the correlation of signal components; i.e., the eigenvalue spread, $\lambda_{max}/\lambda_{min}$. The lattice structure is a decorrelating transform based on a family of prediction error filters as illustrated in Figure 10. The recursive equations that describe the lattice predictor are

$$f_0(n) = b_0(n) = x(n) \tag{17a}$$

$$f_m(n) = f_{m-1}(n) - k_m(n)b_{m-1}(n-1),\ 0 < m <= M \tag{17b}$$

$$b_m(n) = b_{m-1}(n-1) - k_m(n)f_{m-1}(n),\ 0 < m <= M \tag{17c}$$

where $f_m(n)$ represents the forward prediction error, $b_m(n)$ represents the backward prediction error, $k_m(n)$ is the reflection coefficients, m is the stage index, and M is the number of cascaded stages. The lattice structure has the advantage of being order-recursive. This property allows adding or deleting of stages from the lattice without affecting the existing stages.

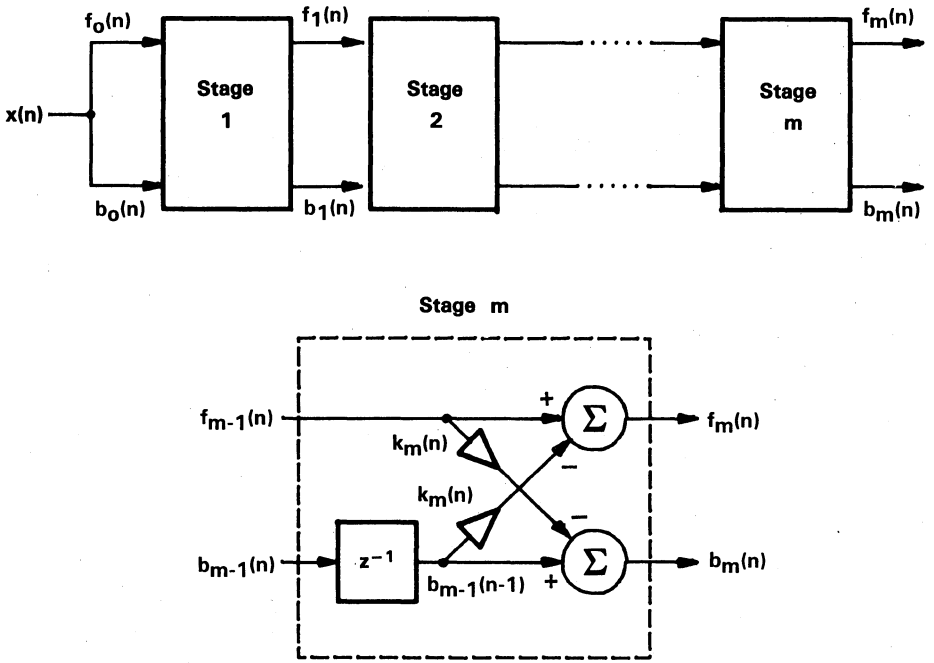


Figure 10. Lattice Structure

To implement the lattice filter for processing actual data, the reflection coefficients $k_m(n)$ are required. These coefficients can be computed according to estimates of the autocorrelation coefficients using Durbin's algorithm. However, it would be more efficient if these reflection coefficients could be estimated directly from the data and updated on a sample-by-sample basis, such as LMS algorithm [6]. The reflection coefficient $k_m(n+1)$ can be recursively computed [7]:

$$k_m(n+1) = k_m(n) + u[f_m(n)b_{m-1}(n-1) + b_m(n)f_{m-1}(n)],\ 0 < m <= M \quad (18)$$

For applications such as noise cancellation, channel equalization, line enhancement, etc., the joint-process estimation [3] illustrated in Figure 11 is required. This device performs two optimum estimations: the lattice predictor and the multiple regression filter. The following equations define the implementation of the regression filter

$$e_0(n) = d(n) - b_0(n)g_0(n) \quad (19a)$$

$$e_m(n) = e_{m-1}(n) - b_{m-1}(n)g_{m-1}(n),\ 0 < m <= M \quad (19b)$$

$$g_m(n+1) = g_m(n) + u_{em}(n)b_m(n), \quad 0 <= m <= M \quad (20)$$

where the LMS algorithm is used to update the coefficients of the regression filter. For noise cancellation application, $e_m(n)$ corresponds to the output e(n) in Figure 5. For applications such as adaptive line enhancer and channel equalizer, filter output y(n) is obtained as

$$y(n) = \sum_{m=0}^{M} g_m(n)\, b_m(n) \quad (21)$$

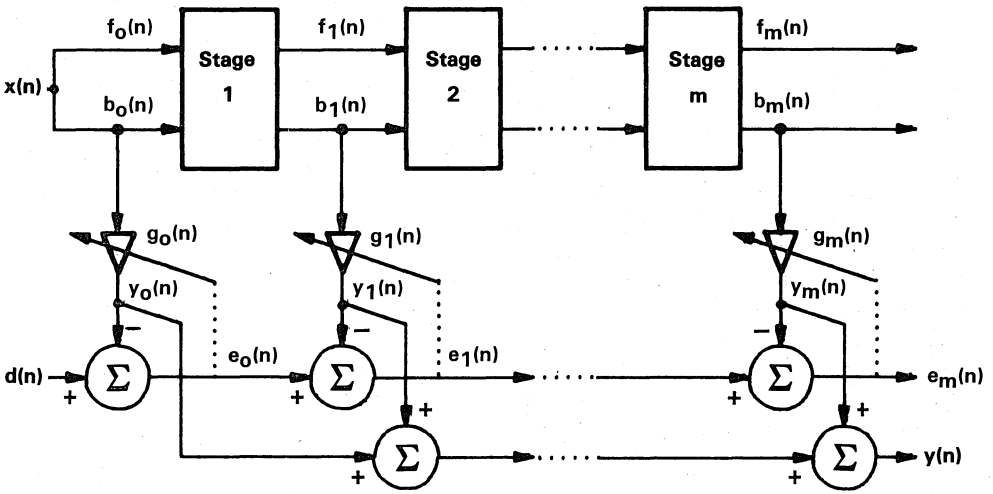


Figure 11. Lattice Structure with Joint Process Estimation

TMS320C25/TMS320C30 Implementation

There are five memory locations—$f_m(n)$, $b_m(n)$, $b_m(n-1)$, $k_m(n)$, and $g_m(n)$—required for each stage. The limitation of on-chip data RAM is 544 words for the TMS320C25 and 2K words for the TMS320C30. A maximum of 102 stages can therefore be implemented on a single TMS320C25 for the highest throughput. Here, another advantage of TMS320C30 architecture design is shown. Since the operands of the mathematic operations can be either memory or register on the TMS320C30, and there is no need to preserve the values of f_m array for the next iteration (refer to Equations (17) and (18)), the f_m array can be replaced by an extended precision register. Thus, for the most efficient codes, the stage limitation of lattice structure for TMS320C30 is 512, or one-fourth of the 2K on-chip RAM.

Lattice structures have superior convergence properties relative to transversal structures and good stability properties; e.g., low sensitivity to coefficient quantization, low roundoff noise, and the ability to check stability by inspection. The disadvantages of lattice filter algorithms are that they are numerically complex and require mathematical sophistication to thoroughly understand their derivations. Furthermore, as shown in Appendixes C1 and C2, lattice structures cannot take advantage of the TMS320C25 and TMS320C30's pipeline architecture to achieve high throughput. The total number of instruction cycles needed is 33M+32 for TMS320C25 and 14M+4 for TMS320C30.

Modified LMS Algorithms [5]

The LMS algorithm described in previous sections is the most widely used algorithm in practical applications today. In this section, a set of LMS-type algorithms (all direct variants of the LMS algorithm) are presented and implemented. The motivation for each is some practical consideration, such as faster convergence, simplicity in implementation, or robustness in operation. The description of these algorithms is based on the transversal structure. However, these algorithms can be applied to the symmetric transversal structure and the lattice structure as well.

Normalized LMS Algorithm

The stability, convergence time, and fluctuation of the adaptation process is governed by the step size u and the input power to the adaptive filter. In some practical applications, you may need an automatic gain control (AGC) on the input to the adaptive filter. The normalized LMS algorithm is one important technique used to improve the speed of convergence. This is accomplished while maintaining the steady-state performance independent of the input signal power. This algorithm uses a variable convergence factor u(n), which represents a u that is a function of the time index,

$$u(n) = a \,/\, \mathrm{var}(n) \tag{22}$$

and

$$\underline{w}(n+1) = \underline{w}(n) + u(n)e(n)\underline{x}(n) \tag{23}$$

where a is a convergence parameter, and var(n) is an estimate of the input average power at time n using the recursive equation

$$\text{var}(n) = (1 - b)\ \text{var}(n-1) + b\ x^2(n) \tag{24}$$

where $0 < b << 1$ is a smoothing parameter. In practice, a is chosen equal to b.

For fixed-point processors, there is a way to reduce the computation of power estimation. Since b in Equation (24) doesn't have to be an exact number, it is computationally convenient to make b a power of 2. If $b = 2^{-m}$, the multiplication of b can be implemented by shifting right m bits. Therefore, the var(n) in Equation (24) is computed by

$$\begin{aligned}\text{var}(n) &= \text{var}(n-1) - b\ \text{var}(n-1) + b\ x^2(n) \\ &= \text{var}(n-1) - \text{var}(n-1) * 2^{-m} + x^2(n) * 2^{-m}\end{aligned}$$

Then, assuming the variance var(n) of input signal is stored in the data memory VAR and its initial value is 0.99997 ($= 1 - 2^{-15}$), The implementation of this equation using TMS320C25 assembly code is

```
LARP    AR3
LRLK    AR3,FRSTAP    ; Point to input signal x
SQRA    *             ; Square input signal
SPH     ERRF
ZALH    VAR           ; ACC = var(n-1)
SUB     VAR,SHIFT     ; ACC = (1-b) var(n-1)
ADD     ERRF,SHIFT    ; ACC = (1-b) var(n-1) + b x2(n)
SACH    VAR           ; Store var(n)
```

The normalized LMS algorithm can be implemented as

```
var = b1 * var + b * xn[0] * xn[0];
unen = e[n] * a / var;
for (i = 0; i< N; i++)
wn[i] += unen * xn[i];
```

where $b_1 = (1-b)$, xn[0] = x(n), and unen = u(n)*e(n). This normalized technique reduces the dependency of convergence speed on input signal power at the cost of increased computational complexity, especially the division in Equation (22). The algorithms of implementing the fixed-point and floating-point division on the TMS320C25 and

TMS320C30 can be found in the user's guide for each device [13, 14]. Since the power of input signal is always positive, those codes can be simplified to save computation time.

Since the power estimation in Equation (24) and step size normalization in Equation (22) are performed once for each sample x(n), the computation increase can be ignored when N is large. As shown in Appendixes D1 and D2, the total number of instruction cycles needed for the normalized LMS algorithm (7N+57 for the TMS320C25 and 3N+47 for the TMS320C30) is slightly higher than for the LMS algorithm (7N+34 and 3N+15) when N is large.

Sign LMS Algorithms

The LMS algorithm requires 2N multiplications and additions for each iteration; this amount is much lower than the requirements for many other complicated adaptive algorithms, such as Kalman and Recursive Least Square (RLS) [3]. However, there are three simplified versions of the LMS algorithm (sign-error LMS, sign-data LMS, and sign-sign LMS) that save the number of multiplications required and extend the real-time bandwidth for some applications [5, 27].

First, the sign-error LMS algorithm can be expressed as

$$\underline{w}(n+1) = \underline{w}(n) + u\ \text{sign}[e(n)]\ \underline{x}(n) \tag{25}$$

where $$\text{sign}[e(n)] = \begin{cases} 1, & \text{if } e(n) \geq 0 \\ -1, & \text{if } e(n) < 0 \end{cases}$$

The C program implementation of sign-error LMS algorithm is

```
tu = u;
if (e[n] < 0.) {
    tu = -u; }
for (i=0; i<N; i++) {
    wn[i] += tu * xn[i];
}
```

As shown in Appendixes E1 and E2, the instruction sequence to implement weight update with the sign-error LMS algorithm is identical to that with the LMS algorithm. The difference is that the sign-error LMS algorithm uses the sign [e(n)]*u instead of e(n)*u before the update loop. Note that, for fixed-point processors, if u is chosen to be a power of two, the u x(n) can be accomplished by shifting right the elements in x(n). This algorithm keeps the same convergence direction as the LMS algorithm. Thus, the sign-error LMS algorithm should remain efficient, provided the variable gain u(n) is matched to this change. However, the use of constant step size u to reduce computation comes at the expense of a slow convergence rate since smaller u is normally used for stability reasons.

The programs in Appendixes E1 and E2 implement a transversal filter with sign-error LMS algorithm in looped code. The total number of instruction cycles needed for this algorithm using the TMS320C25 is 7N+26, which is slightly less than for the LMS algorithm's 7N+28. Computing u*e(n) takes 5 instruction cycles. The sign-error LMS algorithm determines the sign of the u by checking the sign of e(n), which takes only 3 instruction cycles. The total number of instruction cycles needed for the sign-error LMS algorithm using the TMS320C30 is 3N+16, which is slightly higher than for the LMS algorithm. This occurs because the TMS320C30 takes only one instruction cycle to compute u*e(n) and two instruction cycles to determine the sign of the u.

Secondly, the sign-data LMS algorithm is

$$\underline{w}(n+1) = \underline{w}(n) + u\, e(n)\, \text{sign}[\underline{x}(n)] \tag{26}$$

This equation can be implemented as

$$\begin{aligned} w_i(n+1) &= w_i(n) + ue(n)\ , \text{ if } x(n-i) >= 0 \\ &= w_i(n) - ue(n)\ , \text{ if } x(n-i) < 0 \end{aligned}$$

for i=0,1,...,N−1. Since the sign determination is required inside the adaptation loop to determine the sign of x(n−i), slower throughput is expected. The total number of instruction cycles needed is 11N+26 for the TMS320C25 and 5N+16 for the TMS320C30.

Finally, the sign-sign LMS algorithm is

$$\underline{w}(n+1) = \underline{w}(n) + u\, \text{sign}[e(n)]\, \text{sign}[\underline{x}(n)] \tag{27}$$

which requires no multiplications at all and is used in the CCITT standard for ADPCM transmission. As we can see from the above equations, the number of multiplications is reduced. This simplified LMS algorithm looks promising and is designed for VLSI or discrete IC implementation to save multiplications.

The sign-sign LMS algorithm can be implemented as

```
for (i=0; i<N; i++) {
     if (e[n] >= 0.) {
          if (xn[i] >= 0.)
               wn[i] += u;
          else
               wn[i] -= u; }
                    else {
          if (xn[i]> = 0.)
               wn[i]-= u;
```

```
        else
            wn[i] += u; } }
```

When this algorithm is implemented on TMS320C25 and TMS320C30 with pipeline architecture and a parallel multiplier, the performance of sign-sign LMS algorithm is poor compared to standard LMS algorithm due to the determination of sign of data, which can break the instruction pipeline and can severely reduce the execution speed of the processors.

In order to avoid double branches inside the loop, the XOR instruction is utilized to check the sign bit of e(n) and x(n−i). The sign-sign LMS algorithm can be implemented as

$$w_i(n+1) = w_i(n) + u \text{ , if sign[e(n)] = sign[x(n-i)]}$$
$$= w_i(n) - u \text{ , otherwise}$$

The following TMS320C25 instruction sequence implements this algorithm without branching (assuming that the current address register used is AR3):

```
        LRLK    AR1,N-1         ; Set up counter
        LRLK    AR2,COEFFD      ; Point to wi(n)
        LRLK    AR3,LASTAP+1    ; Point to x(n-i)
ADAP    LAC     *-,0,AR2        ; Load x(n-i)
        XOR     ERR             ; XOR with e(n)
        SACL    ERRF            ; Save sign bit, sign = 0 if same signs
                                ; Sign = 1 if different signs
        LAC     ERRF            ; Sign extension to ACCH,
                                ; ACCH = 0 If ERRF > = 0
                                ; ACCH = 0FFFFh if ERRF < 0
        XORK    MU,15           ; Take one's complement of m
                                ; If sign = 1
        ADD     *,15            ; Weight update
        SACH    *+,1,AR1        ; Save new weight
        BANZ    ADAP,*-,AR3
```

The one's complement of u is used instead of −u, because they are only slightly different and the step size does not require the exact number. The weight update with this technique requires 10N instruction cycles and FIR filtering requires N instruction cycles so that the total number of instruction cycles needed is 11N+21. The complete TMS320C25 assembly program is given in Appendix F1.

To determine whether a positive or negative u should be used without branching is trickier in the TMS320C30. Fortunately, the extended precision registers of TMS320C30 interpret the 32 most-significant bits of the 40-bit data as the floating-point number and the 32 least-significant bits of the 40-bit data as an integer. When a floating-point number

changes its sign, its exponent remains the same. Therefore, the sign of step size u can be determined by using XOR logic on its mantissa. The following code shows how the sign-sign LMS algorithm is implemented on the TMS320C30.

```
        ASH     -31,R7          ; R7 = Sign[e(n)]
        XOR3    R0,R7,R5        ; R5 = Sign[e(n)] * u
        LDF     *AR0++(1)%,R6   ; R6 = x(n)
        ASH     -31,R6          ; R6 = Sign[x(n-i)]
        XOR3    R5,R6,R4        ; R4 = Sign[x(n-i)]*Sign[e(n)] * u
        ADDF3   *AR1,R4,R3      ; R3 = wi(n) + R4

        LDI     order-3,RC      ; Initialize repeat counter
        RPTB    SSLMS           ; Do i = 0, N-3
        LDF     *AR0++(1)%,R6   ; Get next data
||      STF     R3,*AR1++(1)%   ; Update wi(n+1)
        ASH     -31,R6          ; Get the sign of data
        XOR3    R5,R6,R4        ; Decide the sign of u
SSLMS   ADDF3   *AR1,R4,R3      ; R3 = wi(n) + R4

        LDF     *AR0,R6         ; Get last data
||      STF     R3,*AR1++(1)%   ; Update wN-2(n+1)
        ASH     -31,R6          ; Get the sign of data
        XOR3    R5,R6,R4        ; Decide the sign of u
        ADDF3   *AR1,R4,R3      ; Compute wN-1(n+1)
        STF     R3,*AR1++(1)%   ; Store last w(n+1)
```

Here, R0, R4, and R5 contain the value of u before updating. AR0 and AR1 point to x array and w array, respectively. R7 contains the value of error signal e(n). The complete program is given in Appendix F2. The total number of instruction cycles is 5N+16, which is much higher than LMS algorithm.

The sign-sign LMS algorithm is developed to reduce the multiplication requirement of the LMS algorithm. Since DSPs provide the hardware multiplier as a standard feature, this modification does not provide any advantage when implementing this algorithm on the DSPs. On the contrary, it causes some disadvantages since decision instructions will destroy the instruction pipeline. If you use the XOR logic operation in order to avoid using the decision instructions, the complexity of the program will be increased and the total number of instruction cycles will be greater than the regular LMS algorithm.

Leaky LMS Algorithm

When adaptive filters are implemented on signal processors with fixed word lengths, roundoff noise is fed back to adaptive weights and accumulates in time without bound. This leads to an overflow that is unacceptable for real-time applications. One solution is

based upon adding a small forcing function, which tends to bias each filter weight toward zero. The leaky LMS algorithm has the form

$$\underline{w}(n+1) = r\ \underline{w}(n) + u\ e(n)\ \underline{x}(n) \tag{28a}$$

where r is slightly less than 1.

Since r can be expressed as $1 - c$ and $c << 1$, the TMS320C25 can take advantage of the built-in shifters to implement this algorithm. Therefore, Equation (28a) can be changed to

$$\underline{w}(n+1) = \underline{w}(n) - c\ \underline{w}(n) + u\ e(n)\ \underline{x}(n) \tag{28b}$$

In order to achieve the highest throughput by using ZALR and MPYA, cw(n) can be implemented by shifting $w_i(n)$ right by m bits where 2^{-m} is close to c. Since the length of the accumulator is 32 bits and the high word (bits 16 to 31) is used for updating w(n), shifting right m bits of $w_i(n)$ can be implemented by loading $w_i(n)$ and shifting left $16 - m$ bits. The sequence of TMS320C25 instructions to implement Equation (28b) is shown as

```
        LRLK    AR1,N-1         ; Set up counter
        LRLK    AR2,COEFFD      ; Point to wi(n)
        LRLK    AR3,LASTAP+1    ; Point to x(n - i)
        LT      ERRF            ; T = ERRF =u*e(n)
        MPY     *-,AR2
ADAPT   ZALR    *,AR3
        MPYA    *-,AR2
        SUB     *,LEAKY         ; LEAKY=16-m
        SACH    *+,0,AR1
        BANZ    ADAPT,*-,AR2
```

For each iteration, 7N instruction cycles are needed to perform the adaptation process (6N for the LMS algorithm). The total number of instruction cycles needed is 8N+28 (see Appendix G1 for the complete program). The leaky factor r has the same effect as adding a white noise to the input. This technique not only can solve adaptive weights overflow problem, but also can be beneficial in an insufficient spectral excitation and stalling situation [5].

The method used above is especially for the TMS320C25, which has a free shift feature. Since TMS320C30 is a floating-point processor, r can simply multiply to filter coefficient. However, in order to reduce the instruction cycles, this multiplication can combine with another instruction to be a parallel instruction inside the loop. The following code shows how to rearrange the instructions from the LMS algorithm to include this multiplication without an extra instruction cycle.

```
        MPYF    @u__r,R7            ; R7 = e(n)*u/r
        MPYF3   *AR0++(1)%,R7,R1    ; R1 = e(n)*u*x(n)/r
        MPYF3   *AR0++(1)%,R7,R1    ; R1 = e(n)*u*x(n-1)/r
||      ADDF3   *AR1,R1,R2          ; R2 = w0(n) + e(n)*u*x(n)/r
        LDI     order-4,RC          ; Initialize repeat counter
        RPTB    LLMS                ; do i = 0, N-4
        MPYF3   *AR2,R2,R0          ; R0 = r*wi(n) + e(n)*u*x(n-i)
||      ADDF3   *+AR1(1),R1,R2      ; R2 = wi+1(n) + e(n)*u*x(nz-i-1)/r
LLMS    MPYF3   *AR0++(1)%,R7,R1    ; R1 = e(n)*u*x(n-i-2)/r
||      STF     R0,*AR1++(1)%       ; Store wi(n+1)

        MPYF3   *AR2,R2,R0          ; R0 = r*wN-3(n) + e(n)*u*x(n-N+3)
||      ADDF3   *+AR1(1),R1,R2      ; R2 = wN-2(n) + e(n)*u*x(n-N+2)/r
        MPYF3   *AR0,R7,R1          ; R1 = e(n)*u*x(n-N+1)/r
||      STF     R0,*AR1++(1)%       ; Store wN-3(n+1)
        MPYF3   *AR2,R2,R0          ; R0 = r*wi(n) + e(n)*u*x(n-N+2)
||      ADDF3   *+AR1(1),R1,R2      ; R2 = wN-1(n) +
*                                   ;        e(n)*u*x(n-N+1)/r
        MPYF3   *AR2,R2,R0          ; R0 = r*wi(n) + e(n)*u*x(n-N+1)
||      STF     R0,*AR1++(1)%       ; Store wN-2(n+1)
        STF     R0,*AR1++(1)%       ; Update last w
```

Auxiliary registers AR0 and AR1 point to x and w arrays. AR2 points to the memory location that contains value r. R7 contains the value of error signal e(n). R1 and R2 are updated before the loop because the parallel instructions inside the loop use the previous values in R1 and R2. Note that R1 is updated twice before the loop because the updating of R2 requires the previous value of R1. In order to update x array pointer to the new beginning of the data buffer for next iteration, two of the loop instruction sets have been taken out of loop and modified by eliminating the incrementation of AR0. The TMS320C30 assembly program of an adaptive transversal filter with the leakage LMS algorithm is listed in Appendix G2 as an example. The total number of instruction cycles for this algorithm is 3N+15, which is the same as the LMS algorithm. This example shows the power and flexibility of the TMS320C30.

Implementation Considerations

The adaptive filter structures and algorithms discussed previously were derived on the basis of infinite precision arithmetic. When implementing these structures and algorithms on a fixed integer machine, there is a limitation on the accuracy of these filters due to the fact that the DSP operates with a finite number of bits. Thus, designers must pay attention to the effects of finite word length. In general, these effects are input quantization, roundoff in the arithmetic operation, dynamic range constraints, and quantization of filter coefficients. These effects can either cause deviations from the original design criteria or create an effective noise at the filter output. These problems have been investigated extensively, and techniques to solve these problems have been developed [28, 29].

The effects of finite precision in adaptive filters is an active research area, and some significant results have been reported [30 through 32]. There are three categories of finite word length effects in adaptive filters:

- Dynamic Range Constraint (scaling to avoid overflow). Since this is not applicable for a floating-point processor, the TMS320C30 is not mentioned in this portion.
- Finite Precision Errors (errors introduced by roundoff in the arithmetic).
- Design Issues (design of the optimum step size u that minimizes system noise).

Dynamic Range Constraint

As shown in Figure 1, the most widely used LMS transversal filter is specified by the difference equations

$$y(n) = \sum_{i=0}^{N-1} w_i(n)\, x(n-i) \tag{29}$$

and

$$w_i(n+1) = w_i(n) + u*e(n)*x(n-i), \text{ for } i = 0, 1, \ldots, N-1 \tag{30}$$

where $x(n-i)$ is the input sequence and $w_i(n)$ are the filter coefficients.

If the input sequence and filter coefficients are properly normalized so that their values lie between -1 and 1 using Q15 format, no error is introduced into the addition. However, the sum of two numbers may become larger than one. This is known as overflow. The TMS320C25 provides four features that can be applied to handle overflow management [13]:

A. Branch on overflow conditions.
B. Overflow mode (saturation arithmetic).
C. Product register right shift.
D. Accumulator right shift.

One technique to inhibit the probability of overflow is scaling, i.e., constraining each node within an adaptive filter to maintain a magnitude less than unity. In Equation (29), the condition for $|y(n)| < 1$ is

$$x_{max} < 1 / \sum_{i=0}^{N-1} |w_i(n)| \tag{31}$$

where x_{max} denotes the maximum of the absolute value of the input. The right shifter of the TMS320C25, which operates with no cycle overhead, can be applied to implement scaling to prevent overflow of multiply-accumulate operations in Equation (29). By setting the PM bits of status register ST1 to 11 using the SPM or LST1 instructions, the P register output is right-shifted 6 places. This allows up to 128 accumulations without the possibility of an overflow. SFR instruction can also be used to right shift one bit of the accumulator when it is near overflow.

Another effective technique to prevent overflow in the computation of Equation (29) is using saturation arithmetic. As illustrated in Figure 12, if the result of an addition overflows, the output is clamped at the maximum value. If saturation arithmetic is used, it is common practice [28] to permit the amplitude of $x(n-i)$ to be larger than the upper bound given in Equation (31). Saturation of the filter represents a distortion, and the choice of scaling on the input depends on how often such distortion is permissible. The saturation arithmetic on the TMS320C25 is controlled by the OVM bit of status register ST0 and can be changed by the SOVM (set overflow mode), ROVM (reset overflow mode), or LST (load status register).

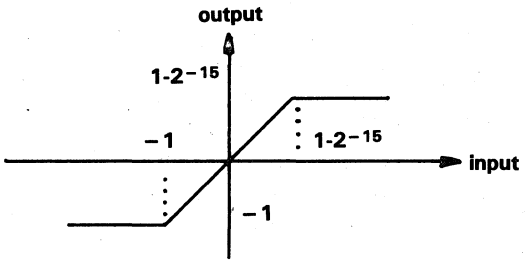


Figure 12. Saturation Arithmetic

Filter coefficients are updated using Equation (30). As illustrated in Figure 13, a new technique presented in reference 31 uses the scaling factor a to prevent filter's coefficients overflow during the weight updating operation. Suppose you use $a = 2^{-m}$. A right shift by m bits implements multiplication by a, while a left shift by m bits implements the scaling factor 1/a. Usually, the required value of a is not expected to be very small and depends on the application. Since a scales the desired signal, it does not affect the rate of convergence.

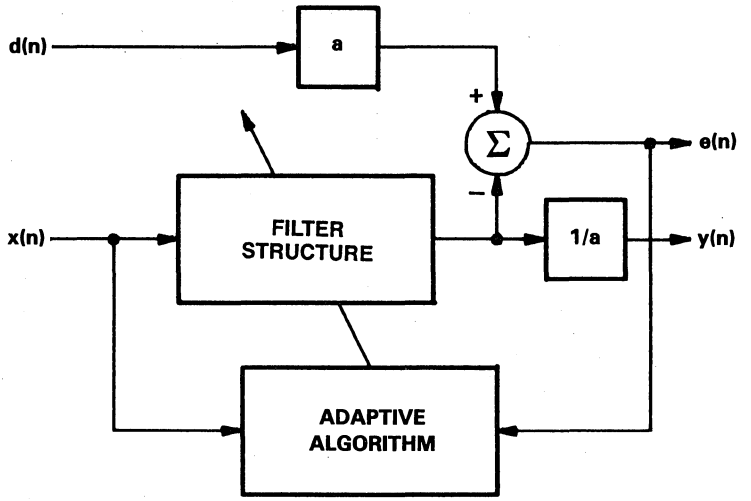


Figure 13. Fixed-Point Arithmetic Model of the Adaptive Filter

Finite Precision Errors

The TMS320C25 is a 16/32-bit fixed point processor. Each data sample is represented by a fractional number that uses 15 magnitude bits and one sign bit. The quantization interval

$$\delta = 2^{-b}, \tag{32}$$

(b = 15), is called the width of quantization since the numbers are quantized in steps of δ.

The products of the multiplications of data by coefficients within the filter must be rounded or truncated to store in memory or a CPU register. As shown in Figure 14, the roundoff error can be modeled as the white noise injected into the filter by each rounding operation. This white noise has a uniform distribution over a quantization interval and for rounding

$$-1/2\ \delta < e \leq 1/2\ \delta \tag{33a}$$

and

$$\delta_e^2 = (1/12)\ \delta^2 \tag{33b}$$

where δ_e^2 is the variance of the white noise.

In general, roundoff noise occurs after each multiplication. However, the TMS320C25 has a full precision accumulator, i.e., a 16×16-bit multiplier with a 32-bit accumulator, so there is no roundoff when you implement a set of summations and multiplications as in Equation (29). Rounding is performed when the result is stored back to memory location y(n), so that only one noise source is presented in a given summation node.

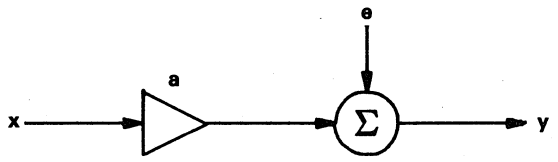


Figure 14. Fixed-Point Roundoff Noise Model

For floating-point arithmetic, the variance of the roundoff noise [31] is slightly different from Equation (33b),

$$\sigma_e^2 = 0.18\ \delta^2 \tag{33c}$$

Since TMS320C30 has a 40/32-bit floating-point multiplier and ALU, the result from arithmetic operation has the mantissa of [31] bits plus one sign bit. Therefore, the δ in Equation (33c) is equal to 2^{-31}. Another roundoff noise is introduced when you restore the result back to memory. This noise has the power of 2^{-23} because the mantissa of TMS320C30 floating-point data is 23 bits plus one sign bit. Therefore, unless the filter order is high, the roundoff noise from arithmetic operation is relatively small.

The steady-state output error of the LMS algorithm due to the finite precision arithmetic of a digital processor was analyzed in reference [31]. It was found that the power of arithmetic errors is inversely proportional to the adaptation step size u. The significance of this result in the adaptive filter design is discussed next. Furthermore, roundoff noise is found to accumulate in time without bound, leading to an eventual overflow [32]. The leaky LMS algorithm presented in the previous section can be used to prevent the algorithm overflow.

Design Issues

The performance of digital adaptive algorithms differs from infinite precision adaptive algorithms. The finite precision LMS algorithm is given as

$$\underline{w}(n+1) = \underline{w}(n) + Q[u*e(n)*\underline{x}(n)] \tag{34}$$

where Q [.] denotes the operation of fixed point quantization. Whenever any correction term u*e(n)*x(n−i) in the update of the weight vector in Equation (34) is too small, the quantized value of that term is zero, and the corresponding weight $w_i(n)$ remains unchanged. The condition for the ith component of the vector w(n) not to be updated when the algorithm is implemented with the TMS320C25 is

$$| u\ e(n)\ x(n-i) | < \delta/2 \tag{35a}$$

where $\delta = 2^{-15}$. The condition for TMS320C30 is

$$| u\ e(n)\ x(n-i) | < 2^{exp} * \delta/2 \tag{35b}$$

where exp is the exponent of $w_i(n)$ and $\delta = 2^{-23}$.

Since the adaptive algorithms are designed to minimize the mean squared value of the error signal, e(n) decreases with time. If u is small enough, most of the time the weights are not updated. This early termination of the adaptation may not allow the weight values to converge to the optimum set, resulting in a mean square error larger than its minimum value. The conditions for the adaptation to converge completely [30] is $u > u_{min}$ where

$$u^2_{min} = \frac{\delta^2}{4\sigma_x^2 \epsilon_{min}} \tag{36a}$$

for the TMS320C25 and the TMS320C30

$$u^2_{min} = \frac{\delta^2 * 2^{exp}}{4\sigma_x^2 \epsilon_{min}} \tag{36b}$$

where σ_x^2 is the power of input signal x(n) and ϵ_{min} is the minimum mean squared error at steady state.

In the Leaky LMS Algorithm section, it was mentioned that the excess MSE given in Equation (14) is minimized by using small u. However, this may result in a large quantization error since the most significant term in the total output quantization error is [31]

$$\frac{N\sigma_e^2}{2\,a^2\,u} \tag{37}$$

The optimum step size u_0 reflects a compromise between these conflicting goals. The value of u_0 is shown to be too small to allow the adaptive algorithm to converge completely and also to give a slow convergence. In practice, $u > u_0$ is used for faster convergence. Hence, the excess MSE becomes larger, and the roundoff noise can typically be neglected when compared with the excess mean square error.

Finally, recall Equations (11) and (12). The step size u has an upper limit to guarantee the stability and convergence. Therefore, the adaptive algorithm requires

$$0 < u < \frac{1}{N\sigma_x^2} \tag{38}$$

On the other hand, the step size u also has a lower limit. The optimum u_0, which minimizes the sum of the excess MSE and roundoff noise, is smaller than u_{min}, i.e., too small to allow the adaptive weight to converge. For an algorithm implemented on the TMS320C25, the word-length of 16 bits is fixed, and the minimum step-size that can be used is given in Equation (36). The most important design issue is to find the best u to satisfy

$$u_{min} < u < \frac{1}{N\sigma_x^2} \tag{39}$$

Therefore, in order to make the condition in Equation (39) valid, the initial values of filter coefficients are better close to zero for the floating-point processor if the situation in unknown.

Software Development

The TMS320C25 and TMS320C30 combine the high performance and the special features needed in adaptive signal processing applications. The processors are supported by a full set of software and hardware development tools. The software development tools include an assembler, a linker, a simulator, and a C compiler. The most universal software development tool available is a macro assembler. However, the assembly language programming for DSP can be tedious and costly. For adaptive filter applications, an assembly language programmer must have knowledge of adaptive signal processing. The challenge lies in compressing a great deal of complex code into the fairly small space and most efficient code dictated by the real-time applications typical of adaptive signal processing.

Recently, C compilers for the processors were developed to make DSP programming easier, quicker, and less costly compared with the work associated with programming in assembly language. Due to the general characteristics of a compiler, the code it generates is not the most efficient. Since the program efficiency consideration is important for adaptive filter implementation, the code generated from the C compiler has to be modified before implementing. Thus, two alternative ways, besides writing an assembly program, to implement adaptive signal processing on DSP are presented. First is the automatic adaptive filter code generator [12], which can be found on Texas Instruments TMS320 Bulletin Board Service (BBS), and second are the adaptive filter function libraries that support assembly and C programming languages.

In this report, two adaptive filter libraries have been developed: one can be called from an assembly main program; the other can be called from the C main program. Note that, for the TMS320C25 only, certain data memory locations have been reserved for storing the necessary filter coefficients, previous delayed signal, etc. In other words, these data memories are used as global variables.

Assembly Function Libraries

The basic concept of creating an assembly subroutine for an adaptive filter is to modify in module the assembly programs discussed above. Then, the user can implement the adaptive filter by writing his own assembly main program that calls the subroutine.

TMS320C25 Assembly Subroutine

The TMS320C25 has an eight-level deep hardware stack. The CALL and CALA subroutine calls store the current contents of the program counter (PC) on the top of the stack. The RET (return from subroutine) instruction pops the top of the stack back to the PC. For computational convenience, the processor needs to be set as follows before calling the assembly callable subroutine.

1. PM status bits equal to 01.
2. SXM status bit set to 1.
3. The current DP (data memory page pointer) is 0.

The following example is the TMS320C25 assembly main routine, which performs an adaptive line enhancement by calling the LMS algorithm subroutine. The filter order is 64, delay is equal to one, and the convergence factor u is 0.01.

```
*       DEFINE AND REFER SYMBOLS
*
              .global  ORDER,U,ONE,D,Y,ERR,XN,WN,LMS
*
```

```
        DEFINE SAMPLING RATE, ORDER, AND MU
*
ORDER:  .equ    20
MU:     .equ    327             ; mu = 0.01 in Q15 format
PAGE0:  .equ    0
*
        DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:     .usect  "buffer",ORDER-1
XN:     .usect  "buffer",1
WN:     .usect  "coeffs",ORDER
*
*       RESERVE ADDRESSES FOR PARAMETERS
*
ONE:    .usect  "parameters",1
U:      .usect  "parameters",1
ERR:    .usect  "parameters",1
Y:      .usect  "parameters",1
D:      .usect  "parameters",1
ERRF:   .usect  "parameters",1
*
*       INITIALIZATION
*
START   LDPK    PAGE0           ; Set DP = 0
        SPM     1               ; Set PM equal to 1
        SSXM                    ; Set sign extension mode
        LRLK    AR7,X0          ; AR7 point to >300
        LACK    1               ; Initialize ONE = 1
        SACL    ONE
        LALK    MU              ; Initialize U = MU = 0.01
        SACL    U
*****************************************************************************
*       PERFORM THE PREDICTOR
*****************************************************************************
INPUT:  IN      D,PA2           ; Get the input
*
        CALL    LMS             ; Call subroutine
*
OUTPUT: OUT     Y,PA2           ; Output the signal
*
        LAC     D               ; Insert the newest sample
        LARP    AR7
        SACL    *
        B       INPUT
        .end
```

The symbols, such as ORDER, U, ONE, D, LMS, Y, and ERR, are defined and referred to for the purpose of modular programming. The uninitialized sections specified by the directive .usect can be placed in any location of memory according to the linker command file. Note that MACD instruction requires the sources of the operands on program memory and data memory separately, and CNFP instruction configures RAM block 0 as program memory. Therefore, the coeffs section has to be in data RAM block 0, and the buffer has to be in RAM block 1. Appendix H1 contains the adaptive transversal filter with LMS algorithm subroutine using the TMS320C25, and Appendix H2 contains an example of a linker command file.

TMS320C30 Assembly Subroutine

Instead of a hardware stack, TMS320C30 uses a software stack, which is more flexible and convenient for a high-level language compiler. The stack memory location is pointed to by the stack pointer SP. In order to maintain the proper program sequence, the programmer must make certain that no data is lost and that the stack pointer always points to proper location. The PUSH, PUSHF, POP, POPF, CALL, CALLcond, RETIcond, and RETScond instructions will change the value of the stack pointer; in addition, writing data into it and using the interrupt will also change that value. It is the programmer's responsibility to initialize the stack pointer in the beginning of the program. The same adaptive line enhancer example above using TMS320C30 is listed below. The adapfltr.int program that initializes the stack pointer and the data RAM is given in Appendix H3.

```
*
*       DEFINE GLOBAL VARIABLES AND CONSTANTS
*
              .copy   "adapfltr.int"
              .global LMS30,order,u,d,y,e
N             .set    20
mu            .set    0.01
*
*       INITIALIZE POINTERS AND ARRAYS
*
              .text
begin         .set    $
              LDI     N,BK              ; Set up circular buffer
              LDP     @xn_addr          ; Set data page
              LDI     @xn_addr,AR0      ; Set pointer for x[]
              LDI     @wn_addr,AR1      ; Set pointer for w[]
              LDF     0.0,R0            ; R0 = 0.0
              RPTS    N-1
              STF     R0,*AR0++(1)%     ; x[] = 0.
```

```
          ||STF    R0,*AR1++(1)%   ; w[] = 0.
            LDI    @in__addr,AR6   ; Set pointer for input ports
            LDI    @out__addr,AR7  ; Set pointer for output ports
*
*       PERFORM ADAPTIVE LINE ENHANCER
*
input:
            LDF    *AR6,R7         ; Input d(n)
          ||LDF    *+AR6(1),R6     ; Input x(n)
            STF    R7,@d           ; Insert d(n)
            STF    R6,*AR0         ; Insert x(n) to buffer
*
*       CALL ASSEMBLY SUBROUTINE
*
*           CALL   LMS30
*       OUTPUT y(n) AND e(n) SIGNALS
*
            LDF    @y,R6           ; Get y(n)
            BD     input           ; Delay branch
            LDF    @e,R7           ; Get e(n)
            STF    R6,*AR7         ; Send out y(n)
            STF    R7,*+AR7(1)     ; Send out e(n)
*
*       DEFINE CONSTANTS
*
n           .usect  "buffer",N
wn          .usect  "coeffs",N
in__addr    .usect  "vars",1
out__addr   .usect  "vars",1
xn__addr    .usect  "vars",1
wn__addr    .usect  "vars",1
u           .usect  "vars",1
order       .usect  "vars",1
d           .usect  "vars",1
y           .usect  "vars",1
e           .usect  "vars",1
cinit       .sect   ".cinit"
            .word   6,in__addr
            .word   0804000h
            .word   0804002h
            .word   xn
            .word   wn
```

```
.float   mu
.word    N-2
.end
```

In the above example, data memory order is initialized to N−2 for computation convenience. The linker command files and the subroutine that implements the LMS transversal filter can be found in Appendixes H4 and H5.

C Function Libraries

The TMS320C25 and TMS320C30 C language compilers provide high-level language support for these processors. The compilers allow application developers without an extensive knowledge of the device's architecture and instruction set to generate assembly code for the device. Also, since C programs are not device-specific, it is a relatively straightforward task to port existing C programs from other systems.

To allow fast development of efficient programs for adaptive signal processing applications, C function libraries have been developed. These libraries include functions for adaptive transversal, symmetric transversal, and lattice structures.

TMS320C25 C-Callable Subroutines

In a C program, the memory assignments are chosen by the compiler. There are two ways to use the most efficient instruction MACD:

A. Use inline assembly code to assign memory locations for filter coefficients and buffers.
B. Reserve the desired memory locations for them and do the assignment in the linker command file.

The latter method is used in this report.

For a C main program, the parameters passed to and returned from the subroutines are all within the parentheses following the subroutine name, as shown below:

```
lms(n,mu,d,x,&y,&e)     n - Filter order
                        mu - Convergence factor
                        d - Desired signal
                        x - Input signal
                        y - Address of output signal
                        e - Address of error signal
```

Since the TMS320C25 C compiler pushes the parameters from right to left into software stack pointed by AR1 , the subroutine gets the parameters in reverse order, as shown below:

```
MAR     *-          ; Set pointer for getting parameters
LAC     *-          ; ACC = N
```

```
SUBK   1
SACL   ORDER        ; ORDER = N - 1
LAC    *-           ; Getting and storing the mu
SACL   U
LAC    *-           ; Getting and storing the D
SACL   D
LAC    *-,0,A-R3    ; Insert the newest sample
LRLK   AR3,FRSTAP
SACL   *
```

The assembly subroutine returns the parameters y and e as follows:

```
LARP   AR1
LAR    AR2,*-,AR2   ; Get the address of y in main
LAC    Y
SACL   *,0,AR1      ; Store y
LAR    AR2,*,AR2    ; Get the address of e in main
LAC    ERR
SACL   *,0,AR1      ; Store e
```

Therefore, the parameters should be entered in the order given above. If there are other parameters, they should be inserted right after the convergence factor mu. The leaky LMS algorithm subroutine is given as an example.

llms(n,mu,r,d,x,&y,&e)

the r is defined in Equation (28a). Note that the values of the AR registers, which will be used in subroutine, and the status registers must be saved at the beginning of the subroutine and restored right before returning to calling routine. An example of a C-callable program is given in Appendix I1. Memory locations 0200h to 0200h+N−1 and 0300h to 0300h+N−1 are reserved for filter coefficients and buffers, respectively. N denotes the filter order.

TMS320C30 C Subroutine

As previously mentioned, the TMS320C30 architecture has features designed for a high-level language compiler. Note that the callable word is dropped in this section title because the TMS320C30 is so flexible that the restrictions for the TMS320C25 no longer exist. Since the memory locations of filter buffers and coefficients are determined by the parameters that pass from the calling routine, the same subroutine can be used in different places. However, the only restriction is that the memory locations of filter buffers must align to the circular addressing boundary [14]. The features of TMS320C30 architecture that make a major contribution toward these improvements are dual data address buses, software stack, and flexible addressing mode. The parameters passed to subroutine are pushed into the stack. Therefore, after returning from the subroutine, the stack pointer, SP, must be updated to point to the location where SP pointed before pushing the parameters

into the stack. However, this will be done by the C compiler. The usage example of the C function subroutine is given as follows:

```
tlms(n,u,d,&w,&x,&y,&e) where    n - Filter order
                                 u - Step size
                                 d - Desired signal
                                 &w - Filter coefficients
                                 &x - Input signal buffers
                                 &y - Addr of output signal
                                 &e - Addr of error signal
```

The example below shows how the C subroutine receives and manipulates the parameters passed from the caller program and how the result is returned to the caller routine.

```
*
*       SET FRAME POINTER FP
*
FP      .set    AR3
        PUSH    FP
        LDI     SP,FP
*
*       GET FILTER PARAMETERS
*
        LDI     *-FP(2),R4      ; Get filter order
        LDI     *-FP(6),AR0     ; Get pointer for x[]
        LDI     *--FP(5),AR1    ; Get pointer for w[]
*
*       COMPUTE ERROR SIGNAL e(n) AND STORE y(n) AND e(n)
*
        LDI     *-FP(2),AR2     ; Get y(n) address
        SUBF3   R2,*+FP(1),R7   ; e(n) = d(n) - y(n)
    ||STF       R2,*AR2         ; Send out y(n)
        LDI     *-FP(3),AR2     ; Get e(n) address
        STF     R7,*AR2         ; Send out e(n)
        MPYF    *+FP(2),R7      ; R7 = e(n) * u
        POP     FP
```

Note that AR3 is used as the frame pointer in TMS320C30 C compiler. Appendix I2 contains the complete LMS transversal filter example subroutine program.

Development Process and Environment

Following a four stage procedure [33] to minimize the amount of finite word length effect analysis and real-time debugging, adaptive structures and algorithms are implemented

on the TMS320C25. Figure 15 illustrates the flowchart of this procedure. Since the implementation on TMS320C30 is done only by the simulator, the last stage, real-time testing, is not implemented.

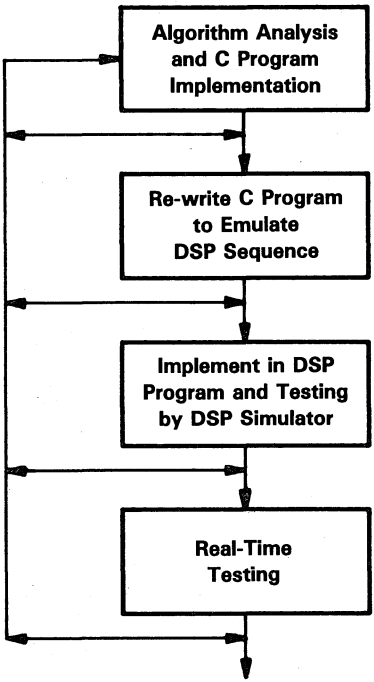


Figure 15. Adaptive Filter Implementation Procedure

In the first stage, algorithm design and study is performed on a personal computer. Once the algorithm is understood, the filter is implemented using a high-level C program with double precision coefficients and arithmetic. This filter is considered an ideal filter.

In the second stage, the C program is rewritten in a way that emulates the same sequence of operations with the same parameters and state variables that will be implemented in the processors. This program then serves as a detailed outline for the DSP assembly language program or can be compiled using TMS320C25 or TMS320C30 C compiler. The effects of numerical errors can be measured directly by means of the technique shown in Figure 16, where H(z) is the ideal filter implemented in the first stage and H'(z) is a real filter. Optimization is performed to minimize the quantization error and produce stable implementation.

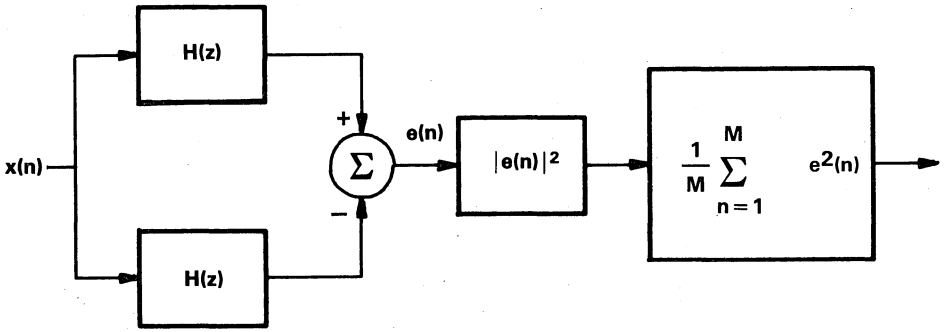


Figure 16. A Commutational Technique for Evaluating Quantization Effects

In the third stage, the TMS320C25 and TMS320C30 assembly programs are developed; then they are tested using the simulators with test data from a disk file. Note that the simulation of TMS320C25 can also be implemented on the SWDS with the data logging option. This test data is a short version of the data used in stage 2 that can be internally generated from a program or data digitized from a real application environment. Output from the simulation is compared against the equivalent output of the C program in the second stage. Since the simulation requires data files to be in Q15 format, certain precision is lost during data conversion. When a one-to-one agreement within tolerable range is obtained between these two outputs, the processor software is assured to be essentially correct.

The final stage is applied only to the TMS320C25. First, you download this assembled program into the target TMS320C25 system (SWDS) to initiate real-time operation. Thus, the real-time debugging process is constrained primarily to debugging the I/O timing structure of the algorithm and testing the long-term stability of the algorithm. Figure 17 shows an experimental setup for verification, in which the adaptive filter is configured for a one-step adaptive predictor illustrated in Figure 18. The data used for real-time testing is a sinusoid generated by a Tektronix FG504 Function Generator embedded in white noise generated by an HP Precision Noise Generator. The DSP gets a quantized signal from the Analog Interface Board (AIB), performs adaptive prediction routines, and outputs an enhanced sinusoid to the analog interface board. The corrupted input and predicted (enhanced) output waveforms are compared on the oscilloscope or on the HP 4361 Dynamic Signal Analyzer. The corresponding spectra of input and output can be compared on the signal analyzer. The signal-to-noise ratio (SNR) improvement can be measured from the analyzer, which is connected to an HP plotter.

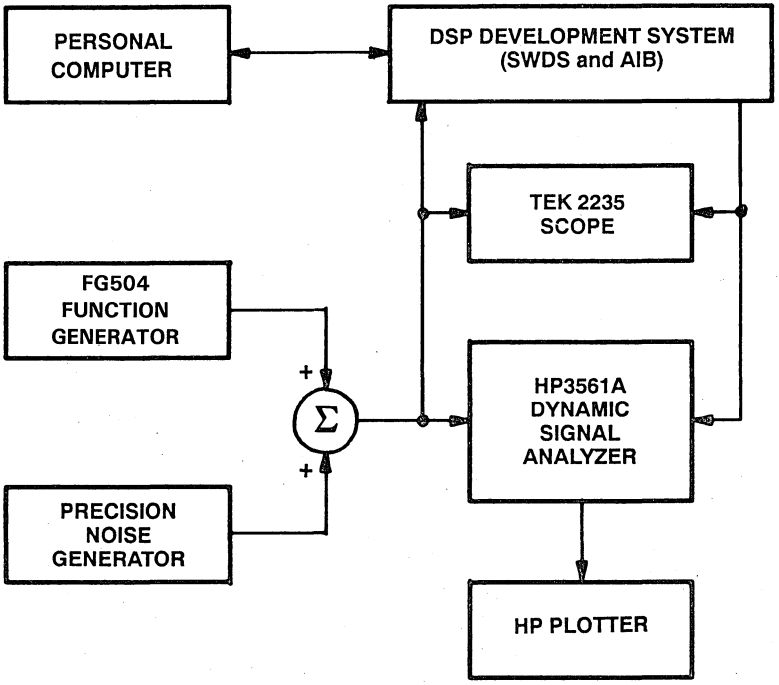


Figure 17. Real-Time Experiment Setup

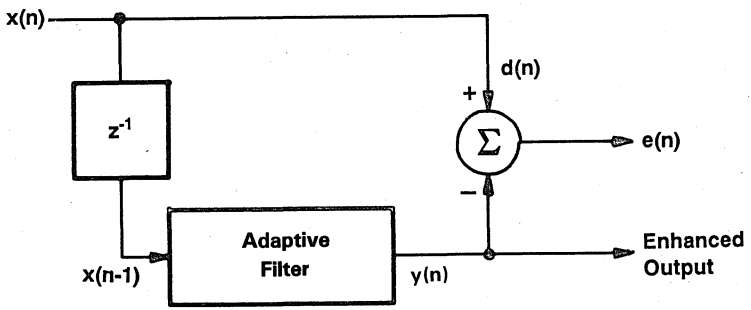


Figure 18. Block Diagram of a One-Step Adaptive Predictor

To illustrate the operation in a nonstationary environment, the adaptive predictor is implemented using a TMS320C25, and the following experiment is performed. The input signal is swept from 1287 Hz to 4025 Hz, then jumps back to 1287 Hz. The time for each sweep is one second. The input spectra at every second are shown in Figure 19a; the corresponding output spectra are shown in Figure 19b. From the observations on the

oscilloscope and signal analyzer, the significant SNR improvement, convergence speed, ability to track nonstationary signals, and long-term stability of the adaptive predictor are observed.

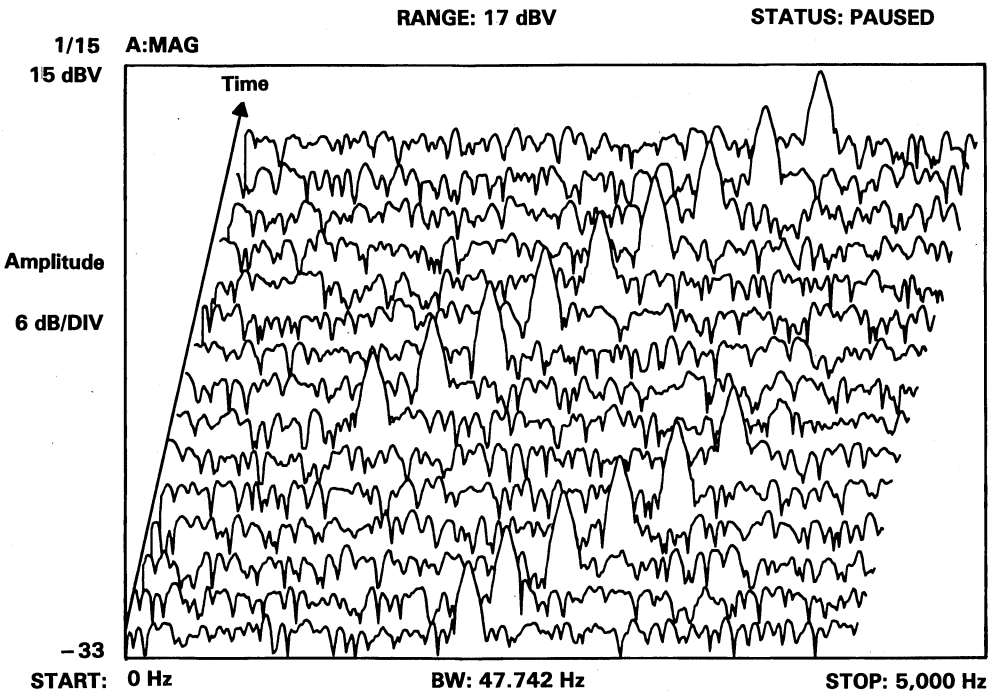


Figure 19(a). Spectrum of Input Signal

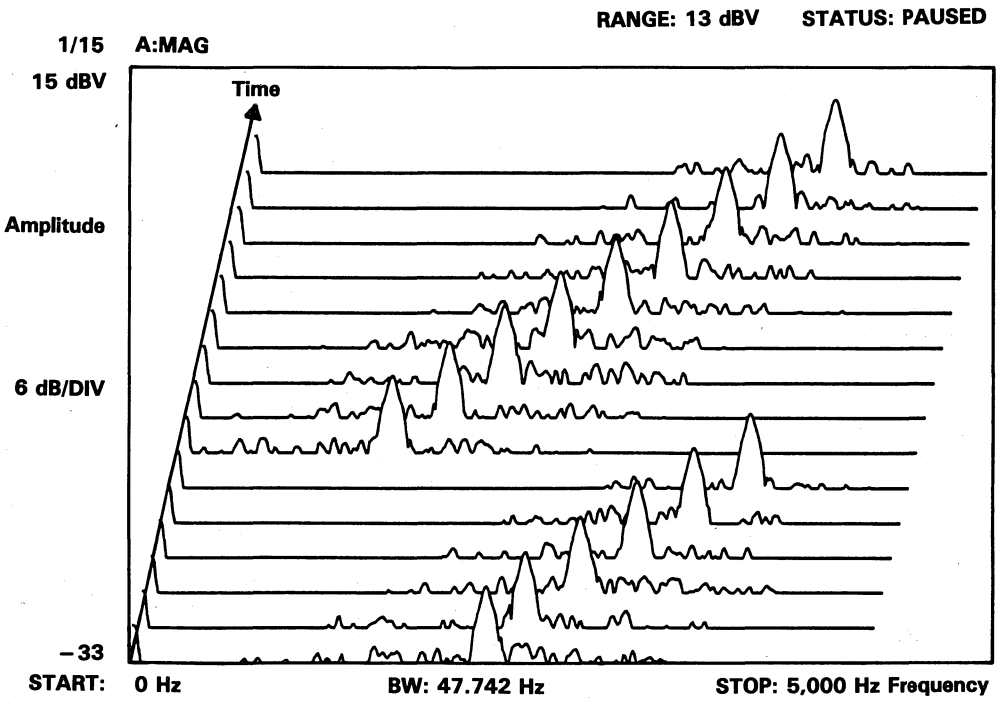


Figure 19(b). Spectrum of Enhanced Output Signal

Summary

Three adaptive structures and six update algorithms are implemented with the TMS320C25 and TMS320C30. Applications of adaptive filters and implementation considerations have been discussed. Two subroutine libraries that support both C language and assembly language for two processors were developed. These routines can be readily incorporated into TMS320C25 or TMS320C30 users' application programs.

The advancements in the TMS320C25 and TMS320C30 devices have made the implementation of sophisticated adaptive algorithms oriented toward performing real-time processing tasks feasible. Many adaptive signal processing algorithms are readily available and capable of solving real-time problems when implemented on the DSP. These programs provide an efficient way to implement the widely used structures and algorithms on the TMS320C25 and TMS320C30, based on assembly-language programming. They are also extremely useful for choosing an algorithm for a given application. The performances of adaptive structures and algorithms that have been implemented using the TMS320C25 and TMS320C30 have been summarized in Tables 1 and 2.

Table 1. The Performance of Adaptive Structures and Algorithms of TMS320C25

TMS320C25			
Transversal Structure	LMS	Instruction Cycles	7N + 28
		Program Memory (Word)	33
	Leaky LMS	Instruction Cycles	8N + 28
		Program Memory (Word)	34
	Sign-Data LMS	Instruction Cycles	11N + 26
		Program Memory (Word)	41
	Sign-Error LMS	Instruction Cycles	7N + 26
		Program Memory (Word)	30
	Sign-Sign LMS	Instruction Cycles	11N + 21
		Program Memory (Word)	30
	Normalized LMS	Instruction Cycles	7N + 57
		Program Memory (Word)	47
Symmetric Transversal Structure	LMS	Instruction Cycles	7.5N + 38
		Program Memory (Word)	50
	Leaky LMS	Instruction Cycles	8N + 38
		Program Memory (Word)	51
	Sign-Data LMS	Instruction Cycles	9.5N + 36
		Program Memory (Word)	58
	Sign-Error LMS	Instruction Cycles	7.5N + 36
		Program Memory (Word)	47
	Sign-Sign LMS	Instruction Cycles	9.5N + 31
		Program Memory (Word)	47
	Normalized LMS	Instruction Cycles	7.5N + 69
		Program Memory (Word)	66
Lattice Structure	LMS	Instruction Cycles	33N + 32
		Program Memory (Word)	63
	Leaky LMS	Instruction Cycles	35N + 32
		Program Memory (Word)	65
	Sign-Error LMS	Instruction Cycles	36N + 32
		Program Memory (Word)	65
	Normalized LMS	Instruction Cycles	90N + 34
		Program Memory (Word)	92

Note: N represents filter order.

Table 2. The Performance of Adaptive Structures and Algorithms of TMS320C30

TMS320C30			
Transversal Structure	LMS	Instruction Cycles	3N + 15
		Program Memory (Word)	17
	Leaky LMS	Instruction Cycles	3N + 15
		Program Memory (Word)	19
	Sign-Data LMS	Instruction Cycles	5N + 16
		Program Memory (Word)	24
	Sign-Error LMS	Instruction Cycles	3N + 16
		Program Memory (Word)	18
	Sign-Sign LMS	Instruction Cycles	5N + 16
		Program Memory (Word)	24
	Normalized LMS	Instruction Cycles	3N + 47
		Program Memory (Word)	49
Symmetric Transversal Structure	LMS	Instruction Cycles	2.5N + 15
		Program Memory (Word)	23
	Leaky LMS	Instruction Cycles	2.5N + 19
		Program Memory (Word)	26
	Sign-Data LMS	Instruction Cycles	3.5N + 18
		Program Memory (Word)	30
	Sign-Error LMS	Instruction Cycles	2.5N + 18
		Program Memory (Word)	24
	Sign-Sign LMS	Instruction Cycles	3.5N + 17
		Program Memory (Word)	30
	Normalized LMS	Instruction Cycles	2.5N + 50
		Program Memory (Word)	56
Lattice Structure	LMS	Instruction Cycles	14N + 9
		Program Memory (Word)	20
	Leaky LMS	Instruction Cycles	16N + 9
		Program Memory (Word)	22
	Sign-Error LMS	Instruction Cycles	16N + 9
		Program Memory (Word)	22
	Normalized LMS	Instruction Cycles	67N + 9
		Program Memory (Word)	73

Note: N represents filter order.

References

[1] B. Widrow and S. Stearns, *Adaptive Signal Processing*, Prentice-Hall, 1985.

[2] R. Lucky, J. Salz, and E. Weldon, *Principles of Data Communications*, McGraw-Hill, 1968.

[3] S. Haykin, *Adaptive Filter Theory*, Prentice-Hall, 1986.

[4] M. Honig and D. Messerschmit, *Adaptive Filters: Structures, Algorithms, and Applications*, Kluwer Academic, 1984.

[5] J.R. Treichler, C.R. Johnson, and M.G. Larimore, *Theory and Design of Adaptive Filters*, Wiley, 1987.

[6] T. Alexander, *Adaptive Signal Processing*, Springer-Verlag, 1986.

[7] G. Goodwin and K. Sin, *Adaptive Filtering Prediction and Control*, Prentice-Hall, 1984.

[8] M. Bellanger, *Adaptive Digital Filters and Signal Analysis*, Marcel Dekker, 1987.

[9] J. Proakis, *Digital Communications*, McGraw-Hill, 1983.

[10] C. Chen and S. Kuo, "An Interactive Software Package for Adaptive Signal Processing on an IBM Person Computer," *19th Pittsburgh Conference on Modeling and Simulation*, May 1988.

[11] S. Kuo, G. Ranganathan, P. Gupta, and C. Chen, "Design and Implementation of Adaptive Filters," *IEEE 1988 International Conference on Circuits and Systems*, June 1988.

[12] S. Kuo, G. Ma, and C. Chen, "An Advanced DSP Code Generator for Adaptive Filters," *1988 ASSP DSP workshop*, Sept. 1988.

[13] Texas Instruments, *Second-Generation TMS320 User's Guide*, 1987.

[14] Texas Instruments, *Third-Generation TMS320 User's Guide*, 1988.

[15] S. Qureshi, "Adaptive Equalization," Invited Paper, *Proceedings of the IEEE*, Sept. 1985.

[16] L. Rabiner and R. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[17] N. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice-Hall, 1984.

[18] J. Makhoul, "Linear Prediction: A Tutorial Review," *Proceedings of the IEEE*, April 1975.

[19] C. Cowan and P. Grant, *Adaptive Filters*, Prentice-Hall, 1985.

[20] C. Gritton and D. Lin, "Echo Cancellation Algorithms," *IEEE ASSP Magazine*, April 1984.

[21] D. Messerschmitt, et al, "Digital Voice Echo Canceller with a TMS32020," in *Digital Signal Processing Applications with the TMS320 Family*, Prentice-Hall, 1986.

[22] B. Widrow, et al, "Adaptive Noise Cancelling: Principles and Applications," *Proceedings of the IEEE*, December 1975.

[23] A. Lovrich and R. Simar, "Implementation of FIR/IIR Filter with the TMS32010/TMS32020," in *Digital Signal Processing Applications with the TMS320 Family*, Texas Instruments, 1986.

[24] S. Orfanidis, *Optimum Signal Processing*, MacMillan, 1985.
[25] G. Frantz, K. Lin, J. Reimer, and J. Bradley, "The Texas Instruments TMS320C25 Digital Signal Microcomputer," *IEEE Micro*, December 1986.
[26] B. Friedlander, "Lattice Filters for Adaptive Processing," *Proceedings of the IEEE*, August 1982.
[27] A. Gersho, "Adaptive Filtering with Binary Reinforcement," *IEEE Transactions on Information Theory*, March 1984.
[28] A. Oppenheim and R. Schafer, *Digital Signal Processing*, Chap. 9, Prentice-Hall, 1975.
[29] L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Chap. 5, Prentice-Hall, 1975.
[30] J. R. Gitlin et al, "On the Design of Gradient Algorithms for Digitally Implemented Adaptive Filters," *IEEE Transactions on Circuit Theory*, March 1973.
[31] C. Caraiscos and B. Liu, "A Roundoff Error Analysis of the LMS Adaptive Algorithm," *IEEE Transactions on ASSP*, February, 1984.
[32] J. Cioffi, "Limited-Precision Effects in Adaptive Filtering," *IEEE Transactions on Circuits and Systems*, July 1987.
[33] R. Crochier, R. Cox, and J. Johnson, "Real-Time Speech Coding," *IEEE Transactions on Communications*, April 1982.

List of Appendices for Implementation of Adaptive Filters with the TMS320C25 and TMS320C30

Appendix A1. Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title   'TLMS'
*****************************************************************
*
*   TLMS :  Adaptive Filter Using Transversal Structure
*           and LMS Algorithm, Looped Code
*
*   d(n) ---------------------------¦
*                                   ¦
*                                   ¦+
*                                  (SUM)------> e(n)
*                                   ¦-
*                ------             ¦
*   x(n) ---------¦ AF ¦-----------¦--------> y(n)
*                ------
*
*   Algorithm:
*
*             63
*      y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*             k=0
*
*      e(n) = d(n) - y(n)
*
*      w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,..63
*
*      Where we use filter order = 64 and mu = 0.01.
*
*   Note:  This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*   Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory ONE should be 1.
*         5) Data memory U should be 327.
*
*                 Chen, Chein-Chung  February, 1989
*
*****************************************************************
*
*   DEFINE PARAMETERS
*
ORDER:    .equ     64
PAGE0:    .equ     0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
XO:       .usect   "buffer",ORDER-1
XN:       .usect   "buffer",1
WN:       .usect   "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:        .usect   "parameters",1
Y:        .usect   "parameters",1
ERR:      .usect   "parameters",1
ONE:      .usect   "parameters",1
U:        .usect   "parameters",1
ERRF:     .usect   "parameters",1
*********************************
*   PERFORM THE ADAPTIVE FILTER
*********************************
          .text
*
*   ESTIMATE THE SIGNAL Y
*
          LARP     AR3
          CNFP                        ; Configure B0 as program memory
          MPYK     0                  ; Clear the P register
          LAC      ONE,15             ; Using rounding
          LRLK     AR3,XN             ; point to the oldest sample
FIR       RPTK     ORDER-1            ; Repeat N times
          MACD     WN+0fd00h,*-       ; Estimate Y(n)
          CNFD                        ; Configure B0 as data memory
          APAC
          SACH     Y                  ; Store the filter output
*
*   COMPUTE THE ERROR
*
          NEG                         ; ACC = - Y(n)
          ADDH     D
          SACH     ERR                ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
          LT       ERR                ; T = ERR(n)
          MPY      U                  ; P = U * ERR(n)
          PAC
          ADD      ONE,15             ; Round the result
          SACH     ERRF               ; ERRF = U * ERR(n)
*
          LARK     AR1,ORDER-1        ; Set up counter
          LRLK     AR2,WN             ; Point to the coefficients
          LRLK     AR3,XN+1           ; Point to the data sample
          LT       ERRF               ; T register = U * ERR(n)
          MPY      *-,AR2             ; P = U * ERR(n) * X(n-k)
ADAPT     ZALR     *,AR3              ; Load ACCH with A(k,n) & round
          MPYA     *-,AR2             ; W(k,n+1) = W(k,n) + P
*                                     ; P = U * ERR(n) * X(n-k)
          SACH     *+,0,AR1           ; Store W(k,n+1)
          BANZ     ADAPT,*-,AR2
*
FINISH    .end
```

Appendix A2. Transversal Structure with LMS Algorithm Using the TMS320C30

```
*************************************************************
*
*   T30 - Adaptive transversal filter with LMS algorithm
*         using the TMS320C30
*
*   I/O configuration:
*
*    d(n) --------------------------¦
*                                   ¦
*                                   ¦+
*                                  (SUM)------> e(n)
*                                   ¦-
*                  -----            ¦
*    x(n) ---------¦ AF ¦-----------¦--------> y(n)
*                  -----
*
*    Algorithm:
*
*              63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,..63
*
*       Where we use filter order = 64 and mu = 0.01.
*
*                  Chen, Chein-Chung  March, 1989
*
*************************************************************
            .copy     "adapfltr.int"
*******************************************************
*    PERFORM ADAPTIVE FILTER
*******************************************************
order       .set      64
mu          .set      01
*
*    INITIALIZE POINTERS AND ARRAYS
*
            .text
begin       .set      $
            LDI       order,BK          ; Set up circular buffer
            LDP       @xn_addr          ; Set data page
            LDI       @xn_addr,AR0      ; Set pointer for x[]
            LDI       @wn_addr,AR1      ; Set pointer for w[]
            LDF       0.0,R0            ; R0 = 0.0
            RPTS      order-1
            STF       R0,*AR0++(1)%     ; x[] = 0
         ;| STF       R0,*AR1++(1)%     ; w[] = 0
            LDI       @in_addr,AR6      ; Set pointer for input ports
            LDI       @out_addr,AR7     ; Set pointer for output ports

input:
            LDF       *AR6,R7           ; Input d(n)
         ;| LDF       *+AR6(1),R6       ; Input x(n)
            STF       R6,*AR0           ; Insert x(n) to buffer
*
*    COMPUTE FILTER OUTPUT y(n)
*
            LDF       0.0,R2            ; R2 = 0.0
            MPYF3     *AR0++(1)%,*AR1++(1)%,R1
            RPTS      order-2
            MPYF3     *AR0++(1)%,*AR1++(1)%,R1
         ;| ADDF3     R1,R2,R2          ; y(n) = w[].x[]
            ADDF      R1,R2             ; Include last result
*
*    COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) AND e(n) SIGNALS
*
            SUBF      R2,R7             ; e(n) = d(n) - y(n)
            STF       R2,*AR7           ; Send out y(n)
         ;| STF       R7,*+AR7(1)       ; Send out e(n)
*
*    UPDATE WEIGHTS w(n)
*
            MPYF      @u,R7             ; R7 = e(n) * u
            MPYF3     *AR0++(1)%,R7,R1  ; R1 = e(n) * u * x(n)
            LDI       order-3,RC        ; Initialize repeat counter
            RPTB      LMS               ; Do i = 0, N-3
            MPYF3     *AR0++(1)%,R7,R1  ; R1 = e(n) * u * x(n-i-1)
         ;| ADDF3     *AR1,R1,R2        ; R2 = wi(n) + e(n) * u * x(n-i)
LMS         STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
            MPYF3     *AR0,R7,R1        ; For i = N - 2
         ;| ADDF3     *AR1,R1,R2
            BD        input             ; Delay branch
            STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
            ADDF3     *AR1,R1,R2
            STF       R2,*AR1++(1)%     ; Update last w
*
*    DEFINE CONSTANTS
*
xn          .usect    "buffer",order
wn          .usect    "coeffs",order
in_addr     .usect    "vars",1
out_addr    .usect    "vars",1
xn_addr     .usect    "vars",1
wn_addr     .usect    "vars",1
u           .usect    "vars",1
cinit       .sect     ".cinit"
            .word     5,in_addr
            .word     0804000h
            .word     0804002h
            .word     xn
            .word     wn
            .float    mu
            .end
```

Appendix B1. Symmetric Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title    'Y25'
*****************************************************************
*
*   Y25 :  Adaptive Filter Using Symmetry Transversal Structure
*          and LMS Algorithm, Looped Code
*
*   d(n) ------------------------------------------+
*                                                  |
*                                ---------  y(n)    +  |+
*                               |  A. F. |---------->(SUM)---> e(n)
*                               ---------
*                     z1(n)     | | | |    z1(n-k)
*                   |-----------| | | |-------------|
*                   |           |---| |-----|       |
*                   |           |           |       |
*                   |   |---|   |   |---|   |   |---|   |
*    x(n) ----------|---| Z |---|---| Z |...|---| Z |---|-------|
*                   | | |---|   | | |---|   | | |---|   | |   |
*                   |/          |/          |/          |/  |---|
*                 (SUM)       (SUM)       (SUM)       (SUM) | Z |
*                   |           |           |           |   |---|
*                    \  |---|   \  |---|    \  |---|   \    |
*                     |-| Z |---|-| Z |...|--|-| Z |---|----|
*                       |---|     |---|       |---|
*
*   Algorithm:
*
*     z1(n-k) = x(n-k) + x(n-63+k) k=0,1,...,31
*
*               31
*     y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,31
*              k=0
*
*     e(n) = d(n) - y(n)
*
*     w(k) = w(k) + u*e(n)*z1(n-k) k=0,1,2,..31
*
*     Where we use filter order = 64 and mu = 0.01.
*
*   Note:  This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*   Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to logic 1.
*          3) The current DP (data memory page pointer should be page 0.
*          4) Data memory ONE should be 1.
*          5) Data memory U should be 327.
*
*                  Chen, Chein-Chung  February, 1989
*****************************************************************
*
*   DEFINE PARAMETERS
*
ORDER:    .equ     64
ORDER2:   .equ     32
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
FRSBUF:   .usect   "buffer",ORDER2-1
LASBUF:   .usect   "buffer",1
WN:       .usect   "coeffs",ORDER2
FRSDAT:   .usect   "coeffs",ORDER-1
LASDAT:   .usect   "coeffs",1
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:        .usect   "parameters",1
Y:        .usect   "parameters",1
ERR:      .usect   "parameters",1
ONE:      .usect   "parameters",1
U:        .usect   "parameters",1
ERRF:     .usect   "parameters",1
*********************************
*   PERFORM THE ADAPTIVE FILTER
*********************************
          .text
*
*   SYMMETRIC BUFFER ADDITION
*
          LARP     AR3
          LARK     AR1,ORDER2-1      ; Set up the counter
          LRLK     AR2,LASDAT        ; Point to oldest data
          LRLK     AR3,FRSDAT        ; Point to newest data
          LRLK     AR4,FRSBUF        ; Point to first buffer
SYM       LAC      *+,0,AR2
          ADD      *-,0,AR4
          SACL     *+,0,AR1          ; Buffer(k) = DAT(n+k) + DAT(n-N+k)
          BANZ     SYM,*-,AR3
*
*   ESTIMATE THE SIGNAL Y
*
          CNFP                       ; Configure B0 as program memory
          MPYK     0                 ; Clear the P register
          LAC      ONE,15            ; Using rounding
          LRLK     AR3,LASBUF        ; Point to the oldest buffer
FIR       RPTK     ORDER2-1          ; Repeat N/2 time
          MACD     WN+0fd00h,*-      ; Estimate Y(n)
          CNFD                       ; Configure B0 as data memory
          APAC
          SACH     Y                 ; Store the filter output
*
*   COMPUTE THE ERROR
```

```
          NEG                       ; ACC = - Y(n)
          ADD       D,15
          SACH      ERR             ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
          LT        ERR             ; T = ERR(n)
          MPY       U               ; P = U * ERR(n)
          PAC
          ADD       ONE,15          ; Round the result
          SACH      ERRF            ; ERRF = U * ERR(n)
*
          LARK      AR1,ORDER2-1    ; Set up counter
          LRLK      AR2,WN          ; Point to the coefficients
          LRLK      AR3,LASBUF      ; Point to the last buffer
          LT        ERRF            ; T register = U * ERR(n)
          MPY       *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT     ZALR      *,AR3           ; Load ACCH with A(k,n) & round
          MPYA      *-,AR2          ; W(k,n+1) = W(k,n) + P
*                                   ; P = U * ERR(n) * X(n-k)
          SACH      *+,0,AR1        ; Store W(k,n+1)
          BANZ      ADAPT,*-,AR2
*
*   UPDATE DATA POSTION FOR NEXT ITERATION
*
FINISH    LRLK      AR2,LASDAT-1    ; Set pointer
DATMOV    RPTK      ORDER-2         ; Repeat N-1 times
          DMOV      *-              ; Shift data for next iteration
*
          .end
```

Appendix B2. Symmetric Transversal Structure with LMS Algorithm Using the TMS320C30

```
*************************************************************
*
*   Y30 - Adaptive symmetric transversal filter with
*         LMS algorithm using the TMS320C30
*
*    Algorithm:
*       z(n-k) = x(n-k-1) + x(n-63+k) k=0,1,...,31
*                 31
*       y(n) = SUM w(k)*z(n-k) k=0,1,2,...,31
*                k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*z(n-k) k=0,1,2,..31
*
*       Where we use filter order = 64 and mu = 0.01
*
*************************************************
*     PERFORM ADAPTIVE FILTER
*************************************************
            .copy     "adapfltr.int"
order       .set      64              ; Filter order
mu          .set      0.01            ; Step size
*
*   INITIALIZE POINTERS AND ARRAYS
            .text
begin       .set      $
            LDI       order,BK        ; Set up circular buffer
            LDP       @xn_addr        ; Set data page
            LDI       @xn_addr,AR0    ; Set pointer for x[]
            LDI       @wn_addr,AR1    ; Set pointer for w[]
            LDI       @zn_addr,AR2    ; Set pointer for z[]
            LDI       order/2-1,IR0   ; Set index pointer
            LDF       0.0,R0          ; S0 = 0.0
            RPTS      order-1
            STF       R0,*AR0++(1)%   ; x[] = 0
            RPTS      order/2-2
            STF       R0,*AR1++(1)    ; w[] = 0
         || STF       R0,*AR2++(1)    ; z[] = 0
            STF       R0,*AR1--(IR0)  ; w[] = 0
         || STF       R0,*AR2--(IR0)  ; z[] = 0
            LDI       @in_addr,AR6    ; Set pointer for input ports
            LDI       @out_addr,AR7   ; Set pointer for output ports
input:
            LDF       *AR6,R7         ; Input d(n)
         || LDF       *+AR6(1),R6     ; Input x(n)
            LDI       AR0,AR4         ; Set forward pointer for x[]
            STF       R6,*AR0--(1)%   ; Insert x(n) to buffer
*
*   COMPUTE FILTER OUTPUT y(n)
*
            LDF       0.0,R2          ; R2 = 0.0
            LDI       AR0,AR5         ; Set backward pointer for x[]
            LDI       order/2-2,RC
            RPTB      INNER
*
            ADDF3     *AR4++(1)%,*AR5--(1)%,R1
*                                     ; z(n) = x[n-i] + x[n+N-i]
*
            MPYF3     R1,*AR1++(1),R3 ; y[] = w[].z[]
         || STF       R1,*AR2++(1)    ; Store z(n)
INNER       ADDF3     R3,R2,R2        ; Accumulate the result
*
            ADDF3     *AR4++(1)%,*AR5--(1)%,R1
*                                     ; z(n) = x[n-i] + x[n+N-i]
*
            MPYF3     R1,*AR1--(IR0),R3 ; y[] = w[].z[]
         || STF       R1,*AR2--(IR0)  ; Store z(n)
            ADDF      R3,R2           ; Include last result
*
*   COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) and e(n) SIGNALS
*
            SUBF      R2,R7           ; e(n) = d(n) - y(n)
            STF       R2,*AR7         ; Send out y(n)
         || STF       R7,*+AR7(1)     ; Send out e(n)
*
*   UPDATE WEIGHTS w(n)
*
            MPYF      @u,R7           ; R7 = e(n) * u
            MPYF3     *AR2++(1),R7,R1 ; R1 = e(n) * u * z(n)
            LDI       order/2-3,RC    ; Initialize repeat counter
            RPTB      LMS             ; Do i = 0, N-3
            MPYF3     *AR2++(1),R7,R1 ; R1 = e(n) * u * z(n-i-1)
         || ADDF3     *AR1,R1,R2      ; R2 = wi(n) + e(n) * u * z(n-i)
LMS         STF       R2,*AR1++(1)    ; wi(n+1) = wi(n) + e(n) * u * z(n-i)
            MPYF3     *AR2--(IR0),R7,R1 ; For i = N - 2
         || ADDF3     *AR1,R1,R2
            BD        input           ; Delay branch
            STF       R2,*AR1++(1)    ; wi(n+1) = wi(n) + e(n) * u * z(n-i)
            ADDF3     *AR1,R1,R2      ; Include last w
            STF       R2,*AR1--(IR0)  ; Update last w
*
*   DEFINE CONSTANTS
*
xn          .usect    "buffer",order
wn          .usect    "coeffs",order/2
zn          .usect    "coeffs",order/2
in_addr     .usect    "vars",1
out_addr    .usect    "vars",1
xn_addr     .usect    "vars",1
wn_addr     .usect    "vars",1
zn_addr     .usect    "vars",1
u           .usect    "vars",1
cinit       .sect     ".cinit"
            .word     6,in_addr
```

```
.word   0804000h
.word   0804002h
.word   xn
.word   wn
.word   zn
.float  mu
.end
```

Appendix C1. Lattice Structure with LMS Algorithm Using the TMS320C25

```
        .title   'L25'
*****************************************************************
*
*   L25 :  Adaptive Filter Using Lattice Structure
*          and LMS Algorithm, Looped Code
*
*     f0(n)    +       f1(n)    fi-1(n)       +
*       ----¦--)(SUM)-------)....----------¦--)(SUM)-----)fi(n)
*       ¦   ¦    ¦-                        ¦    ¦-
*       ¦   ¦-----¦--¦                     ¦----¦--¦
*       ¦  *k0    ¦  ¦                   *ki-1   ¦  ¦
*  x(n)--¦         ¦  ¦                           ¦  ¦
*       ¦    *k0  ¦  ¦                     *ki-1 ¦  ¦
*       ¦    ¦---¦  ¦                           ¦---¦  ¦
*       ¦ ¦-¦ ¦    -¦                         ¦-¦ ¦    -¦
*       --¦Z¦-¦--)(SUM)-----)....--------¦Z¦-¦--)(SUM)---)bi(n)
*    b0(n)¦-¦    +     b1(n)    bi-1(n) ¦-¦     +
*
*   Algorithm:
*
*      fi(n) = fi-1(n) - Ki(n) * bi-1(n-1)  i=1,2,...,64
*
*      bi(n) = bi-1(n-1) - Ki(n) * fi-1(n)  i=1,2,...,64
*
*                    i-1
*      ei(n) = d(n) - SUM yk(n) = ei-1 - bi-1(n)*Gi-1(n)  i=1,2,...,64
*                    k=0
*
*              64          64
*      y(n) = SUM yi(n) = SUM bi(n)*Gi(n)
*             i=0         i=0
*
*      Ki(n+1) = Ki(n) + mu * [ fi(n)*bi-1(n-1) + bi(n)*fi-1(n) ]
*
*      Gi(n+1) = Gi(n) + mu * ei(n) * bi(n)  i=1,2,..64
*
*      Where filter order = 64 and mu = 0.01.
*
*   Note! This source program is the generic version; I/O configuration has
*         not been set up. User has to modify the main routine for specific
*         application.
*
*   Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to logic 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory U should be 327.
*         5) The B1 & BD1 pointer (AR3 & AR4) should be exchanged every
*            iteration. For example,
*               For odd iteration:  AR3 --> B1
*                                   AR4 --> BD1
*               For even iteration: AR3 --> BD1
*                                   AR4 --> B1
*
*                  Chen, Chein-Chung  February, 1989
*****************************************************************
*
*   DEFINE PARAMETERS
*
ORDER:      .equ     64
*
*   DEFINE ADDRESSES OF BUFFERS AND COEFFICIENTS
*
G1:         .usect   "coeffs",ORDER
K1:         .usect   "coeffs",ORDER
F1:         .usect   "coeffs",ORDER+1
B1:         .usect   "buffer",ORDER+1
BD1:        .usect   "buffer",ORDER+1
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect   "parameters",1
X:          .usect   "parameters",1
Y:          .usect   "parameters",1
E:          .usect   "parameters",1
U:          .usect   "parameters",1
TEMP:       .usect   "parameters",1
*********************************
*   PERFORM THE ADAPTIVE FILTER
*********************************
            .text
*
*   INITIALIZE THE POINTERS
*
            LARP     AR3
            LARK     AR1,ORDER-1
            LRLK     AR2,F1
            LRLK     AR3,B1
            LRLK     AR4,BD1
            LRLK     AR5,G1
            LRLK     AR6,K1
*
*   INITIALIZE THE B1 AND F1
*
            LAC      X
            SACL     *,0,AR2
            SACL     *,0,AR3
*
*   INITIALIZATION
*
            LT       *,AR5          ; T = B1
            MPY      *,AR2          ; P = B1 * G1
            PAC                     ; ACC = B1 * G1
            SACH     Y              ; Initialize Y(0) = B1 * G1
            NEG                     ; ACC = -(B1 * G1)
            ADDH     D              ; ACC = D(n) - B1 * G1
            SACH     E              ; Initialize E(0) = D(n) - B1 * G1
*
```

```
*  COMPUTE THE F(i) AND B(i)
*
LAT1    ZALR    *,AR6           ; ACC = Fi-1
        LT      *,AR4           ; T = Ki
        MPY     *,AR2           ; P = Ki * Bi-1
        MPYS    *+              ; ACC = Fi-1 - Ki*Bi-1, P = Ki*Fi-1
        SACH    *-,0,AR4        ; Store Fi
        ZALR    *,AR3           ; ACC = Bi-1
        MPYS    *+              ; ACC = Bi-1 - Ki * Fi-1
        SACH    *-              ; Store Bi
*
*  COMPUTE GAIN G(n)
*
        LT      U               ; T = MU
        MPY     E               ; P = MU * Ei-1
        SPH     TEMP            ; Store MU * Ei-1
        LT      TEMP            ; T = MU * Ei-1
        MPY     *+,AR5          ; P = MU * Ei-1 * Bi-1
        ZALR    *,AR3           ; ACC = Gi(n)
        ITA     *,AR3           ; ACC = Gi(n) + MU*Ei-1*Bi-1, T = Bi
NEXT1   SACH    *+,0,AR5        ; Store Gi(n+1)
*
*  COMPUTE E AND UPDATE Ki
*
        MPY     *,AR2           ; P = Bi * Gi
        ZALR    E               ; ACC = Ei-1
        MPYS    *+,AR2          ; ACC = Ei-1 - Bi * Gi, P = Bi*Fi-1
        SACH    E               ; Store Ei
        LTP     *,AR4           ; T = Fi, ACC = Bi*Fi-1
        MPY     *+,AR6          ; P = Fi * Bi-1
        APAC                    ; ACC = Bi*Fi-1 + Bi-1 * Fi
        SACH    TEMP
        LT      TEMP            ; T = Bi*Fi-1 + Bi-1 * Fi
        MPY     U               ; P = MU * (Fi*Bi-1+Fi-1*Bi)
        ZALR    *               ; ACC = Ki(n)
        APAC                    ; ACC = Ki(n) + MU
*                               ; * (Fi*Bi-1+Fi-1*Bi)
        SACH    *+,0,AR1        ; Store Ki(n+1)
        BANZ    LAT1,*-,AR2
*
*  COMPUTE Y
*
        CNFP                    ; Configure B0 as program memory
        MPYK    0               ; Clear the P register
        DAC                     ; Clear accumulator
        LALK    AR2,B1          ; Set the pointer
FIR     MPYK    ORDER-1         ; Repeat N times
        RPTK    Gi+0*800h,**    ; Complete Y
        CNFD                    ; Configure B0 as data memory
        APAC                    ; Include last data
        SACH    Y               ; Store the filter output
*
        .end
```

Appendix C2. Lattice Structure with LMS Algorithm Using the TMS320C30

```
*************************************************************
*    L30 :  Adaptive Lattice Structure Filter with LMS Algorithm
*           using the TMS320C30
*
*    Algorithm:
*
*       fi(n) = fi-1(n) - Ki(n) * bi-1(n-1)  i=1,2,...,64
*
*       bi(n) = bi-1(n-1) - Ki(n) * fi-1(n)  i=1,2,...,64
*
*       ei(n) = d(n) - SUM yk(n) = ei-1 - bi-1(n)*Gi-1(n)  i=1,2,...,64
*                      k=0
*                 64          64
*       y(n) = SUM yi(n) = SUM bi(n)*Gi(n)
*                i=1         i=1
*
*       Ki(n+1) = Ki(n) + mu * [ fi(n)*bi-1(n-1) + bi(n)*fi-1(n) ]
*
*       Gi(n+1) = Gi(n) + mu * ei(n) * bi(n)  i=1,2,..64
*
*       Where filter order = 64 and mu = 0.04.
*
*                    Chen, Chein-Chung  March, 1989
*
*************************************************************
            .copy       adapfltr.int"
*********************************************
*     PERFORM ADAPTIVE FILTER
*********************************************
order       .set      64                ; Filter order
mu          .set      0.04              ; Step size
*
*    INITIALIZE POINTERS AND ARRAYS
*
            .text
begin       .set      $
            LDI       order*2,BK        ; Set up circular buffer
            LDP       @kn_addr          ; Set data page
            LDI       @kn_addr,AR0      ; Set pointer for k[]
            LDI       @bn_addr,AR1      ; Set pointer for b[]
            LDI       @gn_addr,AR2      ; Set pointer for g[]
            LDI       order,IR0
            LDF       0.0,R0            ; R0 = 0.0
            RPTS      order*2-1
            STF       R0,*AR0++(1)%     ; k[] = 0.0 and g[] = 0.0
         || STF       R0,*AR1++(1)%     ; b[] = 0.0 and bd[] = 0.0
            ADDI      AR1,IR0,AR4
            LDI       @in_addr,AR6      ; Set pointer for input ports
            LDI       @out_addr,AR7     ; Set pointer for output ports
input:
            LDF       *AR6,R7           ; Input d(n)
         || LDF       *++AR6(1),R5      ; Input x(n)
            MPYF3     R5,*AR2,R6        ; B1 * G1
         || STF       R5,*AR1           ; Insert B1
            SUBF      R6,R7             ; E = D - B1 * G1
*
            LDI       order-1,RC
            RPTB      lattice
            MPYF3     *AR0,R5,R3        ; R3 = kFi-1
            MPYF3     R7,*AR1++(1)%,R0  ; R0 = Ei-1 * Bi-1
         || SUBF3     R3,*AR4,R3        ; R3 = Bi = BDi-1 - kFi-1
            MPYF      @u,R0             ; R0 = u * Ei-1 * Bi-1
            ADDF3     R0,*AR2,R0        ; R0 = Gi-1 + u * Ei-1 * Bi-1
         || STF       R3,*AR1           ; Store Bi
            MPYF3     R5,*AR1,R1        ; R1 = Fi-1 * Bi
         || STF       R0,*AR2++(1)      ; Store Gi
            MPYF3     *AR0,*AR4,R0      ; R0 = kBDi-1
            SUBF      R0,R5             ; R5 = Fi
            MPYF3     R5,*AR4++(1)%,R0  ; R1 = Fi * BDi-1
            ADDF      R1,R0             ; R0 = Fi*BDi-1 + Fi-1*Bi
            MPYF      @u,R0             ; R0 = u * (Fi*BDi-1 + Fi-1*Bi)
            ADDF3     R0,*AR0,R0        ; ki = ki-1 + R0
            MPYF3     R3,*AR2,R4        ; R4 = Yi
         || STF       R0,*AR0++(1)      ; Store ki
            ADDF      R4,R6             ; Compute y(n)
lattice     SUBF      R4,R7             ; Compute e(n)
*
*    OUTPUT y(n) AND e(n) SIGNALS
*
            BD        input             ; Delay branch
            SUBF      R4,R6             ; Take out last term
            STF       R6,*AR7           ; Send out y(n)
         || STF       R7,*+AR7(1)       ; Send out e(n)
            LDI       *AR0--(IR0),R5    ; Update k[] pointer
         || LDI       *AR2--(IR0),R7    ; Update g[] pointer
*
*   DEFINE CONSTANTS
*
kn          .usect    "coeffs",order
gn          .usect    "coeffs",order
bn          .usect    "buffer",2*order
in_addr     .usect    "vars",1
out_addr    .usect    "vars",1
kn_addr     .usect    "vars",1
bn_addr     .usect    "vars",1
gn_addr     .usect    "vars",1
u           .usect    "vars",1
cinit       .sect     ".cinit"
            .word     6,in_addr
            .word     0804000h
            .word     0804002h
            .word     kn
            .word     bn
            .word     gn
            .float    mu
            .end
```

Appendix D1. Transversal Structure with Normalized LMS Algorithm Using the TMS320C25

```
        .title   'TN25'
*****************************************************************
*
*   TN25 :  Adaptive Filter Using Transversal Structure
*           and Normalized LMS Algorithm ,Looped Code
*
*   Algorithm:
*
*                63
*       y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*                k=0
*
*       e(n) = d(n) - y(n)
*
*       var(k) = (1.-r) * var(K-1) + r * x(n) * x(n)
*
*       w(k) = w(k) + u*e(n)*x(n-k)/var(k)  k=0,1,2,..63
*
*       Where we use filter order = 64 and mu = 0.01.
*
*   Note:  This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*   Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to 1.
*          3) The current DP (data memory page pointer) should be page 0.
*          4) Data memory ONE should be 1.
*          5) Data memory U should be 327.
*          6) Data memory VAR should be initialized to 07fffh.
*
*                  Chen, Chein-Chung  February, 1989
*
*************************************************
*
*   DEFINE PARAMETERS
*
ORDER:      .equ     64
SHIFT:      .equ     7
PAGE0:      .equ     0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:         .usect   "buffer",ORDER-1
XN:         .usect   "buffer",1
WN:         .usect   "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect   "parameters",1
Y:          .usect   "parameters",1
ERR:        .usect   "parameters",1
ONE:        .usect   "parameters",1
U:          .usect   "parameters",1
ERRF:       .usect   "parameters",1
VAR:        .usect   "parameters",1
*********************************
*   PERFORM THE ADAPTIVE FILTER
*********************************
        .text
*
*   ESTIMATE THE POWER OF SIGNAL
*
        LARP    AR3
        LRLK    AR3,X0          ; Point to input signal X
        SQRA    *               ; Square input signal
        SPH     ERRF
        ZALH    VAR             ; ACC = VAR(n-1)
        SUB     VAR,SHIFT       ; ACC = (1-r) * VAR(n-1)
        ADD     ERRF,SHIFT      ; ACC = (1-r) * VAR(n-1) + r * X(n)
*                               ; * X(n)
        SACH    VAR             ; Store VAR(n)
*
*   ESTIMATE THE SIGNAL Y
*
        CNFP                    ; Configure B0 as program memory
        MPYK    0               ; Clear the P register
        LAC     ONE,15          ; Using rounding
        LRLK    AR3,XN          ; Point to the oldest sample
FIR     RPTK    ORDER-1         ; Repeat N times
        MACD    WN+0fd00h,*-    ; Estimate Y(n)
        CNFD                    ; Configure B0 as data memory
        APAC
        SACH    Y               ; Store the filter output
*
*   COMPUTE THE ERROR
*
        NEG                     ; ACC = - Y(n)
        ADDH    D
        SACH    ERR             ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
        LT      ERR             ; T = ERR(n)
        MPY     U               ; P = U * ERR(n)
        PAC
        ADD     ONE,15          ; Round the result
*
*   NORMALIZE CONVERGE FACTOR
*
        ABS                     ; Make dividend positive
        RPTK    14              ; Repeat 15 times
        SUBC    VAR             ; Perform U * |ERR(n)| / VAR
        BIT     ERR,0           ; Check sign of ERR(n)
```

```
        BBZ     NEXT
        NEG                     ; ERRF = - U * |ERR(n)| / VAR
NEXT    SACL    ERRF            ; Store ERRF
*
        LARK    AR1,ORDER-1     ; Set up counter
        LRLK    AR2,WN          ; Point to the coefficients
        LRLK    AR3,XN+1        ; Point to the data samples
        LT      ERRF            ; T register = U * ERR(n)
        MPY     *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT   ZALR    *,AR3           ; Load ACCH with A(k,n) & round
        MPYA    *-,AR2          ; W(k,n+1) = W(k,n) + P
*                               ; P = U * ERR(n) * X(n-k)
        SACH    *+,0,AR1        ; Store W(k,n+1)
        BANZ    ADAPT,*-,AR2
FINISH  .end
```

Appendix D2. Transversal Structure with Normalized LMS Algorithm Using the TMS320C30

```
*************************************************************
*
*   TN30 - Adaptive transversal filter with Normalized LMS algorithm
*          using the TMS320C30
*
*    Algorithm:
*
*                63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*                k=0
*
*       var(n) = r*var(n-1) + (1-r)*x(n)*x(n)
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*x(n-k)/var(n)  k=0,1,2,..63
*
*       Where we use filter order = 64 and mu = 0.01.
*
*                  Chen, Chein-Chung  March, 1989
*
*************************************************
          .copy     "adapfltr.int"
*********************************************
*     PERFORM ADAPTIVE FILTER
*********************************************
order     .set      64              ; Filter order
mu        .set      0.01            ; Step size
power     .set      1.0             ; Input signal power
alpha     .set      0.996
alpha1    .set      0.004           ; 1.0 - alpha
*
*   INITIALIZE POINTERS AND ARRAYS
*
          .text
begin     .set      $
          LDI       order,BK        ; Set up circular buffer
          LDP       @xn_addr        ; Set data page
          LDI       @xn_addr,AR0    ; Set pointer for x[]
          LDI       @wn_addr,AR1    ; Set pointer for w[]
          LDF       0.0,R0          ; R0 = 0.0
          RPTS      order-1
          STF       R0,*AR0++(1)%   ; x[] = 0
       ;| STF       R0,*AR1++(1)%   ; w[] = 0
          LDI       @in_addr,AR6    ; Set pointer for input ports
          LDI       @out_addr,AR7   ; Set pointer for output ports

input:
          LDF       *AR6,R7         ; Input d(n)
       ;| LDF       *+AR6(1),R6     ; Input x(n)
          STF       R6,*AR0         ; Insert x(n) to buffer

*
*  ESTIMATE THE POWER OF THE INPUT SIGNAL
*
          MPYF      R6,R6           ; R6 = x2
          MPYF      @r_1,R6         ; R6 = (1-r) * x2
          LDF       @r,R3
          MPYF      @var,R3         ; R3 = r * var(n-1)
*
*  COMPUTE FILTER OUTPUT y(n)
*
          LDF       0.0,R2          ; R2 = 0.0
*
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
       ;| ADDF      R6,R3
          STF       R3,@var         ; Restore var(n)
          RPTS      order-2
*
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
       ;| ADDF3     R1,R2,R2        ; y(n) = w[].x[]
          ADDF      R1,R2           ; Include last result
*
*  COMPUTE ERROR SIGNAL e(n)
*
          SUBF      R2,R7           ; e(n) = d(n) - y(n)
*
*  OUTPUT y(n) AND e(n) SIGNALS
*
          STF       R2,*AR7         ; Send out y(n)
       ;| STF       R7,*+AR7(1)     ; Send out e(n)
*
*  UPDATE WEIGHTS w(n)
*
          PUSHF     R3              ; Compute  1/var(n)
          POP       R2              ; var(n) = a * 2e
          ASH       -24,R2
          NEGI      R2
          SUBI      1,R2            ; Now we have 2-e-1
          ASH       24,R2
          PUSH      R2
          POPF      R2              ; Now R2 = x[0] = 1.0 * 2-e-1.
*
          MPYF      R2,R3,R0        ; R0 = v * x[0]
          SUBRF     2.0,R0          ; R0 = 2.0 - v * x[0]
          MPYF      R0,R2           ; R2 = x[1] = x[0] * (2.0 - v * x[0])
*
          MPYF      R2,R3,R0        ; R0 = v * x[1]
          SUBRF     2.0,R0          ; R0 = 2.0 - v * x[1]
          MPYF      R0,R2           ; R2 = x[2] = x[1] * (2.0 - v * x[1])
*
          MPYF      R2,R3,R0        ; R0 = v * x[2]
          SUBRF     2.0,R0          ; R0 = 2.0 - v * x[2]
          MPYF      R0,R2           ; R2 = x[3] = x[2] * (2.0 - v * x[2])
*
          MPYF      R2,R3,R0        ; R0 = v * x[3]
```

```
            SUBRF   2.0,R0          ; R0 = 2.0 - v * x[3]
            MPYF    R0,R2           ; R2 = x[4] = x[3] * (2.0 - v * x[3])
            RND     R2              ; This minimizes error in the LSBs.
*
            MPYF    R2,R3,R0        ; R0 = v * x[4] = 1.0..01.. => 1
            SUBRF   1.0,R0          ; R0 = 1.0 - v * x[4] =
*                                   ; 0.0..01... => 0
            MPYF    R2,R0           ; R0 = x[4] * (1.0 - v * x[4])
            ADDF    R0,R2           ; x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
            RND     R2,R0           ; Round since this is followed
*                                   ; by a MPYF.
*
*
            MPYF    @u,R7           ; R7 = e(n) * u
            MPYF    R0,R7           ; R7 = e(n) * u / var(n)
            MPYF3   *AR0++(1)%,R7,R1 ; R1 = e(n) * u * x(n) / var(n)
            LDI     order-3,RC      ; Initialize repeat counter
            RPTB    LMS             ; Do i = 0, N-3
            MPYF3   *AR0++(1)%,R7,R1 ; R1 = e(n) * u * x(n-i-1) / var(n)
        ||  ADDF3   *AR1,R1,R2      ; R2 = wi(n) + R1
LMS         STF     R2,*AR1++(1)%   ; Store wi(n+1)
            MPYF3   *AR0,R7,R1      ; For i = N - 2
        ||  ADDF3   *AR1,R1,R2
            BD      input           ; Delay branch
            STF     R2,*AR1++(1)%   ; Store wi(n+1)
            ADDF3   *AR1,R1,R2
            STF     R2,*AR1++(1)%   ; Update last w
*
*   DEFINE CONSTANTS AND VARIABLES
*
xn          .usect  "buffer",order
wn          .usect  "coeffs",order
in_addr     .usect  "vars",1
out_addr    .usect  "vars",1
xn_addr     .usect  "vars",1
wn_addr     .usect  "vars",1
u           .usect  "vars",1
var         .usect  "vars",1
r           .usect  "vars",1
r_1         .usect  "vars",1
cinit       .sect   ".cinit"
            .word   8,in_addr
            .word   0804000h
            .word   0804002h
            .word   xn
            .word   wn
            .float  mu
            .float  power
            .float  alpha
            .float  alpha1
            .end
```

Appendix E1. Transversal Structure with Sign-Error LMS Algorithm Using the TMS320C25

```
          .title    'TSE25'
*****************************************************************
*
*    TSE25 :  Adaptive Filter Using Transversal Structure
*             and Sign-Error LMS Algorithm ,Looped Code
*
*    Algorithm:
*
*                63
*       y(n) = SUM w(k)*x(n-k)   k=0,1,2,...,63
*               k=0
*
*       e(n) = d(n) - y(n)
*
*       For k = 0,1,2,...,63
*             w(k) = w(k) + u*x(n-k) if e(n) >= 0
*             w(k) = w(k) - u*x(n-k) if e(n) <  0
*
*       Where we use filter order = 64 and mu = 0.01.
*
*    Note: This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*     Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to 1.
*          3) The current DP (data memory page pointer) should be page 0.
*          4) Data memory ONE should be 1.
*          5) Data memory U should be 327.
*          6) Data memory NEGMU should be -327.
*
*                  Chen, Chein-Chung  February, 1989
*
*************************************************
*
*    DEFINE PARAMETERS
*
ORDER:    .equ     64
PAGE0:    .equ     0
*
*    DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:       .usect   "buffer",ORDER-1
XN:       .usect   "buffer",1
WN:       .usect   "coeffs",ORDER
*
*    RESERVE ADDRESSES FOR PARAMETERS
*
D:        .usect   "parameters",1
Y:        .usect   "parameters",1
ERR:      .usect   "parameters",1
ONE:      .usect   "parameters",1
U:        .usect   "parameters",1
ERRF:     .usect   "parameters",1
NEGMU:    .usect   "parameters",1
*********************************
*    PERFORM THE ADAPTIVE FILTER
*********************************
          .text
*
*    ESTIMATE THE SIGNAL Y
*
          LARP     AR3
          CNFP                        ; Configure B0 as program memory
          MPYK     0                  ; Clear the P register
          LAC      ONE,15             ; Using rounding
          LRLK     AR3,XN             ; Point to the oldest sample
FIR       RPTK     ORDER-1            ; Repeat N times
          MACD     WN+0fd00h,*-       ; Estimate Y(n)
          CNFD                        ; Configure B0 as data memory
          APAC
          SACH     Y                  ; Store the filter output
*
*    CHECK THE SIGN OF ERROR
*
          LT       U                  ; T register = U
          NEG                         ; ACC = - Y(n)
          ADDH     D                  ; ACC = D(n) - Y(n)
          BGEZ     NEXT
          LT       NEGMU              ; T register = -U
*
*    UPDATE THE WEIGHTS
*
NEXT      LARK     AR1,ORDER-1        ; Set up counter
          LRLK     AR2,WN             ; Point to the coefficients
          LRLK     AR3,XN+1           ; Point to the data sample
          MPY      *-,AR2             ; P = U * X(n-k)
ADAPT     ZALR     *,AR3              ; Load ACCH with W(k,n) & round
          MPYA     *-,AR2             ; W(k,n+1) = W(k,n) + P
*                                     ; P = U * X(n-k)
          SACH     *+,0,AR1           ; Store W(k,n+1)
          BANZ     ADAPT,*-,AR2
*
FINISH    .end
```

Appendix E2. Transversal Structure with Sign-Error LMS Algorithm Using the TMS320C30

```
*************************************************************
*
*   TSE30 - Adaptive transversal filter with Sign-Error LMS
*     algorithm using the TMS320C30
*
*    Algorithm:
*
*              63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       for k=0,1,2,..63
*       w(k) = w(k) + u*x(n-k) if e(n) >= 0.0
*       w(k) = w(k) - u*x(n-k) if e(n) <  0.0
*
*       Where we use filter order = 64 and mu = 0.01.
*
*                  Chen, Chein-Chung  March, 1989
*
*************************************************
          .copy     "adapfltr.int"
*************************************************
*     PERFORM ADAPTIVE FILTER
*************************************************
order     .set      64
mu        .set      0.01
*
*   INITIALIZE POINTERS AND ARRAYS
*
          .text
begin     .set      $
          LDI       order,BK          ; Set up circular buffer
          LDP       @xn_addr          ; Set data page
          LDI       @xn_addr,AR0      ; Set pointer for x[]
          LDI       @wn_addr,AR1      ; Set pointer for w[]
          LDF       0.0,R0            ; R0 = 0.0
          RPTS      order-1
          STF       R0,*AR0++(1)%     ; x[] = 0
       || STF       R0,*AR1++(1)%     ; w[] = 0
          LDI       @in_addr,AR6      ; Set pointer for input ports
          LDI       @out_addr,AR7     ; Set pointer for output ports
          LDF       @u,R4             ; R4 = mu
          LDF       @u,R5             ; R5 = mu
*
input:
          LDF       *AR6,R7           ; Input d(n)
       || LDF       *+AR6(1),R6       ; Input x(n)
          STF       R6,*AR0           ; Insert x(n) to buffer
*
*   COMPUTE FILTER OUTPUT y(n)
*
          LDF       0.0,R2 II         ; R2 = 0.0
*
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
          RPTS      order-2
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
*
       || ADDF3     R1,R2,R2IIII      ; y(n) = w[].x[]
          ADDF      R1,R2             ; Include last result
*
*   COMPUTE ERROR SIGNAL e(n)
*
          SUBF      R2,R7             ; e(n) = d(n) - y(n)
*
*   OUTPUT y(n) AND e(n) SIGNALS
*
          STF       R2,*AR7           ; Send out y(n)
       || STF       R7,*+AR7(1)       ; Send out e(n)
*
*   UPDATE WEIGHTS w(n)
*
          ASH       -31,R7            ; Get Sign[e(n)]
          XOR3      R4,R7,R5          ; R5 = S[e(n)] * u
          MPYF3     *AR0++(1)%,R5,R1  ; R1 = S[e(n)] * u * x(n)
          LDI       order-3,RC        ; Initialize repeat counter
          RPTB      SELMS             ; Do i = 0, N-3
          MPYF3     *AR0++(1)%,R5,R1  ; R1 = S[e(n)] * u * x(n-i-1)
       || ADDF3     *AR1,R1,R2        ; R2 = wi(n) + S[e(n)] * u * x(n-i)
SELMS     STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + S[e(n)]*u*x(n-i)
          MPYF3     *AR0,R5,R1        ; For i = N - 2
       || ADDF3     *AR1,R1,R2
          BD        input             ; Delay branch
          STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + S[e(n)]*u*x(n-i)
          ADDF3     *AR1,R1,R2
          STF       R2,*AR1++(1)%     ; Update last w
*
*   DEFINE CONSTANTS
*
xn        .usect    "buffer",order
wn        .usect    "coeffs",order
in_addr   .usect    "vars",1
out_addr  .usect    "vars",1
xn_addr   .usect    "vars",1
wn_addr   .usect    "vars",1
u         .usect    "vars",1
cinit     .sect     ".cinit"
          .word     5,in_addr
          .word     0804000h
          .word     0804002h
          .word     xn
          .word     wn
          .float    mu
          .end
```

Appendix F1. Transversal Structure with Sign-Sign LMS Algorithm Using the TMS320C25

```
        .title   'TSS25'
*****************************************************************
*
*   TSS :  Adaptive Filter Using Transversal Structure
*          and Sign-Sign LMS Algorithm ,Looped Code
*
*   Algorithm:
*
*             63
*      y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*             k=0
*
*      e(n) = d(n) - y(n)
*
*      For k = 0,1,2,...,63
*           w(k) = w(k) + u if e(n)*x(n-k) >= 0
*           w(k) = w(k) - u if e(n)*x(n-k) <  0
*
*      Where we use filter order = 64 and mu = 0.01.
*
*   Note: This source program is the generic version; I/O configuration has
*         not been set up. User has to modify the main routine for specific
*         application.
*
*   Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory ONE should be 1.
*         5) Data memory U should be 327.
*
*                 Chen, Chein-Chung  February, 1989
*
*************************************************
*
*   DEFINE PARAMETERS
*
ORDER:    .equ     64
PAGE0:    .equ     0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:       .usect   "buffer",ORDER-1
XN:       .usect   "buffer",1
WN:       .usect   "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:        .usect   "parameters",1
Y:        .usect   "parameters",1
ERR:      .usect   "parameters",1
ONE:      .usect   "parameters",1
U:        .usect   "parameters",1
ERRF:     .usect   "parameters",1
*********************************
*   PERFORM THE ADAPTIVE FILTER
*********************************
        .text
*
*   ESTIMATE THE SIGNAL Y
*
        LARP    AR3
        CNFP                    ; Configure B0 as program memory
        MPYK    0               ; Clear the P register
        LAC     ONE,15          ; Using rounding
        LRLK    AR3,XN          ; Point to the oldest sample
FIR     RPTK    ORDER-1         ; Repeat N times
        MACD    WN+0fd00h,*-    ; Estimate Y(n)
        CNFD                    ; Configure B0 as data memory
        APAC
        SACH    Y               ; Store the filter output
*
*   SET UP THE POINTERS
*
        LARK    AR1,ORDER-1     ; Set up counter
        LRLK    AR2,WN          ; Point to the coefficients
        LRLK    AR3,XN+1        ; Point to the data sample
*
*   CHECK THE SIGN OF ERROR
*
        NEG
        ADDH    D               ; ACC = D(n) - Y(n)
        SACH    ERR
*
*   UPDATE THE WEIGHTS
*
ADAPT   LAC     *-,0,AR2        ; ACC = X(n-k)
        XOR     ERR             ; Get the sign of ERR(n) * X(n-k)
        SACL    ERRF            ; Store the sign
        LAC     ERRF            ; Get the sign with its sign extension
        XORK    MU,15           ; Get the convergent factor MU or -MU
        ADD     *,15            ; Update W(k)
        SACH    *+,1,AR1
        BANZ    ADAPT,*-,AR3
*
FINISH  .end
```

Appendix F2. Transversal Structure with Sign-Sign LMS Algorithm Using the TMS320C30

```
*************************************************************
*
*   TSS30 - Adaptive transversal filter with Sign-Sign LMS
*           algorithm using the TMS320C30
*
*    Algorithm:
*
*              63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       for k=0,1,2,...,63
*           w(k) = w(k) + u, if x(n-k)*e(n) >= 0.0
*           w(k) = w(k) - u, if x(n-k)*e(n) <  0.0
*
*       Where we use filter order = 64 and mu = 0.01.
*
*                  Chen, Chein-Chung  March, 1989
*
*************************************************************
            .copy     "adapfiltr.int"
order       .set      64
mu          .set      0.01
*
*   INITIALIZE POINTERS AND ARRAYS
*
            .text
begin       .set
            LDI       order,BK          ; Set up circular buffer
            LDP       @xn_addr          ; Set data page
            LDI       @xn_addr,AR0      ; Set pointer for x[]
            LDI       @wn_addr,AR1      ; Set pointer for w[]
            LDF       @u,R0             ; R0 = mu
            LDF       @u,R4             ; R4 = mu
            LDF       @u,R5             ; R5 = mu
            LDF       0.0,R0            ; R0 = 0.0
            RPTS      order-1
            STF       R0,*AR0++(1)%     ; x[] = 0
         || STF       R0,*AR1++(1)%     ; w[] = 0
            LDI       @in_addr,AR6      ; Set pointer for input ports
            LDI       @out_addr,AR7     ; Set pointer for output ports
input:
            LDF       *AR6,R7           ; Input d(n)
         || LDF       *+AR6(1),R6       ; Input x(n)
            STF       R6,*AR0           ; Insert x(n) to buffer
*
*   COMPUTE FILTER OUTPUT y(n)
*
            LDF       0.0,R2            ; R2 = 0.0
*
            MPYF3     *AR0++(1)%,*AR1++(1)%,R1
            RPTS      order-2
            MPYF3     *AR0++(1)%,*AR1++(1)%,R1
         || ADDF3     R1,R2,R2          ; y(n) = w[].x[]
            ADDF      R1,R2             ; Include last result
*
*   COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) AND e(n) SIGNALS
*
            SUBF      R2,R7             ; e(n) = d(n) - y(n)
            STF       R2,*AR7           ; Send out y(n)
         || STF       R7,*+AR7(1)       ; Send out e(n)
*
*   UPDATE WEIGHTS w(n)
*
            ASH       -31,R7            ; R7 = Sign[e(n)]
            XOR3      R0,R7,R5          ; R5 = Sign[e(n)] * u
            LDF       *AR0++(1)%,R6     ; R6 = x(n)
            ASH       -31,R6            ; R6 = Sign[x(n-i)]
            XOR3      R5,R6,R4          ; R4 = Sign[x(n-i)]*Sign[e(n)] * u
            ADDF3     *AR1,R4,R3        ; R3 = wi(n) + R4

            LDI       order-3,RC        ; Initialize repeat counter
            RPTB      SSLMS             ; Do i = 0, N-3
            LDF       *AR0++(1)%,R6     ; Get next data
         || STF       R3,*AR1++(1)%     ; Update wi(n+1)
            ASH       -31,R6            ; Get the sign of data
            XOR3      R5,R6,R4          ; Decide the sign of u
SSLMS       ADDF3     *AR1,R4,R3        ; R3 = wi(n) + R4

            LDF       *AR0,R6           ; Get last data
         || STF       R3,*AR1++(1)%     ; Update wN-2(n+1)
            ASH       -31,R6            ; Get the sign of data
            BD        input             ; Delay branch
            XOR3      R5,R6,R4          ; Decide the sign of u
            ADDF3     *AR1,R4,R3        ; Compute wN-1(n+1)
            STF       R3,*AR1++(1)%     ; Store last w(n+1)
*
*   DEFINE CONSTANTS
*
xn          .usect    "buffer",order
wn          .usect    "coeffs",order
in_addr     .usect    "vars",1
out_addr    .usect    "vars",1
xn_addr     .usect    "vars",1
wn_addr     .usect    "vars",1
u           .usect    "vars",1
cinit       .sect     ".cinit"
            .word     5,in_addr
            .word     0804000h
            .word     0804002h
            .word     xn
            .word     wn
            .float    mu
.end
```

Appendix G1. Transversal Structure with Leaky LMS Algorithm Using the TMS320C25

```
          .title    'TL25'
*****************************************************************
*
*    TL25 :  Adaptive Filter Using Transversal Structure
*             and Leaky-LMS Algorithm, Looped Code
*
*    Algorithm:
*
*                63
*       y(n) = SUM w(k)*x(n-k)   k=0,1,2,...,63
*                k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = v*w(k) + u*e(n)*x(n-k) k=0,1,2,..63
*
*       Where we use filter order = 64 and mu = 0.01.
*
*    Note: This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*    Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to 1.
*          3) The current DP (data memory page pointer) should be page 0.
*          4) Data memory ONE should be 1.
*          5) Data memory U should be 327.
*
*                  Chen, Chein-Chung  February, 1989
*
*************************************************
*
*    DEFINE PARAMETERS
*
ORDER:    .equ      64
LEAKY:    .equ      7
PAGE0:    .equ      0
*
*    DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:       .usect    "buffer",ORDER-1
XN:       .usect    "buffer",1
WN:       .usect    "coeffs",ORDER
*
*    RESERVE ADDRESSES FOR PARAMETERS
*
D:        .usect    "parameters",1
Y:        .usect    "parameters",1
ERR:      .usect    "parameters",1
ONE:      .usect    "parameters",1
U:        .usect    "parameters",1
ERRF:     .usect    "parameters",1
*********************************
*    PERFORM THE ADAPTIVE FILTER
*********************************
          .text
*
*    ESTIMATE THE SIGNAL Y
*
          LARP      AR3
          CNFP                        ; Configure B0 as program memory
          MPYK      0                 ; Clear the P register
          LAC       ONE,15            ; Using rounding
          LRLK      AR3,XN            ; Point to the oldest sample
FIR       RPTK      ORDER-1           ; Repeat N times
          MACD      WN+0fd00h,*-      ; Estimate Y(n)
          CNFD                        ; Configure B0 as data memory
          APAC
          SACH      Y                 ; Store the filter output
*
*    COMPUTE THE ERROR
*
          NEG                         ; ACC = - Y(n)
          ADDH      D
          SACH      ERR               ; ERR(n) = D(n) - Y(n)
*
*    UPDATE THE WEIGHTS
*
          LT        ERR               ; T = ERR(n)
          MPY       U                 ; P = U * ERR(n)
          PAC
          ADD       ONE,15            ; Round the result
          SACH      ERRF              ; ERRF = U * ERR(n)
*
          LARK      AR1,ORDER-1       ; Set up counter
          LRLK      AR2,WN            ; Point to the coefficients
          LRLK      AR3,XN+1          ; Point to the data sample
          LT        ERRF              ; T register = U * ERR(n)
          MPY       *-,AR2            ; P = U * ERR(n) * X(n-k)
ADAPT     ZALR      *,AR3             ; Load ACCH with A(k,n) & round
          MPYA      *-,AR2            ; W(k,n+1) = W(k,n) + P
*                                     ; P = U * ERR(n) * X(n-k)
          SUB       *,LEAKY           ; ACC = R * W(k,n) + P
          SACH      *+,0,AR1          ; Store W(k,n+1)
          BANZ      ADAPT,*-,AR2
*
FINISH    .end
```

Appendix G2. Transversal Structure with Leaky LMS Algorithm Using the TMS320C30

```
*************************************************************
*   TL30 - Adaptive transversal filter with Leaky LMS algorithm
*          using the TMS320C30
*
*    Algorithm:
*
*              63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = r*w(k) + u*e(n)*x(n-k) k=0,1,2,...,63
*
*       Where we use filter order = 64, r = 0.995 and mu = 0.01.
*
*                  Chen, Chein-Chung  March, 1989
*
*************************************************
          .copy     "adapfltr.int"
*************************************************
*     PERFORM ADAPTIVE FILTER
*************************************************
order     .set      64
mu_leaky  .set      0.01005          ; mu / leaky
leaky     .set      0.995
*
*    INITIALIZE POINTERS AND ARRAYS
*
          .text
begin     .set      $
          LDI       order,BK         ; Set up circular buffer
          LDP       @xn_addrm        ; Set data page
          LDI       @xn_addr,AR0     ; Set pointer for x[]
          LDI       @wn_addr,AR1     ; Set pointer for w[]
          LDI       @r_addr,AR2      ; Set pointer for r
          LDF       0.0,R0           ; R0 = 0.0
          RPTS      order-1
          STF       R0,*AR0++(1)%    ; x[] = 0
       ||  STF      R0,*AR1++(1)%    ; w[] = 0
          LDI       @in_addr,AR6     ; Set pointer for input ports
          LDI       @out_addr,AR7    ; Set pointer for output ports
input:
          LDF       *AR6,R7          ; Input d(n)
       ||  LDF      *++AR6(1),R6     ; Input x(n)
          STF       R6,*AR0          ; Insert x(n) to buffer
*
*    COMPUTE FILTER OUTPUT y(n)
*
          LDF       0.0,R2           ; R2 = 0.0
*
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
          RPTS      order-2
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
       ||  ADDF3    R1,R2,R2         ; y(n) = w[].x[]
          ADDF      R1,R2            ; include last result
*
*    COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) AND e(n) SIGNALS
*
          SUBF      R2,R7            ; e(n) = d(n) - y(n)
          STF       R2,*AR7          ; Send out y(n)
       ||  STF      R7,*+AR7(1)      ; Send out e(n)
*
*    UPDATE WEIGHTS w(n)
*
          MPYF      @u_r,R7          ; R7 = e(n)*u/r
          MPYF3     *AR0++(1)%,R7,R1 ; R1 = e(n)*u*x(n)/r
          MPYF3     *AR0++(1)%,R7,R1 ; R1 = e(n)*u*x(n-1)/r
       ||  ADDF3    *AR1,R1,R2       ; R2 = w0(n) + e(n)*u*x(n)/r
          LDI       order-4,RC       ; Initialize repeat counter
          RPTB      LLMS             ; Do i = 0, N-4
          MPYF3     *AR2,R2,R0       ; R0 = r*wi(n) + e(n)*u*x(n-i)
       ||  ADDF3    *+AR1(1),R1,R2   ; R2 = wi+1(n) + e(n)*u*x(n-i-1)/r
LLMS      MPYF3     *AR0++(1)%,R7,R1 ; R1 = e(n)*u*x(n-i-2)/r
       ||  STF      R0,*AR1++(1)%    ; store wi(n+1)
          MPYF3     *AR2,R2,R0       ; R0 = r*wN-3(n) + e(n)*u*x(n-N+3)
       ||  ADDF3    *+AR1(1),R1,R2   ; R2 = wN-2(n) + e(n)*u*x(n-N+2)/r
          MPYF3     *AR0,R7,R1       ; R1 = e(n)*u*x(n-N+1)/r
       ||  STF      R0,*AR1++(1)%    ; Store wN-3(n+1)
          BD        input            ; Delay branch
          MPYF3     *AR2,R2,R0       ; R0 = r*wi(n) + e(n)*u*x(n-N+2)
       ||  ADDF3    *+AR1(1),R1,R2   ; R2 = wN-1(n) + e(n)*u*x(n-N+1)/r
          MPYF3     *AR2,R2,R0       ; R0 = r*wi(n) + e(n)*u*x(n-N+1)
       ||  STF      R0,*AR1++(1)%    ; Store wN-2(n+1)
          STF       R0,*AR1++(1)%    ; Update last w
*
*    DEFINE CONSTANTS
*
xn        .usect    "buffer",order
wn        .usect    "coeffs",order
in_addr   .usect    "vars",1
out_addr  .usect    "vars",1
xn_addr   .usect    "vars",1
wn_addr   .usect    "vars",1
u_r       .usect    "vars",1
r         .usect    "vars",1
r_addr    .usect    "vars",1
cinit     .sect     ".cinit"
          .word     7,in_addr
          .word     0804000h
          .word     0804002h
          .word     xn
          .word     wn
          .float    mu_leaky
          .float    leaky
          .word     r
          .end
```

Appendix H1. Assembly Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title   'BLMS'
*****************************************************************
*
*   BLMS :  Adaptive Filter subroutine using Transversal Structure
*           and LMS Algorithm, Looped Code
*
*   Algorithm:
*
*             N-1
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*             k=0
*
*       e(n) = d(n) - y(n)-
*
*       w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*       Where we use filter order = N
*
*   Note: This subroutine performs Adaptive Filter using the LMS Algorithm.
*         There are some initial conditions to meet before calling it.
*
*   Initial conditions:
*         1) Data memory ONE should be equal to 1.
*         2) Data memory U should be equal to MU (Q15 format).
*         3) PM status bit should be equal to 01.
*         4) SXM status bit should be set to logic 1.
*         5) OVM status bit should be set to 1.
*         6) The current DP (data memory page pointer) should be page 0.
*
*    p.s. 1) The return current auxiliary register will be AR2.
*         2) AR1  AR3 have been used in this subroutine.
*
*                 Chen, Chein-Chung  February, 1989
*
************************************************
*
*   DEFINE AND REFER SYMBOLS
*
        .global  LMS,ORDER,U,D,ONE,Y,ERR,XN,WN
*
*   RESERVE ADDRESS FOR PARAMETER
*
SAVE1:  .usect   "parameters",1
SAVE2:  .usect   "parameters",1
SAVE3:  .usect   "parameters",1
ERRF:   .usect   "parameters",1
**********************************
*   PERFORM THE ADAPTIVE FILTER
**********************************
*
*   ESTIMATE THE SIGNAL Y
        .text
LMS     LARP    AR3             ; Set current register
        SAR     AR1,SAVE1       ; Save register AR1
        SAR     AR2,SAVE2       ; Save register AR2
        SAR     AR3,SAVE3       ; Save register AR3
        CNFP                    ; Configure B0 as program memory
        MPYK    0               ; Clear the P register
        LAC     ONE,15          ; Using rounding
        LRLK    AR3,XN          ; Point to the oldest sample
FIR     RPTK    ORDER-1         ; Repeat N times
        MACD    WN+0fd00h,*-    ; Estimate Y(n)
        CNFD                    ; Configure B0 as data memory
        APAC
        SACH    Y               ; Store the filter output
*
*   COMPUTE THE ERROR
*
        NEG                     ; ACC = - Y(n)
        ADDH    D
        SACH    ERR             ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
        LT      ERR             ; T = ERR(n)
        MPY     U               ; P = U * ERR(n)
        PAC
        ADD     ONE,15          ; round the result
        SACH    ERRF            ; ERRF = U * ERR(n)
*
        LARK    AR1,ORDER-1     ; Set up counter
        LRLK    AR2,WN          ; Point to the coefficients
        LRLK    AR3,XN+1        ; Point to the data sample
        LT      ERRF            ; T register = U * ERR(n)
        MPY     *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT   ZALR    *,AR3           ; Load ACCH with A(k,n) & round
        MPYA    *-,AR2          ; W(k,n+1) = W(k,n) + P
*                               ; P = U * ERR(n) * X(n-k)
        SACH    *+,0,AR1        ; Store W(k,n+1)
        BANZ    ADAPT,*-,AR2
*
        LAR     AR1,SAVE1       ; Restore register AR1
        LAR     AR2,SAVE2       ; Restore register AR2
        LAR     AR3,SAVE3       ; Restore register AR3
*
FINISH RET
*
        .end
```

Appendix H2. Linker Command File for Assembly Main Program Calling a TMS320C25 Adaptive LMS Transversal Filter Subroutine

```
/*****************************************************************/
/* ALINK.CMD - COMMAND FILE FOR LINKING A TMS320C25 ASSEMBLY PROGRAM  */
/*             Copyright 1988, 1989 Texas Instruments Incorporated    */
/*                                                                    */
/*     Usage: dsplnk <obj files...>,-o <out file) -m <map file) c.cmd  */
/*                                                                    */
/* Description: This file is a sample command file that can be used   */
/*              for linking the TMS320C25 assembly programs; use it as a */
/*              guideline. You may want to change the allocation      */
/*              scheme according to the size of the program and the   */
/*              memory configuration of your TMS320C25.               */
/*                                                                    */
/* Notes:       MEMORY SPECIFICATION                                  */
/*                                                                    */
/*              Block B0 is configured as data memory (DNPD) and      */
/*              MP/MC = 1 (microprocessor mode). Data memory locations */
/*              0h-5Fh and 800h-1FFh are not configured.              */
/*                                                                    */
/**********************************************************************/
MEMORY
{
    PAGE 0 : Ints    : origin =   0h, length =  020h     /* Program */
             Ext_Prog: origin = 020h, length = 0FEE0h

    PAGE 1 : Regs    : origin =   0h, length =   06h     /* Data  */
             Block_B2: origin =  060h, length =  020h
             Int_RAM : origin = 0200h, length = 0100h    /* B0    */
             Int_RAM1: origin = 0300h, length = 0100h    /* B1    */
             Ext_Data: origin = 0400h, length = 0FC00h
}

/*--------------------------------------------------------------------*/
/* SECTIONS ALLOCATION                                                */
/*--------------------------------------------------------------------*/
SECTIONS
{
   vectors  : { } > Ints     PAGE 0   /* Interrupt vector table   */
   .text    : { } > Ext_Prog PAGE 0   /* Code                     */
   parameters: { } > Block_B2 PAGE 1  /* Parameters               */
   coeffs   : { } > Int_RAM  PAGE 1   /* Block B0                 */
   buffer   : { } > Int_RAM1 PAGE 1   /* Block B1                 */
   .bss     : { } > Ext_Data PAGE 1   /* Global VARS, STACK, HEAP */
}
```

```
          .width    132
*************************************************************
*
*   This is the initial boot routine for TMS320C30 adaptive
*        filter Programs.
*
*   This module performs the following actions:
*        1) Allocates and initializes the system stack.
*        2) Performs auto-initialization, which copies section
*           ".const" data from ROM to DATA RAM.
*        3) Prepare to start the user's assembly program.
*
*************************************************************
STACK_SIZE .set     40h             ; Size of system stack
FP         .set     AR3             ; Frame pointer
*
           .sect    "vectors"
RESET      .word    adap_init
*
* ALLOCATE SPACE FOR THE SYSTEM STACK. INITIALIZE THE FIRST WORDS IN
* .text TO POINT TO THE STACK AND INITIALIZATION TABLES.
*
stack      .usect   ".stack",STACK_SIZE
           .text
*
stack_addr .word    stack           ; Address of stack
init_addr  .word    cinit           ; Address of init tables
*************************************************************
* ADAPTIVE FILTER INITIALIZATION ENTRY POINT FUNCTION
*************************************************************
adap_init:
*
* SET UP THE INITIAL STACK POINTER
*
           LDP      stack_addr      ; Get page of stored address
           LDI      @stack_addr,SP  ; Load the address into SP
           LDI      SP,FP           ; And into FP too
*
* DO AUTOINITIALIZATION
*
           LDP      init_addr       ; Get page of stored address
           LDI      @init_addr,AR0  ; Get address of init tables
           CMPI     -1,AR0          ; If RAM model, skip init
           BEQ      done
           LDI      *AR0++,R1       ; Get first count
           BZD      done            ; If 0, nothing to do
           LDI      *AR0++,AR1      ; Get dest address
           LDI      *AR0++,R0       ; Get first word
           SUBI     1,R1            ; Count - 1
*
do_init:
           RPTS     R1              ; Block copy
             STI    R0,*AR1++
          || LDI    *AR0++,R0
             LDI    R0,R1           ; Move next count into R1
             BNZD   do_init         ; If there is more, repeat
             LDI    *AR0++,AR       ; Get next dest address
             LDI    *AR0++,R0       ; Get next first word
             SUBI   1,R1            ; Count - 1
*
done:
 BR          begin
             .end
```

Appendix H4. Assembly Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C30

```
*************************************************************
*
*  BT30 - TMS320C30 adaptive transversal filter with
*      LMS algorithm assembly subroutine.
*
*   Algorithm:
*
*            N-1
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*            k=0
*
*      e(n) = d(n) - y(n)
*
*      w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*      Where we use filter order = N and mu = 0.01.
*
*   Initial condition:
*
*       1) AR0 and AR1 should point to x[0] and w[0].
*       2) Data memory u should contain step size.
*       3) Data memory order should contain N-2, where N is filter order.
*       4) Data memories d, y, and e should be defined in caller routine.
*
*                Chen, Chein-Chung  March, 1989
*
************************************************
          .global   LMS30,u,d,y,e,order
************************************************
*    PERFORM ADAPTIVE FILTER
************************************************
          .text
LMS30     .set      $
          PUSH      R1
          PUSHF     R1
          PUSHF     R2
          PUSH      R3
          PUSHF     R3
*
*  COMPUTE FILTER OUTPUT y(n)
*
          LDF       0.0,R3            ; R3 = 0.0
*
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
          RPTS      @order
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
        ||ADDF3     R1,R3,R3          ; y(n) = w[].x[]
          ADDF      R1,R3             ; Include last result
*
*  COMPUTE ERROR SIGNAL e(n) AND STORE y(n) AND e(n)
*
          STF       R3,@y             ; Store y(n)
          SUBRF     @d,R3             ; e(n) = d(n) - y(n)
          STF       R3,@e             ; Store e(n)
*
*  UPDATE WEIGHTS w[] AND SHIFT x[]
*
          MPYF      @u,R3             ; R3 = e(n) * u
          MPYF3     *AR0++(1)%,R3,R1  ; R1 = e(n) * u * x(n)
          LDI       @order,RC         ; Initialize repeat counter
          SUBI      1,RC
          RPTB      LMS               ; Do i = 0, N-3
          MPYF3     *AR0++(1)%,R3,R1  ; R1 = e(n) * u * x(n-i-1)
        ||ADDF3     *AR1,R1,R2        ; R2 = wi(n) + e(n) * u * x(n-i)
LMS       STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
          MPYF3     *AR0,R3,R1        ; for i = N - 2
        ||ADDF3     *AR1,R1,R2
          STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
          ADDF3     *AR1,R1,R2
          STF       R2,*AR1++(1)%     ; Update last w
*
          POPF      R3
          POP       R3
          POPF      R2
          POPF      R1
          POP       R1
*
          RETS
          .end
```

Appendix H5. Linker Command/file for Assembly Main Program Calling the TMS320C30 Adaptive LMS Transversal Filter Subroutine

```
/*************************************************************************/
/* ADAP.CMD  -  COMMAND FILE FOR LINKING TMS320C30 ADAPTIVE FILTER       */
/*              PROGRAMS                                                 */
/*                                                                       */
/*     Usage: lnk30 Obj files..> -o Out file) -m Map file) adap.cmd       */
/*                                                                       */
/* Description: This file is a sample command file that can be used      */
/*              for linking adaptive filter assembly programs.           */
/*              All the adaptive filter programs have to link with the   */
/*              ADAPTINIT.ASM file to do the auto_initialization.        */
/*                                                                       */
/* Notes:       When using the small (default) memory model, be sure     */
/*              that the ENTIRE .bss section fits within a single page.  */
/*              To satisfy this, you must be smaller than 64K words and  */
/*              must not cross a 64K boundary.                           */
/*************************************************************************/

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    VECS:  org = 0          len = 0xc0
    ROM:   org = 0xc0       len = 0x7ff40
    RAM2:  org = 0x809300 len = 0x2c0   /* RAM block 0                  */
    _STACK: org = 0x8096c0 len = 0x40   /* System stack                 */
    RAM1:  org = 0x809c00 len = 0x3c0   /* RAM block 1                  */
    _VARS: org = 0x8094c0 len = 0x1040  /* VARS block - 40h + 4k of EXT */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    vectors# (.) > VECS          /* Interrupt vectors                  */
    .text:   (.) > ROM           /* Code                               */
    .cinit:  (.) > ROM           /* Initialization tables              */
    .stack:  (.) > _STACK        /* System stack                       */
    .vars:   (.) > _VARS         /* Memory for variables               */
    buffer#  (.) > RAM1          /* Memory for data buffer             */
    coeffs#  (.) > RAM1          /* Memory for filter coefficients     */
    gains align(32) + (.) > RAM1 /* Memory for lattice filter gains    */
}
```

Appendix I1. C Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C25

```
          .title    'CLMS'
*****************************************************************
*
*    CLMS :  Adaptive Filter C subroutine using Transversal Structure
*         and LMS Algorithm, Looped Code
*
*    Algorithm:
*
*              N-1
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*       Where we use filter order = N
*
*    Usage:    lms(n,mu,d,x,&y,&e)
*              n  - order of filter
*              mu - convergence factor
*              d  - desired signal
*              x  - input signal
*              &y - addr of output signal
*              &e - addr of error signal
*
*    Note:     Data memory 0200h  0200h+N-1 & 0300h  0300h+N-1 are reserved.
*
*                  Chen, Chein-Chung  February, 1989
*
*****************************************************
          .def      _lms
*
*    RESERVE ADDRESSES FOR PARAMETERS
*
DST0:       .usect    "parameters",1
DST1:       .usect    "parameters",1
SAVE1:      .usect    "parameters",1
SAVE2:      .usect    "parameters",1
SAVE3:      .usect    "parameters",1
SAVE4:      .usect    "parameters",1
ORDER:      .usect    "parameters",1
X:          .usect    "parameters",1
D:          .usect    "parameters",1
U:          .usect    "parameters",1
Y:          .usect    "parameters",1
ERR:        .usect    "parameters",1
ERRF:       .usect    "parameters",1
ADRLST:     .usect    "parameters",1
*
*    DEFINE ADDRESSES OF BUFFER AND COEFFICIENTSS
*
COEFFP:     .equ      0ff00h
COEFFD:     .equ      0200h
FRSTAP:     .equ      0300h
*********************************
*    PERFORM THE ADAPTIVE FILTER
*********************************
*
*    SAVE THE VALUES OF THE REGISTERS
*
          .text
_lms      SAR       AR1,SAVE1
          SAR       AR2,SAVE2
          SAR       AR3,SAVE3
          SAR       AR4,SAVE4
          SST       DST0
          SST1      DST1
*
*    GET THE ADAPTIVE FILTER PARAMETERS
*
          SPM       1               ; Set P register shift mode
          SSXM                      ; Set sign extension mode
          SOVM                      ; Set overflow mode
          LDPK      0               ; Set data page = 0
          MAR       *-              ; Set pointer for getting papameter
          LAC       *-              ; ACC = N
          SUBK      1
          SACL      ORDER           ; ORDER = N - 1
          ADLK      FRSTAP
          SACL      ADRLST          ; Store address of last tap
          LAC       *-
          SACL      U               ; Get and store the MU
          LAC       *-
          SACL      D               ; Get and store the D
          LAC       *-,0,AR3
          LRLK      AR3,FRSTAP
          SACL      *               ; Insert newest sample
*
*    ESTIMATE THE SIGNAL Y
*
          CNFP                      ; Configure B0 as program memory
          MPYK      0               ; Clear the P register
          LALK      1,15            ; Using rounding
          LAR       AR3,ADRLST      ; Point to the oldest sample
FIR       RPT       ORDER           ; Repeat N times
          MACD      COEFFP,*-       ; Estimate Y(n)
          CNFD                      ; Configure B0 as data memory
          APAC
          SACH      Y               ; Store the filter output
*
*    COMPUTE THE ERROR
*
          NEG                       ; ACC = - Y(n)
          ADDH      D
```

```
        SACH    ERR             ; ERR(n) = D(n) - Y(n)
*
*  UPDATE THE WEIGHTS
*
        LT      ERR             ; T = ERR(n)
        MPY     U               ; P = U * ERR(n)
        PAC
        ADLK    1,15            ; Round the result
        SACH    ERRF            ; ERRF = U * ERR(n)
*
        LAR     AR4,ORDER       ; Set up counter
        LBLK    AR2,COEFFD      ; Point to the coefficients
        LAR     AR3,AUXLST      ; Point to the data sample
        MAR     *+
        LT      ERRF            ; T register = U * ERR(n)
        MPY     *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT   ZALR    *,AR3           ; Load ACCH with A(k,n) & round
        MPYA    *-,AR2          ; W(k,n+1) = W(k,n) + P
*                               ; P = U * ERR(n) * X(n-k)
        SACH    *+,0,AR4        ; Store W(k,n+1)
        BANZ    ADAPT,*+,AR2
*
*  STORE THE Y AND ERR
*
        LARP    AR1
        LAR     AR2,*+,AR2      ; Get the address of Y (in MAIN)
        LAC     Y
        SACL    *,0,AR1         ; Store Y
        LAR     AR2,*,AR2       ; Get the address of ERR (in MAIN)
        LAC     ERR
        SACL    *,0,AR1         ; Store ERR
*
*  RESTORE THE REGISTERS
*
        LST     #STO
        LST1    #ST1
        LAR     AR1,SAVE1
        LAR     AR2,SAVE2
        LAR     AR3,SAVE3
        LAR     AR4,SAVE4
*
FINISH RET
*
        .end
```

Appendix I2. C Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C30

```
*****************************************************
*
*  CT30 - TMS320C30 C subroutine adaptive transversal filter with
*      LMS algorithm.
*
*  Algorithm:
*
*            N-1
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*            k=0
*
*      e(n) = d(n) - y(n)
*
*      w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*      Where we use filter order = N and mu = 0.01.
*
*  Usage:  tlms(n,mu,d,&w,&x,&y,&e)
*          n  - order of filter
*          mu - convergence factor
*          d  - desired signal
*          &w - filter coefficients
*          &x - input signal buffer
*          &y - addr of output signal
*          &e - addr of error signal
*
*                Chen, Chein-Chung  March,  1989
*
*****************************************************
          .global  _tlms
FP        .set     AR3
*****************************************************
*    PERFORM ADAPTIVE FILTER
*****************************************************
          .text
_tlms     .set     $
          PUSH     FP
          LDI      SP,FP
          PUSH     AR0
          PUSH     AR1
          PUSH     AR2
          PUSH     R1
          PUSHF    R1
          PUSH     R2
          PUSHF    R2
          PUSH     R4
          PUSHF    R6
          PUSHF    R7
*
*  GET FILTER PARAMETERS
*
          LDI      *-FP(2),R4      ; Get filter order
          LDI      *-FP(6),AR0     ; Get pointer for x[]
          LDI      *-FP(5),AR1     ; Get pointer for w[]
          SUBI     2,R4            ; Set loop counter
*
*  COMPUTE FILTER OUTPUT y(n)
*
          LDF      0.0,R2          ; R2 = 0.0
*
          MPYF3    *AR0++(1),*AR1++(1),R1
          RPTS     R4
          MPYF3    *AR0++(1),*AR1++(1),R1
       ||ADDF3     R1,R2,R2        ; y(n) = w[].x[]
          ADDF     R1,R2           ; Include last result
*
*  COMPUTE ERROR SIGNAL e(n) AND STORE y(n) AND e(n)
*
          LDI      *-FP(2),AR2     ; Get y(n) address
          SUBF3    R2,*+FP(1),R7   ; e(n) = d(n) - y(n)
       ||STF       R2,*AR2         ; Send out y(n)
          LDI      *-FP(3),AR2     ; Get e(n) address
          STF      R7,*AR2         ; Send out e(n)
*
*  UPDATE WEIGHTS w[] AND SHIFT x[]
*
          MPYF     *+FP(2),R7      ; R7 = e(n) * u
          MPYF3    *--AR0(1),R7,R1 ; R1 = e(n) * u * x(n-N+1)
          LDI      R4,RC           ; Initialize repeat counter
          RPTB     LMS             ; Do i = 1, N-1
          MPYF3    *--AR0(1),R7,R1 ; R1 = e(n) * u * x(n-i+1)
       ||ADDF3     *--AR1(1),R1,R2 ; R2 = wi(n) + e(n) * u * x(n-i)
          LDF      *AR0,R6         ; Get x((n+i-N+1)
       ||STF       R2,*AR1         ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
LMS       STF      R6,*+AR0(1)     ; Shift x[]
          ADDF3    *--AR1(1),R1,R2 ; R2 = wi(n) + e(n) * u * x(n)
          STF      R2,*AR1         ; Update last w
*
          POPF     R7
          POPF     R6
          POP      R4
          POPF     R2
          POP      R2
          POPF     R1
          POP      R1
          POP      AR2
          POP      AR1
          POP      AR0
          POP      FP
          RETS
*
          .end
```

A Collection of Functions for the TMS320C30

Gary Sitton

Gaslight Software

Introduction

This report presents a collection of efficient machine language programs for advanced applications with the TMS320C30. These programs provide basic math and transcendental functions. Other routines include vector functions, FFTs and linear algebra.

Library Overview

The set of programs fall into six categories:

I. Normal precision floating point math functions,
II. Extended precision floating point math functions,
III. Integer arithmetic routines,
IV. Vector utility routines,
V. Radix 2 FFT routines, and
VI. Linear algebra routines.

Categories I and II are programs which implement a minimal set of elementary mathematical functions for advanced applications. In these categories, the functions **FPINV** and **SQRT** are improved versions of the programs in the *TMS320C3x User's Guide* [1]. In category III, **IMULT** and **IDIV** are improved versions of the programs **EXTMPY** and **DIVI** in [1]. In category IV, ***FMIEEE** and ***TOIEE** are array versions of the **TOIEEE** and **FMIEEE** scalar programs from the User's Guide.

The names and short descriptions of these routines use some special notation:

Categories I and II:	**xd** — indicates that the relative accuracy of the implemented function is x decimal digits.
Categories IV and VI:	* — program name prefix stands for M or R.
	M — selects the memory based parameter entry point.
	R — selects the register based parameter entry point.
Categories II and VI:	**X** — indicates the extended precision program version.

Consult the program source listings for more details.

The following are brief descriptions of the programs by category:

I. Normal floating-point (32-bit) math functions **($MATH.ASM)**:

A.	**SIN**	—computes a 7d sine(x) for all x in radians.
B.	**COS**	—computes a 7d cosine(x) for all x in radians.
C.	**EXP**	—computes a 7d exp(x) for all $\|x\| \leq 88$.
D.	**LN**	—computes a 7d ln(x) for all $x > 0$.
E.	**ATAN**	—computes a 7d atan(x) in radians for all x.
F.	**SQRT**	—computes an 8d sqrt(x) for all $x \geq 0$.
G.	**FPINV**	—computes an 8d 1/x for all $x \neq 0$.
H.	**FDIV**	—computes an 8d x/y for all x and all $y \neq 0$.

II. Extended-precision, floating-point (40-bit) math functions **($MATHX.ASM)**:

A.	**SINX**	—computes a 9d sine(x) for all x in radians.
B.	**COSX**	—computes a 9d cosine(x) for all x in radians.
C.	**EXPX**	—computes a 9d exp(x) for all $\|x\| \leq 88$.
D.	**LNX**	—computes an 8d ln(x) for all $x > 0$.
E.	**ATANX**	—computes an 8d atan(x) in radians for all x.
F.	**SQRTX**	—computes a 10d sqrt(x) for all $x \geq 0$.
G.	**FPINVX**	—computes a 10d 1/x for all $x \neq 0$.
H.	**FDIVX**	—computes a 10d x/y for all x and all $y \neq 0$.
I.	**FMULTX**	—computes a 10d x*y for all x and y.

III. Integer (32-bit) math routines **($MATHI.ASM)**:

A.	**ILOG2**	—computes $m = \log2(n)$, $n \leq 2^m$ for use with radix 2 FFT programs.
B.	**IMULT**	—computes 64-bit product of two 32-bit numbers.
C.	**IDIV**	—computes quotient and remainder of two 32-bit numbers.

IV. Vector utilities **($VECTOR.ASM)**:

A.	***CORMULT**	—in-place computation of the complex vector product of two complex arrays using the complex conjugate of the second array.
B.	***CONMULT**	—in-place computation of the complex vector product of two complex arrays.
C.	***CBITREV**	—in-place bit reverse permutation on a complex array with separate real and imaginary arrays.
D.	***FMIEEE**	—in-place fast conversion of an IEEE array to a TMS320C30 array.

E. ***TOIEEE** —in-place fast conversion of a TMS320C30 array to an IEEE array.

F. ***VECMULT** —in-place multiplies a constant times an array.

G. ***CONMOV** —moves (fills) a constant into an array.

H. ***VECMOV** —moves (copies) an array into another array.

V. Radix 2 FFT routines (**$FFT2.ASM**):

A. **CFFFT2** —Complex DIF forward radix 2 FFT using separate real and imaginary arrays and 3/4 cycle sine table.

B. **CIFFT2** —Complex DIT inverse radix 2 FFT using separate real and imaginary arrays and 3/4 cycle sine table (does not include the 1/N scale factor).

VI. Linear algebra routines (**$LINALG.ASM**):

A. ***SOLUTN** —Solves a well conditioned system of linear equations with any number of dependent variable sets. Uses no (diagonal) pivoting with normal-precision floating-point math.

B. ***SOLUTNX** —Solves a well conditioned system of linear equations with any number of dependent variable sets. Uses no (diagonal) pivoting with extended-precision floating-point math.

Extended vs. Normal Precision

Categories I, II, and VI represent a dual collection of programs implemented with 32-bit single- or normal-precision TMS320C30 floating-point arithmetic, and with 40-bit extended-precision TMS320C30 floating-point arithmetic. Some of the normal-precision programs (category I, for example) have been written using the TMS320C30 **RND** instruction for rounding to obtain the optimal precision from the standard floating point TMS320C30 instruction set. This has been done with a slight loss of speed. Such rounding can be carefully eliminated by the user if the additional speed is necessary at the expense of some accuracy.

Extended-precision was implemented on the TMS320C30 by the simple implementation of the 40-by-40 floating-point multiply routine, **FMULTX**. This was necessary since the TMS320C30 has 40-bit addition and subtraction instructions, but the multiply operates only on 32-bit inputs. By using the native add and subtract **FMULTX** and the extended-precision registers R0 to R7, 40-bit floating-point math was effected. All 40-bit constants are stored in two consecutive words in memory. The first word is the normal truncated 32-bit floating-point number. The least significant byte of the second word contains the remaining bottom 8 bits of the extended mantissa. The programs are coded to properly load extended-precision registers with these double-word constants.

The extended-precision versions of the programs in this report may be slower than their normal precision counterparts. When using extended-precision results in R0 from category II programs, note that the results may be stored in memory with or without rounding. A more accurate normal-precision result will generally be obtained by rounding. You should never round before using an extended-precision result as input to another extended-precision program unless special circumstances exist. Note that truncation, not rounding, will occur if an extended-precision register is moved to any 32-bit register or any memory location. This will generally cause loss of accuracy in the amount of the value of the least significant bit of the mantissa.

Program Utilization

Since all programs in this collection are intended to be invoked by a **CALL** instruction, you must have the stack pointer (SP register) appropriately set to an available memory area, preferably in internal RAM. Programs in categories I and II save and restore the data page register DP by using the stack area pointed to by SP. Programs in category III do not alter or use the DP register at all. The programs in categories IV through VI alter but do not restore the DP register.

All of the programs in categories I through III, except for **ILOG2**, are implemented as straight line code. You may wish to disable the instruction cache while these programs are being executing. This will cause no loss of execution speed and will avoid flushing out potentially reusable instructions in the cache. It is beneficial to have the cache enabled when using most of the remaining programs (categories IV through VI) as they generally contain multi-instruction loops.

Programs in categories IV through VI allow input through externally defined variables addresses. The **.global** references indicate these addresses, where the input variable values and/or addresses are located. The starting address of these memory locations is given by the external variable **$PARAMS**. All of the addresses are assumed to be in the same TMS320C30 memory page as **$PARAMS**. If this is not the case, the addresses or the programs should be changed assure that the DP register gets set properly.

Programs in categories IV and VI also allow the use of registers to hold input parameters. The exact registers to be used are found in the program source listings. When using the register input entry point, refer to the program using the *R* prefix on the program name, e.g. *R*SOLUTN. The memory based parameter input entry uses the *M* prefix, e.g. *M*SOLUTN. The **.global** references to the *R* prefix entry points may be deleted if they are not needed.

Function Approximation Techniques

Categories I and II are made up of a collection of elementary mathematical functions numerically approximated using two basic methods. The functions **SIN**, **COS**, **EXP**, **LN**, and **ATAN** are approximated by using polynomials fitted to the various functions over a limited range of the independent variable. The functions **SQRT** and **FPINV** are approximated by iteratively solving a particular non-linear equation. The extended precision versions of these programs (category II) use the same approach with extended-precision arithmetic and resort to more accurate polynomials or more iterations to achieve the desired precision.

Polynomial Approximations

The polynomial approximation method is fundamentally very simple. A limited part of a function is approximated by a polynomial of some order sufficient to obtain the desired accuracy. The polynomial is generally a series of the form:

$$P(n, x) = \sum_{i=0}^{n} \{a[i]x^i\}, \tag{1}$$

where x is the independent variable, n the polynomial order (a fixed integer), and a[i] is a set of n+1 fixed coefficients.

The desired function, say f(x), is then approximated by a particular P(n, x) such that:

$$f(x) = P(n, x) + e(x), \quad x1 < x < xu, \tag{2}$$

where x1 and xu are the limits of the domain of x, and e(x) or e(x)/f(x) is the error function which has been usually minimized in the min-max (equi-ripple) sense. This is done by selecting an appropriate means of calculating the coefficients a[i].

Various techniques and schemes are used in the selection of:

- the approximation interval,
- transformations on the function,
- selection of the polynomial form,
- error minimization criteria, and
- calculation of the coefficients.

See Hastings [2] for an excellent tutorial on this numerical methodology. All of the polynomial approximations used in here were obtained from the National Bureau of Standards reference edited by Abramowitz and Stegun [3].

Non-Linear Equation Approximation

The second method of approximation, using the solution of non-linear equations, is easier to understand. This method requires that a solution for the equation $g(x) = 0$ be found. One means for solving this equation is by Newton-Raphson iteration. This can be understood by considering the Taylor series expansion for $g(x)$:

$$g(x + h) = g(x) + hg'(x) + r(x, h), \tag{3}$$

where $r(x, h)$ is the remainder of the series (which can be assumed to be small), and $g'(x)$ is the derivative of the function $g(x)$. Leaving off the remainder in (3) we get, in terms of incremental values of x, the approximation:

$$g(x[i+1]) = g(x[i]) + \{x[i+1]-x[i]\}g'(x[i]). \tag{4}$$

Solving for $x[i+1]$ in (4) with $g(x[i+1]) = 0$ yields the approximation:

$$x[i+1] = x[i] - g(x[i])/g'(x[i]). \tag{5}$$

Thus, $x[i+1]$ will converge to a solution of $g(x) = 0$. Convergence can be shown to be quadratic, i.e. the error in the approximation at each iteration is proportional to the square of the error in the previous iteration. Minimally, this requires a sufficiently close starting value for x[0] and the condition that $|g'(x)| > 0$ for all iterated values of x.

Math Functions Details

The approximation techniques can be applied to each of the classes of functions. The following sections describe the approximations as they are applied to each function.

Inverse and Square Root Functions

For the problem of computing good approximations to sqrt(c) (**SQRT** and **SQRTX** routines) and 1/c (**FPINV** and **FPINVX** routines), both g(x) and g'(x) must be derived and then use the iteration of equation (5). This is complicated by the restriction that division should be avoided since the TMS320C30 has no divide instructions. For the iteration to find the inverse of c, you can write:

$$g(x[i]) = 1/x[i] - c = 0, \tag{6}$$

which is solved when $1/x = c$ or $x = 1/c$. Taking the derivative of (6) and substituting into (5) and simplifying gives us:

$$x[i+1] = x[i]\{2 - cx[i]\}, \tag{7}$$

which needs no division.

Thus, (7) will converge to 1/c with the accuracy (in digits) for each iteration equal to twice that of the preceding one. Thus, if x[0] approximates 1/c to 3 bits of precision, only three iterations of (7) will yield about $24 = 3(2^3)$ bits of accuracy.

A similar iteration from $f(x) = x^2$ for sqrt(c) can be derived from the formulation:

$$g(x[i]) = x[i]^2 - c = 0, \tag{8}$$

which is solved when $x^2 = c$ or $x = \text{sqrt}(c)$. The solution for (8) leads to the classic square root formula:

$$x[i+1] = 0.5\{c/x[i] + x[i]\}, \tag{9}$$

but this equation uses division. However, the iteration from $f(x) = 1/x^2$ for 1/sqrt(c) can be shown to be:

$$x[i+1] = x[i]\{1.5 - c'x[i]^2\}, \tag{10}$$

where $c' = c/2 = 0.5c$. Though (10) needs no division, the final desired result must be transformed by an extra multiplication by the input c because:

$$\text{sqrt}(c) = c\{1/\text{sqrt}(c)\}. \tag{11}$$

Formula (10) will also converge, in the precision doubling fashion of the Newton-Raphson iteration, given a suitable close starting value for x[0] and the use of sufficiently accurate arithmetic. Note that the extended-precision version routines **FPINVX** and **SQRTX** both use an extra iteration (for a total of 4) to achieve the needed 32-bit accuracy for the 40-bit format.

The initial guess x[0], for the iterations of 1/sqrt(c) and 1/c, may be obtained using an interesting approximation. A TMS320C30 floating-point number $c = (1 + m)2^e$, where $0 \leq m < 1$ and $-127 \leq e \leq 127$. The extra 1, added to the fractional mantissa m, is the implied bit. Then we can write the inverse of c as:

$$1/c = 1/(1 + m)2^{-e}. \tag{12}$$

An excellent approximation for the inverse of the mantissa is:

$$1/(1 + m) = 1 - m/2, \tag{13}$$

which is exact at the end points: $m = 0$ and $m = 1$. Then the approximation for the reciprocal would be:

$$1/c = (1 - m/2)2^{-e}. \tag{14}$$

It turns out that this approximation can be achieved in a single logical operation. If you compute the unlikely value of c' = c XOR 0FF7FFFFFFh, you would complement all bits in c except the sign bit. Including the implied bit and taking the effect of one's complement arithmetic into account results in a final value of:

$$c' = \{1 + (1 - m)\}2^{-(e+1)}, \tag{15}$$

or the desired approximation:

$$c' = (1 - m/)2^{-e} = 1/c. \tag{16}$$

c' gives about 3 bits of precision, which is an excellent seed x[0] for the 1/c iteration. Using e/2, you have a start for the 1/sqrt(c) iteration as well.

Sine and Cosine Functions

The **SIN**, **COS**, **SINX**, and **COSX** (sine and cosine) routines all use the same basic approximation (section 4.3.98, p. 76 in [3]). The series is for sin(x)/x but is obviously transformed by multiplying by x. The polynomial of even terms then is of the form:

$$\sin(x) = x \sum_{i=0}^{5} \{a[2i]x^{2i}\} + xe(x), \tag{16}$$

where $|x| \leq Pi/2$ and $|xe(x)| \leq 2(10^{-9})$. Instead of using another power series for cos(x), you can use the fact that:

$$\cos(x) = \sin(x + Pi/2). \tag{17}$$

The series given by (16) is only accurate in the 1st and 4th quadrants, i.e. $|x| \leq Pi/2$. Sin(x) in the other two quadrants is found from:

$$\sin(x) = \sin(Pi - x). \tag{18}$$

The case for $x < 0$ is expediently handled by using $|x|$ for all calculations except for the final multiply by x in (16).

Exponential Functions

The **EXP** and **EXPX** (exponential) routines use an approximation (see Section 4.2.45, p. 71, in [3]). The expansion is of the form

$$\exp(x) = \sum_{i=0}^{7} \{a[i]x^{i}\} + e(x), \tag{19}$$

where $0 \leq x \leq \ln(2)$ and $|e(x)| \leq 2(10^{-10})$. The series for 2^y is found by substituting $y = x/\ln(2)$ since:

$$\exp(x) = \exp(\ln(2)y) = 2^{y}. \tag{20}$$

The new expansion then becomes:

$$2^y = \sum_{i=0}^{7} \{b[i]y^i\} + e(x), \tag{21}$$

where b[i] = a[i](ln(2)^i). See the coefficients in the **EXP** routine.

Values of exp(x) for x outside the convergent range are found by two means. First for $x < 0$, note the relationship:

$$\exp(-x) = 1/\exp(x), \tag{22}$$

which does require an inverse (see the **FPINV** and **FPINVX** routines). For $y > 1$, let $y = n + f$ where $n = 1, 2, \ldots$ and $0 \leq f < 1$. By substituting y in (20), you get

$$\exp(x) = 2^{n+f} = (2^f)(2^n). \tag{23}$$

Natural Log Functions

The **LN** and **LNX** (natural or base e logarithm) routines use the approximation from [3] (section 4.1.44, p. 69). The expansion comes in the form:

$$\ln(1 + x) = \sum_{i=1}^{8} \{a[i]x^i\} + e(x), \tag{24}$$

where $0 \leq x \leq 1$ and $|e(x)| \leq 3(10^{-8})$. The expansion for ln(y) can be used if the transformation $y = x - 1$ is applied.

Values of ln(x) for x outside the convergent range are found in the following way. First, make the substitution $x = f(2^n)$ for $1 \leq f < 2$ and $n = 0, 1, \ldots$, and then write:

$$\log2(x) = \log2(f2^n) = n + \log2(f), \tag{25}$$

where log2(x) is the log base 2 of x. Using the relationship that log2(x) = ln(x)/ln(2), you get the equation

$$\ln(x) = \ln(f) + n\ln(2). \tag{26}$$

Arctangent Functions

The **ATAN** and **ATANX** (arc or inverse tangent) routines use the approximation from section 4.4.49, p. 81 in [3]. The series with only even terms for atan(x)/x is transformed to

$$\text{atan}(x) = x \sum_{i=0}^{8} \{a[2i]x^{2i}\} + xe(x), \tag{27}$$

where $-1 \leq x \leq 1$ and $|xe(x)| \leq 2(10^{-8})$. Values for atan(x) for x outside the convergent range are obtained by noting the following identity:

$$\text{atan}(x) = \text{atan}((x - 1)/(x + 1)) + \text{Pi}/4. \tag{28}$$

Using the bilinear transformation $y = (x - 1)/(x + 1)$ assures, at the expense of a divide operation, that $y \leq 1$ for $x \geq 1$. The case for $x < 0$ is expediently handled by using $|x|$ for all calculations except for the final multiply by x in (27).

Divide and Multiply Functions

The last group of routines in category I and II are those for the additional arithmetic functions **FDIV** and **FDIVX** (floating-point divides), and **FMULTX** (extended-precision floating-point multiply). The divide operation for the TMS320C30, a = b/c is done by calculating the reciprocal or inverse of the divisor c. Then you compute

$$a = b(1/c). \tag{29}$$

For a normal-precision divide, FDIV finds 1/c by a call to FPINV. A subsequent normal TMS320C30 floating-point multiply of the rounded inverse provides a suitable quotient. For an extended-precision divide, FDIVX finds 1/c by a call to FPINVX. The inverse is then extended-precision multiplied by the dividend using FMULTX.

The extended-precision floating-point multiply simulated by FMULTX is the key to the implementation of virtually all of the extended-precision functions. The extended multiply is achieved using the normal floating-point multiply of the TMS320C30. For two extended-precision numbers **xa** and **xb**, you can represent each as the sum of two floating-point numbers: $xa = a + ea(2^{-24})$ and $xb = b + eb(2^{-24})$. The quantities **ea** and **eb** are the one-byte extensions of **xa** and **xb** respectively.

Thus the complete product xc = (xa)(xb) can be expanded and written as

$$xc = (a)(b) + [(a)(eb) + (b)(ea)]2^{-24} + (ea)(eb)2^{-48}. \tag{30}$$

The last term in (30) is always less than the 32-bit precision in the mantissa of the final result. Therefore, you need only to compute the first two terms in the product **xc**. Also, note that all the indicated products in (30) may be computed using a normal-precision native TMS320C30 multiply as long as the terms are collected in extended-precision registers. The additions are also done using the native TMS320C30 add as it is implemented in extended-precision.

Integer Arithmetic Program Details

Integer routines differ from the floating-point versions because they produce only integer results. If the computation can produce fractional values, then the fraction must be truncated to leave only the integer result.

Integer Result Log Base 2

The routine **ILOG2** is a useful utility for computing integer value **m** of the log base 2 of the integer **n**. The result is computed by successive multiplies by 2 (implemented as shifts by 1). The resulting relationship is $n \leq 2m$, such that if log2(n) is not an exact integer, m is rounded up to the next largest integer. This is useful as it allows the determination of m from any value $n > 0$ (e.g. not a power of two) which might require the padding of additional values (zeros) for a radix 2 FFT. This program is very fast because of a delayed branch loop and internally requires only $4(m+1)$ cycles (cached) to do the calculation.

Extended Precision Integer Multiply

The IMULT routine is a modified version of the program EXTMPY in the *TMS320C3x User's Guide* [1]. It has been modified and slightly speeded up. The negation of the final 64-bit product is done in two instructions by direct two's complement negation rather than by using one's complement to simulate the same result. The product is computed by breaking the multiplier and multiplicand up into two 16 bit integers each. Thus the full product **c** of the numbers $a = au(2^{16}) + al$, and $b = au(2^{16}) + bl$ is

$$c = (au)(bu)2^{32} + [(au)(bl) + (bu)(al)]2^{16} + (al)(bl), \qquad (31)$$

where the powers of two indicated are accomplished by shifts. Note that each product in (31) must be represented as a 32-bit integer. The adds in the sum must be done with care to facilitate the carry between the two final 32-bit components of the product.

Integer Divide

The IDIV routine is a modified version of the program DIVI in the *TMS320C3x User's Guide* [1]. It has been modified to return the absolute value of the remainder of the integer division. The remainder was originally computed, but was discarded during the extraction process for the quotient. A few more instructions allow the extraction of both the quotient and remainder from the result of the **SUBC** process. The program **IDIV** may be used for the computation of the modulo function. The output of **IDIV** is the pair $\{q, |r|\} = a/b$, with the property:

$$0 \leq r = (a \text{ modulo } b) < a, \qquad (32)$$

for $a > 0$ and $b > 0$. The complete relationship is, by definition, $a = bq + r$, for positive a and b.

Vector Utility Routines

Vector utilities are functions which operate on arrays of numbers. Some utilities, like dot products and convolutions, are simple. Other utilities, like those presented here, are more involved.

Complex and Complex Conjugate Array Multiplies

The array routine ***CORMULT** computes the point-by-point complex conjugate multiply of two complex arrays. If the arrays are c1 and c2, and are of length n, then:

$$c1[k] \leftarrow c1[k]\text{conj}(c2[k]),\ k = 1, \ldots, n, \tag{33}$$

where $\leftarrow$ means replaces. Each complex array is assumed to be stored as two separate arrays, i.e. $\{c1\} = \{x1, y1\}$ and $\{c2\} = \{x2, y2\}$. In cartesian complex representation, (33) becomes

$$(x1 + iy1) \leftarrow (x1 + iy1)(x2 - iy2), \tag{34}$$

where i represents the imaginary constant sqrt(−1). Separating the real and imaginary parts, we have:

$$x1 \leftarrow x1x2 + y1y2,\ y1 \leftarrow y1x2 - y2x1 \tag{35}$$

This operation can be used for the frequency domain correlation of two FFTs to implement time domain correlation.

On the other hand, the array routine ***CONMULT** computes the point-by-point complex multiply of two complex arrays. If the arrays are c1 and c2, and are each of length n, then

$$c1[k] \leftarrow c1[k](c2[k]),\ k = 1, \ldots, n, \tag{36}$$

In cartesian complex representation, (36) becomes

$$(x1 + iy1) \leftarrow (x1 + iy1)(x2 + iy2). \tag{37}$$

Separating the real and imaginary parts results in

$$x1 \leftarrow x1x2 - y1y2,\ y1 \leftarrow y1x2 + y2x1. \tag{38}$$

This operation can be used for the frequency domain convolution of two FFTs to implement digital filtering.

Complex Array Bit Reversal

The array routine ***CBITREV** executes an in-place bit reverse permutation on two arrays simultaneously. This operation is generally used for index scrambling before a DIT FFT (decimation in time, see CIFFT2), or after a DIF FFT (decimation in frequency, see CFFFT2) for index unscrambling. Therefore, ***CBITREV** is useful in permuting complex arrays stored as two separate arrays which are associated with radix 2 FFTs. The program uses the bit reverse indexing feature of the TMS320C30 to achieve this function. The loop in ***CBITREV** is nearly as efficient in permuting two arrays together as permuting one array alone. This is due to the use of parallel load and store instructions and a delayed (single cycle) conditional branch.

Floating Point Conversions

The array routines ***FMIEEE** and ***TOIEEE** are vectorized versions of their original scalar counterparts **FMIEEE** and **TOIEEE**. Both routines do fast conversions from or to IEEE format by avoiding dealing with special rare cases. Also, both programs convert the numbers in the arrays in-place which destroys the original data. These array versions of the format conversion routines are much faster than calling the scalar version routines in a special loop. These routines also have their own internal, shared constant table for conversions.

Vector Primitives

The array routines ***VECMULT**, ***CONMOV**, and ***VECMOV** are a useful suite of efficient programs for simple array operations. The first routine, ***VECMULT**, performs the simple operation x[k] ← x[k]c which is a scalar-vector multiply useful in uniformly scaling an array by a constant c. You can use this for scaling arrays after an inverse FFT by choosing c = 1/n. The next routine, ***CONMOV**, performs the operation x[k] ← c which is useful in filling or initializing any portion of an array to a single constant c. The last routine, ***VECMOV** performs the simple operation x[k] ← y[k], an array move, and is, therefore, generally useful.

FFT Routines

This category contains the two complementary radix 2 complex FFT programs **CFFFT2** and **CIFFT2**. These programs differ from previously available TMS320C30 FFT programs in that they operate on complex arrays which are stored as two separate and independent real arrays. Both routines do the FFTs in-place and do no index permutations or constant scaling (multiplication). Also these programs require only a 3/4 cycle external, pre-computed sine table. As with previous FFT programs, these, too, have a special multiply-less butterfly loop for the occurrence of unity twiddle or complex rotation factors.

The routine **CFFFT2** is a DIF radix 2 complex forward FFT program and thus assumes a normally indexed pair of input arrays. The output array is bit-reverse permuted and normally must be unscrambled to be of any use (see ***CBITREV**). The routine **CIFFT2** is a DIT radix 2 inverse FFT program and thus assumes a bit-reverse indexed pair of input arrays. A normally indexed complex frequency spectrum must be bit-reverse scrambled before using **CIFFT2** (again, see ***CBITREV**). On the other hand, the output from this inverse FFT is in normal indexed order, but lacks the traditional scaling by the factor of 1/n. Therefore, back-to-back calls of **CFFFT2** and **CIFFT2** will return the original complex array (in proper order) but multiplied by a factor of n. Consult the handbook by Burrus and Parks [4] for additional FFT algorithm details.

Linear Algebra Routines

The routines ***SOLUTN** and ***SOLUTNX** are the normal- and extended-precision implementations of the algorithm for solving simultaneous linear equations. This algorithm is the modified Gauss-Jordan elimination without (off diagonal) pivoting. This is a simple algorithm which is intended for use with *well-conditioned* systems of dense linear equations of moderate size. Well conditioned means that the system of linear equations is linearly independent or non-singular. This subject and further algorithm details are to be found in chapter 2 of [5] by Press et al, or any other book on the numerical techniques of linear algebra. This algorithm is suitable for a wide range of problems requiring the solution of a system of linear equations, e.g. exact or least squares polynomial fitting.

A simple system of linear equations has the form:

$$\begin{aligned}
&A[1,\ 1]x[1] + A[1,\ 2]x[2] + \ldots + A[1,\ n]x[n] = y[1], \\
&A[2,\ 1]x[1] + A[2,\ 2]x[2] + \ldots + A[2,\ n]x[n] = y[2], \\
&\quad\vdots \qquad\qquad\quad \vdots \qquad\qquad\quad \vdots \qquad\quad \vdots \\
&A[n,\ 1]x[1] + A[n,\ 2]x[2] + \ldots + A[n,\ n]x[n] = y[n].
\end{aligned} \tag{39}$$

Symbolically, you may write A = A[i, j] as the n x n matrix of coefficients, and x = x[i] as the unknown independent variable (column) vector, and y = y[j] as the dependent variable (row) vector. Thus (39) can be written in short hand form as Ax = y or Ax − y = 0, where the multiplication indicated is a matrix-vector multiply. The fundamental problem in linear algebra, then, is to find the solution vector x. In fact, you may desire to find the m different solutions to m sets of linear equations which share the same coefficient matrix A, i.e. Ax[k] = y[k], for k = 1, . . . , m.

You can solve the general problem just stated by using *SOLUTN, or with more accuracy with *SOLUTNX. This is done by constructing a tableau B (table of coefficients) which is simply the coefficient matrix A (in row major storage format) with the negative of the y vector(s) appended (:) as m extra columns to A. Thus you would have B = A : −y, as your problem, where B is a n by n+m matrix and typically m = 1. Thus, for the common case of m = 1, the input array B can be written as:

$$\begin{array}{l} A[1,\ 1],\ A[1,\ 2],\ \ldots,\ A[1,\ n],\ -y[1], \\ A[2,\ 1],\ A[2,\ 2],\ \ldots,\ A[2,\ n],\ -y[2], \\ \quad\vdots \qquad\quad \vdots \qquad\qquad \vdots \qquad\quad \vdots \\ A[n,\ 1],\ A[n,\ 2],\ \ldots,\ A[n,\ n],\ -y[n]. \end{array} \tag{40}$$

After the ***SOLUTN** routine is executed, the matrix C = A' : x appears, where the column(s) beyond the original coefficients A (the y[k] vectors) have been replaced by the solution vector(s) x[k]. The new matrix A' is a partially computed version of the inverse of the matrix A. The complete inverse of A, which is normally computed by the standard Gauss-Jordan scheme, is rarely needed. Therefore, a faster modified algorithm has been used which does about half the work.

This simple method used for solving systems of linear equations has two restrictions.

1. As the pivoting operation (exchange of x and y variables) always starts with A[1, 1] and proceeds down the diagonal, A[1, 1] must be non-zero. This is because, in the exchange process, you must divide by the pivot element. A zero coefficient at A[1, 1] may be moved by reordering the variable indices by appropriately swapping rows and columns in A and in y.

2. The maximum absolute value of the elements in A must be approximately unity. This is necessary to assure that no pivot element is encountered which is smaller in magnitude than 10^{-8} for ***SOLUTN**, and 10^{-10} for ***SOLUTNX**. This restriction monitors the system condition and assures an adequately accurate solution, but the final solution should always be verified by substitution. This is done by inspecting the elements of the error vector e = Ax − y computed by using the solution x, and the original A and y.

Summary

This report presented a set of routines that can be used in digital signal processing applications. The appendix contains the source code of these routines. This source code can also be obtained from the Texas Instruments Electronic Bulletin Board (713) 274-2323. If there are comments or corrections, please contact the author of this report:

Mr. Gary Sitton
Gas Light Software
5211 Yarwell
Houston, TX 77096
Tel (713) 729-1257

References

(1) *TMS320C3x User's Guide* (literature number SPRU031), Texas Instruments, Dallas, TX, August 1988.

(2) Hastings, C. Jr., "Approximations for Digital Computers", Princeton University Press, Princeton N.J., 1955.

(3) Abramowitz, M. and Stegun, I.A. (Editors), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards (Applied Mathematics Series 55), Washington D.C., 1964.

(4) Burrus, C.S. and Parks, T.W., "DFT/FFT and Convolution Algorithms", John Wiley and Sons, New York N.Y., 1985.

(5) Press, W.H., Flannery, B.P., Teukolsy, S.A., and Vettering, W.T., *Numerical Recipes in C - The Art of Scientific Programming*, Cambridge University Press, Cambridge England, 1988.

Appendix

Program Library

I. $MATH.ASM
II. $MATHX.ASM
III. $MATHI.ASM
IV. $VECTOR.ASM
V. $FFT2.ASM
VI. $LINALG.ASM

Appendix A

```
*************************************************************************
*
*   PROGRAM: $MATH.ASM
*
*   NORMAL FLOATING-POINT (32-BIT) MATH FUNCTIONS
*
*   $MATH.ASM CONSISTS OF THE FOLLOWING ROUTINES:
*
*       SIN   - COMPUTES A 7D SINE(X) FOR ALL X IN RADIANS.
*
*       COS   - COMPUTES A 7D COSINE(X) FOR ALL X IN RADIANS.
*
*       EXP   - COMPUTES A 7D EXP(X) FOR ALL |X| =< 88.
*
*       LN    - COMPUTES A 7D LN(X) FOR ALL X > 0.
*
*       ATAN  - COMPUTES A 7D ATAN(X) FOR ALL X IN RADIANS.
*
*       SQRT  - COMPUTES AN 8D SQRT(X) FOR ALL X >= 0.
*
*       FPINV - COMPUTES AN 8D 1/X FOR ALL X /= 0.
*
*       FDIV  - COMPUTES AN 8D X/Y FOR ALL X AND ALL Y /= 0.
*
*************************************************************************
```

```
*****************************************************
*   PROGRAM: SIN                                    *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   SINE FUNCTION: R0 <= SIN(R0).                   *
*                                                   *
*   APPROXIMATE ACCURACY: 7 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: NONE.                       *
*   REGISTERS FOR INPUT: R0 (ARGUMENT IN RADIANS).  *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: AR0, IR0, AND R0-4.          *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: NONE.                          *
*   EXECUTION CYCLES (MIN, MAX): 44 , 44.           *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  SIN
        .GLOBL  ECOS

;       INTERNAL CONSTANTS

        .DATA

NORM    .FLOAT  0.636619772     ; 2/PI

;       POLYNOMIAL COEFFS. FOR SIN(X*2/PI), -1 < X < 1

SHFT    .FLOAT  1.570796327     ; C1 (PI/2)
        .FLOAT  -0.6459640968   ; C3
        .FLOAT  0.07969260878   ; C5
        .FLOAT  -0.00468166687  ; C7
        .FLOAT  0.00016025884   ; C9
COF     .FLOAT  -0.000003433338 ; C11

ACOF    .WORD   COF             ; ADDRESS OF COEFFS.

CON     .FLOAT  -1.0, 0.0, 1.0, 0.0 ; MAPPING CONSTANTS

ACON    .WORD   CON             ; ADDRESS OF CONSTS.

        .TEXT

;       START OF SIN PROGRAM

SIN:

        PUSH    DP              ; SAVE DP
        LDP     @ACOF           ; LOAD DATA PAGE POINTER
        RND     R0              ; ROUND X
        LDF     R0,R4           ; R4 <= X

;       COSINE ENTRY POINT

ECOS:

;       SCALE AND MAP VARIABLE X

        ABSF    R0              ; R0 <= |X|
        LDF     R0,R1           ; R1 <= RND |X|
        MPYF    @NORM,R1        ; R1 <= X*2/PI
        FIX     R1,IR0          ; IR0 <= INTEGER QUADRANT Q
        FLOAT   IR0,R2          ; R2 <= FLOATING QUADRANT Q
        SUBF    R2,R1,R0        ; R0 <= X, -1 < X < 1
        NEGF    R0,R3           ; R3 <= -X
        ADDI    1,IR0           ; IR0 <= Q + 1
        AND     3,IR0           ; IR0 <= TABLE INDEX
        TSTB    2,IR0           ; LOOK AT 2ND LSB
        LDFNZ   R3,R0           ; IF 1 THEN R0 <= -X
        LDI     @ACON,AR0       ; AR1 -> CONST. TABLE
        ADDF    *+AR0(IR0),R0   ; FINAL MAPPING, R0 <= X + C
        NEGF    R0,R3           ; R3 <= -X
        LDI     @ACOF,AR0       ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED (ODD) SERIES

        MPYF    R0,R0,R2        ; R2 <= X**2
        RND     R2              ; ROUND X**2

        MPYF    *AR0--,R2,R1    ; R1 <= X**2*C11
        ADDF    *AR0--,R1       ; R1 <= C9 + R1

        MPYF    R2,R1           ; R1 <= X**2*(C9 + R1)
        ADDF    *AR0--,R1       ; R1 <= C7 + R1

        MPYF    R2,R1           ; R1 <= X**2*(C7 + R1)
        ADDF    *AR0--,R1       ; R1 <= C5 + R1

        RND     R1              ; ROUND BEFORE *
        MPYF    R2,R1           ; R1 <= X**2*(C5 + R1)
        ADDF    *AR0--,R1       ; R1 <= C3 + R1

        RND     R1              ; ROUND BEFORE *
        MPYF    R2,R1           ; R1 <= X**2*(C3 + R1)
        ADDF    *AR0,R1         ; R1 <= C1 + R1
```

```
;       FINISH UP SERIES AND RETURN

        LDF     R4,R4           ; TEST ORIGINAL X
        LDFN    R3,R0           ; IF X < 0 THEN R0 <= -X
        POP     R2              ; R2 <= RETURN ADDRESS
        BUD     R2              ; RETURN (DELAYED)
        RND     R0              ; ROUND BEFORE *
        RND     R1              ; ROUND BEFORE *
        MPYF    R1,R0           ; R1 <= X*(C1 + R1)
```

```
*****************************************************
*   PROGRAM: COS                                    *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   COSINE FUNCTION: R0 <= COS(R0).                 *
*                                                   *
*   APPROXIMATE ACCURACY: 7 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: NONE.                       *
*   REGISTERS FOR INPUT: R0 (ARGUMENT IN RADIANS).  *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: AR0, IR0, AND R0-4.          *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: ECOS (SIN).                    *
*   EXECUTION CYCLES (MIN, MAX): 46 , 46.           *
*                                                   *
*   NOTE: USES SHFT CONSTANT FROM SIN PROGRAM!      *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  COS
        .GLOBL  ECOS

        .TEXT

;       START OF COS PROGRAM

COS:

        PUSH    DP              ; SAVE DP
        LDP     @ACOF           ; LOAD DATA PAGE POINTER

        BRD     ECOS            ; R0 <= COS(X) = SIN(X'), (DELAYED)
        RND     R0              ; ROUND X
        ADDF    @SHFT,R0        ; R0 <= X' = X + PI/2
        LDF     R0,R4           ; R4 <= X'

;       RETURN OCCURS FROM SIN !
```

```
*****************************************************
*  PROGRAM: EXP                                     *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              MARCH 1989.                          *
*                                                   *
*  EXPONENTIAL FUNCTION: R0 <= EXP(R0).             *
*                                                   *
*  APPROXIMATE ACCURACY: 7 DECIMAL DIGITS.          *
*  INPUT RESTRICTIONS: |R0| <= 88.0.                *
*  REGISTERS FOR INPUT: R0.                         *
*  REGISTERS USED AND RESTORED: DP AND SP.          *
*  REGISTERS ALTERED: R0-4.                         *
*  REGISTERS FOR OUTPUT: R0.                        *
*  ROUTINES NEEDED: FPINV.                          *
*  EXECUTION CYCLES (MIN, MAX): 44 (R0 <= 0), 70.   *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  EXP
        .GLOBL  FPINV

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFF. FOR 2**-X

ENRM    .FLOAT  1.442695041     ; 1/LN(2)

;       POLYNOMIAL COEFFS. FOR 2**-X, 0 <= X < 1.

        .FLOAT  1.0000000000    ; C0
        .FLOAT  -0.693147180    ; C1
        .FLOAT  0.240226469     ; C2
        .FLOAT  -0.055503654    ; C3
        .FLOAT  0.009615978     ; C4
        .FLOAT  -0.001328240    ; C5
        .FLOAT  0.000147491     ; C6
C7      .FLOAT  -0.000010863    ; C7

AC7     .WORD   C7

        .TEXT

;       START OF EXP PROGRAM

EXP:

;       SCALE VARIABLE X

        PUSH    DP              ; SAVE DP
        LDP     @AC7            ; LOAD DATA PAGE POINTER
        RND     R0              ; ROUND X
        NEGF    R0,R2           ; R2 <= -X
        LDF     R0,R1           ; R1 <= X
        LDFN    R2,R1           ; IF X < 0 THEN R1 <= |X|
        MPYF    @ENRM,R1        ; R1 <= X = |X|/LN(2)
        FIX     R1,R3           ; R3 <= I = INTEGER OF X
        FLOAT   R3,R0           ; R0 <= FLT. PT. I
        SUBF    R0,R1           ; R1 <= FRACTION OF |X|, 0 <= X < 1
        NEGI    R3              ; R3 <= -I
        LSH     24,R3           ; MOVE -I TO EXP.
        PUSH    R3              ; SAVE AS INT.
        POPF    R3              ; R3 <= FLT. PT. 2**-I
        LDI     @AC7,AR0        ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED SERIES

        RND     R1              ; ROUND BEFORE *
        MPYF    *AR0--,R1,R0    ; R0 <= X*C7
        ADDF    *AR0--,R0       ; R0 <= C6 + R0

        MPYF    R1,R0           ; R0 <= X*(C6 + R0)
        ADDF    *AR0--,R0       ; R0 <= C5 + R0

        MPYF    R1,R0           ; R0 <= X*(C5 + R0)
        ADDF    *AR0--,R0       ; R0 <= C4 + R0

        MPYF    R1,R0           ; R0 <= X*(C4 + R0)
        ADDF    *AR0--,R0       ; R0 <= C3 + R0

        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X*(C3 + R0)
        ADDF    *AR0--,R0       ; R0 <= C2 + R0

        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X*(C2 + R0)
        ADDF    *AR0--,R0       ; R0 <= C1 + R0

        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X*(C1 + R0)

;       TEST FOR X < 0 AND RETURN

        LDF     R2,R2           ; TEST ORIGINAL -X
        BND     FPINV           ; IF -X < 0 THEN R0 <= 1/X, (DELAYED)
        ADDF    *AR0,R0         ; R0 <= 2**-X = C0 + R0
        RND     R0              ; ROUND BEFORE *
        MPYF    R3,R0           ; R0 <= 2**-(I + X)

        RETS                    ; RETURN (IF NO FPINV BRANCH)
```

```
*************************************************
*  PROGRAM: LN                                  *
*                                               *
*  WRITTEN BY: GARY A. SITTON                   *
*              GAS LIGHT SOFTWARE               *
*              HOUSTON, TEXAS                   *
*              MARCH 1989.                      *
*                                               *
*  LOGARITHM FUNCTION BASE E: R0 <= LN(R0).     *
*                                               *
*  APPROXIMATE ACCURACY: 7 DECIMAL DIGITS.      *
*  INPUT RESTRICTIONS: R0 > 0.0.                *
*  REGISTERS FOR INPUT: R0.                     *
*  REGISTERS USED AND RESTORED: DP AND SP.      *
*  REGISTERS ALTERED: AR0 AND R0-3.             *
*  REGISTERS FOR OUTPUT: R0.                    *
*  ROUTINES NEEDED: NONE.                       *
*  EXECUTION CYCLES (MIN, MAX): 43 , 43.        *
*************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  LN

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFFS. FOR LN(1+X)

LNRM    .FLOAT  0.6931471806    ; LN(2)
C0      .FLOAT  1.0000000000    ; C0 (1.0)

;       POLYNOMIAL COEFFS. FOR LN(1+X), 0 <= X < 1.

        .FLOAT  0.9999964239    ; TOP OF C1
        .FLOAT  -0.4998741238   ; TOP OF C2
        .FLOAT  0.3317990258    ; TOP OF C3
        .FLOAT  -0.2407338084   ; TOP OF C4
        .FLOAT  0.1676540711    ; TOP OF C5
        .FLOAT  -0.0953293897   ; TOP OF C6
        .FLOAT  0.0360884937    ; TOP OF C7
C8      .FLOAT  -0.0064535442   ; TOP OF C8

AC8     .WORD   C8

        .TEXT

;       START OF LN PROGRAM

LN:

        LDF     R0,R0           ; TEST X
        RETSLE                  ; RETURN NOW IF X <= 0

;       SCALE VARIABLE X

        PUSH    DP              ; SAVE DP
        LDP     @AC8            ; LOAD DATA PAGE POINTER
        PUSHF   R0              ; SAVE AS FLT. PT.
        POP     R3              ; R3 <= INTEGER FORMAT
        ASH     -24,R3          ; R3 <= E = SIGNED EXP.
        FLOAT   R3,R1           ; R1 <= FLT. PT. E VALUE
        LDF     @C0,R2          ; R2 <= 1.0
        LDE     R2,R0           ; EXP. R0 <= 0 (1 <= X < 2)
        SUBRF   R0,R2           ; R2 <= X - 1 (0 <= X < 1)
        LDF     @LNRM,R0        ; R0 <= LN(2)
        MPYF    R1,R0           ; R0 <= E*LN(2)
        LDF     R0,R3           ; R3 <= E*LN(2)
        LDI     @AC8,AR0        ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED SERIES

        RND     R2,R1           ; R1 <= RND X
        MPYF    *AR0--,R1,R0    ; R0 <= X*C8
        ADDF    *AR0--,R0       ; R0 <= C7 + R0

        MPYF    R1,R0           ; R0 <= X*(C7 + R0)
        ADDF    *AR0--,R0       ; R0 <= C6 + R0

        MPYF    R1,R0           ; R0 <= X*(C6 + R0)
        ADDF    *AR0--,R0       ; R0 <= C5 + R0

        MPYF    R1,R0           ; R0 <= X*(C5 + R0)
        ADDF    *AR0--,R0       ; R0 <= C4 + R0

        MPYF    R1,R0           ; R0 <= X*(C4 + R0)
        ADDF    *AR0--,R0       ; R0 <= C3 + R0

        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X*(C3 + R0)
        ADDF    *AR0--,R0       ; R0 <= C2 + R0

        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X*(C2 + R0)
        ADDF    *AR0--,R0       ; R0 <= C1 + R0

;       ADD IN SCALED EXPONENT AND RETURN

        POP     R2              ; R2 <= RETURN ADDRESS
        BUD     R2              ; RETURN (DELAYED)
        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X*(C1 + R0)
        ADDF    R3,R0           ; R0 <= LN(X) + E*LN(2)
```

```
*****************************************************
*  PROGRAM: ATAN                                    *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              MARCH 1989.                          *
*                                                   *
*  ARC TANGENT FUNCTION: R0 <= ATAN(R0).            *
*                                                   *
*  APPROXIMATE ACCURACY: 7 DECIMAL DIGITS.          *
*  INPUT RESTRICTIONS: NONE.                        *
*  REGISTERS FOR INPUT: R0.                         *
*  REGISTERS USED AND RESTORED: DP AND SP.          *
*  REGISTERS ALTERED: AR0, IR0, AND R0-4.           *
*  REGISTERS FOR OUTPUT: R0 (IN RADIANS).           *
*  ROUTINES NEEDED: FDIV.                           *
*  EXECUTION CYCLES (MIN, MAX): 30 (|ATAN| <= 1), 69.  *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  ATAN
        .GLOBL  FDIV

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFFS. FOR ATAN(X)

        .FLOAT  -0.7853981635   ; -PI/4
        .FLOAT  0.7853981635    ; PI/4
        .FLOAT  0.0000000000    ; ZERO

;       POLYNOMIAL COEFFS. FOR ATAN(X), -1 <= X <= 1.

C1      .FLOAT  1.0000000000    ; C1
        .FLOAT  -0.3333314528   ; C3
        .FLOAT  0.1999355085    ; C5
        .FLOAT  -0.1420889944   ; C7
        .FLOAT  0.1065626393    ; C9
        .FLOAT  -0.0752896400   ; C11
        .FLOAT  0.0429096138    ; C13
        .FLOAT  -0.0161657367   ; C15
C17     .FLOAT  0.0028662257    ; C17

AC17    .WORD   C17

        .TEXT

;       START OF ATAN PROGRAM

ATAN:

;       SCALE VARIABLE X

        PUSH    DP              ; SAVE DP
        LDP     @AC17           ; LOAD DATA PAGE POINTER
        ABSF    R0,R2           ; R2 <= |X|
        SUBF    @C1,R2          ; R2 <= |X| - 1
        BLED    SKIP            ; IF |X| > 1 THEN SCALE (DELAYED)
        RND     R0,R3           ; R3 <= RND X
        RND     R0,R1           ; R1 <= RND X
        LDI     0,IR0           ; IR0 <= 0, POST SCALE INDEX

;       SCALE FOR |X| > 1

        PUSHF   R1              ; SAVE RND X
        ABSF    R0,R1           ; R1 <= |X|
        ADDF    @C1,R1          ; R1 <= |X| + 1
        LDF     R2,R0           ; R0 <= |X| - 1
        CALL    FDIV            ; R0 <= (|X| - 1)/(|X| + 1)

;       TEST FOR X' < 0

        POPF    R2              ; GET ORIGINAL X
        BGED    SKIP            ; IF X < 0 THEN R0 <= -X' (DELAYED)
        RND     R0,R3           ; R3 <= RND X'
        RND     R0,R1           ; R1 <= RND X'
        SUBI    1,IR0           ; IR0 <= -1, (PI/4)

        NEGF    R3,R3           ; R3 <= -X'
        SUBI    1,IR0           ; IR0 <= -2, (-PI/4)

SKIP:   MPYF    R1,R1,R0        ; R0 <= X**2
        LDI     @AC17,AR0       ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED (ODD) SERIES

        RND     R0,R1           ; R1 <= RND X**2
        MPYF    *AR0--,R1,R0    ; R0 <= X**2*C17
        ADDF    *AR0--,R0       ; R0 <= C15 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C15 + R0)
        ADDF    *AR0--,R0       ; R0 <= C13 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C13 + R0)
        ADDF    *AR0--,R0       ; R0 <= C11 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C11 + R0)
        ADDF    *AR0--,R0       ; R0 <= C9 + R0

        RND     R0              ; ROUND BEFORE *
        MPYF    R1,R0           ; R0 <= X**2*(C9 + R0)
        ADDF    *AR0--,R0       ; R0 <= C7 + R0
```

```
RND     R0              ; ROUND BEFORE *
MPYF    R1,R0           ; R0 <= X**2*(C7 + R0)
ADDF    *AR0--,R0        ; R0 <= C5 + R0

RND     R0              ; ROUND BEFORE *
MPYF    R1,R0           ; R0 <= X**2*(C5 + R0)
ADDF    *AR0--,R0       ; R0 <= C3 + R0

RND     R0              ; ROUND BEFORE *
MPYF    R1,R0           ; R0 <= X**2*(C3 + R0)
ADDF    *AR0--,R0,R1    ; R1 <= C1 + R0

FINISH UP, POST SCALE BY C AND RETURN

POP     R2              ; R2 <= RETURN ADDRESS
BUD     R2              ; RETURN (DELAYED)
RND     R1              ; ROUND BEFORE *
MPYF    R3,R1,R0        ; R0 <= ATAN(X) = X*(1 + R0)
ADDF    *++AR0(IR0),R0  ; R0 <= ATAN(X) + C (0.0, PI/4 OR -PI/4)
```

```
*****************************************************
*   PROGRAM: SQRT                                   *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   SQUARE ROOT FUNCTION: R0 <= SQRT(R0).           *
*                                                   *
*   APPROXIMATE ACCURACY: 8 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: R0 >= 0.0.                  *
*   REGISTERS FOR INPUT: R0.                        *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: R0-4.                        *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: NONE.                          *
*   EXECUTION CYCLES (MIN, MAX): 49 , 49.           *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  SQRT

;       INTERNAL CONSTANTS

        .DATA

CNST1   .SET    0.5
CNST2   .SET    1.5
CNST3   .FLOAT  1.103553391     ; ADJUSTED 1.0
CNST4   .FLOAT  0.780330086     ; ADJUSTED SQRT(1/2)

SMSK    .WORD   0FF7FFFFFH

        .TEXT

;       START OF SQRT PROGRAM.

SQRT:

        LDF     R0,R3           ; TEST AND SAVE V
        RETSLE                  ; RETURN NOW IF V <= 0

;       GET APPROXIMATION TO 1/V.  FOR V = (1+M)*2**E
;       AND 0 <= M < 1, FOR E EVEN: X[0] = (1-M/2)*2**-E/2
;       AND FOR E ODD: X[0] = SQRT(1/2)*(1-M/2)*2**-E/2

        PUSH    DP              ; SAVE DP
        LDP     @SMSK           ; LOAD DATA PAGE POINTER
        PUSHF   R0              ; SAVE V AS FLT. PT. V = (1+M)*2**E
        POP     R2              ; R2 <= V AS INTEGER
        XOR     @SMSK,R2        ; R2 <= COMPLEMENT ALL BUT SIGN
        LDI     R2,R1           ; R1 <= (1-M/2)*2**-E
```

```
        LDI     R2,R4           ; R4 <= R1
        LSH     8,R1            ; R1 <= R1 EXP. REMOVED
        ASH     -1,R2           ; R2 <= R2 WITH -E/2 EXP.
        PUSH    R2              ; SAVE R2 AS INTEGER
        POPF    R2              ; R2 <= FLT. PT.
        LDE     R2,R1           ; R1 <= (1-M/2)*2**-E/2
        LDF     @CNST3,R2       ; R2 <= 1.1... FOR ODD E
        LSH     7,R4            ; TEST LSB OF E (AS SIGN)
        LDFNN   @CNST4,R2       ; IF E EVEN R2 <= 0.78...
        MPYF    R2,R1           ; R1 <= CORRECTED ESTIMATE
        POP     DP              ; UNSAVE DP

;       GENERATE V/2 (USES MPYF).

        MPYF    CNST1,R0        ; R0 <= V/2 TRUNC.
        RND     R0              ; R0 <= RND V/2

;       NEWTON ITERATION FOR Y(X) = X - V**-2 = 0 ...

        MPYF    R1,R1,R2        ; R2 <= X[0]**2
        MPYF    R0,R2           ; R2 <= (V/2) * X[0]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[0]**2
        MPYF    R2,R1           ; R1 <= X[1] = X[0] * (1.5 - (V/2)*X[0]**2)
;
        MPYF    R1,R1,R2        ; R2 <= X[1]**2
        MPYF    R0,R2           ; R2 <= (V/2) * X[1]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[1]**2
        MPYF    R2,R1           ; R1 <= X[2] = X[1] * (1.5 - (V/2)*X[1]**2)
;
        MPYF    R1,R1,R2        ; R2 <= X[2]**2
        MPYF    R0,R2           ; R2 <= (V/2) * X[2]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[2]**2
        MPYF    R2,R1           ; R1 <= X[3] = X[2] * (1.5 - (V/2)*X[2]**2)
;
        RND     R1              ; ROUND BEFORE *
        MPYF    R1,R1,R2        ; R2 <= X[3]**2
        RND     R2              ; ROUND BEFORE *
        MPYF    R0,R2           ; R2 <= (V/2) * X[3]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[3]**2
        RND     R2              ; ROUND BEFORE *
        MPYF    R2,R1           ; R1 <= X[4] = X[3] * (1.5 - (V/2)*X[3]**2)

;       INVERT FINAL RESULT AND RETURN

        POP     R2              ; R2 <= RETURN ADDRESS
        BUD     R2              ; RETURN (DELAYED)
        RND     R3              ; ROUND BEFORE *
        RND     R1              ; ROUND BEFORE *
        MPYF    R1,R3,R0        ; R0 = SQRT(V) = V*SQRT(1/V)
```

```
*************************************************
*  PROGRAM: FPINV                               *
*                                               *
*  WRITTEN BY: GARY A. SITTON                   *
*              GAS LIGHT SOFTWARE               *
*              HOUSTON, TEXAS                   *
*              MARCH 1989.                      *
*                                               *
*  FLOATING POINT INVERSE: R0 <= 1/R0           *
*                                               *
*  APPROXIMATE ACCURACY: 8 DECIMAL DIGITS.      *
*  INPUT RESTRICTIONS: R0 != 0.0.               *
*  REGISTERS FOR INPUT: R0.                     *
*  REGISTERS USED AND RESTORED: DP AND SP.      *
*  REGISTERS ALTERED: R0-2 AND R4.              *
*  REGISTERS FOR OUTPUT: R0.                    *
*  ROUTINES NEEDED: NONE.                       *
*  EXECUTION CYCLES (MIN, MAX): 33 , 33.        *
*************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  FPINV

;       INTERNAL CONSTANTS

        .DATA

ONE     .SET    1.0
TWO     .SET    2.0

MSK     .WORD   0FF7FFFFFH

        .TEXT

;       START OF FPINV PROGRAM

FPINV:

        LDF     R0,R0           ; TEST F
        RETSZ                   ; RETURN NOW IF F = 0

;       GET APPROXIMATION TO 1/F.  FOR F = (1+M) * 2**E
;       AND 0 <= M < 1, USE: X[0] = (1-M/2) * 2**-E

        PUSH    DP              ; SAVE DATA PAGE POINTER
        LDP     @MSK            ; LOAD DATA PAGE POINTER
        PUSHF   R0              ; SAVE AS FLT. PT. F = (1+M) * 2**E
        POP     R1              ; FETCH BACK AS INTEGER
        XOR     @MSK,R1         ; COMPLEMENT E & M BUT NOT SIGN BIT
        PUSH    R1              ; SAVE AS INTEGER, AND BY MAGIC...
        POPF    R1              ; R1 <= X[0] = (1-M/2) * 2**-E.
        POP     DP              ; UNSAVE DP
```

```
;       NEWTON ITERATION FOR: Y(X) = X - 1/F = 0 ...

        MPYF    R1,R0,R4        ; R4 <= F * X[0]
        SUBRF   TWO,R4          ; R4 <= 2 - F * X[0]
        MPYF    R4,R1           ; R1 <= X[1] = X[0] * (2 - F * X[0])

        MPYF    R1,R0,R4        ; R4 <= F * X[1]
        SUBRF   TWO,R4          ; R4 <= 2 - F * X[1]
        MPYF    R4,R1           ; R1 <= X[2] = X[1] * (2 - F * X[1])

        MPYF    R1,R0,R4        ; R4 <= F * X[2]
        SUBRF   TWO,R4          ; R4 <= 2 - F * X[2]
        MPYF    R4,R1           ; R1 <= X[3] = X[2] * (2 - F * X[2])

;       FOR THE LAST ITERATION: X[4] = (X[3] * (1 - (F * X[3]))) + X[3]

        RND     R0,R4           ; ROUND F BEFORE LAST MULTIPLY
        RND     R1,R0           ; ROUND X[3] BEFORE MULTIPLIES
        MPYF    R0,R4           ; R4 <= F * X[3] = 1 + EPS

;       FINISH ITERATION AND RETURN

        POP     R2              ; R2 <= RETURN ADDRESS
        BUD     R2              ; RETURN (DELAYED)
        SUBRF   ONE,R4          ; R4 <= 1 - F * X[3] = EPS
        MPYF    R0,R4           ; R4 <= X[3] * EPS
        ADDF    R4,R1,R0        ; R0 <= X[4] = (X[3]*(1 - (F*X[3]))) + X[3]
```

```
*****************************************************
*   PROGRAM: FDIV                                   *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               APRIL 1989.                         *
*                                                   *
*   FLOATING POINT DIVIDE FUNCTION: R0 <= R0/R1.    *
*                                                   *
*   APPROXIMATE ACCURACY: 8 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: R1 != 0.0.                  *
*   REGISTERS FOR INPUT: R0 (DIVIDEND) AND R1 (DIVISOR).*
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: R0-4.                        *
*   REGISTERS FOR OUTPUT: R0 (QUOTIENT).            *
*   ROUTINES NEEDED: FPINV.                         *
*   EXECUTION CYCLES (MIN, MAX): 43 , 43.           *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  FDIV
        .GLOBL  FPINV

        .TEXT

;       START OF FDIV PROGRAM

FDIV:

        RND     R0,R3           ; R3 <= RND X
        LDF     R1,R0           ; R1 <= Y
        CALL    FPINV           ; R0 <= 1/Y
        RND     R0              ; ROUND BEFORE *
        MPYF    R3,R0           ; R0 <= X

        RETS                    ; RETURN

        .END
```

```
*************************************************************************
*                                                                       *
*   PROGRAM: $MATHX.ASM                                                 *
*                                                                       *
*   EXTENDED-PRECISION, FLOATING-POINT (40-BIT) MATH FUNCTIONS          *
*                                                                       *
*   $MATHX.ASM CONSISTS OF THE FOLLOWING ROUTINES:                      *
*                                                                       *
*       SINX   - COMPUTES A 9D SIN(X) FOR ALL X IN RADIANS.             *
*                                                                       *
*       COSX   - COMPUTES A 9D COSINE(X) FOR ALL X IN RADIANS.          *
*                                                                       *
*       EXPX   - COMPUTES A 9D EXP(X) FOR ALL !X! =< 88.                *
*                                                                       *
*       LNX    - COMPUTES AN 8D LN(X) FOR ALL X > 0.                    *
*                                                                       *
*       ATANX  - COMPUTES AN 8D ATAN(X) FOR ALL X IN RADIANS.           *
*                                                                       *
*       SQRTX  - COMPUTES A 10D SQRT(X) FOR ALL X >= 0.                 *
*                                                                       *
*       FPINVX - COMPUTES A 10D 1/X FOR ALL X /= 0.                     *
*                                                                       *
*       FDIVX  - COMPUTES A 10D X/Y FOR ALL X AND ALL Y /= 0.           *
*                                                                       *
*       FMULTX - COMPUTES A 10D X*Y FOR ALL X AND ALL Y.                *
*                                                                       *
*************************************************************************
```

```
*******************************************************
*  PROGRAM: SINX                                      *
*                                                     *
*  WRITTEN BY: GARY A. SITTON                         *
*              GAS LIGHT SOFTWARE                     *
*              HOUSTON, TEXAS                         *
*              MARCH 1989.                            *
*                                                     *
*  EXTENDED PRECISION SINE FUNCTION: R0 <= SIN(R0).   *
*                                                     *
*  APPROXIMATE ACCURACY: 9 DECIMAL DIGITS.            *
*  INPUT RESTRICTIONS: NONE.                          *
*  REGISTERS FOR INPUT: R0 (ARGUMENT IN RADIANS).     *
*  REGISTERS USED AND RESTORED: DP AND SP.            *
*  REGISTERS ALTERED: AR0, IR0, AND R0-7.             *
*  REGISTERS FOR OUTPUT: R0.                          *
*  ROUTINES NEEDED: FMULTX.                           *
*  EXECUTION CYCLES (MIN, MAX): 160, 160.             *
*******************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  SINX
        .GLOBL  ECOSX
        .GLOBL  FMULTX

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFFS. FOR SIN(X)

NRM2    .WORD   00000006FH      ; BOTTOM OF 2/PI
NRM1    .WORD   0FF22F983H      ; TOP OF 2/PI

;       POLYNOMIAL COEFFS. FOR SIN(X)

SHF2    .WORD   0000000A3H      ; BOTTOM OF C1 (PI/2)
SHF1    .WORD   000490FDAH      ; TOP OF C1 (PI/2)
        .WORD   0000000D1H      ; BOTTOM OF C3
        .WORD   0FFDAA218H      ; TOP OF C3
        .WORD   0000000E3H      ; BOTTOM OF C5
        .WORD   0FC2335E0H      ; TOP OF C5
        .WORD   0F8E69754H      ; TOP OF C7
        .WORD   0F3260B28H      ; TOP OF C9
COF     .WORD   0ED9997B4H      ; TOP OF C11

ACOF    .WORD   COF             ; ADDRESS OF COEFFS.

CON     .FLOAT  -1.0, 0.0, 1.0, 0.0 ; MAPPING CONSTS.

ACON    .WORD   CON             ; ADDRESS OF CONSTS.

        .TEXT
```

```
;       START OF SINX PROGRAM

SINX:

        PUSH    DP              ; SAVE DP
        LDP     @NRM1           ; LOAD DATA PAGE POINTER

;       COSX ENTRY POINT

ECOSX:

;       SCALE AND MAP VARIABLE X

        PUSHF   R0              ; SAVE ORIGINAL X
        ABSF    R0              ; R0 <= |X|
        LDF     @NRM1,R1        ; R1 <= TOP OF 2/PI
        OR      @NRM2,R1        ; OR IN BOTTOM OF 2/PI
        CALL    FMULTX          ; R0 <= |X|*2/PI
        FIX     R0,IR0          ; IR0 <= INTEGER QUADRANT Q
        FLOAT   IR0,R1          ; R1 <= FLOATING QUADRANT Q
        SUBF    R1,R0           ; R0 <= X, -1 < X < 1
        NEGF    R0,R3           ; R3 <= -X
        ADDI    1,IR0           ; R2 <= Q + 1
        AND     3,IR0           ; IR0 <= TABLE INDEX
        TSTB    2,IR0           ; LOOK AT 2ND LSB
        LDFNZ   R3,R0           ; IF 1 THEN R0 <= -X
        LDP     @ACON           ; LOAD DATA PAGE POINTER
        LDI     @ACON,AR0       ; AR0 -> CONST. TABLE
        ADDF    **AR0(IR0),R0   ; FINAL MAPPING, R0 <= X + C
        NEGF    R0,R3           ; R3 <= -X
        LDI     @ACOF,AR0       ; AR0 -> COEFF. TABLE

;       EVALUATE TRUNCATED SERIES

        LDF     R0,R1           ; R1 <= X
        CALL    FMULTX          ; R0 <= X**2
        LDF     R0,R1           ; R1 <= X**2

        MPYF    *AR0--,R1,R0    ; R0 <= X**2*C11
        ADDF    *AR0--,R0       ; R0 <= C9 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C9 + R0)
        ADDF    *AR0--,R0       ; R0 <= C7 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C7 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C5
        OR      *AR0--,R2       ; OR IN BOTTOM OF C5
        ADDF    R2,R0           ; R0 <= C5 + R0

        CALL    FMULTX          ; R0 <= X**2*(C5 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C3
        OR      *AR0--,R2       ; OR IN BOTTOM OF C3
        ADDF    R2,R0           ; R0 <= C3 + R0

        CALL    FMULTX          ; R0 <= X**2*(C3 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C1
        OR      *AR0,R2         ; OR IN BOTTOM OF C1
        ADDF    R2,R0,R1        ; R1 <= C1 + R0

;       TEST FOR X < 0 AND RETURN

        NEGF    R3,R0           ; R0 <= X
        BRD     FMULTX          ; R0 <= X*R1 = SIN(X), (DELAYED)
        POPF    R5              ; TEST ORIGINAL X
        LDFN    R3,R0           ; IF X < 0 THEN R0 <= -X
        POP     DP              ; UNSAVE DP

;       RETURN OCCURS FROM FMULTX !
```

```
*****************************************************
*   PROGRAM: COSX                                   *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   EXTENDED PRECISION COSINE FUNCTION: R0 <= COS(R0).  *
*                                                   *
*   APPROXIMATE ACCURACY: 9 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: NONE.                       *
*   REGISTERS FOR INPUT: R0 (ARGUMENT IN RADIANS).  *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: AR0, IR0, AND R0-7.          *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: ECOSX (SINX).                  *
*   EXECUTION CYCLES (MIN, MAX): 165, 165.          *
*                                                   *
*   NOTE: USES SHF1 AND SHF2 FROM SINX PROGRAM!     *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  COSX
        .GLOBL  ECOSX

        .TEXT

;       START OF COSX PROGRAM

COSX:

        PUSH    DP              ; SAVE DP
        LDP     @NRM1           ; LOAD DATA PAGE POINTER

        BRD     ECOSX           ; R0 <= COS(X) = SIN(X'), (DELAYED)
        LDF     @SHF1,R1        ; R1 <= TOP OF PI/2
        OR      @SHF2,R1        ; OR IN BOTTOM OF PI/2
        ADDF    R1,R0           ; R0 <= X' = X + PI/2

;       RETURN OCCURS FROM SINX (ALIAS FMULTX) !
```

```
*****************************************************
*   PROGRAM: EXPX                                   *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   EXTENDED PREC. EXPONENTIAL: R0 <= EXP(R0).      *
*                                                   *
*   APPROXIMATE ACCURACY: 9 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: |R0| <= 88.0.               *
*   REGISTERS FOR INPUT: R0.                        *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: AR0 AND R0-7.                *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: FMULTX AND FPINVX.             *
*   EXECUTION CYCLES (MIN, MAX): 115 (R0 <=0 ), 160.  *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  EXPX
        .GLOBL  FMULTX
        .GLOBL  FPINVX

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFFS. FOR 2**X

ENRM2   .WORD   000000029H      ; BOTTOM OF 1/LN(2)
ENRM1   .WORD   00038AA3BH      ; TOP OF 1/LN(2)

;       POLYNOMIAL COEFFS. FOR 2**-X, 0 <= X < 1.

        .WORD   000000000H      ; C0 (1.0)
        .WORD   00000000AH      ; BOTTOM OF C1
        .WORD   0FFCE8DE8H      ; TOP OF C1
        .WORD   00000006EH      ; BOTTOM OF C2
        .WORD   0FD75FDEDH      ; TOP OF C2
        .WORD   000000046H      ; BOTTOM OF C3
        .WORD   0FB9CA833H      ; TOP OF C3
        .WORD   0F91D8C56H      ; TOP OF C4
        .WORD   0F6D1E7A9H      ; TOP OF C5
        .WORD   0F31AA7D7H      ; TOP OF C6
C7      .WORD   0EFC9BD9CH      ; TOP OF C7

AC7     .WORD   C7

        .TEXT

;       START OF EXPX PROGRAM
```

```
EXPX:

;       SCALE VARIABLE X

        PUSH    DP              ; SAVE DP
        LDP     @AC7            ; LOAD DATA PAGE POINTER
        NEGF    R0,R2           ; R2 <= -X
        LDF     R0,R1           ; R1 <= X
        LDFN    R2,R0           ; IF X < 0 THEN R1 <= |X|
        LDF     @ENRM1,R1       ; R1 <= TOP OF 1/LN(2)
        OR      @ENRM2,R1       ; OR IN BOTTOM OF 1/LN(2)
        CALL    FMULTX          ; R0 <= X = |X|/LN(2)
        FIX     R0,R3           ; R3 <= I = INTEGER OF X
        FLOAT   R3,R1           ; R1 <= FLT. PT. I
        SUBF    R1,R0,R1        ; R1 <= FRACTION OF |X|, 0 <= X < 1
        NEGI    R3              ; R3 <= -I
        LSH     24,R3           ; MOVE -I TO EXP.
        PUSH    R3              ; SAVE AS INT.
        POPF    R3              ; R3 <= FLT. PT. 2**-I
        LDI     @AC7,AR0        ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED SERIES

        MPYF    *AR0--,R1,R0    ; R0 <= X*C7
        ADDF    *AR0--,R0       ; R0 <= C6 + R0

        MPYF    R1,R0           ; R0 <= X*(C6 + R0)
        ADDF    *AR0--,R0       ; R0 <= C5 + R0

        MPYF    R1,R0           ; R0 <= X*(C5 + R0)
        ADDF    *AR0--,R0       ; R0 <= C4 + R0

        MPYF    R1,R0           ; R0 <= X*(C4 + R0)
        LDF     *AR0--,R4       ; R4 <= TOP OF C3
        OR      *AR0--,R4       ; OR IN BOTTOM OF C3
        ADDF    R4,R0           ; R0 <= C3 + R0

        MPYF    R1,R0           ; R0 <= X*(C3 + R0)
        LDF     *AR0--,R4       ; R4 <= TOP OF C2
        OR      *AR0--,R4       ; OR IN BOTTOM OF C2
        ADDF    R4,R0           ; R0 <= C2 + R0

        CALL    FMULTX          ; R0 <= X*(C2 + R0)
        LDF     *AR0--,R4       ; R4 <= TOP OF C1
        OR      *AR0--,R4       ; OR IN BOTTOM OF C1
        ADDF    R4,R0           ; R0 <= C1 + R0

        CALL    FMULTX          ; R0 <= X*(C1 + R0)

;       TEST FOR X < 0 AND RETURN

        LDF     R2,R2           ; TEST ORIGINAL -X

        BND     FPINVX          ; IF -X < 0 THEN R0 <= 1/X, (DELAYED)
        ADDF    *AR0,R0,R1      ; R1 <= 2**-X = C0 + R0
        MPYF    R3,R1,R0        ; R0 <= 2**-(I + X) TRUNC.
        LDM     R1,R0           ; R0 <= FULL MANTISSA

        RETS                    ; RETURN (IF NO FPINVX BRANCH)
```

```
*****************************************************
*   PROGRAM: LNX                                    *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   EXTENDED PREC. LOGARITHM BASE E: R0 <= LN(R0).  *
*                                                   *
*   APPROXIMATE ACCURACY: 8 DECIMAL DIGITS.         *
*   INPUT RESTRICTIONS: R0 > 0.0.                   *
*   REGISTERS FOR INPUT: R0.                        *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: AR0 AND R0-7.                *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: FMULTX.                        *
*   EXECUTION CYCLES (MIN, MAX): 193, 193.          *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  LNX
        .GLOBL  FMULTX

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFFS. FOR LN(1+X)

LNRM2   .WORD   0000000F7H      ; BOTTOM OF LN(2)
LNRM1   .WORD   0FF317217H      ; TOP OF LN(2)

;       POLYNOMIAL COEFFS. FOR LN(1+X), 0 <= X < 1.

C0      .FLOAT  1.0             ; C0 (1.0)

        .WORD   0000000FFH      ; BOTTOM OF C1
        .WORD   0FF7FFFC3H      ; TOP OF C1
        .WORD   000000084H      ; BOTTOM OF C2
        .WORD   0FE80107FH      ; TOP OF C2
        .WORD   0000000D9H      ; BOTTOM OF C3
        .WORD   0FE29E18FH      ; TOP OF C3
        .WORD   000000097H      ; BOTTOM OF C4
        .WORD   0FD897D13H      ; TOP OF C4
        .WORD   000000041H      ; BOTTOM OF C5
        .WORD   0FD2BAD82H      ; TOP OF C5
        .WORD   0000000E7H      ; BOTTOM OF C6
        .WORD   0FC8CC3F1H      ; TOP OF C6
        .WORD   000000043H      ; BOTTOM OF C7
        .WORD   0FB13D187H      ; TOP OF C7
C8      .WORD   0F8AC87BFH      ; TOP OF C8

AC8     .WORD   C8

        .TEXT

;       START OF LNX PROGRAM

LNX:

        LDF     R0,R0           ; TEST X
        RETSLE                  ; RETURN NOW IF X <= 0

;       SCALE VARIABLE X

        PUSH    DP              ; SAVE DP
        LDP     @AC8            ; LOAD DATA PAGE POINTER
        PUSHF   R0              ; SAVE AS FLT. PT.
        POP     R3              ; R3 <= INTEGER FORMAT
        ASH     -24,R3          ; R3 <= E = SIGNED EXP.
        FLOAT   R3,R1           ; R1 <= FLT. PT. E VALUE
        LDF     @C0,R2          ; R2 <= 1.0
        LDE     R2,R0           ; EXP. R0 <= 0 (1 <= X < 2)
        SUBRF   R0,R2           ; R2 <= X - 1 (0 <= X < 1)
        LDF     @LNRM1,R0       ; R0 <= TOP OF LN(2)
        OR      @LNRM2,R0       ; OR IN BOTTOM OF LN(2)
        CALL    FMULTX          ; R0 <= E*LN(2)
        LDF     R0,R3           ; R3 <= E*LN(2)
        LDI     @AC8,AR0        ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED SERIES

        LDF     R2,R1           ; R1 <= X
        MPYF    *AR0--,R1,R0    ; R0 <= X*C8
        LDF     *AR0--,R2       ; R2 <= TOP OF C7
        OR      *AR0--,R2       ; OR IN BOTTOM OF C7
        ADDF    R2,R0           ; R0 <= C7 + R0

        MPYF    R1,R0           ; R0 <= X*(C7 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C6
        OR      *AR0--,R2       ; OR IN BOTTOM OF C6
        ADDF    R2,R0           ; R0 <= C6 + R0

        MPYF    R1,R0           ; R0 <= X*(C6 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C5
        OR      *AR0--,R2       ; OR IN BOTTOM OF C5
        ADDF    R2,R0           ; R0 <= C5 + R0

        CALL    FMULTX          ; R0 <= X*(C5 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C4
        OR      *AR0--,R2       ; OR IN BOTTOM OF C4
        ADDF    R2,R0           ; R0 <= C4 + R0

        CALL    FMULTX          ; R0 <= X*(C4 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C3
```

```
        OR      *AR0--,R2       ; OR IN BOTTOM OF C3
        ADDF    R2,R0           ; R0 <= C3 + R0

        CALL    FMULTX          ; R0 <= X*(C3 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C2
        OR      *AR0--,R2       ; OR IN BOTTOM OF C2
        ADDF    R2,R0           ; R0 <= C2 + R0

        CALL    FMULTX          ; R0 <= X*(C2 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C1
        OR      *AR0--,R2       ; OR IN BOTTOM OF C1
        ADDF    R2,R0           ; R0 <= C1 + R0

        CALL    FMULTX          ; R0 <= X*(C1 + R0)

;       ADD IN SCALED EXPONENT.

        ADDF    R3,R0           ; R0 <= LN(X) + E*LN(2)

        RETS                    ; RETURN
```

```
*****************************************************
*  PROGRAM: ATANX                                   *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              MARCH 1989.                          *
*                                                   *
*  EXTENDED PRECISION ARC TANGENT: R0 <= ATAN(R0).  *
*                                                   *
*  APPROXIMATE ACCURACY: 8 DECIMAL DIGITS.          *
*  INPUT RESTRICTIONS: NONE.                        *
*  REGISTERS FOR INPUT: R0.                         *
*  REGISTERS USED AND RESTORED: DP AND SP.          *
*  REGISTERS ALTERED: AR0, IR0, AND R0-7.           *
*  REGISTERS FOR OUTPUT: R0 (IN RADIANS).           *
*  ROUTINES NEEDED: FMULTX, AND FDIVX.              *
*  EXECUTION CYCLES (MIN, MAX): 210 (!ATANX!<=1), 332. *
*****************************************************
```

```
;       EXTERNAL PROGRAM NAMES

        .GLOBL  ATANX
        .GLOBL  FMULTX
        .GLOBL  FDIVX

;       INTERNAL CONSTANTS

        .DATA

;       SCALING COEFFS. FOR ATAN(X)

        .WORD   00000005DH      ; BOTTOM OF -PI/4
        .WORD   0FFB6F025H      ; TOP OF -PI/4
        .WORD   0000000A2H      ; BOTTOM OF PI/4
        .WORD   0FF490FDAH      ; TOP OF PI/4
        .WORD   000000000H      ; BOTTOM OF ZERO
        .WORD   080000000H      ; TOP OF ZERO

;       POLYNOMIAL COEFFS. FOR ATAN(X), -1 <= X <= 1.

C1      .WORD   000000000H      ; TOP OF C1 (1.0)
        .WORD   00000006EH      ; BOTTOM OF C3
        .WORD   0FED55594H      ; TOP OF C3
        .WORD   0000000D9H      ; BOTTOM OF C5
        .WORD   0FD4CBBE4H      ; TOP OF C5
        .WORD   0000000FFH      ; BOTTOM OF C7
        .WORD   0FDEE8038H      ; TOP OF C7
        .WORD   000000056H      ; BOTTOM OF C9
        .WORD   0FC5A3D83H      ; TOP OF C9
        .WORD   000000093H      ; BOTTOM OF C11
        .WORD   0FCE5CE8BH      ; TOP OF C11
        .WORD   0000000BFH      ; BOTTOM OF C13
        .WORD   0FB2FC1FDH      ; TOP OF C13
```

```
        .WORD   0FAFB91FEH      ; TOP OF C15
C17     .WORD   0F73BD74AH      ; TOP OF C17

AC17    .WORD   C17

        .TEXT

;       START OF ATANX PROGRAM

ATANX:

;       SCALE VARIABLE X

        PUSH    DP              ; SAVE DP
        LDP     @AC17           ; LOAD DATA PAGE POINTER
        ABSF    R0,R2           ; R2 <= |X|
        SUBF    @C1,R2          ; R2 <= |X| - 1
        BLED    SKIP            ; IF |X| > 1 THEN SCALE (DELAYED)
        LDF     R0,R3           ; R3 <= X
        LDF     R0,R1           ; R1 <= X
        LDI     0,IR0           ; IR0 <= 0, POST SCALE INDEX

;       SCALE FOR |X| > 1

        PUSHF   R0              ; SAVE X
        ABSF    R0,R1           ; R1 <= |X|
        ADDF    @C1,R1          ; R1 <= |X| + 1
        LDF     R2,R0           ; R0 <= |X| - 1
        CALL    FDIVX           ; R0 <= (|X| - 1)/(|X| + 1)

;       TEST FOR X' < 0

        POPF    R4              ; GET ORIGINAL X
        BGED    SKIP            ; IF X < 0 THEN R0 <= -X' (DELAYED)
        LDF     R0,R3           ; R3 <= X'
        LDF     R0,R1           ; R1 <= X'
        SUBI    2,IR0           ; IR0 <= -2, (PI/4)

        NEGF    R0,R3           ; R3 <= -X'
        SUBI    2,IR0           ; IR0 <= -4, (-PI/4)

SKIP:   CALL    FMULTX          ; R0 <= X**2
        LDI     @AC17,AR0       ; AR0 -> COEFF. TABLE
        POP     DP              ; UNSAVE DP

;       EVALUATE TRUNCATED (ODD) SERIES

        LDF     R0,R1           ; R1 <= X**2
        MPYF    *AR0--,R1,R0    ; R0 <= X**2*C17
        ADDF    *AR0--,R0       ; R0 <= C15 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C15 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C13
        OR      *AR0--,R2       ; OR IN BOTTOM OF C13
        ADDF    R2,R0           ; R0 <= C13 + R0

        MPYF    R1,R0           ; R0 <= X**2*(C13 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C11
        OR      *AR0--,R2       ; OR IN BOTTOM OF C11
        ADDF    R2,R0           ; R0 <= C11 + R0

        CALL    FMULTX          ; R0 <= X**2*(C11 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C9
        OR      *AR0--,R2       ; OR IN BOTTOM OF C9
        ADDF    R2,R0           ; R0 <= C9 + R0

        CALL    FMULTX          ; R0 <= X**2*(C9 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C7
        OR      *AR0--,R2       ; OR IN BOTTOM OF C7
        ADDF    R2,R0           ; R0 <= C7 + R0

        CALL    FMULTX          ; R0 <= X**2*(C7 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C5
        OR      *AR0--,R2       ; OR IN BOTTOM OF C5
        ADDF    R2,R0           ; R0 <= C5 + R0

        CALL    FMULTX          ; R0 <= X**2*(C5 + R0)
        LDF     *AR0--,R2       ; R2 <= TOP OF C3
        OR      *AR0--,R2       ; OR IN BOTTOM OF C3
        ADDF    R2,R0           ; R0 <= C3 + R0

        CALL    FMULTX          ; R0 <= X**2*(C3 + R0)

;       FINISH UP

        ADDF    *AR0--,R0,R1    ; R1 <= C1 + R0
        LDF     R3,R0           ; R0 <= X (SIGNED)
        CALL    FMULTX          ; R0 <= ATANX(X) = X*(1 + R0)
        NOP     *AR0++(IR0)     ; AR0 -> C (0.0, PI/4 OR -PI/4)

;       ADD IN POST SCALE VALUE C AND RETURN

        POP     R4              ; R4 <= RETURN ADDRESS
        BUD     R4              ; RETURN (DELAYED)
        LDF     *AR0--,R1       ; R1 <= TOP OF C
        OR      *AR0,R1         ; OR IN BOTTOM OF C
        ADDF    R1,R0           ; R0 <= ATAN(X) + C
```

```
*************************************************
*  PROGRAM: SQRTX                               *
*                                               *
*  WRITTEN BY: GARY A. SITTON                   *
*              GAS LIGHT SOFTWARE               *
*              HOUSTON,'TEXAS                   *
*              MARCH 1989.                      *
*                                               *
*  APPROXIMATE ACCURACY: 10 DECIMAL DIGITS.     *
*  INPUT RESTRICTIONS: R0 >= 0.0.               *
*  REGISTERS FOR INPUT: R0.                     *
*  REGISTERS USED AND RESTORED: DP AND SP.      *
*  REGISTERS ALTERED: R0-7.                     *
*  REGISTERS FOR OUTPUT: R0.                    *
*  ROUTINES NEEDED: FMULTX.                     *
*  EXECUTION CYCLES (MIN, MAX): 138, 138.       *
*************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  SQRTX
        .GLOBL  FMULTX

;       INTERNAL CONSTANTS

        .DATA

CNST1   .SET    0.5
CNST2   .SET    1.5
CNST3   .FLOAT  1.103553391     ; ADJUSTED 1.0
CNST4   .FLOAT  0.780330086     ; ADJUSTED SQRT(1/2)

SMSK    .WORD   0FF7FFFFFH

        .TEXT

;       START OF SQRTX PROGRAM.

SQRTX:

        LDF     R0,R3           ; TEST AND SAVE V
        RETSLE                  ; RETURN NOW IF V <= 0

;       GET APPROXIMATION TO 1/V.  FOR V = (1+M)*2**E
;       AND 0 <= M < 1, FOR E EVEN: X[0] = (1-M/2)*2**-E/2
;       AND FOR E ODD: X[0] = SQRT(1/2)*(1-M/2)*2**-E/2

        PUSH    DP              ; SAVE DP
        LDP     @SMSK           ; LOAD DATA PAGE POINTER
        PUSHF   R0              ; SAVE V AS FLT. PT. V = (1+M)*2**E
        POP     R4              ; R4 <= V AS INTEGER
        XOR     @SMSK,R4        ; R4 <= COMPLEMENT ALL BUT SIGN
        LDI     R4,R1           ; R1 <= (1-M/2)*2**-E
        LDI     R4,R5           ; R5 <= R1
        LSH     8,R1            ; R1 <= R1 EXP. REMOVED
        ASH     -1,R4           ; R4 <= R4 WITH -E/2 EXP.
        PUSH    R4              ; SAVE R4 AS INTEGER
        POPF    R4              ; R4 <= FLT. PT.
        LDE     R4,R1           ; R1 <= (1-M/2)*2**-E/2
        LDF     @CNST3,R2       ; R2 <= 1.1... FOR ODD E
        LSH     7,R5            ; TEST LSB OF E (AS SIGN)
        LDFNN   @CNST4,R2       ; IF E EVEN R2 <= 0.78...
        MPYF    R2,R1           ; R1 <= CORRECTED ESTIMATE

;       GENERATE V/2 (USES MPYF).

        MPYF    CNST1,R0        ; R0 <= V/2 TRUNC.
        LDM     R3,R0           ; R0 <= V/2 FULL PREC.

;       NEWTON ITERATION FOR Y(X) = X - V**-2 = 0 ...

        MPYF    R1,R1,R2        ; R2 <= X[0]**2
        MPYF    R0,R2           ; R2 <= (V/2) * X[0]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[0]**2
        MPYF    R2,R1           ; R1 <= X[1] = X[0] * (1.5 - (V/2)*X[0]**2)
;
        MPYF    R1,R1,R2        ; R2 <= X[1]**2
        MPYF    R0,R2           ; R2 <= (V/2) * X[1]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[1]**2
        MPYF    R2,R1           ; R1 <= X[2] = X[1] * (1.5 - (V/2)*X[1]**2)
;
        MPYF    R1,R1,R2        ; R2 <= X[2]**2
        MPYF    R0,R2           ; R2 <= (V/2) * X[2]**2
        SUBRF   CNST2,R2        ; R2 <= 1.5 - (V/2) * X[2]**2
        MPYF    R2,R1           ; R1 <= X[3] = X[2] * (1.5 - (V/2)*X[2]**2)
;
        LDF     R0,R2           ; R2 <= V/2
        LDF     R1,R0           ; R0 <= X[3]
        CALL    FMULTX          ; R0 <= X[3]**2
        LDF     R1,R4           ; R4 <= X[3]
        LDF     R2,R1           ; R1 <= V/2
        LDF     R4,R2           ; R2 <= X[3]
        CALL    FMULTX          ; R0 <= (V/2) * X[3]**2
        SUBRF   CNST2,R0        ; R0 <= 1.5 - (V/2) * X[3]**2
        LDF     R2,R1           ; R1 <= X[3]
        CALL    FMULTX          ; R0 <= X[4] = X[3] * (1.5 - (V/2)*X[3]**2)

;       INVERT FINAL RESULT AND RETURN

        BRD     FMULTX          ; R0 = SQRT(V) = V*SQRT(1/V) (DELAYED)
        LDF     R3,R1           ; R1 <= V
        POP     DP              ; UNSAVE DP
        NOP                     ; DEAD CYCLE

;       RETURN OCCURS FROM FMULTX !
```

```
*****************************************************
*   PROGRAM: FPINVX                                 *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   EXTENDED PREC. FLT. PT. INVERSE: R0 <= 1/R0.    *
*                                                   *
*   APPROXIMATE ACCURACY: 10 DECIMAL DIGITS.        *
*   INPUT RESTRICTIONS: R0 != 0.0.                  *
*   REGISTERS FOR INPUT: R0.                        *
*   REGISTERS USED AND RESTORED: DP AND SP.         *
*   REGISTERS ALTERED: R0-1 AND R4-7.               *
*   REGISTERS FOR OUTPUT: R0.                       *
*   ROUTINES NEEDED: FMULTX.                        *
*   EXECUTION CYCLES (MIN, MAX): 76, 76.            *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  FPINVX
        .GLOBL  FMULTX

;       INTERNAL CONSTANTS

        .DATA

ONE     .SET    1.0
TWO     .SET    2.0

MSK     .WORD   0FF7FFFFFH

        .TEXT

;       START OF FPINVX PROGRAM

FPINVX:

        LDF     R0,R0           ; TEST F
        RETSZ                   ; RETURN NOW IF F = 0

;       GET APPROXIMATION TO 1/F.  FOR F = (1+M) * 2**E
;       AND 0 <= M < 1, USE: X[0] = (1-M/2) * 2**-E

        PUSH    DP              ; SAVE DP
        LDP     @MSK            ; LOAD DATA PAGE POINTER
        PUSHF   R0              ; SAVE AS FLT. PT. F = (1+M) * 2**E
        POP     R1              ; FETCH BACK AS INTEGER
        XOR     @MSK,R1         ; COMPLEMENT E & M BUT NOT SIGN BIT
        PUSH    R1              ; SAVE AS INTEGER, AND BY MAGIC...
        POPF    R1              ; R1 <= X[0] = (1-M/2) * 2**-E.
        POP     DP              ; UNSAVE DP

;       NEWTON ITERATION FOR: Y(X) = X - 1/F = 0 ...

        MPYF    R1,R0,R4        ; R4 <= F * X[0]
        SUBRF   TWO,R4          ; R4 <= 2 - F * X[0]
        MPYF    R4,R1           ; R1 <= X[1] = X[0] * (2 - F * X[0])

        MPYF    R1,R0,R4        ; R4 <= F * X[1]
        SUBRF   TWO,R4          ; R4 <= 2 - F * X[1]
        MPYF    R4,R1           ; R1 <= X[2] = X[1] * (2 - F * X[1])

        MPYF    R1,R0,R4        ; R4 <= F * X[2]
        SUBRF   TWO,R4          ; R4 <= 2 - F * X[2]
        MPYF    R4,R1           ; R1 <= X[3] = X[2] * (2 - F * X[2])

;       FOR THE LAST ITERATION: X[4] = (X[3] * (1 - (F * X[3]))) + X[3]

        CALL    FMULTX          ; R0 <= F * X[3] = 1 + EPS
        SUBRF   ONE,R0          ; R0 <= 1 - F * X[3] = EPS
        CALL    FMULTX          ; R0 <= X[3] * EPS
        ADDF    R1,R0           ; R0 <= X[4] = (X[3]*(1 - (F*X[3]))) + X[3]

        RETS                    ; RETURN

        .END
```

```
*************************************************
*  PROGRAM: FDIVX                               *
*                                               *
*  WRITTEN BY: GARY A. SITTON                   *
*              GAS LIGHT SOFTWARE               *
*              HOUSTON, TEXAS                   *
*              MARCH 1989.                      *
*                                               *
*  EXTENDED PRECISION DIVIDE: R0 <= R0/R1.      *
*                                               *
*  APPROXIMATE ACCURACY: 10 DECIMAL DIGITS.     *
*  INPUT RESTRICTIONS: R1 != 0.0.               *
*  REGISTERS FOR INPUT: R0 (DIVIDEND) AND R1 (DIVISOR).*
*  REGISTERS USED AND RESTORED: DP AND SP.      *
*  REGISTERS ALTERED: R0-7.                     *
*  REGISTERS FOR OUTPUT: R0 (QUOTIENT).         *
*  ROUTINES NEEDED: FMULTX AND FPINVX.          *
*  EXECUTION CYCLES (MIN, MAX): 107, 107.       *
*************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  FDIVX
        .GLOBL  FPINVX
        .GLOBL  FMULTX

        .TEXT

;       START OF FDIVX PROGRAM

FDIVX:

        LDF     R0,R3           ; R3 <= X
        LDF     R1,R0           ; R1 <= Y
        CALL    FPINVX          ; R0 <= 1/Y
        LDF     R3,R1           ; R1 <= X
        BR      FMULTX          ; R0 <= X/Y

;       RETURN OCCURS FROM FMULTX !
```

```
*************************************************
*  PROGRAM: FMULTX                              *
*                                               *
*  WRITTEN BY: GARY A. SITTON                   *
*              GAS LIGHT SOFTWARE               *
*              HOUSTON, TEXAS                   *
*              MARCH 1989.                      *
*                                               *
*  EXTENDED PRECISION MULTIPLY: R0 <= R0*R1.    *
*                                               *
*  APPROXIMATE ACCURACY: 10 DECIMAL DIGITS.     *
*  INPUT RESTRICTIONS: NONE.                    *
*  REGISTERS FOR INPUT: R0.                     *
*  REGISTERS USED AND RESTORED: DP AND SP.      *
*  REGISTERS ALTERED: R0 AND R4-7.              *
*  REGISTERS FOR OUTPUT: R0.                    *
*  ROUTINES NEEDED: NONE.                       *
*  EXECUTION CYCLES (MIN, MAX): 20, 20.         *
*************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  FMULTX

        .TEXT

;       START OF FMULTX PROGRAM

FMULTX:

        ABSF    R0,R4           ; R4 <= |XA|
        XOR     R1,R0           ; R0 <= SIGN INFO.
        ABSF    R1,R7           ; R7 <= |XB|
        MPYF    R4,R7,R6        ; R6 <= A*B
        LDF     R4,R5           ; R5 <= |XA|
        ANDN    0FFH,R5         ; R5 <= A = XA - EA*2**-24
        SUBRF   R4,R5           ; R5 <= EA*2**-24
        MPYF    R7,R5           ; R5 <= B*EA*2**-24
        ADDF    R6,R5           ; R5 <= A*B + B*EA*2**-24
        LDF     R7,R6           ; R6 <= |XB|
        ANDN    0FFH,R6         ; R6 <= B = XB - EB*2**-24
        SUBRF   R7,R6           ; R6 <= EB*2**-24
        MPYF    R4,R6           ; R6 <= A*EB*2**-24
        ADDF    R6,R5           ; R5 <= |XA*XB| = A*B + (B*EA+A*EB)*2**-24
        NEGF    R5,R6           ; R6 <= - |XA*XB|

        TEST FOR XA*XB < 0 AND RETURN

        POP     R4              ; R4 <= RETURN ADDRESS
        BUD     R4              ; RETURN (DELAYED)
        LDF     R0,R0           ; TEST ORIGINAL (XA ^ XB)
        LDFN    R6,R5           ; IF XA*XB < 0 THEN R5 <= -|XA*XB|
        LDF     R5,R0           ; R0 <= XA*XB
```

```
*************************************************************
*                                                           *
*   PROGRAM: $MATHI.ASM                                     *
*                                                           *
*   INTEGER (32-BIT) MATH ROUTINES                          *
*                                                           *
*   $MATHI.ASM CONSISTS OF THE FOLLOWING ROUTINES:          *
*                                                           *
*       ILOG2 - COMPUTES M = LOG2(N), N=< 2**M FOR USE WITH RADIX 2 FFT
*               PROGRAMS.                                   *
*                                                           *
*       IMULT - COMPUTES A 64-BIT PRODUCT OF TWO 32-BIT NUMBERS.
*                                                           *
*       IDIV  - COMPUTES THE QUOTIENT AND REMAINDER OF TWO 32-BIT NUMBERS.
*                                                           *
*************************************************************
```

```
*************************************************
*  PROGRAM: ILOG2                               *
*                                               *
*  WRITTEN BY: GARY A. SITTON                   *
*              GAS LIGHT SOFTWARE               *
*              HOUSTON, TEXAS                   *
*              MARCH 1989.                      *
*                                               *
*  INTEGER LOG BASE 2: R0 <= (INTEGER) LOG2(R0). *
*                                               *
*  INPUT RESTRICTIONS: R0 > 0.                  *
*  REGISTERS FOR INPUT: R0.                     *
*  REGISTERS USED AND RESTORED: SP.             *
*  REGISTERS ALTERED: IR0-1 AND R0.             *
*  REGISTERS FOR OUTPUT: R0.                    *
*  ROUTINES NEEDED: NONE.                       *
*************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  ILOG2

        .TEXT

;       START OF ILOG2 PROGRAM

ILOG2:

        LDI     1,IR0           ; IR0 <= I (INIT. 1)
        LDI     -1,IR1          ; IR1 <= M (INIT. -1)

        CMPI    IR0,R0          ; COMPARE I TO N
LOOP:   BGTD    LOOP            ; LOOP IF N > I (DELAYED)
        LSH     1,IR0           ; I <= 2*I
        ADDI    1,IR1           ; M = M + 1
        CMPI    IR0,R0          ; COMPARE I TO N

        LDI     IR1,R0          ; R0 <= LOG2(N)
        RETS                    ; RETURN
```

```
*****************************************************
*  PROGRAM: IMULT                                   *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              MARCH 1989.                          *
*                                                   *
*  INTEGER 32 X 32 MULTIPLY: R1, R0 <= R0*R1.       *
*  RESULT IS THE 64 BIT PRODUCT OF TWO 32 BIT INPUTS.*
*                                                   *
*  INPUT RESTRICTIONS: NONE.                        *
*  REGISTERS FOR INPUT: R0 AND R1.                  *
*  REGISTERS USED AND RESTORED: SP.                 *
*  REGISTERS ALTERED: AR0-1 AND R0-4.               *
*  REGISTERS FOR OUTPUT: R1 (UPPER) AND R0 (LOWER). *
*  ROUTINES NEEDED: NONE.                           *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  IMULT

        .TEXT

;       START OF IMULT PROGRAM

IMULT:

        XOR     R0,R1,AR0       ; AR0 <= SIGNUM (R0*R1)
        ABSI    R0              ; R0 <= |X|
        ABSI    R1              ; R1 <= |Y|

;       SEPARATE MULTIPLIER AND MULTIPLICAND IN TWO PARTS

        LDI     -16,AR1         ; AR1 <= -16 (FOR SHIFTS)
        LSH     AR1,R0,R2       ; R2 <= X1 = UPPER 16 BITS OF |X|
        AND     0FFFFH,R0       ; R0 <= X0 = LOWER 16 BITS OF |X|
        LSH     AR1,R1,R3       ; R3 <= Y1 = UPPER 16 BITS OF |Y|
        AND     0FFFFH,R1       ; R1 <= Y0 = LOWER 16 BITS OF |Y|

;       CARRY OUT THE MULTIPLICATION

        MPYI    R0,R1,R4        ; R4 <= X0*Y0 = P1
        MPYI    R3,R0           ; R0 <= X0*Y1 = P2
        MPYI    R2,R1           ; R1 <= X1*Y0 = P3
        ADDI    R0,R1           ; R1 <= P2+P3
        MPYI    R2,R3           ; R3 <= X1*Y1 = P4

;       PUT THE PRODUCTS TOGETHER

        LDI     R1,R2           ; R2 <= P2+P3
        LSH     16,R2           ; R2 <= LOWER 16 BITS OF P2+P3
        CMPI    0,AR0           ; CHECK THE SIGN OF THE PRODUCT
        BGED    DONE            ; IF >= 0 THEN DONE (DELAYED)
        LSH     AR1,R1          ; R1 <= UPPER 16 BITS OF P2+P3
        ADDI    R4,R2,R0        ; R0 <= W0 = LOWER WORD OF THE PRODUCT
        ADDC    R3,R1           ; R1 <= W1 = UPPER WORD OF THE PRODUCT

;       NEGATE THE PRODUCT IF NUMBERS WERE OF OPPOSITE SIGN

        SUBRI   0,R0            ; R0 <= -W0
        SUBRB   0,R1            ; R1 <= -W1 (WITH BORROW)

DONE:   RETS                    ; RETURN
```

```
*******************************************************
*  PROGRAM: IDIV                                      *
*                                                     *
*  WRITTEN BY: GARY A. SITTON                         *
*              GAS LIGHT SOFTWARE                     *
*              HOUSTON, TEXAS                         *
*              MARCH 1989.                            *
*                                                     *
*                                                     *
*  INTEGER 32 / 32 DIVIDE: R0, R1 <= R0/R1.           *
*  RESULT IS A 32 BIT QUOTIENT AND |REMAINDER|.       *
*                                                     *
*  INPUT RESTRICTIONS: R1 != 0.                       *
*  REGISTERS FOR INPUT: R0 (DIVIDEND) AND R1 (DIVISOR).*
*  REGISTERS USED AND RESTORED: SP.                   *
*  REGISTERS ALTERED: IR0-1 AND R0-3.                 *
*  REGISTERS FOR OUTPUT: R0 (QUOTIENT) AND            *
*     R1 (|REMAINDER|).                               *
*  ROUTINES NEEDED: NONE.                             *
*******************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  IDIV

;       START OF IDIV PROGRAM

        .TEXT

IDIV:

;       DETERMINE SIGN OF RESULT.  GET ABSOLUTE VALUE OF OPERANDS.

        XOR     R0,R1,R2        ; R2 <= SIGNUM (R0/R1)
        ABSI    R0              ; R0 <= |X|
        ABSI    R1              ; R1 <= |Y|

;       TEST INPUT VALUES

        CMPI    R0,R1           ; COMPARE DIVISOR TO DIVIDEND
        BHID    ZERO            ; IF R1 > R0 THEN RETURN 0 (DELAYED)

;       NORMALIZE OPERANDS.  USE DIFFERENCE IN EXPONENTS AS
;       SHIFT COUNT FOR DIVISOR, AND AS REPEAT COUNT FOR SUBC.

        FLOAT   R0,R3           ; R3 <= NORMALIZED DIVIDEND
        PUSHF   R3              ; PUSH AS FLOAT
        POP     IR1             ; IR1 <= INTEGER
        LSH     -24,IR1         ; IR1 <= DIVIDEND EXPONENT

        FLOAT   R1,R3           ; R3 <= NORMALIZED DIVISOR
        PUSHF   R3              ; PUSH AS FLOAT
        POP     IR0             ; IR0 <= INTEGER
        LSH     -24,IR0         ; IR0 <= DIVISOR EXPONENT

        SUBI    IR0,IR1         ; IR1 <= DIFFERENCE IN EXPONENTS
        LSH     IR1,R1          ; R1 <= ALIGNED DIVISOR WITH DIVIDEND

;       DO IR1+1 SUBTRACT & SHIFTS.

        RPTS    IR1             ; REPEAT IR1+1 TIMES
        SUBC    R1,R0           ; R0 <= 2*(R0 - R1)

;       MASK OFF THE LOWER IR1+1 BITS OF R0

        LDI     R0,R1           ; R1 <= |REMAINDER, QUOTIENT|
        SUBRI   31,IR1          ; IR1 <= 32 - (IR1+1)
        LSH     IR1,R0          ; R0 <= R0 SHIFT LEFT IR1
        NEGI    IR1             ; IR1 <= -IR1
        LSH     IR1,R0          ; R0 <= |X|/|Y|
        SUBRI   -32,IR1         ; IR1 <=  -(IR1+1)
        LSH     IR1,R1          ; R1 <= |REMAINDER|

;       CHECK SIGN AND NEGATE RESULT IF NECESSARY.

        NEGI    R0,R3           ; R3 <= -|X|/|Y|
        ASH     -31,R2          ; TEST SIGN BIT
        LDINZ   R3,R0           ; IF SET R0 <= -R0
        CMPI    0,R0            ; SET STATUS FROM RESULT
        RETS                    ; RETURN

;       RETURN ZERO QUOTIENT.

ZERO:   LDI     R0,R1           ; R1 <= |REMAINDER|
        LDI     0,R0            ; R0 <= 0 QUOTIENT
        RETS                    ; RETURN

        .END
```

```
*************************************************************
*                                                           *
*   PROGRAM: $VECTOR.ASM                                    *
*                                                           *
*   VECTOR UTILITIES                                        *
*                                                           *
*   $VECTOR.ASM CONSISTS OF THE FOLLOWING ROUTINES:         *
*                                                           *
*       #CORMULT - IN-PLACE COMPUTATION OF THE COMPLEX VECTOR PRODUCT OF TWO
*                  COMPLEX ARRAYS USING THE COMPLEX CONJUGATE OF THE SECOND
*                  ARRAY.                                   *
*                                                           *
*       #CONMULT - IN-PLACE COMPUTATION OF THE COMPLEX VECTOR PRODUCT OF TWO
*                  COMPLEX ARRAYS.                          *
*                                                           *
*       #CBITREV - IN-PLACE BIT REVERSE PERMUTATION ON A COMPLEX ARRAY WITH
*                  SEPARATE REAL AND IMAGINARY ARRAYS.      *
*                                                           *
*       #FMIEEE  - IN-PLACE FAST CONVERSION OF AN IEEE ARRAY TO A TMS320C30
*                  ARRAY.                                   *
*                                                           *
*       #TOIEEE  - IN-PLACE FAST CONVERSION OF A TMS320C30 ARRAY TO AN IEEE
*                  ARRAY.                                   *
*                                                           *
*       #VECMULT - IN-PLACE MULTIPLIES A CONSTANT TIMES AN ARRAY.
*                                                           *
*       #CONMOV  - MOVES (FILLS) A CONSTANT INTO AN ARRAY.  *
*                                                           *
*       #VECMOV  - MOVES (COPIES) AN ARRAY INTO ANOTHER ARRAY.
*                                                           *
*************************************************************
```

```
*************************************************************
*   PROGRAM: #CORMULT                                       *
*                                                           *
*   WRITTEN BY: GARY A. SITTON                              *
*               GAS LIGHT SOFTWARE                          *
*               HOUSTON, TEXAS                              *
*               FEBRUARY 1989.                              *
*                                                           *
*   COMPLEX IN-PLACE FREQUENCY DOMAIN CORRELATION:          *
*   C1 <= C1 * CONJ(C2), C1 AND C2 ARE BOTH OF LENGTH       *
*   N, AND C1 = (X1 + I*Y1) AND CONJ(C2) = (X2 - I*Y2).     *
*                                                           *
*   MCORMULT ENTRY PROTOCOL:                                *
*       VARIABLES FOR INPUT:                                *
*           $IAD1 -> X1[0], $IAD2 -> Y1[0],                 *
*           $SAD1 -> X2[0], $SAD2 -> Y2[0],                 *
*           $N = N (LENGTH), $PARMS = DATA PAGE.            *
*       INPUT RESTRICTIONS: $N > 0.                         *
*       REGISTERS ALTERED: RC, DP, AR0-3 AND R0-3.          *
*                                                           *
*   RCORMULT ENTRY PROTOCOL:                                *
*       REGISTERS FOR INPUT:                                *
*           AR0 -> X1[0], AR1 -> Y1[0], AR2 -> X2[0],       *
*           AR3 -> Y2[0], RC = N (LENGTH).                  *
*       INPUT RESTRICTIONS: RC > 0.                         *
*       REGISTERS ALTERED: RC, AR0-3 AND R0-3.              *
*                                                           *
*   REGISTERS USED AND RESTORED: SP.                        *
*   REGISTERS FOR OUTPUT: NONE.                             *
*   ROUTINES NEEDED: NONE.                                  *
*************************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X1
        .GLOBL  $IAD2       ; ADDRESS OF INPUT Y1
        .GLOBL  $SAD1       ; ADDRESS OF INPUT X2
        .GLOBL  $SAD2       ; ADDRESS OF INPUT Y2

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MCORMULT    ; MEMORY ENTRY FOR COMPLEX (CORR.) MULTIPLY
        .GLOBL  RCORMULT    ; REGISTER ENTRY FOR COMPLEX (CORR.) MULTIPLY

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY
```

```
MCORMULT:

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
        LDI     @$IAD1,AR0      ; AR0 -> X1[0]
        LDI     @$IAD2,AR1      ; AR1 -> Y1[0]
        LDI     @$SAD1,AR2      ; AR2 -> X2[0]
        LDI     @$SAD2,AR3      ; AR3 -> Y2[0]

;       REGISTER BASED PARAMETER ENTRY

RCORMULT:

;       COMPLEX MULTIPLY (CORRELATION) LOOP

        SUBI    1,RC            ; RC <= N - 1

        RPTB    LOOP1           ; REPEAT BLOCK N TIMES
        MPYF    *AR0,*AR2,R1    ; R1 <= X1[I]*X2[I]
        MPYF    *AR1,*AR3,R3    ; R3 <= Y1[I]*Y2[I]
        MPYF    *AR2++,*AR1,R0  ; R0 <= Y1[I]*X2[I], INCR. AR2 AND...
||      ADDF    R1,R3,R2        ; R2 <= X1[I]*X2[I] + Y1[I]*Y2[I]
        MPYF    *AR0,*AR3++,R1  ; R1 <= X1[I]*Y2[I], INCR. AR3
        SUBF    R1,R0,R3        ; R3 <= Y1[I]*X2[I] - X1[I]*Y2[I]
LOOP1:  STF     R2,*AR0++       ; X1[I] <= R2, INCR. AR0 AND...
||      STF     R3,*AR1++       ; Y1[I] <= R3, INCR. AR1

        RETS                    ; RETURN
```

```
*************************************************
*   PROGRAM: *CONMULT                           *
*                                               *
*   WRITTEN BY: GARY A. SITTON                  *
*               GAS LIGHT SOFTWARE              *
*               HOUSTON, TEXAS                  *
*               APRIL 1989.                     *
*                                               *
*   COMPLEX IN-PLACE FREQUENCY DOMAIN CONVOLUTION:*
*   C1 <= C1 * C2, C1 AND C2 ARE BOTH OF LENGTH *
*   N, AND C1 = (X1 + I*Y1) AND C2 = (X2 + I*Y2).*
*                                               *
*   MCONMULT ENTRY PROTOCOL:                    *
*      VARIABLES FOR INPUT:                     *
*         $IAD1 -> X1[0], $IAD2 -> Y1[0],       *
*         $SAD1 -> X2[0], $SAD2 -> Y2[0],       *
*         $N = N (LENGTH), $PARMS = DATA PAGE.  *
*      INPUT RESTRICTIONS: $N > 0.              *
*      REGISTERS ALTERED: RC, DP, AR0-3 AND R0-3.*
*                                               *
*   RCONMULT ENTRY PROTOCOL:                    *
*      REGISTERS FOR INPUT:                     *
*         AR0 -> X1[0], AR1 -> Y1[0], AR2 -> X2[0],*
*         AR3 -> Y2[0], RC = N (LENGTH).        *
*      INPUT RESTRICTIONS: RC > 0.              *
*      REGISTERS ALTERED: RC, AR0-3 AND R0-3.   *
*                                               *
*   REGISTERS USED AND RESTORED: SP.            *
*   REGISTERS FOR OUTPUT: NONE.                 *
*   ROUTINES NEEDED: NONE.                      *
*************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X1
        .GLOBL  $IAD2       ; ADDRESS OF INPUT Y1
        .GLOBL  $SAD1       ; ADDRESS OF INPUT X2
        .GLOBL  $SAD2       ; ADDRESS OF INPUT Y2

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MCONMULT    ; MEMORY ENTRY FOR COMPLEX (CONV.) MULTIPLY
        .GLOBL  RCONMULT    ; REGISTER ENTRY FOR COMPLEX (CONV.) MULTIPLY

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY
```

```
MCONMULT:

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
        LDI     @$IAD1,AR0      ; AR0 -> X1[0]
        LDI     @$IAD2,AR1      ; AR1 -> Y1[0]
        LDI     @$SAD1,AR2      ; AR2 -> X2[0]
        LDI     @$SAD2,AR3      ; AR3 -> Y2[0]

;       REGISTER BASED PARAMETER ENTRY

RCONMULT:

;       COMPLEX MULTIPLY (CONVOLUTION) LOOP

        SUBI    1,RC            ; RC <= N - 1

        RPTB    LOOP2           ; REPEAT BLOCK N TIMES
        MPYF    *AR0,*AR2,R1    ; R1 <= X1[I]*X2[I]
        MPYF    *AR1,*AR3,R3    ; R3 <= Y1[I]*Y2[I]
        MPYF    *AR2++,*AR1,R0  ; R0 <= Y1[I]*X2[I], INCR. AR2 AND...
||      SUBF    R3,R1,R2        ; R2 <= X1[I]*X2[I] - Y1[I]*Y2[I]
        MPYF    *AR0,*AR3++,R1  ; R1 <= X1[I]*Y2[I], INCR. AR3
        ADDF    R1,R0,R3        ; R3 <= Y1[I]*X2[I] + X1[I]*Y2[I]
LOOP2:  STF     R2,*AR0++       ; X1[I] <= R2, INCR. AR0 AND...
||      STF     R3,*AR1++       ; Y1[I] <= R3, INCR. AR1

        RETS                    ; RETURN
```

```
*****************************************************
*  PROGRAM: *CBITREV                                *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              MARCH 1989.                          *
*                                                   *
*  BIT REVERSE INDEX MAP TWO REAL ARRAYS AS A SINGLE *
*  COMPLEX ARRAY WITH THE SWAPPING DONE IN-PLACE.   *
*  X[I], Y[I] <-> X[J], Y[J], WHERE J = BR(I).      *
*  LENGTH OF ARRAYS N >= 4 IS ABSOLUTELY REQUIRED.  *
*                                                   *
*  MCBITREV ENTRY PROTOCOL:                         *
*     VARIABLES FOR INPUT:                          *
*        $IAD1 -> X[0], $IAD2 -> Y[0],              *
*        $N = N (LENGTH), $PARMS = DATA PAGE.       *
*     INPUT RESTRICTIONS: $N >= 4.                  *
*     REGISTERS ALTERED: RC, DP, IR0, AR0-3 AND R0-3. *
*                                                   *
*  RCBITREV ENTRY PROTOCOL:                         *
*     REGISTERS FOR INPUT:                          *
*        AR0 -> X[0], AR1 -> Y[0], RC = N (LENGTH). *
*     INPUT RESTRICTIONS: RC >= 4.                  *
*     REGISTERS ALTERED: RC, IR0, AR0-3 AND R0-3.   *
*                                                   *
*  REGISTERS USED AND RESTORED: SP.                 *
*  REGISTERS FOR OUTPUT: NONE.                      *
*  ROUTINES NEEDED: NONE.                           *
*****************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X
        .GLOBL  $IAD2       ; ADDRESS OF INPUT Y

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MCBITREV    ; MEMORY ENTRY FOR COMPLEX BIT REVERSE
        .GLOBL  RCBITREV    ; REGISTER ENTRY FOR COMPLEX BIT REVERSE

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MCBITREV:
```

```
        LDP     @$PARMS     ; LOAD DATA PAGE POINTER
        LDI     @$N,RC      ; RC <= N
        LDI     @$IAD1,AR0  ; AR0 -> ARRAY X
        LDI     @$IAD2,AR1  ; AR1 -> ARRAY Y

;       REGISTER BASED PARAMETER ENTRY

RCBITREV:

        LDI     RC,IR0      ; IR0 <= N
        SUBI    3,RC        ; RC <= N - 3
        LSH     -1,IR0      ; IR0 <= N/2 FOR BIT REVERSE
        LDI     AR0,AR2     ; AR2 -> ARRAY X (BIT REV.)
        NOP     *AR2++(IR0)B    ; INCR. BR(AR2) (OUTSIDE LOOP)
        NOP     *AR0++      ; INCR. AR0 (OUTSIDE LOOP)
        LDI     AR1,AR3     ; AR3 -> ARRAY Y (BIT REV.)

;       DO BIT REVERSE SWAP ON BOTH ARRAYS
;       SKIPPING THE 0TH AND N-1ST ELEMENTS

        RPTB    LOOP3       ; REPEAT LOOP N-2 TIMES
        CMPI    AR2,AR0     ; COMPARE AR2 TO AR0
        BGED    LOOP3       ; IF AR0 >= AR2, LOOP (DELAYED)
        NOP     *AR1++      ; INCR. AR1
        NOP     *AR3++(IR0)B    ; INCR. BR(AR3)
        LDF     *AR0++,R0   ; R0 <= X[I], INCR. AR0

        LDF     *AR2,R2     ; R2 <= X[J]
        LDF     *AR1,R1     ; R1 <= Y[I]
||      LDF     *AR3,R3     ; R3 <= Y[J]
        STF     R0,*AR2     ; X[J] <= R0
||      STF     R2,*-AR0    ; X[I] <= R2
        STF     R1,*AR3     ; Y[J] <= R1
||      STF     R3,*AR1     ; Y[I] <= R3
LOOP3:  NOP     *AR2++(IR0)B    ; INCR. BR(AR2)

        RETS                ; RETURN
```

```
*****************************************************
*   PROGRAM: *FMIEEE                                *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   CONVERT AN ARRAY OF IEEE FLOATING-POINT NUMBERS TO  *
*   TMS320C30 FLOATING-POINT FORMAT.  ASSUMES NO: INF., *
*   NAN, OR DENORMALIZED NUMBERS.                   *
*                                                   *
*   MFMIEEE ENTRY PROTOCOL:                         *
*       VARIABLES FOR INPUT:                        *
*           $IAD1 -> X[0], $N = N (LENGTH),         *
*           $PARMS = DATA PAGE.                     *
*       INPUT RESTRICTIONS: $N > 0.                 *
*       REGISTERS ALTERED: RC, DP, AR0-1 AND R0-1.  *
*                                                   *
*   RFMIEEE ENTRY PROTOCOL:                         *
*       REGISTERS FOR INPUT:                        *
*           AR0 -> X[0], RC = N (LENGTH).           *
*       INPUT RESTRICTIONS: RC > 0.                 *
*       REGISTERS ALTERED: RC, AR0-1 AND R0-1.      *
*                                                   *
*   REGISTERS USED AND RESTORED: SP.                *
*   REGISTERS FOR OUTPUT: NONE.                     *
*   ROUTINES NEEDED: NONE.                          *
*****************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MFMIEEE     ; MEMORY ENTRY FOR IEEE -> 'C30 CONVERSION
        .GLOBL  RFMIEEE     ; REGISTER ENTRY FOR IEEE -> 'C30 CONVERSION

;       CONSTANTS FOR BOTH CONVERSIONS

        .DATA

CTAB    .WORD       0FF800000H
        .WORD       0FF000000H
        .WORD       07F000000H
        .WORD       080000000H
        .WORD       081000000H
```

```
TABA    .WORD       CTAB

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MFMIEEE:

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
        LDI     @$IAD1,AR0      ; AR0 -> IEEE ARRAY

;       REGISTER BASED PARAMETER ENTRY

RFMIEEE:

        SUBI    1,RC            ; RC <= N - 1
        LDP     @CTAB           ; LOAD DATA PAGE POINTER
        LDI     @TABA,AR1       ; AR1 -> CONSTANT TABLE

;       IEEE -> 'C30 CONVERSION LOOP

        RPTB    LOOP4           ; REPEAT LOOP N TIMES
        AND     *AR0,*AR1,R0    ; REPLACE FRACTION WITH 0
        ADDI    *AR0,R0         ; SHIFT SIGN AND EXPONENT INSERTING 0
        LDIZ    *+AR1(1),R0     ; IF ALL ZERO, LOAD 'C30 0.0
        LDI     *AR0,R1         ; TEST ORIGINAL NUMBER
        BGED    LOOP4           ; IF >= 0, STORE NUMBER (DELAYED)
        SUBI    *+AR1(2),R0     ; REMOVE EXPONENT BIAS (127)
        PUSH    R0              ; SAVE AS AN INTEGER
        POPF    R0              ; UNSAVE AS A FLT. PT. NUMBER

        NEGF    R0              ; NEGATE 'C30 NUMBER

LOOP4:  STF     R0,*AR0++       ; STORE 'C30 NUMBER, INCR. AR0

        RETS                    ; RETURN
```

```
*****************************************************
*   PROGRAM: *TOIEEE                                *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               APRIL 1989.                         *
*                                                   *
*   CONVERT AN ARRAY OF TMS320C30 FLOATING-POINT    *
*   NUMBERS TO IEEE FLOATING-POINT FORMAT.  ZERO    *
*   IS THE ONLY SPECIAL CASE.                       *
*                                                   *
*   MTOIEEE ENTRY PROTOCOL:                         *
*       VARIABLES FOR INPUT:                        *
*           $IAD1 -> X[0], $N = N (LENGTH),         *
*           $PARMS = DATA PAGE.                     *
*       INPUT RESTRICTIONS: $N > 0.                 *
*       REGISTERS ALTERED: RC, DP, AR0-1 AND R0-1.  *
*                                                   *
*   RTOIEEE ENTRY PROTOCOL:                         *
*       REGISTERS FOR INPUT:                        *
*           AR0 -> X[0], RC = N (LENGTH).           *
*       INPUT RESTRICTIONS: RC > 0.                 *
*       REGISTERS ALTERED: RC, AR0-1 AND R0-1.      *
*                                                   *
*   REGISTERS USED AND RESTORED: SP.                *
*   REGISTERS FOR OUTPUT: NONE.                     *
*   ROUTINES NEEDED: NONE.                          *
*                                                   *
*   NOTE: *TOIEEE SHARES THE CTAB TABLE FROM *FMIEEE  *
*****************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MTOIEEE     ; MEMORY ENTRY FOR 'C30 -> IEEE CONVERSION
        .GLOBL  RTOIEEE     ; REGISTER ENTRY FOR 'C30 -> IEEE CONVERSION

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MTOIEEE:
```

```
        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
        LDI     @$IAD1,AR0      ; AR0 -> 'C30 ARRAY

;       REGISTER BASED PARAMETER ENTRY

RTOIEEE:

        SUBI    1,RC            ; RC <= N - 1
        LDP     @CTAB           ; LOAD DATA PAGE POINTER
        LDI     @TABA,AR1       ; AR1 -> CONSTANT TABLE

;       'C30 -> IEEE CONVERSION LOOP

        RPTB    LOOP5           ; REPEAT LOOP N TIMES
        ABSF    *AR0,R0         ; TEST !NUMBER!
        LDFZ    *++AR1(4),R0    ; IF == 0, LOAD FAKE 0.0
        LSH     1,R0            ; SHIFT OFF SIGN BIT
        PUSHF   R0              ; SAVE AS A FLT. PT.
        LDF     *AR0,R1         ; TEST ORIGINAL NUMBER
        BGED    LOOP5           ; IF >= 0, STORE NUMBER (DELAYED)
        POP     R0              ; UNSAVE AS AN INTEGER
        ADDI    *++AR1(2),R0    ; ADD EXPONENT BIAS (127)
        LSH     -1,R0           ; ADJUST FOR SIGN BIT

        OR      *++AR1(3),R0    ; NEGATE IEEE NUMBER

LOOP5:  STI     R0,*AR0++       ; STORE IEEE NUMBER, INCR. AR0

        RETS                    ; RETURN
```

```
*************************************************************
*  PROGRAM: *VECMULT                                        *
*                                                           *
*  WRITTEN BY: GARY A. SITTON                               *
*              GAS LIGHT SOFTWARE                           *
*              HOUSTON, TEXAS                               *
*              FEBRUARY 1989.                               *
*                                                           *
*  SCALAR - VECTOR MULTIPLY: X[I] <= X[I]*C, C IS A         *
*  CONSTANT AND THE ARRAY X IS OF LENGTH N >= 1.            *
*                                                           *
*  MVECMULT ENTRY PROTOCOL:                                 *
*      VARIABLES FOR INPUT:                                 *
*          $IAD1 -> X[0], $N = N (LENGTH),                  *
*          $CNST = C, $PARMS = DATA PAGE.                   *
*      INPUT RESTRICTIONS: $N > 0.                          *
*      REGISTERS ALTERED: RC, DP, AR0 AND R0-1.             *
*                                                           *
*  RVECMULT ENTRY PROTOCOL:                                 *
*      REGISTERS FOR INPUT:                                 *
*          AR0 -> X[0], R0 = C, RC = N (LENGTH).            *
*      INPUT RESTRICTIONS: RC > 0.                          *
*      REGISTERS ALTERED: RC, AR0 AND R1.                   *
*                                                           *
*  REGISTERS USED AND RESTORED: SP.                         *
*  REGISTERS FOR OUTPUT: NONE.                              *
*  ROUTINES NEEDED: NONE.                                   *
*************************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $CNST       ; ADDRESS OF CONSTANT C
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MVECMULT    ; MEMORY ENTRY FOR SCALAR - VECTOR MULTIPLY
        .GLOBL  RVECMULT    ; REGISTER ENTRY FOR SCALAR - VECTOR MULTIPLY

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MVECMULT:

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
```

```
        LDI     @$IAD1,AR0      ; AR0 -> X[0]
        LDF     @$CNST,R0       ; R0 <= C

;       REGISTER BASED PARAMETER ENTRY

RVECMULT:

        SUBI    2,RC            ; RC <= N - 2
        MPYF    R0,*AR0,R1      ; R1 <= C*X[0]
        CMPI    0,RC            ; COMPARE RC TO 0
        BLT     SKIP1           ; IF RC < 0 THEN SKIP LOOP

;       SCALAR - VECTOR MULTIPLY LOOP

        RPTS    RC              ; REPEAT INST. N-1 TIMES
        MPYF    R0,*++AR0,R1    ; R1 <= C*X[I+1]
||      STF     R1,*AR0         ; X[I] <= C*X[I]

SKIP1:  STF     R1,*AR0         ; X[N-1] <= C*X[N-1]

        RETS                    ; RETURN
```

```
*****************************************************
*  PROGRAM: *CONMOV                                 *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              FEBRUARY 1989.                       *
*                                                   *
*  SCALAR -> VECTOR MOVE: X[I] <= C, C IS A         *
*  CONSTANT AND THE ARRAY X IS OF LENGTH N.         *
*                                                   *
*  MCONMOV ENTRY PROTOCOL:                          *
*      VARIABLES FOR INPUT:                         *
*          $IAD1 -> X[0], $N = N (LENGTH),          *
*          $CNST = C, $PARMS = DATA PAGE.           *
*      INPUT RESTRICTIONS: $N > 0.                  *
*      REGISTERS ALTERED: RC, DP, AR0, AND R0.      *
*                                                   *
*  RCONMOV ENTRY PROTOCOL:                          *
*      REGISTERS FOR INPUT:                         *
*          AR0 -> X[0], R0 = C, RC = N (LENGTH).    *
*      INPUT RESTRICTIONS: RC > 0.                  *
*      REGISTERS ALTERED: RC, AR0.                  *
*                                                   *
*  REGISTERS USED AND RESTORED: SP.                 *
*  REGISTERS FOR OUTPUT: NONE.                      *
*  ROUTINES NEEDED: NONE.                           *
*****************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $CNST       ; ADDRESS OF CONSTANT C
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MCONMOV     ; MEMORY ENTRY FOR CONSTANT TO VECTOR MOVE
        .GLOBL  RCONMOV     ; REGISTER ENTRY FOR CONSTANT TO VECTOR MOVE

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MCONMOV:

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
```

```
        LDI     @$IAD1,AR0      ; AR0 -> X[0]
        LDF     @$CNST,R0       ; R0 <= C

;       REGISTER BASED PARAMETER ENTRY

RCONMOV:

        SUBI    1,RC            ; RC <= N - 1

;       SCALAR TO VECTOR MOVE LOOP

        RPTS    RC              ; REPEAT INST. N TIMES
        STF     R0,*AR0++       ; X[I] <= C

        RETS                    ; RETURN
```

```
*****************************************************
*   PROGRAM: *VECMOV                                *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               FEBRUARY 1989.                      *
*                                                   *
*   VECTOR MOVE: Y[I] <= X[I], I = 0,...,N-1 (N >= 1).*
*                                                   *
*   MVECMOV ENTRY PROTOCOL:                         *
*       VARIABLES FOR INPUT:                        *
*           $IAD1 -> X[0], $IAD2 -> Y[0],           *
*           $N = N (LENGTH), $PARMS = DATA PAGE.    *
*       INPUT RESTRICTIONS: $N > 0.                 *
*       REGISTERS ALTERED: RC, DP, AR0-1, AND R0.   *
*                                                   *
*   RVECMOV ENTRY PROTOCOL:                         *
*       REGISTERS FOR INPUT:                        *
*           AR0 -> X[0], AR1 -> Y[0], RC = N (LENGTH).*
*       INPUT RESTRICTIONS: RC > 0.                 *
*       REGISTERS ALTERED: RC, AR0-1, AND R0.       *
*                                                   *
*   REGISTERS USED AND RESTORED: SP.                *
*   REGISTERS FOR OUTPUT: NONE.                     *
*   ROUTINES NEEDED: NONE.                          *
*****************************************************

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $PARMS      ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N          ; ARRAY LENGTH N
        .GLOBL  $IAD1       ; ADDRESS OF INPUT X
        .GLOBL  $IAD2       ; ADDRESS OF INPUT Y

;       EXTERNAL PROGRAM NAMES

        .GLOBL  MVECMOV     ; MEMORY ENTRY FOR VECTOR TO VECTOR MOVE
        .GLOBL  RVECMOV     ; REGISTER ENTRY FOR VECTOR TO VECTOR MOVE

;       START OF PROGRAM AREA

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MVECMOV:

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,RC          ; RC <= N
        LDI     @$IAD1,AR0      ; AR0 -> X[0]
```

```
        LDI     @$IAD2,AR1      ; AR1 -> Y[0]

;       REGISTER BASED PARAMETER ENTRY

RVECMOV:

        SUBI    2,RC            ; RC <= N - 2
        LDF     *AR0++,R0       ; R0 <= X[0]
        CMPI    0,RC            ; COMPARE RC TO 0
        BLT     SKIP2           ; IF RC < 0 THEN SKIP LOOP

;       VECTOR MOVE LOOP

        RPTS    RC              ; REPEAT INST. N-1 TIMES
        LDF     *AR0++,R0       ; R0 <= X[I+1]
||      STF     R0,*AR1++       ; MOVE X[I] TO Y[I]

SKIP2:  STF     R0,*AR1         ; MOVE X[N-1] TO Y[N-1]

        RETS                    ; RETURN

        .END
```

```
*****************************************************************
*
*   PROGRAM: $FFT2.ASM
*
*   RADIX 2 FFT ROUTINES
*
*   $FFT2.ASM CONSISTS OF THE FOLLOWING ROUTINES:
*
*       CFFFT2 - COMPLEX DIF FORWARD RADIX 2 FFT USING SEPARATE REAL AND
*                IMAGINARY ARRAYS AND 3/4 CYCLE SINE TABLE.
*
*       CIFFT2 - COMPLEX DIT INVERSE RADIX 2 FFT USING SEPARATE REAL AND
*                IMAGINARY ARRAYS AND 3/4 CYCLE SINE TABLE (DOES NOT INCLUDE
*                THE 1/N SCALE FACTOR.
*
*****************************************************************
```

```
*****************************************************
*                                                   *
*  PROGRAM: CFFFT2                                  *
*                                                   *
*  WRITTEN BY: GARY A. SITTON                       *
*              GAS LIGHT SOFTWARE                   *
*              HOUSTON, TEXAS                       *
*              MARCH 1989.                          *
*                                                   *
*  SPECIAL VERSION USES 3/4 SINE TABLE LOOKUP WITH  *
*  THE PARAMETERS PASSED IN PREDEFINED MEMORY LOCATIONS.  *
*  COMPLEX RADIX-2 DIF FORWARD FFT FOR THE TMS320C30.     *
*  THIS PROGRAM ASSUMES NORMAL ORDERED DATA AS INPUT,     *
*  BUT LEAVES THE OUTPUT INDEXED IN BIT REVERSED ORDER.   *
*  TWO POINTERS ARE USED FOR SEPARATE REAL AND IMAGINARY  *
*  ARRAYS.                                          *
*                                                   *
*  VARIABLES FOR INPUT:                             *
*      $IAD1 -> REAL[0], $IAD2 -> IMAG[0],          *
*      $N = N (LENGTH), $M = M (LOG2(N)),           *
*      $SINE -> SINE TABLE, $PARMS = DATA PAGE.     *
*  INPUT RESTRICTIONS: $N > 1.                      *
*  REGISTERS ALTERED: RC, DP, IR0-1, AR0-7, AND R0-7.     *
*  REGISTERS USED AND RESTORED: SP.                 *
*  REGISTERS FOR OUTPUT: NONE.                      *
*  ROUTINES NEEDED: NONE.                           *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  CFFFT2          ; ENTRY POINT FOR EXECUTION

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $SINE           ; SINE TABLE ADDRESS
        .GLOBL  $PARMS          ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N              ; FFT LENGTH, N = 2**M
        .GLOBL  $M              ; M = LOG2(N) >= 2
        .GLOBL  $IAD1           ; REAL INPUT ARRAY ADDRESS
        .GLOBL  $IAD2           ; IMAGINARY INPUT ARRAY ADDRESS
        .
        .TEXT

;       START OF DIF FFFT PROGRAM

CFFFT2:

;       INITIALIZE LOOP VARIABLES

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,IR0         ; IR0 <= N1 (INIT. N)
        LDI     IR0,IR1         ; IR1 <= N
        LSH     -2,IR1          ; IR1 <= N/4, OFFSET FOR COSINE
        LDI     0,AR6           ; AR6 <= K (INIT. 0)
        LDI     IR0,R7          ; R7 <= N1
        LSH     -1,R7           ; R7 <= N2 (INIT. N/2)
        LDI     1,R5            ; R5 <= IE (INIT. 1)

;       OUTER LOOP

FLOOP:  ADDI    1,AR6           ; K <= K + 1
        LDI     @$IAD1,AR0      ; AR0 -> X(0)
        ADDI    R7,AR0,AR1      ; AR1 -> X(L)
        LDI     @$IAD2,AR2      ; AR2 -> Y(0)
        ADDI    R7,AR2,AR3      ; AR3 -> Y(L)
        LDI     R5,RC           ; SETUP 1ST INNER LOOP REPEAT COUNTER.
        SUBI    1,RC            ; RC (ONE LESS THAN THE DESIRED #)

;       FIRST INNER LOOP (UNITY TWIDDLE FACTOR)

        RPTB    FBLK1           ; REPEAT BLOCK IE TIMES
        ADDF    *AR0,*AR1,R0    ; R0 <= X(I) + X(L)
        SUBF    *AR1,*AR0,R1    ; R1 <= X(I) - X(L)
        ADDF    *AR2,*AR3,R2    ; R2 <= Y(I) + Y(L) AND...
        SUBF    *AR3,*AR2,R3    ; R3 <= Y(I) - Y(L)
        STF     R0,*AR0++(IR0)  ; X(I) <= R0, INCR. AR0 AND...
;;      STF     R1,*AR1++(IR0)  ; X(L) <= R1, INCR. AR1
FBLK1:  STF     R2,*AR2++(IR0)  ; Y(I) <= R2, INCR. AR2 AND...
;;      STF     R3,*AR3++(IR0)  ; Y(L) <= R3, INCR. AR3

;       PROGRAM EXIT TEST

        CMPI    @$M,AR6         ; COMPARE M TO K
        RETSGE                  ; IF K >= M THEN RETURN

;       MAIN INNER LOOP

        LDI     2,AR7           ; J <= 2, (PRE-INCREMENTED)
        LDI     1,AR0           ; AR0 <= I (INIT. 1)
        LDI     1,AR2           ; AR2 <= I (INIT. 1)
        LDI     @$SINE,AR5      ; AR5 <= SINTAB[IA] (INIT. IA = 0)

FINLOP: ADDI    R5,AR5          ; AR5 -> SINTAB[IA <= IA + IE]
        LDF     *AR5,R6         ; R6 <= SIN(X), (X = (2*PI/N)*IA)
        ADDI    AR5,IR1,AR4     ; AR4 -> COS(X)
        ADDI    @$IAD1,AR0      ; AR0 -> X(I)
        ADDI    @$IAD2,AR2      ; AR2 -> Y(I)
        ADDI    R7,AR0,AR1      ; AR1 -> X(L)
        ADDI    R7,AR2,AR3      ; AR3 -> Y(L)
        LDI     R5,RC           ; SETUP 2ND INNER LOOP REPEAT COUNTER.
        SUBI    1,RC            ; RC (ONE LESS THAN THE DESIRED #)

;       SECOND INNER LOOP (DOES TWIDDLE ROTATION)

        RPTB    FBLK2           ; REPEAT BLOCK IE TIMES
```

```
        SUBF    *AR1,*AR0,R2    ; R2 <= XT = X(I) - X(L)
        SUBF    *AR3,*AR2,R1    ; R1 <= YT = Y(I) - Y(L)
        MPYF    R2,R6,R0        ; R0 <= XT*SIN AND...
||      ADDF    *AR2,*AR3,R3    ; R3 <= Y(I) + Y(L)
        MPYF    R1,*AR4,R3      ; R3 <= YT*COS AND...
||      STF     R3,*AR2++(IR0)  ; Y(I) <= Y(I) + Y(L), INCR. AR2
        SUBF    R0,R3,R4        ; R4 <= COS*YT - SIN*XT
        MPYF    R1,R6,R0        ; R0 <= SIN*YT AND...
||      ADDF    *AR0,*AR1,R3    ; R3 <= X(I) + X(L)
        MPYF    R2,*AR4,R3      ; R3 <= COS*XT AND...
||      STF     R3,*AR0++(IR0)  ; X(I) <= X(I) + X(L) , INCR. AR0
        ADDF    R0,R3           ; R3 <= COS*XT + SIN*YT
FBLK2:  STF     R3,*AR1++(IR0)  ; X(L) <= COS*XT + SIN*YT, INCR. AR1 AND...
||      STF     R4,*AR3++(IR0)  ; Y(L) <= COS*YT - SIN*XT, INCR. AR3

        CMPI    R7,AR7          ; COMPARE N2 TO J

        BLTD    FINLOP          ; IF J < N2 THEN LOOP (DELAYED)
        LDI     AR7,AR0         ; AR0 <= J
        LDI     AR7,AR2         ; AR2 <= J
        ADDI    1,AR7           ; J <= J + 1

        BRD     FLOOP           ; NEXT FFT STAGE (DELAYED)
        LSH     1,R5            ; IE <= 2*IE
        LDI     R7,IR0          ; N1 <= N2
        LSH     -1,R7           ; N2 <= N2/2

;       END OF OUTER LOOP
```

```
*****************************************************
*                                                   *
*   PROGRAM: CIFFT2                                 *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MARCH 1989.                         *
*                                                   *
*   SPECIAL VERSION USES 3/4 SINE TABLE LOOKUP WITH *
*   THE PARAMETERS PASSED IN PREDEFINED MEMORY LOCATIONS.
*   COMPLEX RADIX-2 DIT INVERSE FFT FOR THE TMS320C30.
*   THIS PROGRAM ASSUMES BIT REVERSED ORDERED DATA AS
*   INPUT, BUT LEAVES THE OUTPUT INDEXED IN NORMAL ORDER.
*   TWO POINTERS ARE USED FOR SEPARATE REAL AND IMAGINARY
*   ARRAYS.                                         *
*                                                   *
*   VARIABLES FOR INPUT:                            *
*       $IAD1 -> REAL[0], $IAD2 -> IMAG[0],         *
*       $N = N (LENGTH), $M = M (LOG2(N)),          *
*       $SINE -> SINE TABLE, $PARMS = DATA PAGE.    *
*   INPUT RESTRICTIONS: $N > 1.                     *
*   REGISTERS ALTERED: RC, DP, IR0-1, AR0-7, AND R0-7.
*   REGISTERS USED AND RESTORED: SP.                *
*   REGISTERS FOR OUTPUT: NONE.                     *
*   ROUTINES NEEDED: NONE.                          *
*****************************************************

;       EXTERNAL PROGRAM NAMES

        .GLOBL  CIFFT2          ; ENTRY POINT FOR EXECUTION

;       EXTERNAL MEMORY ADDRESSES

        .GLOBL  $SINE           ; SINE TABLE ADDRESS
        .GLOBL  $PARMS          ; PARAMETER PAGE ADDRESS

;       EXTERNAL VARIABLE ADDRESSES

        .GLOBL  $N              ; FFT LENGTH, N = 2**M
        .GLOBL  $M              ; M = LOG2(N) >= 2
        .GLOBL  $IAD1           ; REAL INPUT ARRAY ADDRESS
        .GLOBL  $IAD2           ; IMAGINARY INPUT ARRAY ADDRESS

;       START OF DIT IFFT PROGRAM

        .TEXT

CIFFT2:

;       INITIALIZE LOOP VARIABLES

        LDP     @$PARMS         ; LOAD DATA PAGE POINTER
        LDI     @$N,IR0         ; IR0 <= N
```

```
        LDI     IR0,IR1         ; IR1 <= N
        LSH     -2,IR1          ; IR1 <= N/4, OFFSET FOR COSINE
        LDI     @$M,AR6         ; AR6 <= K (INIT. M)
        LDI     1,R7            ; R7 <= N2 (INIT. 1)
        LDI     IR0,R5          ; R5 <= N
        LSH     -1,R5           ; R5 <= IE (INIT. N/2)
        LDI     2,IR0           ; IR0 <= N1 (INIT. 2)

;       OUTER LOOP

ILOOP:  LDI     @$IAD1,AR0      ; AR0 -> X(0)
        ADDI    R7,AR0,AR1      ; AR1 -> X(L)
        LDI     @$IAD2,AR2      ; AR2 -> Y(0)
        ADDI    R7,AR2,AR3      ; AR3 -> Y(L)
        LDI     R5,RC           ; SETUP 1ST INNER LOOP REPEAT COUNTER.
        SUBI    1,RC            ; RC (ONE LESS THAN THE DESIRED #)

;       FIRST INNER LOOP (UNITY TWIDDLE FACTOR)

        RPTB    IBLK1           ; REPEAT BLOCK IE TIMES
        ADDF    *AR0,*AR1,R0    ; R0 <= X(I) + X(L)
        SUBF    *AR1,*AR0,R1    ; R1 <= X(I) - X(L)
        ADDF    *AR2,*AR3,R2    ; R2 <= Y(I) + Y(L) AND...
        SUBF    *AR3,*AR2,R3    ; R3 <= Y(I) - Y(L)
        STF     R0,*AR0++(IR0)  ; X(I) <= R0, INCR. AR0 AND...
||      STF     R1,*AR1++(IR0)  ; X(L) <= R1, INCR. AR1
IBLK1:  STF     R2,*AR2++(IR0)  ; Y(I) <= R2, INCR. AR2 AND...
||      STF     R3,*AR3++(IR0)  ; Y(L) <= R3, INCR. AR3

        CMPI    @$M,AR6         ; COMPARE M TO K
        BEQD    SKIP            ; IF K == M THEN SKIP TWIDDLED LOOP

;       MAIN INNER LOOP

        LDI     2,AR7           ; J <= 2, (PRE-INCREMENTED)
        LDI     1,AR0           ; AR0 <= I (INIT. 1)
        LDI     1,AR2           ; AR2 <= I (INIT. 1)
        LDI     @$SINE,AR5      ; AR5 <= IA (INIT. 0)

IINLOP: ADDI    R5,AR5          ; AR5 -> SINTAB[IA <= IA + IE]
        LDF     *AR5,R6         ; R6 <= SIN(X), (X = (2*PI/N)*IA)
        ADDI    AR5,IR1,AR4     ; AR4 -> COS(X)
        ADDI    @$IAD1,AR0      ; AR0 -> X(I)
        ADDI    @$IAD2,AR2      ; AR2 -> Y(I)
        ADDI    R7,AR0,AR1      ; AR1 -> X(L)
        ADDI    R7,AR2,AR3      ; AR3 -> Y(L)
        LDI     R5,RC           ; SETUP 2ND INNER LOOP REPEAT COUNTER.
        SUBI    1,RC            ; RC (ONE LESS THAN THE DESIRED #)

;       SECOND INNER LOOP (DOES TWIDDLE ROTATION)

        RPTB    IBLK2           ; REPEAT BLOCK IE TIMES
        MPYF    *AR4,*AR1,R4    ; R4 <= COS*X(L)
        MPYF    R6,*AR3,R3      ; R3 <= SIN*Y(L)
        MPYF    *AR4,*AR3,R0    ; R0 <= COS*Y(L), AND...
||      SUBF    R3,R4,R2        ; R2 <= YT = COS*X(L) - SIN*Y(L)
        MPYF    R6,*AR1,R1      ; R1 <= SIN*X(L), AND...
||      SUBF    R2,*AR0,R3      ; R3 <= X(I) - XT
        ADDF    R0,R1,R4        ; R4 <= YT = COS*Y(L) + SIN*X(L)
        SUBF    R4,*AR2,R3      ; R3 <= Y(I) - YT, AND...
||      STF     R3,*AR1++(IR0)  ; X(L) <= X(I) - XT, INCR. AR1
        ADDF    R2,*AR0,R3      ; R3 <= X(I) + XT, AND...
||      STF     R3,*AR3++(IR0)  ; Y(L) <= Y(I) - YT, INCR. AR3
        ADDF    R4,*AR2,R4      ; R4 <= Y(I) + YT
IBLK2:  STF     R3,*AR0++(IR0)  ; X(I) <= X(I) + XT, INCR. AR0 AND...
||      STF     R4,*AR2++(IR0)  ; Y(I) <= Y(I) + YT, INCR. AR2

        CMPI    R7,AR7          ; COMPARE N2 TO J

        BLTD    IINLOP          ; IF J < N2 THEN LOOP (DELAYED)
        LDI     AR7,AR0         ; AR0 <= J
        LDI     AR7,AR2         ; AR2 <= J
        ADDI    1,AR7           ; J <= J + 1

SKIP:   SUBI    1,AR6           ; K <= K - 1
        CMPI    0,AR6           ; COMPARE 0 TO K
        BGTD    ILOOP           ; IF K > 0 THEN LOOP (DELAYED)
        LSH     -1,R5           ; IE <= IE/2
        LDI     IR0,R7          ; N2 <= N1
        LSH     1,IR0           ; N1 <= 2*N1

;       PROGRAM EXIT POINT

        RETS                    ; RETURN

        .END
```

```
*************************************************************************
*
*   PROGRAM: $LINALG.ASM
*
*   LINEAR ALGEBRA ROUTINES
*
*   $LINALG.ASM CONSISTS OF THE FOLLOWING ROUTINES:
*
*       *SOLUTN  - SOLVES A WELL CONDITIONED SYSTEM OF LINEAR EQUATIONS WITH
*                  ANY NUMBER OF DEPENDENT VARIABLE SETS. USES NO (DIAGONAL)
*                  PIVOTING WITH NORMAL-PRECISION FLOATING-POINT MATH.
*
*       *SOLUTNX - SOLVES A WELL CONDITIONED SYSTEM OF LINEAR EQUATIONS WITH
*                  ANY NUMBER OF DEPENDENT VARIABLE SETS. USES NO (DIAGONAL)
*                  PIVOTING WITH EXTENDED-PRECISION FLOATING-POINT MATH.
*
*************************************************************************
```

```
*****************************************************
*   PROGRAM: *SOLUTN                                *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MAY 1989.                           *
*                                                   *
*   (NORMAL PRECISION VERSION)                      *
*                                                   *
*   SOLVES A SYSTEM OF LINEAR EQUATIONS A*X = Y IN THE *
*   TABLEAU FORMAT B = A:-Y, AN M X N MATRIX.  THIS  *
*   MEANS THAT A IS AN M X M SQUARE MATRIX OF COEFFI- *
*   CIENTS, AND -Y IS AN M X N-M RECTANGULAR MATRIX  *
*   OF N-M VECTORS EACH HAVING M ELEMENTS.  EACH DEPEN- *
*   DENT VARIABLE COLUMN VECTOR IS NEGATED AND APPENDED *
*   TO THE COEFFICIENT MATRIX A.  THE SET OF N-M INDE- *
*   PENDENT SOLUTION VECTORS X WILL APPEAR IN PLACE OF *
*   THE ORIGINAL APPENDED COLUMNS WHEN SOLUTN FINISHES. *
*   ROW MAJOR MATRIX STORAGE FORMAT IS ASSUMED PLUS  *
*   THE PROGRAM ASSUMES N > M > 1 AND B[0, 0] != 0.0 *
*   SINCE THE METHOD USES DIAGONAL PIVOTING AND STARTS *
*   WITH B[0, 0].  ANY PIVOT ELEMENT < 10**-8 IN ITS *
*   ABSOLUTE VALUE WILL IMPLY AN "ILL CONDITIONED"   *
*   SYSTEM OF EQUATIONS, I. E. NOT HAVING SUFFICIENT *
*   LINEAR INDEPENDENCE,  AND WILL RESULT IN AN INCOM- *
*   PLETE SOLUTION.  AN INCOMPLETE SOLUTION WILL BE  *
*   INDICATED BY THE VALUE OF R3 == 0.0 ON EXIT, ELSE *
*   R3 != 0.0 AND EQUALS THE LAST PIVOT ELEMENT VALUE. *
*                                                   *
*   MSOLUTN ENTRY PROTOCOL:                         *
*       VARIABLES FOR INPUT:                        *
*           $IAD1 -> B[0, 0], $NROW = M,            *
*           $NCOL = N, $PARMS = DATA PAGE.          *
*       INPUT RESTRICTIONS: N > M > 1.              *
*       REGISTERS ALTERED: RC, DP, AR0-7, IR0-1,    *
*                          AND R0-7.                *
*                                                   *
*   RSOLUTN ENTRY PROTOCOL:                         *
*       REGISTERS FOR INPUT:                        *
*           AR0 -> B[0, 0], AR1 = M, AR2 = N.       *
*       INPUT RESTRICTIONS: AR2 > AR1 > 1.          *
*       REGISTERS ALTERED: RC, AR0-7, IR0-1, AND R0-7. *
*                                                   *
*   REGISTERS USED AND RESTORED: SP.                *
*   REGISTERS FOR OUTPUT: R3.                       *
*   ROUTINES NEEDED: FPINV (SEE $MATH).             *
*                                                   *
*   NOTE: COMMENTED OUT RND INSTRUCTIONS MAY BE ACTI- *
*   VATED FOR ADDITIONAL ACCURACY WITH LOSS OF SPEED. *
*****************************************************

;       EXTERNAL PROGRAM NAMES
```

```
        .GLOBL  MSOLUTN     ; MEMORY BASED ENTRY
        .GLOBL  RSOLUTN     ; REGISTER BASED ENTRY
        .GLOBL  FPINV       ; RECIPROCAL ROUTINE

;       EXTERNAL PARAMETER NAMES

        .GLOBL  $PARMS      ; PARAMETER SPACE ADDRESS
        .GLOBL  $IAD1       ; POINTER TO MATRIX B, ADDRESS OF B[0, 0]
        .GLOBL  $NROW       ; NUMBER OF ROWS IN B, VALUE OF M
        .GLOBL  $NCOL       ; NUMBER OF COLUMNS IN B, VALUE OF N

;       INTERNAL CONSTANTS

        .DATA

EPSN    .FLOAT  1.0E-8      ; SINGULARITY CRITERION
ZERO    .SET    0.0         ; SINGULARITY FLAG

;       START SOLUTN PROGRAM

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MSOLUTN:

        LDP     @$PARMS     ; LOAD DATA PAGE POINTER
        LDI     @$IAD1,AR0  ; AR0 -> B[0, 0]
        LDI     @$NROW,AR1  ; AR1 <= M
        LDI     @$NCOL,AR2  ; AR2 <= N

;       REGISTER BASED PARAMETER ENTRY

RSOLUTN:

;       SETUP LOOP REGISTERS

        LDP     @EPSN       ; LOAD DATA PAGE POINTER
        LDI     0,IR0       ; IR0 <= K (INIT. 0)
        LDI     AR0,AR3     ; AR3 -> B[0, 0]
        SUBI    1,AR1       ; AR1 <= M-1
        LDI     AR2,AR6     ; AR6 <= N
        SUBI    2,AR6       ; AR6 <= N-2

;       MAIN LOOP (K INDEX)

KLOOP:  LDF     *+AR3(IR0),R3   ; R3 <= B[K, K], NEXT PIVOT
        ABSF    R3,R0           ; R0 <= |R3|
        CMPF    @EPSN,R0        ; COMPARE |B[K, K]| TO EPS
        BLT     SING            ; IF |B[K, K]| < EPS THEN STOP

;       COMPUTE RECIPROCAL OF -PIVOT ELEMENT

        NEGF    R3,R0           ; R0 <= -B[K, K]
        CALL    FPINV           ; R0 <= -1/B[K, K]
        RND     R0              ; ROUND INVERSE

;       DIVIDE RIGHT PART OF PIVOT ROW BY -PIVOT ELEMENT

        ADDI    AR3,IR0,AR7     ; AR7 -> B[K, K]
        LDI     AR6,RC          ; RC <= N-K-2

        RPTB    DLOOP           ; REPEAT DIVIDE LOOP N-K-1 TIMES
        MPYF    R0,*++AR7,R2    ; R2 <= B[K, J]*(-1/B[K, K])
*       RND     R2              ; REMOVE "*" TO ROUND *
DLOOP:  STF     R2,*AR7         ; B[K, J] <= R2

;       START INNER LOOP (I INDEX)

        LDI     0,IR1           ; IR1 <= I (INIT. 0)
        LDI     AR0,AR4         ; AR4 -> B[0, 0]

        CMPI    IR0,IR1         ; COMPARE I TO K
ILOOP:  BEQ     SKIP            ; IF I == K THEN SKIP PIVOT ROW

;       COMPLETE PIVOTING OPERATION

        ADDI    AR4,IR0,AR5     ; AR5 -> B[I, K]
        LDF     *AR5,R0         ; R0 <= B[I, K]
        LDI     AR6,RC          ; RC <= N-K-2
        CMPI    1,RC            ; COMPARE RC TO 1
        BLTD    JUMP            ; IF RC < 1 THEN NO RPTB (DELAYED)

        SUBI    1,RC            ; RC <= N-K-3
        ADDI    AR3,IR0,AR7     ; AR7 -> B[K, J]
        MPYF    R0,*++AR7,R1    ; R1 <= B[K, K+1]*B[I, K]

;       START INNER-INNER LOOP (J INDEX)

        RPTB    JLOOP           ; REPEAT PIVOT LOOP N-K-2 TIMES
        MPYF    R0,*++AR7,R1    ; R1 <= B[K, J]*B[I, K]
||      ADDF    R1,*++AR5,R2    ; R2 <= B[I, J] + R1
*       RND     R2              ; REMOVE "*" TO ROUND +
JLOOP:  STF     R2,*AR5         ; B[I, J] <= R2

;       END OF INNER-INNER LOOP (J INDEX)

JUMP:   ADDF    R1,*++AR5,R2    ; R2 <= B[I, N-1] + R1
*       RND     R2              ; REMOVE "*" TO ROUND +
        STF     R2,*AR5         ; B[I, N-1] <= R2

SKIP:   CMPI    AR1,IR1         ; COMPARE I TO M-1
        BLTD    ILOOP           ; IF I < M-1 THEN LOOP (DELAYED)

        ADDI    AR2,AR4         ; AR4 -> B[I+1, 0]
        ADDI    1,IR1           ; I <= I+1
        CMPI    IR0,IR1         ; COMPARE I TO K
```

```
;       END OF INNER LOOP (I INDEX)

        CMPI    AR1,IR0         ; COMPARE K TO M-1
        BLTD    KLOOP           ; IF K < M-1 THEN LOOP

        ADDI    AR2,AR3         ; AR3 -> B[K+1, 0]
        ADDI    1,IR0           ; K <= K+1
        SUBI    1,AR6           ; AR6 <= N-K-1

;       END OF OUTER LOOP (K INDEX)

        RETS                    ; RETURN

;       SINGULAR SYSTEM EXIT

SING:   LDF     ZERO,R3         ; SET "SINGULAR" FLAG

        RETS                    ; RETURN
```

```
*****************************************************
*   PROGRAM: *SOLUTNX                               *
*                                                   *
*   WRITTEN BY: GARY A. SITTON                      *
*               GAS LIGHT SOFTWARE                  *
*               HOUSTON, TEXAS                      *
*               MAY 1989.                           *
*                                                   *
*   (EXTENDED PRECISION VERSION)                    *
*                                                   *
*   SOLVES A SYSTEM OF LINEAR EQUATIONS A*X = Y IN THE  *
*   TABLEAU FORMAT B = A:-Y, AN M X N MATRIX.  THIS     *
*   MEANS THAT A IS AN M X M SQUARE MATRIX OF COEFFI-   *
*   CIENTS, AND -Y IS AN M X N-M RECTANGULAR MATRIX     *
*   OF N-M VECTORS EACH HAVING M ELEMENTS.  EACH DEPEN- *
*   DENT VARIABLE COLUMN VECTOR IS NEGATED AND APPENDED *
*   TO THE COEFFICIENT MATRIX A.  THE SET OF N-M INDE-  *
*   PENDENT SOLUTION VECTORS X WILL APPEAR IN PLACE OF  *
*   THE ORIGINAL APPENDED COLUMNS WHEN SOLUTNX FINISHES.*
*   ROW MAJOR MATRIX STORAGE FORMAT IS ASSUMED PLUS     *
*   THE PROGRAM ASSUMES N > M > 1 AND B[0, 0] != 0.0    *
*   SINCE THE METHOD USES DIAGONAL PIVOTING AND STARTS  *
*   WITH B[0, 0].  ANY PIVOT ELEMENT < 10**-10 IN ITS   *
*   ABSOLUTE VALUE WILL IMPLY AN "ILL CONDITIONED"      *
*   SYSTEM OF EQUATIONS, I. E. NOT HAVING SUFFICIENT    *
*   LINEAR INDEPENDENCE,  AND WILL RESULT IN AN INCOM-  *
*   PLETE SOLUTION.  AN INCOMPLETE SOLUTION WILL BE     *
*   INDICATED BY THE VALUE OF R3 == 0.0 ON EXIT, ELSE   *
*   R3 != 0.0 AND EQUALS THE LAST PIVOT ELEMENT VALUE.  *
*                                                   *
*   MSOLUTNX ENTRY PROTOCOL:                        *
*       VARIABLES FOR INPUT:                        *
*           $IAD1 -> B[0, 0], $NROW = M,            *
*           $NCOL = N, $PARMS = DATA PAGE.          *
*       INPUT RESTRICTIONS: N > M > 1.              *
*       REGISTERS ALTERED: RC, DP, AR0-7, IR0-1,    *
*                          AND R0-7.                *
*                                                   *
*   RSOLUTNX ENTRY PROTOCOL:                        *
*       REGISTERS FOR INPUT:                        *
*           AR0 -> B[0, 0], AR1 = M, AR2 = N.       *
*       INPUT RESTRICTIONS: AR2 > AR1 > 1.          *
*       REGISTERS ALTERED: RC, AR0-7, IR0-1, AND R0-7.  *
*                                                   *
*   REGISTERS USED AND RESTORED: SP.                *
*   REGISTERS FOR OUTPUT: R3.                       *
*   ROUTINES NEEDED: FPINVX AND FMULTX (SEE $MATHX).    *
*                                                   *
*   NOTE: THE RND INSTRUCTIONS MAY BE REMOVED WITH  *
*   SOME LOSS OF ACCURACY BUT INCREASE IN SPEED.    *
*****************************************************

;       EXTERNAL PROGRAM NAMES
```

```
        .GLOBL  MSOLUTNX    ; MEMORY BASED ENTRY
        .GLOBL  RSOLUTNX    ; REGISTER BASED ENTRY
        .GLOBL  FPINVX      ; RECIPROCAL ROUTINE
        .GLOBL  FMULTX      ; MULTIPLY ROUTINE

;       EXTERNAL PARAMETER NAMES

        .GLOBL  $PARMS      ; PARAMETER SPACE ADDRESS
        .GLOBL  $IAD1       ; POINTER TO MATRIX B, ADDRESS OF B[0, 0]
        .GLOBL  $NROW       ; NUMBER OF ROWS IN B, VALUE OF M
        .GLOBL  $NCOL       ; NUMBER OF COLUMNS IN B, VALUE OF N

;       INTERNAL CONSTANTS

        .DATA

EPSNX   .FLOAT  1.0E-10     ; SINGULARITY CRITERION
ZEROX   .SET    0.0         ; SINGULARITY FLAG

;       START SOLUTNX PROGRAM

        .TEXT

;       MEMORY BASED PARAMETER ENTRY

MSOLUTNX:

        LDP     @$PARMS     ; LOAD DATA PAGE POINTER
        LDI     @$IAD1,AR0  ; AR0 -> B[0, 0]
        LDI     @$NROW,AR1  ; AR1 <= M
        LDI     @$NCOL,AR2  ; AR2 <= N

;       REGISTER BASED PARAMETER ENTRY

RSOLUTNX:

;       SETUP LOOP REGISTERS

        LDP     @EPSNX      ; LOAD DATA PAGE POINTER
        LDI     0,IR0       ; IR0 <= K (INIT. 0)
        LDI     AR0,AR3     ; AR3 -> B[0, 0]
        SUBI    1,AR1       ; AR1 <= M-1
        LDI     AR2,AR6     ; AR6 <= N
        SUBI    2,AR6       ; AR6 <= N-2

;       MAIN LOOP (K INDEX)

KLOOPX: LDF     *+AR3(IR0),R3   ; R3 <= B[K, K], NEXT PIVOT
        ABSF    R3,R0           ; R0 <= |R3|
        CMPF    @EPSNX,R0       ; COMPARE |B[K, K]| TO EPS
        BLT     SINGX           ; IF |B[K, K]| < EPS THEN STOP

;       COMPUTE RECIPROCAL OF -PIVOT ELEMENT

        NEGF    R3,R0           ; R0 <= -B[K, K]
        CALL    FPINVX          ; R0 <= -1/B[K, K]
        LDF     R0,R1           ; R1 <= -1/B[K, K]

;       DIVIDE RIGHT PART OF PIVOT ROW BY -PIVOT ELEMENT

        ADDI    AR3,IR0,AR7     ; AR7 -> B[K, K]
        LDI     AR6,RC          ; RC <= N-K-2

        RPTB    DLOOPX          ; REPEAT DIVIDE LOOP N-K-1 TIMES
        LDF     *++AR7,R0       ; R0 <= B[K, J]
        CALL    FMULTX          ; R0 <= B[K, J]*(-1/B[K, K])
        RND     R0              ; ROUND *
DLOOPX: STF     R0,*AR7         ; B[K, J] <= R0

;       START INNER LOOP (I INDEX)

        LDI     0,IR1           ; IR1 <= I (INIT. 0)
        LDI     AR0,AR4         ; AR4 -> B[0, 0]

        CMPI    IR0,IR1         ; COMPARE I TO K
ILOOPX: BEQ     SKIPX           ; IF I == K THEN SKIP PIVOT ROW

;       COMPLETE PIVOTING OPERATION

        ADDI    AR4,IR0,AR5     ; AR5 -> B[I, K]
        LDF     *AR5,R0         ; R0 <= B[I, K]
        LDI     AR6,RC          ; RC <= N-K-2
        CMPI    1,RC            ; COMPARE RC TO 1
        BLTD    JUMPX           ; IF RC < 1 THEN NO RPTB (DELAYED)

        SUBI    1,RC            ; RC <= N-K-3
        ADDI    AR3,IR0,AR7     ; AR7 -> B[K, J]
        MPYF    R0,*++AR7,R1    ; R1 <= B[K, K+1]*B[I, K]

;       START INNER-INNER LOOP (J INDEX)

        RPTB    JLOOPX          ; REPEAT PIVOT LOOP N-K-2 TIMES
        MPYF    R0,*++AR7,R1    ; R1 <= B[K, J]*B[I, K]
||      ADDF    R1,*++AR5,R2    ; R2 <= B[I, J] + R1
        RND     R2              ; ROUND +
JLOOPX: STF     R2,*AR5         ; B[I, J] <= R2

;       END OF INNER-INNER LOOP (J INDEX)

JUMPX:  ADDF    R1,*++AR5,R2    ; R2 <= B[I, N-1] + R1
        RND     R2              ; ROUND +
        STF     R2,*AR5         ; B[I, N-1] <= R2

SKIPX:  CMPI    AR1,IR1         ; COMPARE I TO M-1
        BLTD    ILOOPX          ; IF I < M-1 THEN LOOP (DELAYED)

        ADDI    AR2,AR4         ; AR4 -> B[I+1, 0]
        ADDI    1,IR1           ; I <= I+1
```

```
        CMPI    IR0,IR1         ; COMPARE I TO K

;       END OF INNER LOOP (I INDEX)

        CMPI    AR1,IR0         ; COMPARE K TO M-1
        BLTD    KLOOPX          ; IF K < M-1 THEN LOOP

        ADDI    AR2,AR3         ; AR3 -> B[K+1, 0]
        ADDI    1,IR0           ; K <= K+1
        SUBI    1,AR6           ; AR6 <= N-K-1

;       END OF OUTER LOOP (K INDEX)

        RETS                    ; RETURN

;       SINGULAR SYSTEM EXIT

SINGX:  LDF     ZEROX,R3        ; SET "SINGULAR" FLAG

        RETS                    ; RETURN

        .END
```

Part III. Digital Signal Processing Interface Techniques

TMS320C30
Hardware Applications

Jon Bradley

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

Introduction

The TMS320C30 is a high-speed, floating-point, digital signal processor. The TMS320C30s advanced interface design allows it to be used to implement a wide variety of system configurations. Its two external buses and DMA capability provide a parallel 32-bit interface to byte- or word-wide devices, while the interrupt interface, dual serial ports, and general purpose digital I/O provide communication with a multitude of peripherals.

This application report describes how to use the TMS320C30s interfaces to connect to various external devices. Specific discussions include implementation of parallel interface to devices with and without wait states, use of general purpose I/O, and system control functions. All interfaces shown in this report have been built and tested to verify proper operation.

Major topics discussed in this report are as follows:

- System Configuration Options Overview
- Primary Bus Interface
  - Zero Wait Interface to RAMs
  - Ready Generation
  - Bank Switching Techniques
- Expansion Bus Interface
  - A/D Converter Interface
  - D/A Converter Interface
- System Control Functions
  - Clock Oscillator Circuitry
  - Reset Signal Generator
- Serial Port Interface
- XDS1000 Target Design Considerations

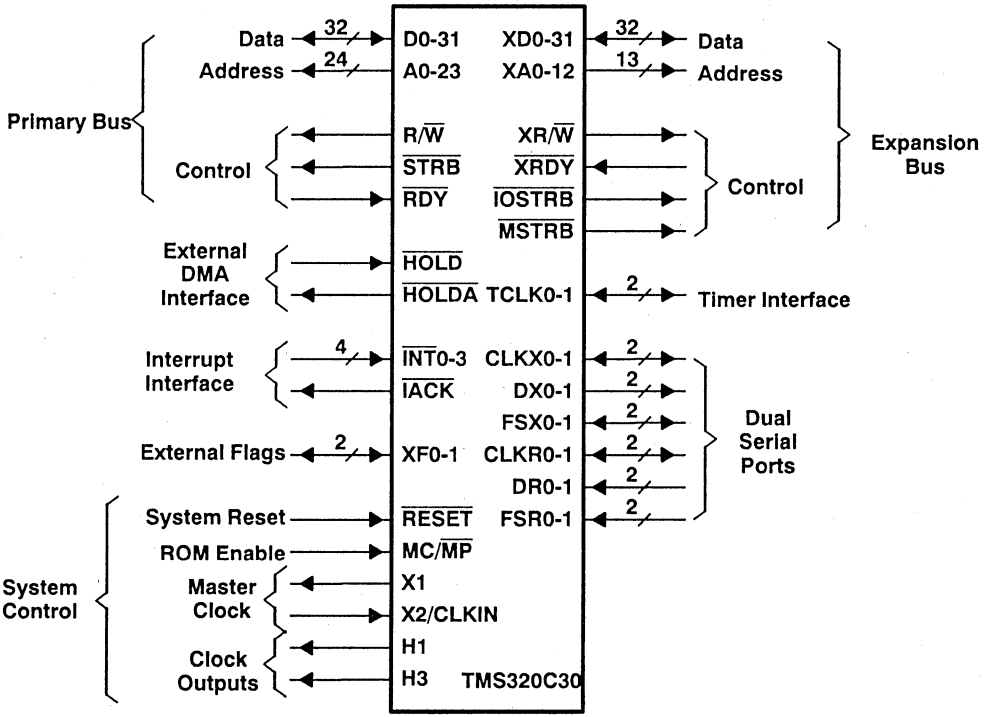
System Configuration Options Overview

The various TMS320C30 interfaces allow connections to a wide variety of different device types. Each of these interfaces is tailored to a particular family of devices.

Categories of Interfaces on the TMS320C30

The interface types on the TMS320C30 fall into several different categories depending on the devices to which they were intended to be connected. Each interface comprises one or more signal lines that transfer information and control its operation. Shown in Figure 1 are the signal line groupings for each of these various interfaces.

Figure 1. External Interfaces on the TMS320C30



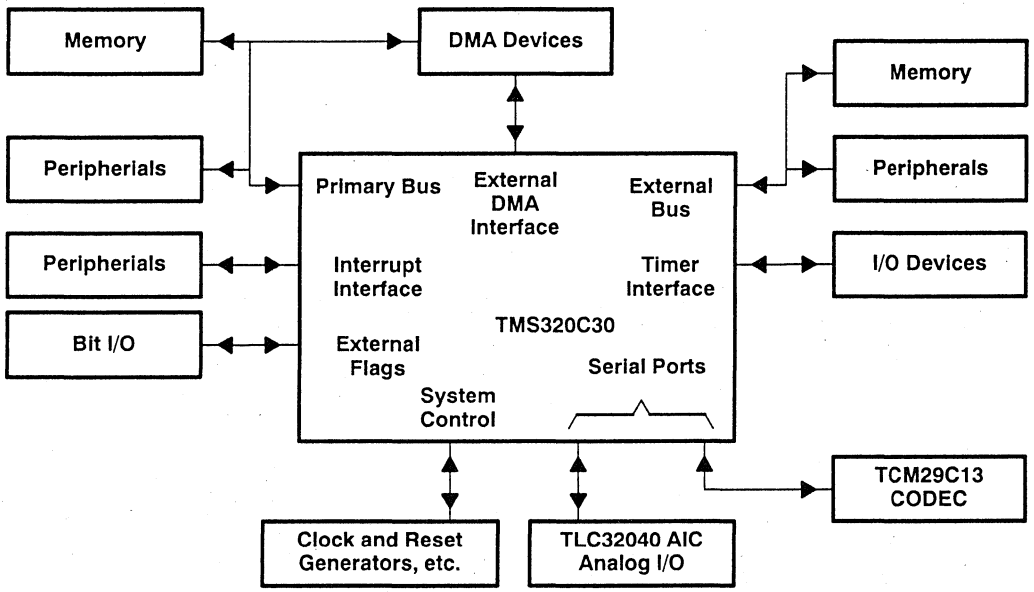
All of the interfaces are independent of one another and different operations may be performed simultaneously on each interface.

The Primary and Expansion buses implement the memory mapped interface to the device. The external DMA interface allows external devices to cause the processor to relinquish the Primary bus and allow direct memory access.

Typical System Block Diagram

The devices that can be interfaced to the TMS320C30 include memory, DMA devices, and numerous parallel and serial peripherals and I/O devices. Figure 2 illustrates a typical configuration of a TMS320C30 system showing different types of external devices and the interfaces to which they are connected.

Figure 2. Possible System Configurations



This block diagram constitutes essentially a fully expanded system. In an actual design, any subset of the illustrated configuration may be used.

Primary Bus Interface

The primary bus is used by the TMS320C30 to access the majority of its memory mapped locations. Therefore, typically when a large amount of external memory is required in a system, it is interfaced to the primary bus. The expansion bus (discussed in the next section) actually comprises two mutually exclusive interfaces, controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals respectively. Cycles on the expansion bus controlled by the $\overline{\text{MSTRB}}$ signal are essentially equivalent to cycles on the primary bus, with the exception that bank switching is not implemented on the expansion bus. Accordingly, the discussion of primary bus cycles in this section applies equally to $\overline{\text{MSTRB}}$ cycles on the expansion bus.

Although both the primary bus and the expansion bus may be used to interface to a wide variety of devices, the devices most commonly interfaced to these buses are memories. Therefore, detailed examples of memory interface will be presented in this section.

Zero Wait State Interface To Static RAMs

For full speed, zero-wait state interface to any device, the TMS320C30 requires a read access time of 30 ns from address stable to data valid. Because, for most memories, access time from chip select is the same as access time from address, it is theoretically possible to use 30 ns memories at full speed with the TMS320C30. This, however, dictates that there be no delays present between the processor and the memories. This is usually not the case in practice, due to interconnection de-

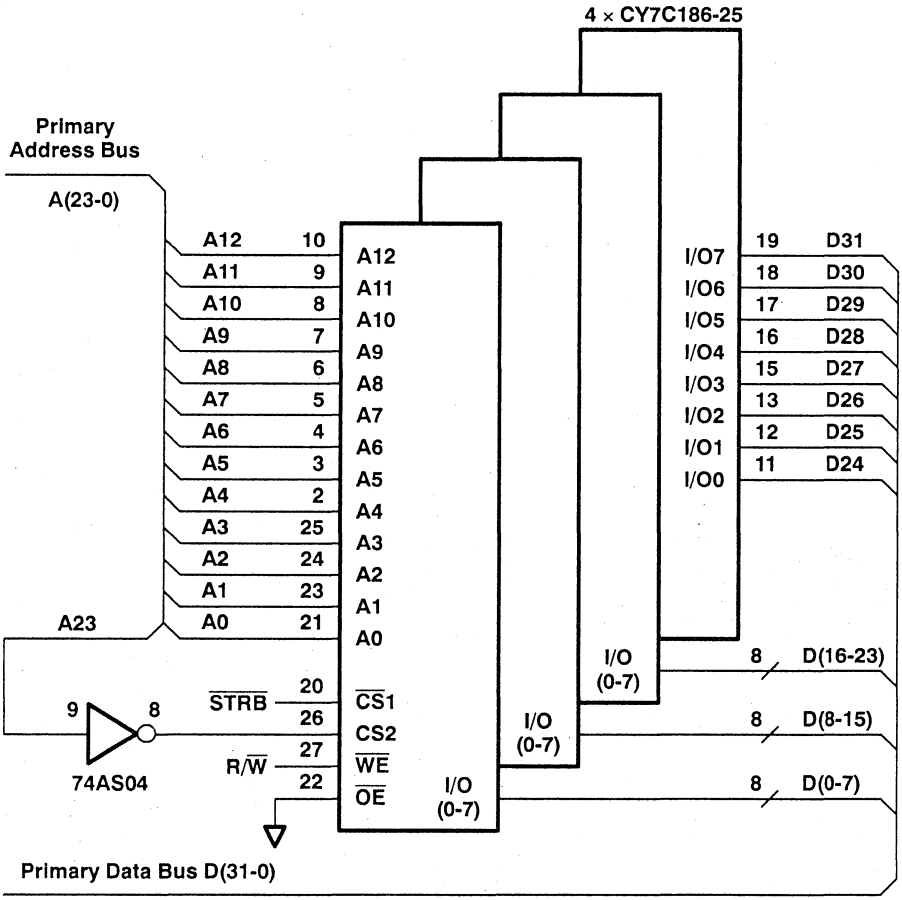
lays and the fact that typically some gating is required for chip select generation. Therefore, slightly faster memories are generally required in most systems. If one level of reasonably high-speed (below 10 ns in propagation delay) gating is used to generate chip select for the memories, 20 ns devices may be used.

Among currently available RAMs, there are two distinct categories of devices with different interface characteristics. These two categories are RAMs without output enable control lines ($\overline{OE}$), which include the 1-bit wide organized RAMs and most of the 4-bit wide RAMs, and those with $\overline{OE}$ controls, which include the byte wide and a few of the 4-bit wide RAMs. Many of the fastest RAMs do not provide $\overline{OE}$ control, and use chip select ($\overline{CS}$) controlled write cycles to insure that data outputs do not turn on for write operations. In $\overline{CS}$ controlled write cycles, the write control line ($\overline{WE}$) goes low prior to $\overline{CS}$ going low, and internal logic holds the outputs disabled until the cycle is completed. Using $\overline{CS}$ controlled write cycles is an efficient way to interface fast RAMs without $\overline{OE}$ controls to the TMS320C30 at full speed.

In the case of RAMs with $\overline{OE}$ controls, the use of this signal can provide added flexibility in many systems. Additionally, many of these devices can be interfaced using $\overline{CS}$ controlled write cycles with $\overline{OE}$ tied low, in the same manner as with RAMs without $\overline{OE}$ controls. There are, however, two requirements for interfacing to $\overline{OE}$ RAMs in this fashion. First, the RAMs $\overline{OE}$ input must be gated with chip select and $\overline{WE}$ internally so that the device's outputs do not turn on unless a read is being performed. Second, the RAM must allow its address inputs to change while $\overline{WE}$ is low, which some RAMs specifically prohibit.

The circuit shown in Figure 3 shows an interface to Cypress Semiconductor's CY7C186 25 ns 8K × 8-bit CMOS static RAMs with the $\overline{OE}$ control input tied low and using a $\overline{CS}$ controlled write cycle.

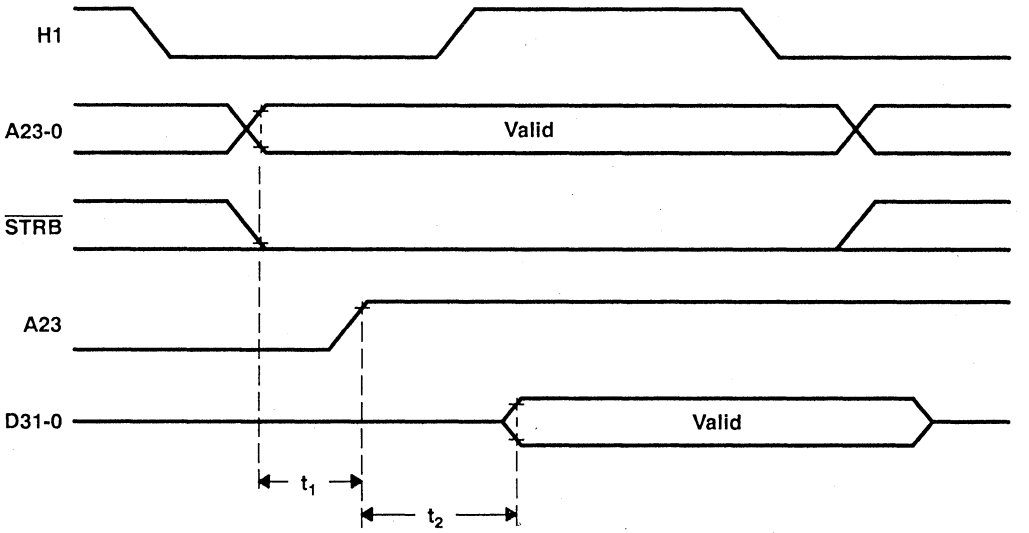
Figure 3. TMS320C30 Interface to Cypress Semiconductor CY7C186 CMOS SRAM



In this circuit, the two chip selects on the RAM are driven by $\overline{STRB}$ and $\overline{A23}$, which are ANDed together internally. The use of $\overline{A23}$ locates the RAM at addresses 00000h through 03FFFh in external memory and $\overline{STRB}$ establishes the $\overline{CS}$ controlled write cycle. The $\overline{WE}$ control input is then driven by the TMS320C30 R/$\overline{W}$ signal, and the $\overline{OE}$ input is not used, and is therefore connected to ground.

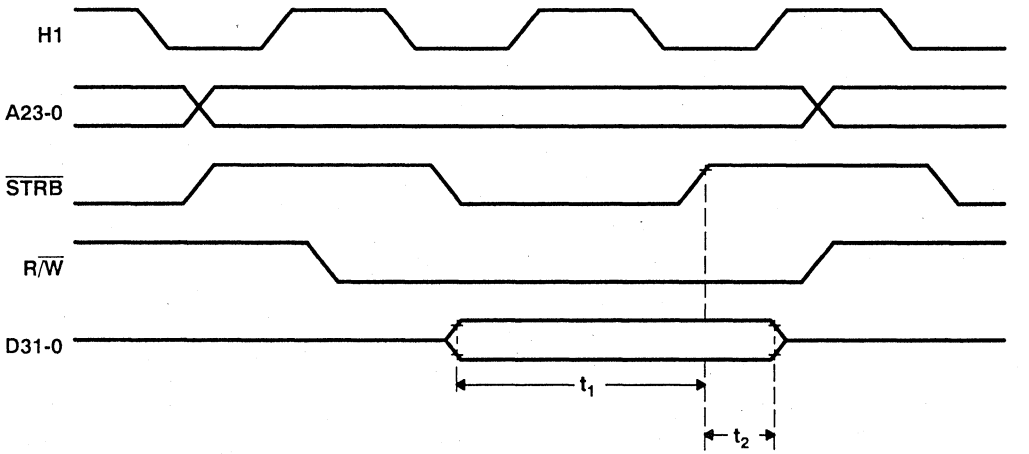
The timing of read operations, shown in Figure 4, is very straightforward since the two chip select inputs are driven directly. The read access time of the circuit is therefore the inverter propagation delay added to the RAMs chip select access time or $t_1 + t_2 = 5 + 25 = 30$ ns. This access time therefore meets the TMS320C30s specified 30 ns requirement.

Figure 4. Read Operations Timing



During write operations, as shown in Figure 5, the RAMs outputs do not turn on at all, due to the use of the chip select controlled write cycles. The chip select controlled write cycles are generated by the fact that R/$\overline{W}$ goes active (low) before the $\overline{STRB}$ term of the chip select input. Because the RAMs output drivers are disabled whenever the $\overline{WE}$ input is low (regardless of the state of the $\overline{OE}$ input) bus conflicts with the TMS320C30 are automatically avoided with this interface. The circuit's data setup and hold times (t_1 and t_2 in the timing diagram) of approximately 50 and 20 ns, respectively, also easily meet the RAMs timing requirements of 10 and 0 ns.

Figure 5. Write Operations Timing



If more complex chip select decode is required than can be accomplished in time to meet zero-wait state timing, wait states or bank switching techniques (discussed in a later section) should be used.

It should be noted that the CY7C186's $\overline{OE}$ control is gated internally with $\overline{CS}$, therefore the RAMs outputs are not enabled unless the device is selected. This is critical if there are any other devices connected to the same bus; if there are no other devices connected to the bus, then $\overline{OE}$ need not be gated internally with chip select.

RAMs without $\overline{OE}$ controls can also be easily interfaced to the TMS320C30 using a similar approach to that used with RAMs with $\overline{OE}$ controls. If there is only one bank of memory implemented, and no other devices are present on the bus, the memories' $\overline{CS}$ input may often be connected to $\overline{STRB}$ directly. If several devices must be selected, however, a gate is generally required to AND the device select and $\overline{STRB}$ to drive the $\overline{CS}$ input to generate the chip select controlled write cycles. In either case, the $\overline{WE}$ input is driven by the TMS320C30 R/$\overline{W}$ signal. Provided sufficiently fast gating is used, 25 ns RAMs may still be used.

As with the case of RAMs with $\overline{OE}$ control lines, this approach works well if only a few banks of memory are implemented where the chip select decode can be accomplished with only one level of gating. If many banks are required to implement very large memory spaces, bank switching can be used to provide for multiple bank select generation while still maintaining full speed accesses within each bank. Bank switching is discussed in detail in a later section.

Ready Generation

The use of wait states can greatly increase system flexibility and reduce hardware requirements over systems without wait state capability. The TMS320C30 has the capability of generating wait states on either the primary bus or the expansion bus and both buses have independent sets of ready control logic. Ready generation is discussed in this subsection from the perspective of the primary bus interface, however, wait state operation on the expansion bus is similar to that of the primary bus, therefore these discussions pertain equally well to expansion bus operation. Thus, ready generation will not be included in the specific discussions of the expansion bus interface.

Wait states are generated on the basis of the internal wait state generator, the external ready input ($\overline{RDY}$), or the logical AND or OR of the two. When enabled, internally generated wait states effect all external cycles, regardless of the address accessed. If different numbers of wait states are required for various external devices, the external $\overline{RDY}$ input may be used to tailor wait state generation to specific system requirements.

If the logical OR (or electrical AND since the signals are true low) of the external and wait count ready signals is selected, the earlier of either of the two signals will generate a ready condition and allow the cycle to be completed. It is not required that both signals be present.

The OR of the two ready signals can be used to implement wait states for devices that require a greater number of wait states than are implemented with external logic (up to seven). This feature is useful, for example, if a system contains some fast and some slow devices. In this case, fast devices can generate a ready signal externally with a minimum of logic, and slow devices can use the internal wait counter for larger numbers of wait states. Thus, when fast devices are accessed, the external hardware responds promptly with a ready signal that terminates the cycle. When slow devices are accessed, the external hardware does not respond, and the cycle is appropriately terminated after the internal wait count.

The OR of the two ready signals may also be used if conditions occur that require termination of bus cycles prior to the number of wait states implemented with external logic. In this case, a

shorter wait count is specified internally than the number of wait states implemented with the external ready logic, and the bus cycle is terminated after the wait count. This feature may also be used as a safeguard against inadvertent accesses to nonexistent memory that would never respond with ready and therefore lock up the TMS320C30.

If the OR of the two ready signals is used, however, and the internal wait state count is less than the number of wait states implemented externally, the external ready generation logic must have the ability to reset its sequencing to allow a new cycle to begin immediately following the end of the internal wait count. This requires that, under these conditions, consecutive cycles must be from independently decoded areas of memory and that the external ready generation logic be capable of restarting its sequence as soon as a new cycle begins. Otherwise, the external ready generation logic may lose synchronization with bus cycles and therefore generate improperly timed wait states.

If the logical AND (electrical OR) of the wait count and external ready signals is selected, the later of the two signals will control the internal ready signal, and both signals must occur. Accordingly, external ready control must be implemented for each wait state device in addition to the wait count ready signal being enabled.

This feature is useful if there are devices in a system that are equipped to provide a ready signal but cannot respond quickly enough to meet the TMS320C30s timing requirements. In particular, if these devices normally indicate a ready condition and, when accessed, respond with a wait until they become ready, the logical AND of the two ready signals can be used to save hardware in the system. In this case, the internal wait counter can be used to provide wait states initially, and become ready after the external device has had time to send a not ready indication. The internal wait counter then remains ready until the external device also becomes ready, which terminates the cycle.

Additionally, the AND of the two ready signals may be used for extending the number of wait states for devices that already have external ready logic implemented but require additional wait states under certain unique circumstances.

In the implementation of external ready generation hardware, the particular technique employed depends heavily on the specific characteristics of the system. The optimum approach to ready generation varies depending on the relative number of wait state and non-wait state devices in the system and the maximum number of wait states required for any one device. The approaches discussed here are intended to be general enough for most applications, and are easily modifiable to comprehend many different system configurations.

In general, ready generation involves the following three functions:

1) Segmentation of the address space in some fashion to distinguish fast and slow devices.
2) Generating properly timed ready indications.
3) Logically ORing all of the separate ready timing signals together to connect to the physical ready input.

Segmentation of the address space is required so that a unique indication of each of the particular areas within the address space that require wait states can be obtained. This segmentation is commonly implemented in a system in the form of chip select generation. Chip select signals may be used to initiate wait states in many cases, however, occasionally chip select decoding considerations may provide signals that will not allow ready input timing requirements to be met. In this case, coarse address space segmentation may be made on the basis of a small number of address lines, where simpler gating allows signals to be generated more quickly. In either case, the signal indicating that a particular area of memory is being addressed is normally used to initiate a ready or wait state indication.

Once the region of address space being accessed has been established, a timing circuit of some sort is normally used to provide a ready indication to the processor at the appropriate point in the cycle to satisfy each device's unique requirements.

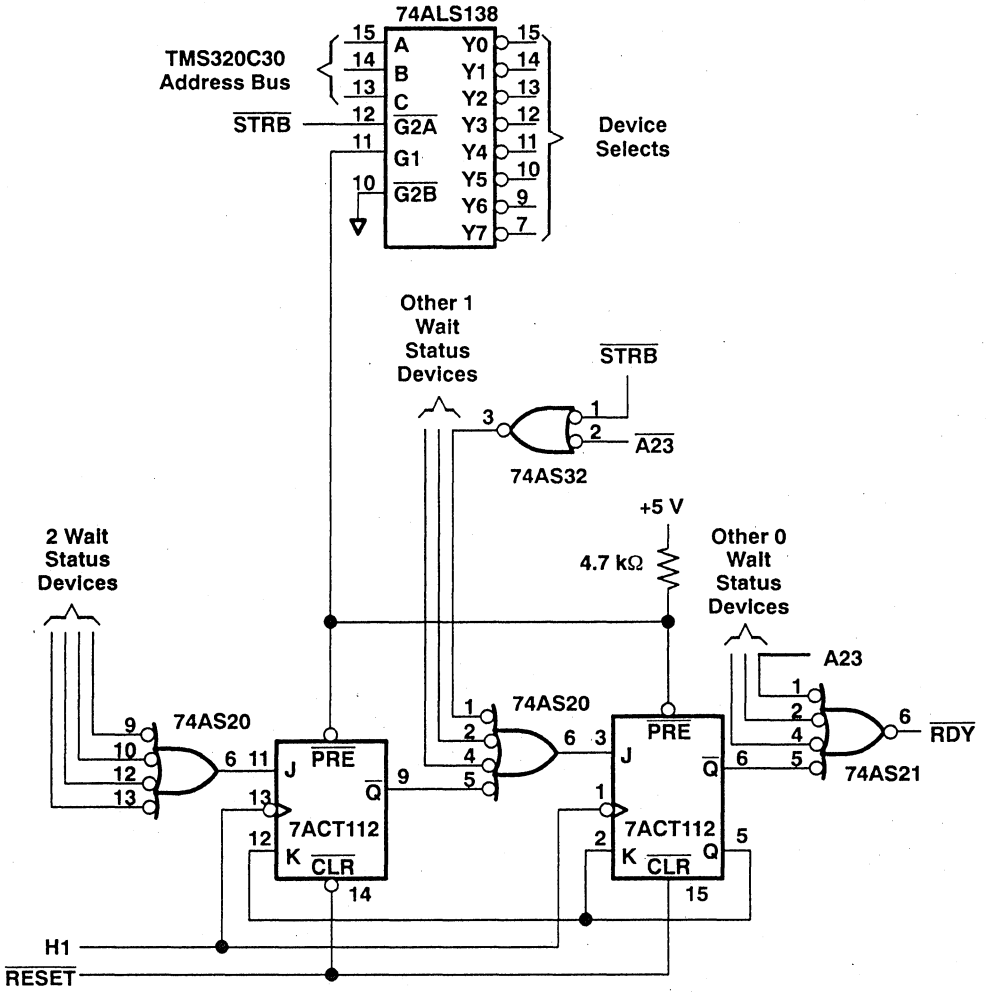
Finally, since indications of ready status from multiple devices are typically present, the signals are logically ORed using a single gate to drive the $\overline{\text{RDY}}$ input.

One of two basic approaches may be taken in the implementation of ready control logic depending upon the state in which the ready input is to be between accesses. If $\overline{\text{RDY}}$ is low between accesses, the processor is always ready unless a wait state is required; if $\overline{\text{RDY}}$ is high between accesses, the processor will always enter a wait state unless a ready indication is generated.

If $\overline{\text{RDY}}$ is low between accesses, control of full speed devices is straightforward; no action is necessary since ready is always active unless otherwise required. Devices requiring wait states, however, must drive ready high fast enough to meet the input timing requirements. Then, after an appropriate delay, a ready indication must be generated. This can be quite difficult in many circumstances since wait state devices are inherently slow and often require complex select decoding.

If $\overline{\text{RDY}}$ is high between accesses, zero wait state devices, which tend to be inherently fast, can usually respond immediately with a ready indication. Wait state devices may simply delay their select signals appropriately to generate a ready. Typically, this approach results in the most efficient implementation of ready control logic. Figure 6 shows a circuit of this type which can be used to generate 0, 1, or 2 wait states for multiple devices in a system.

Figure 6. Circuit For Generation of 0, 1, or 2 Wait States for Multiple Devices



In this circuit, full speed devices drive ready directly through the '74AS21, and the two flip-flops delay wait state devices' select signals one or two H1 cycles to provide 1 or 2 wait states.

Considering the TMS320C30's ready delay time of 8 ns following address, zero wait state devices must use ungated address lines directly to drive the input of the '74AS21, since this gate contributes a maximum propagation delay of 6 ns to the $\overline{\text{RDY}}$ signal. Thus, zero wait state devices should be grouped together within a coarse segmentation of address space if other devices in the system require wait states.

With this circuit, devices requiring wait states may take up to 36 ns from a valid address on the TMS320C30 to provide inputs to the '74AS20s inputs. Typically, this allows sufficient time for any decoding required in generating select signals for slower devices in the system. For exam-

ple, the 74ALS138 driven by address and $\overline{\text{STRB}}$, can generate select decodes in 22 ns, which easily meets the TMS320C30s timing requirements.

With this circuit, unused inputs to either the 74AS20s or the 74AS21 should be tied to a logic high level to prevent noise from generating spurious wait states.

If more than 2 wait states are required by devices within a system, other approaches may be employed for ready generation. If between three and seven wait states are required, additional flip-flops may be included, in the same manner as shown in Figure 6, or internally generated wait states may be used in conjunction with external hardware. If greater than seven wait states are required, an external circuit using a counter may be used to supplement the internal wait–state generator's capabilities.

Bank Switching Techniques

The TMS320C30's programmable bank switching feature can greatly ease system design when large amounts of memory are required. This feature is used to provide a period of time during which all device selects are disabled that would not normally be present otherwise. During this interval, slow devices are allowed time to turn off before other devices have the opportunity to drive the data bus, thus avoiding bus contention.

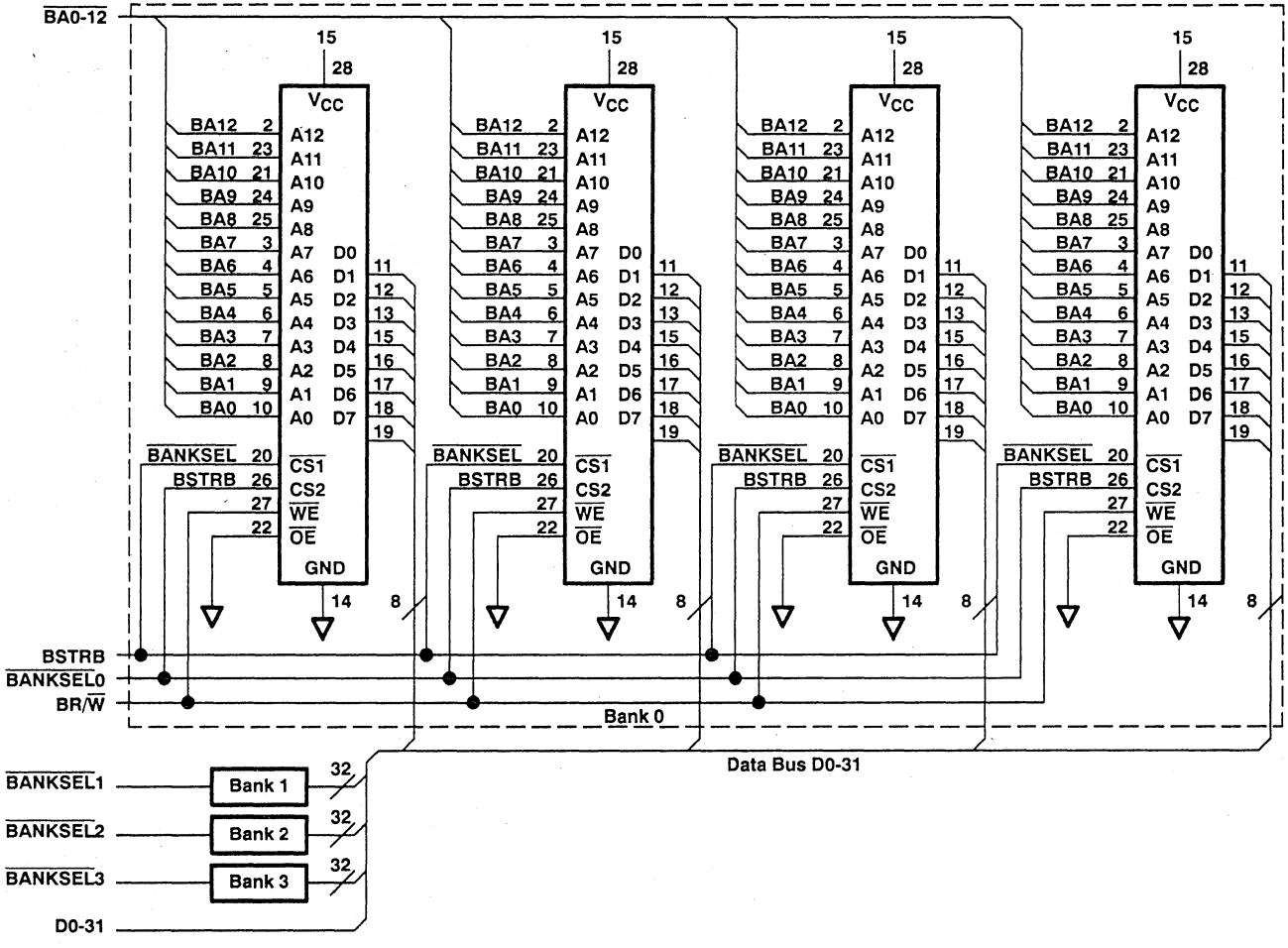
When bank switching is enabled, any time a portion of the high order address lines change, as defined by the contents of the BNKCMPR register, $\overline{\text{STRB}}$ goes high for one full H1 cycle. Provided $\overline{\text{STRB}}$ is included in chip select decodes, this causes all devices to be disabled during this period. The next bank of devices is not enabled until $\overline{\text{STRB}}$ goes low again.

Bank switching is not required during writes since these cycles always exhibit an inherent one-half H1 cycle setup of address information before $\overline{\text{STRB}}$ goes low. Thus, when using bank switching for read/write devices, a minimum of half of one H1 cycle of address setup is provided for all accesses. Therefore, large amounts of memory can be implemented without wait states or extra hardware required for isolation between banks. Also, note that access time for cycles during bank switching is the same as that of cycles without bank switching, and accordingly, full speed accesses may still be accomplished within each bank.

When using bank switching to implement large multiple-bank memory systems, an important consideration is address line fanout. Besides parametric specifications for which account must be made, AC characteristics are also crucial in memory system design. With large memory arrays which commonly require large numbers of address line inputs to be driven in parallel, capacitive loading of address outputs is often quite large. Because all TMS320C30 timing specifications are guaranteed up to a capacitive load of 80 pF, driving greater loads will invalidate guaranteed AC characteristics. Therefore it is often necessary to provide buffering for address lines when driving large memory arrays. AC timings for buffer performance may then be derated according to manufacturer specifications to accomodate a wide variety of memory array sizes.

The circuit shown in Figure 7 illustrates the use of bank switching with Cypress Semiconductor's 'CY7C185 25 ns 8K × 8 CMOS static RAM. This circuit implements 32K 32-bit words of memory with one wait-state accesses within each bank.

Figure 7. Bank Switching For Cypress Semiconductors CY7C185

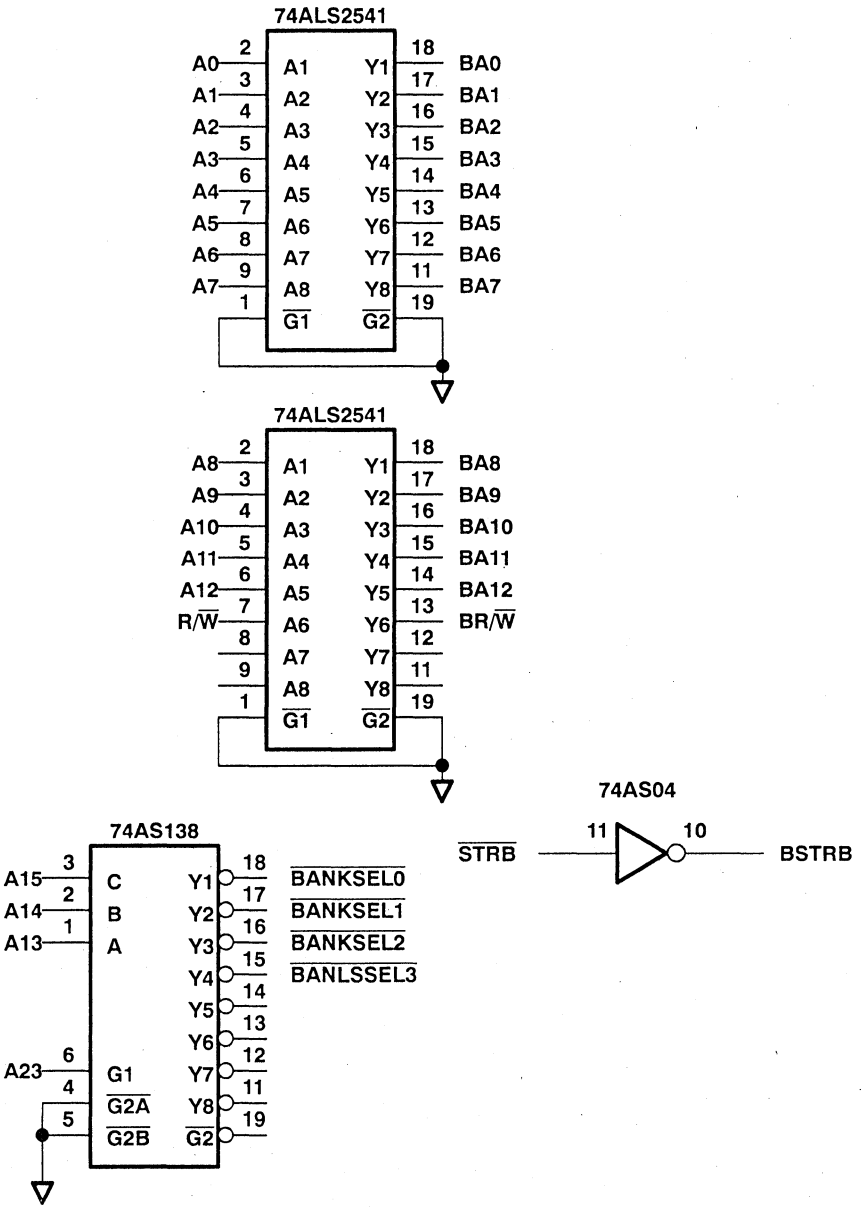


A wait state is required with this implementation of bank memory because of the added propagation delay presented by the address bus buffers used in the circuit. The wait state is not a function of the fact that the memory is organized as multiple banks or the use of bank switching. When bank switching is used, memory access speeds are the same as without bank switching once bank boundaries are crossed. Therefore, no speed penalty is paid when using bank switching except for the occasional extra cycle inserted when bank boundaries are crossed. It should be noted, however, that if the extra cycle inserted when crossing bank boundaries does impact software performance significantly, code can often be restructured to minimize bank boundary crossings, thereby reducing the effect of these boundary crossings on software performance.

The wait state for this bank memory is generated using the wait state generator circuit presented in the previous section. Because A23 is the signal which enables the entire bank memory system, the inverted version of this signal is ANDed with $\overline{\text{STRB}}$ to derive a one wait state device select. This signal is then connected in the circuit along with the other one wait state device selects. Thus, any time a bank memory access is made, one wait state is generated.

Each of the four banks in this circuit is selected using a decode of A15-A13 generated by the 74AS138 (see Figure 8). With the BNKCMPR register set to 0Bh, the banks will be selected on even 8K-word boundaries starting at location 080A000h in external memory space.

Figure 8. Bank Memory Control Logic



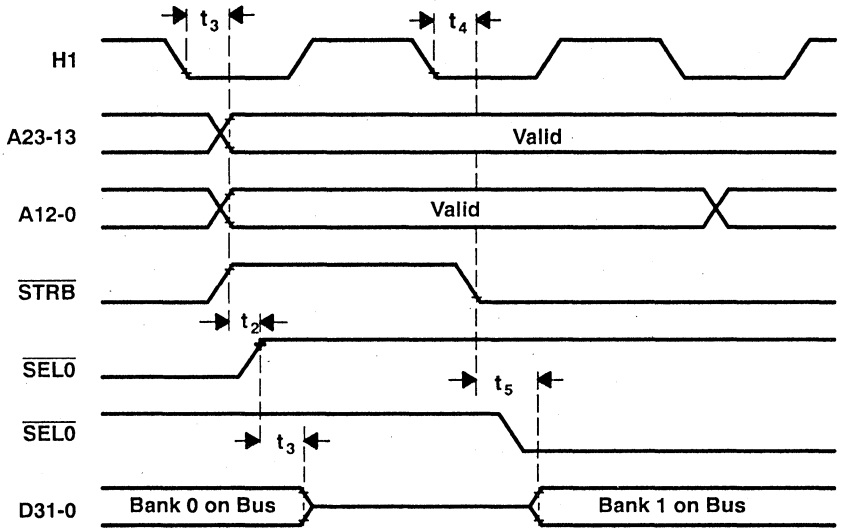
The 74ALS2541 buffers used on the address lines are necessary in this design since the total capacitive load presented to each address line is a maximum of 20 × 5 pF or 100 pF (bank memory plus zero wait–state static RAM), which exceeds the TMS320C30 rated capacitive loading of 80 pF. Using the manufacturers derating curves for these devices at a load of 80 pF (the load presented by the bank memory) predicts propagation delays at the output of the buffers of a maximum of 16 ns. The access time of a read cycle within a bank of the memory is therefore the sum of the memory access time and the maximum buffer propagation delay or 25 + 16 = 41 ns, which, since it falls between 30 and 90 ns, requires one wait state on the TMS320C30.

The 74ALS2541 buffers offer one additional system performance enhancement in that they include 25-ohm resistors in series with each individual buffer output. These resistors greatly improve the transient response characteristics of the buffers especially when driving CMOS loads such as the memories used here. The effect of these resistors is to reduce overshoot and ringing which is common when driving predominantly capacitive loads such as CMOS. The result of this is reduced noise and increased immunity to latchup in the circuit, which in turn results in a more reliable memory system. Having these resistors included in the buffers eliminates the need to put discrete resistors in the system which is often required in high speed memory systems.

This circuit could not have been implemented without bank switching, since data output's turn-on and turn-off delays would have caused bus conflicts. Here, the propagation delay of the 74AS138 is only involved during bank switches, where there is sufficient time between cycles to allow new chip selects to be decoded.

The timing of this circuit for read operations using bank switching is shown in Figure 9. With the BNKCMPR register set to 0Bh, when a bank switch occurs, the bank address on address lines A23-A13, is updated during the extra H1 cycle while $\overline{\text{STRB}}$ is high. Then, after chip select decodes have stabilized, and the previously selected bank has disabled its outputs, $\overline{\text{STRB}}$ goes low for the next read cycle. Further accesses occur at normal bus timings with one wait state as long as another bank switch is not necessary. Write cycles do not require bank switching due to the inherent address setup provided in their timings.

Figure 9. Timing For Read Operations Using Bank Switching



The timing for this interface is summarized in the Table 1.

Table 1. Bank Switching Interface Timing

Time Interval	Event	Time Period
t_1	H1 falling to address/STRB valid	14 ns
t_2	Add to select delay	10 ns
t_3	Memory disable from STRB	10 ns
t_4	H1 falling to STRB	10 ns
t_6	Memory output enable delay	3 ns

Expansion Bus Interface

The TMS320C30s expansion bus interface provides a second complete parallel bus which can be used to implement data transfers concurrently with and independent of operations on the primary bus. The expansion bus comprises two mutually exclusive interfaces controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals, respectively. This section discusses interface to the expansion bus using $\overline{\text{IOSTRB}}$ cycles; $\overline{\text{MSTRB}}$ cycles are essentially equivalent in timing to primary bus cycles, and are discussed in the previous section.

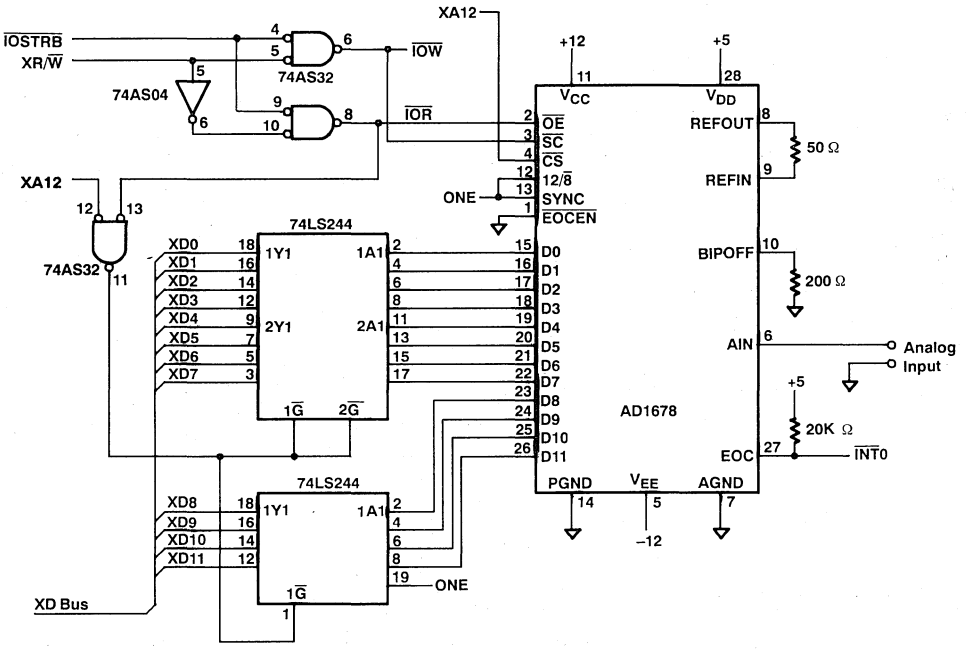
Unlike the primary bus, both read and write cycles on the I/O portion of the expansion bus are two H1 cycles in duration and exhibit the same timing. The XR/$\overline{\text{W}}$ signal is high for reads and low for writes. Since I/O accesses take two cycles, many peripherals that require wait states if interfaced either to the primary bus or using $\overline{\text{MSTRB}}$ may be used in a system without the need for wait states. Specifically, in cases where there is only one device on the expansion bus, devices with access times greater than the 30 ns required by the primary bus, but not more than 59 ns can be interfaced to the I/O bus without wait states.

A/D Converter Interface

A/D and D/A converters are components that are commonly required in DSP systems and interface efficiently to the I/O expansion bus. These devices are available in many speed ranges and with a variety of features, and while some may be used at full speed on the I/O bus, others may require one or more wait states.

Figure 10 shows an interface to an Analog Devices AD1678 analog to digital converter. The AD1678 is a 12-bit, 5 μs converter allowing sample rates up to 200 kHz and with an input voltage range of 10 volts bipolar or unipolar. The converter is connected according to manufacturers specifications to provide 0 to +10 volt operation. This interface illustrates a common approach to connecting devices such as this to the TMS320C30. Note that the interface requires only a minimum amount of control logic.

Figure 10. Interface to AD1678 A/D Converter



The AD1678 is a very flexible converter and is configurable in a number of different operating modes. These operating modes include byte or word data format, continuous or non-continuous conversions, enabled or disabled chip select function, and programmable end of conversion indication. This interface utilizes 12-bit word data format, rather than byte format to be compatible with the TMS320C30. Non-continuous conversions are selected, so that variable sample rates may be used, since continuous conversions occur only at a rate of 200 kHz. With non-continuous conversions, the host processor determines the conversion rate by initiating conversions through write operations to the converter.

The chip select function is enabled, so the chip select input is required to be active when accessing the device. Enabling the chip select function is necessary to allow a mechanism for the AD1678 to be isolated from other peripheral devices connected to the expansion bus. To establish the desired operating modes, the SYNC and 12/$\overline{8}$ inputs to the converter are pulled high and $\overline{\text{EOCEN}}$ is grounded, as specified in the AD1678 data sheet.

In this application, the converter's chip select is driven by XA12, which maps this device at 804000h in I/O address space. Conversions are initiated by writing any data value to the device, and the conversion results are obtained by reading from the device after the conversion is completed. To generate the devices Start Conversion ($\overline{\text{SC}}$) and Output Enable ($\overline{\text{OE}}$) inputs, $\overline{\text{IOSTRB}}$ is ANDed with XR/$\overline{\text{W}}$. Therefore, the converter is selected whenever XA12 is low, and $\overline{\text{OE}}$ is driven when reads are performed, while SC is driven when writes are performed.

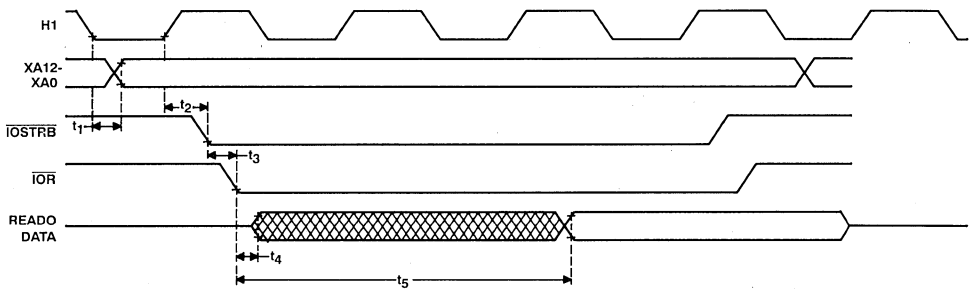
As with many A/D converters, at the end of a read cycle the AD1678 data output lines enter a high impedance state. This occurs after the Output Enable ($\overline{\text{OE}}$) or read control line goes inactive. Also common with these types of devices, is that the data output buffers often require a substantial amount of time to actually attain a full high-impedance state. When used with the TMS320C30, devices must have their outputs fully disabled no later than 65 ns following the rising edge of $\overline{\text{IOSTRB}}$, since the TMS320C30 will begin driving the data bus at this point if the next cycle is a write. If this timing is not met, bus conflicts between the TMS320C30 and the AD1678 may occur, potentially causing degraded system performance and even failure due to damaged data bus drivers. The actual disable time for the AD1678 can be as long as 80 ns, therefore buffers are required to isolate the converter outputs from the TMS320C30. The buffers used here are 74LS244s that are enabled when the AD1678 is read, and turned off 30.8 ns following $\overline{\text{IOSTRB}}$ going high.Therefore, the TMS320C30 requirement of 65 ns is met.

When data is read following a conversion, the AD1678 takes 100 ns after its $\overline{\text{OE}}$ control line is asserted to provide valid data at its outputs. Thus, including the propagation delay of the 74LS244 buffers, the total access time for reading the converter is 118 ns. This requires two wait states on the TMS320C30 expansion I/O bus.

The two wait states required in this case are implemented using software wait states, however, depending on the overall system configuration it may be necessary to implement a separate wait state generator for the expansion bus (refer to section on ready generation). This would be the case if there were multiple devices that required different numbers of wait states connected to the expansion bus.

Figure 11 shows the timing for read operations between the TMS320C30 and the AD1678. At the beginning of the cycle, the address and XR/$\overline{\text{W}}$ lines become valid t_1 = 10 ns following the falling edge of H_1. Then, after t_2 = 10 ns from the next rising edge of H_1, $\overline{\text{IOSTRB}}$ goes low, beginning the active portion of the read cycle. After t_3 = 5.8 ns, the control logic propagation delay, the $\overline{\text{IOR}}$ signal goes low, asserting the $\overline{\text{OE}}$ input to the AD1678. The '74LS244 buffers take t_4 = 30 ns to enable their outputs, and then, following the converters access delay and the buffer propagation delay (t_5 = 100 + 18 = 118 ns) data is provided to the TMS320C30. This provides approximately 46 ns of data setup before the rising edge of $\overline{\text{IOSTRB}}$. Therefore, this design easily satisfies the TMS320C30s requirement of 15 ns of data setup time for reads.

Figure 11. Read Operations Timing Between the TMS320C30 and AD1678



Unlike the primary bus, read and write cycles on the I/O expansion bus are timed the same with the exception that XR/$\overline{W}$ is high for reads and low for writes and that the data bus is driven by the TMS320C30 during writes. When writing to the AD1678, the '74LS244 buffers do not turn on and no data is transferred. The purpose of writing to the converter is only to generate a pulse on the converter's $\overline{SC}$ input, which initiates a conversion cycle. When a conversion cycle is completed, the AD1678's EOC output is used to generate an interrupt on the TMS320C30 to indicate that the converted data may be read.

It should be noted that for different applications, use of TLC1225 or TLC1550 A/D converters from Texas Instruments may be beneficial. The TLC1225 is a self-calibrating 12-bit-plus-sign bipolar or unipolar converter which features 10 μs conversion times. The TLC1550 is a 10-bit, 6 μs converter with a high speed DSP interface. Both converters are parallel-interface devices.

D/A Converter Interface

In many DSP systems, the requirement for generating an analog output signal is a natural consequence of sampling an analog waveform with an A/D converter and then processing the signal digitally internally. Interfacing D/A converters to the the TMS320C30 on the expansion I/O bus is also quite straightforward.

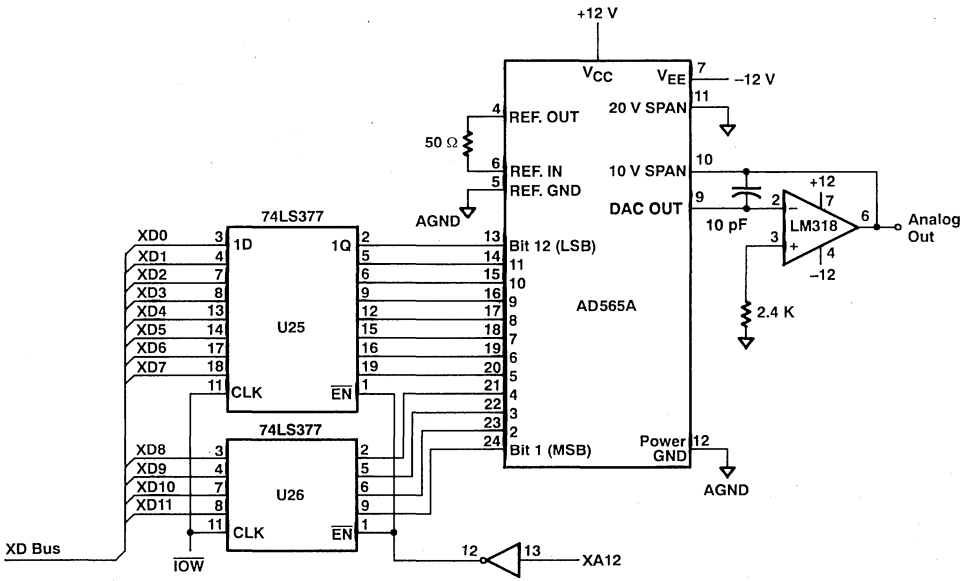
As with A/D converters, D/A converters are also available in a number of varieties. One of the major distinctions between various types of D/A converters is whether or not the converter includes latches to store the digital value to be converted to an analog quantity, and the interface to control those latches. With latches and control logic included with the converter, interface design is often simplified, however, internal latches are often included only in slower D/A converters.

Because slower converters limit signal bandwidths, the converter chosen for this design was selected to allow a reasonably wide range of signal frequencies to be processed, in addition to illustrating the technique of interfacing to a converter using external data latches.

Figure 12 shows an interface to an Analog Devices AD565A digital to analog converter. This device is a 12-bit, 250 ns current output DAC with an on-board 10 volt reference. Using an off-board current-to-voltage conversion circuit connected according to manufacturers specifications,

the converter exhibits output signal ranges 0 to +10 volts, which is compatible with the conversion range of the A/D converter discussed in the previous section.

Figure 12. Interface Between the TMS320C30 and the AD565A

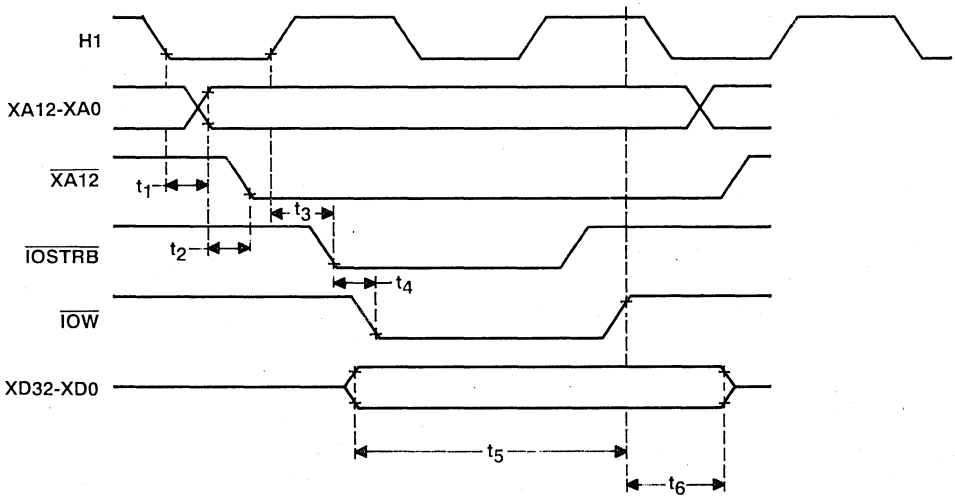


Because this DAC essentially performs continuous conversions based on the digital value provided at its inputs, periodic sampling is maintained by periodically updating the value stored in the external latches. Therefore, between sample updates, the digital value is stored and maintained at the latch outputs that provide the input to the DAC. This results in the analog output remaining stable until the next sample update is performed.

The external data latches used in this interface are '74LS377 devices that have both clock and enable inputs. These latches serve as a convenient interface with the TMS320C30; the enable inputs provide a device select function, and the clock inputs latch the data. Therefore, with the enable input driven by inverted XA12 and the clock input driven by $\overline{\text{IOW}}$, which is the AND of $\overline{\text{IOSTRB}}$ and XR/$\overline{\text{W}}$, data will be stored in the latches when a write is performed to I/O address 805000h. Reading this address has no effect on the circuit.

Figure 13 shows a timing diagram of a write operation to the D/A converter latches.

Figure 13. Write Operation to the D/A Converter Timing Diagram



Because the write is actually being performed to the latches, the key timings for this operation are the timing requirements for these devices. For proper operation, these latches require simply a minimal setup and hold time of data and control signals with respect to the rising edge of the clock input. Specifically, the latches require a data setup time of 20 ns, enable setup of 25 ns, disable setup of 10 ns and data and enable hold times of 5 ns. This design provides approximately 60 ns of enable setup, 30 ns of data setup, and 7.2 ns of data hold time. Therefore, the setup and hold times provided by this design are well in excess of those required by the latches. The key timing parameters for this interface are summarized in Table 2.

Table 2. Key Timing Parameter for D/A Converter Write Operation

Time Interval	Event	Time Period
t_1	H1 falling to address valid	10 ns
t_2	XA12 to XA12 delay	5 ns
t_3	H1 rising to IOSTRB falling	10 ns
t_4	IOSTRB to IOW delay	5.8 ns
t_5	Data setup to IOW	30 ns
t_6	Data hold from IOW	7.2 ns

System Control Functions

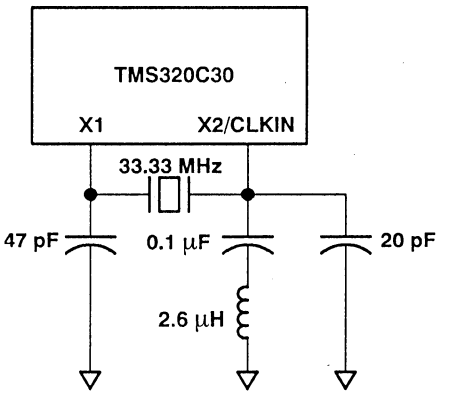
There are several aspects of TMS320C30 system hardware design that are critical to overall system operation. These include such functions as clock and reset signal generation and interrupt control.

Clock Oscillator Circuitry

An input clock may be provided to the TMS320C30 either from an external clock input or by using the on-board oscillator. Unless special clock requirements exist, using the on-board oscillator is generally a convenient method of clock generation. This method requires few external components and can provide stable, reliable clock generation for the device.

Figure 14 shows a clock generator circuit using the internal oscillator. This circuit is designed to operate at 33.33 MHz and since crystals with fundamental oscillation frequencies of 30 MHz and above are not readily available, a parallel-resonant third-overtone circuit is used.

Figure 14. Crystal Oscillator Circuit



In a third-overtone oscillator, the crystal fundamental frequency must be attenuated so that oscillation is at the third harmonic. This is achieved with an LC circuit that filters out the fundamental, thus allowing oscillation at the third harmonic. The impedance of the LC circuit must be inductive at the crystal fundamental and capacitive at the third harmonic. The impedance of the LC circuit is given by:

$$z\ (\omega) = \frac{L/C}{j\ [\omega_L - 1/\omega C\ ]} \tag{1}$$

Therefore, the LC circuit has a pole at:

$$\omega_p = \frac{1}{\sqrt{LC}} \tag{2}$$

At frequencies significantly lower than ω_p, the $1/(\omega C)$ term in (1) becomes the dominating term, while ω_L can be neglected. This gives:

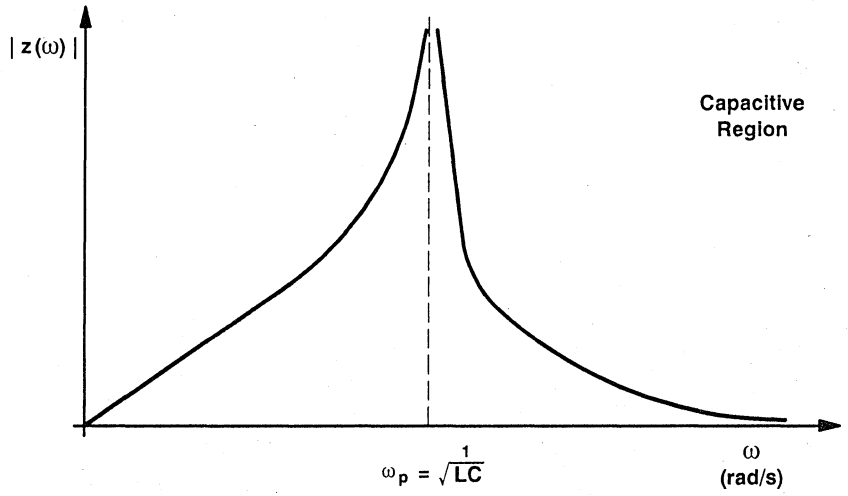
$$z\ (\omega) = j\omega L \text{ for } \omega < \omega_p \tag{3}$$

In (3), the LC circuit appears inductive at frequencies lower than ω_p. On the other hand, at frequencies much higher than ω_p, the ωL term is the dominant term in (1), and $1/(\omega C)$ can be neglected. This gives:

$$z(\omega) = \frac{1}{j\omega C} \quad \text{for } \omega > \omega_p \tag{4}$$

The LC circuit in (4) appears increasingly capacitive as frequency increases above ω_p. This is shown in Figure 15, which is a plot of the magnitude of the impedance of the LC circuit of Figure 14 versus frequency.

Figure 15. Magnitude of the Impedance of the Oscillator LC Network



Based on the discussion above, the design of the LC circuit proceeds as follows:

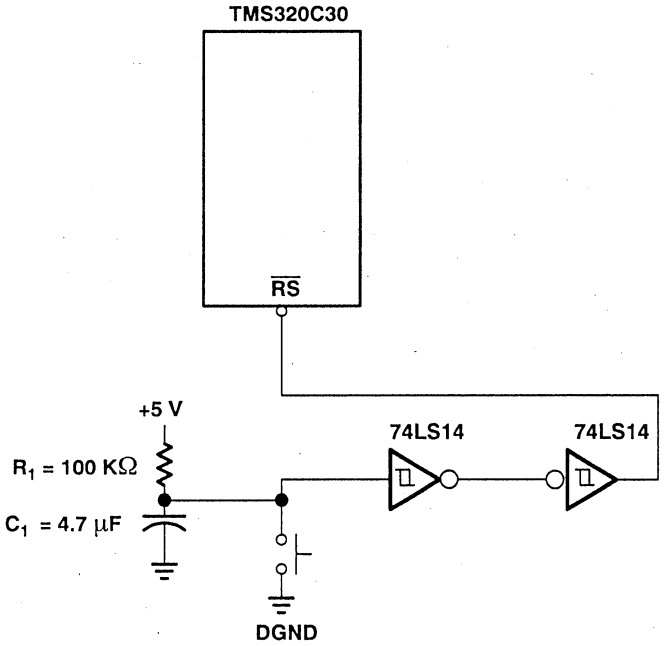
1) Choose the pole frequency ω_p approximately halfway between the crystal fundamental and the third harmonic.
2) The circuit now appears inductive at the fundamental frequency and capacitive at the third harmonic.

In the oscillator of Figure 13, choose ω_p = 22.2 MHz, which is approximately halfway between the fundamental and the third harmonic. Choose C = 20 pF. Then, using (2), L = 2.6 μH.

Reset Signal Generation

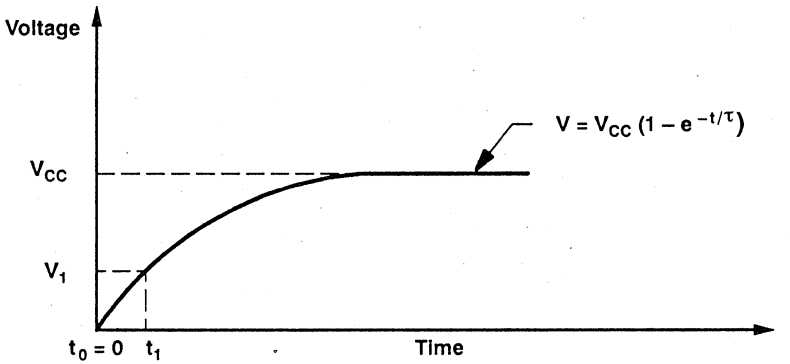
The reset input controls initialization of internal TMS320C30 logic and also causes execution of the system initialization software. For proper system initialization, the reset signal must be applied at least ten H1 cycles, i.e., 600 ns for a TMS320C30 operating at 33.33 MHz. Upon powerup, however, it can take 20 ms or more before the system oscillator reaches a stable operating state. Therefore, the powerup reset circuit should generate a low pulse on the reset line for 100 to 200 ms. Once a proper reset pulse has been applied, the processor fetches the reset vector from location zero which contains the address of the system initialization routine. Figure 16 shows a circuit that will generate an appropiate powerup reset circuit.

Figure 16. Reset Circuit



The voltage on the reset pin ($\overline{\text{RESET}}$) is controlled by the R_1C_1 network. After a reset, this voltage rises exponentially according to the time constant R_1C_1, as shown in Figure 17.

Figure 17. Voltage on the TMS320C30 Reset Pin.



The duration of the low pulse on the reset pin is approximately t_1, which is the time it takes for the capacitor C_1 to be charged to 1.5 V. This is approximately the voltage at which the reset input switches from a logic 0 to a logic 1. The capacitor voltage is given by:

$$V = V_{CC} \left[1 - e^{-\frac{t}{\tau}} \right] \quad (5)$$

where $\tau = R_1C_1$ is the reset circuit time constant. Solving (5) for t gives:

$$t = -R_1C_1 \ln \left[1 - \frac{V}{V_{CC}}\right] \quad (6)$$

Setting the following:

$R_1 = 100\ k\Omega$
$C_1 = 4.7\ \mu F$
$V_{CC} = 5\ V$
$V = V_1 = 1.5\ V$

gives t = 167 ms. Therefore, the reset circuit of Figure 16 provides a low pulse of long enough duration to ensure the stabilization of the system oscillator.

Note that if synchronization of multiple TMS320C30s is required, all processors should be provided with the same input clcock and the same reset signal. After powerup, when the clock has stabilized, all processors may then be synchronized by generating a falling edge on the common reset signal. Because it is in the falling edge of reset that establishes synchronization, reset must be high for a period of time (at least ten H1 cycles) initially. Following the falling edge, reset should remain low for at least ten H1 cycles and then be driven high. This sequencing of reset may be accomplished using additional circuitry, based on either RC time delays or counters.

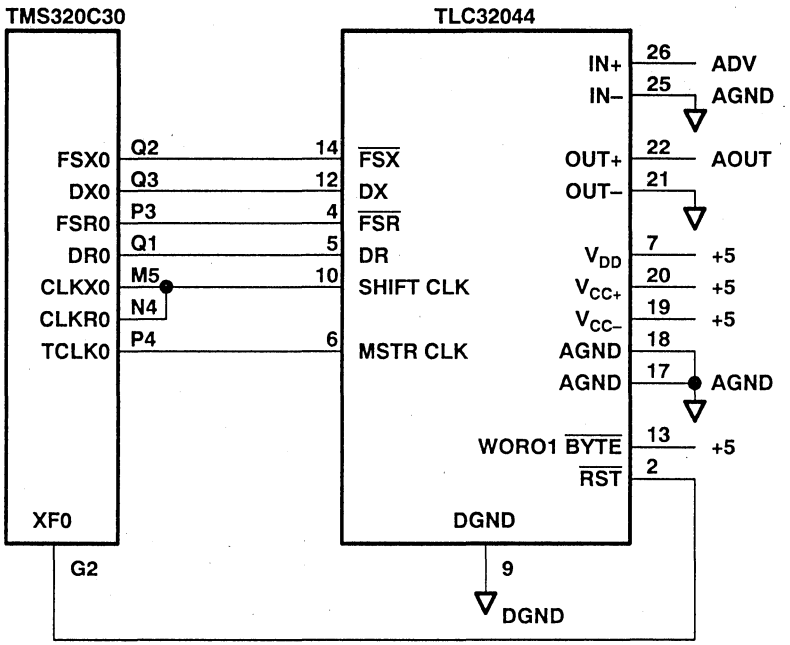
Serial Port Interface to AIC

For applications such as modems, speech, control, instrumentation, and analog interface for DSPs, a complete analog-to-digital (A/D) and digital-to-analog (D/A) input/output system on a single chip may be desired. The TLC32044 analog interface circuit (AIC) integrates on a single monlithic/CMOS chip a bandpass, switched-capacitor, antialiasing-input filter, 14-bit resolution A/D and D/A converters, and a lowpass, switched-capacitor, output-reconstruction filter. The TLC32044 offers numerous combinations of master clock input frequencies and conversion/sampling rates, which can be changed via digital processor control.

Four serial port modes on the TLC32044 allow direct interface to TMS320C30 processors. When the transmit and receive sections of the AIC are operating synchronously, it can interface to two SN54299 or SN74299 serial-to-parallel shift registers. These shift registers can then interface in parallel to the TMS320C30, other TMS320 digital processors, or to external FIFO circuitry. Output data pulses are emitted to inform the processor that data transmission is complete or to allow the DSP to differentiate between two transmitted bytes. A flexible control scheme is provided so that the functions of the AIC can be selected and adjusted coincidentally with signal processing via software control. Refer to the TLC32044 data sheet for detailed information.

When interfacing the AIC to the TMS320C30 via one of the serial ports, no additional logic is required. This interface is shown in Figure 18. The serial data, control and clock signals connect directly between the two devices and the AIC's master clock input is driven from TCLK0, one of the TMS320C30s internal timer outputs. The AIC's WORD/$\overline{\text{BYTE}}$ input is pulled high selecting 16-bit serial port transfers to optimize serial port data transfer rate. The TMS320C30s XF0, configured as an output, is connected to the AIC's reset ($\overline{\text{RST}}$) input to allow the AIC to be reset by the TMS320C30 under program control. This allows the TMS320C30 timer and serial port to be initialized before beginning conversions on the AIC.

Figure 18. AIC to TMS320C30 Interface



To provide the master clock input for the AIC, the TCLK0 timer is configured to generate a clock signal with a 50% duty cycle at a frequency of H1/4 or 4.167 MHz. To accomplish this, the timer 0 global control register is set to the value 3C1h, which establishes the desired operating modes. The timer 0 period register is set to 1 which sets the required division ratio for the H1 clock.

To properly communicate with the AIC the TMS320C30 serial port must be configured appropriately. To configure the serial port, several TMS320C30 registers and memory locations must be initialized. First the serial port should be reset by setting the serial port global control register to 2170300h. (The AIC should also be reset at this time. See description below of resetting the AIC using XF0). This resets the serial port logic and configures the serial port operating modes including data transfer lengths and enables the serial port interrupts. This also configures another important aspect of serial port operation: polarity of serial port signals. Because active polarity of all serial port signals is programmable, it is critical that the bits in the serial port global control register that control this be set appropriately. In this application all polarities are set to positive except FSX and FSR which are driven by the AIC and are true low.

The serial port transmit and receive control registers must also be initialized for proper serial port operation. In this application, both of these registers are set to 111h, which configures all of the serial port pins in the serial port mode, rather than the general purpose digital I/O mode.

With the operations described above completed, interrupts are enabled, and provided the serial port interrupt vector(s) are properly loaded, serial port transfers may begin after the serial port is taken out of reset. This is accomplished by loading E170300h into the global control register.

To begin conversion operations on the AIC and subsequent transfers of data on the serial port, the AIC is first reset by setting XF0 to zero at the beginning of the TMS320C30 initialization rou-

tine. Setting XF0 to zero is accomplished by setting the TMS320C30 IOF register to 2. This sets the AIC to a default configuration and halts serial port transfers and conversion operations until reset is set high. Once the TMS320C30 serial port and timer have been initialized as described above, XF0 is set high by setting the IOF register to 6. This allows the AIC to begin operating in its default configuration, which in this application is the desired mode. In this mode all internal filtering is enabled, sample rate is set at approximately 6.4 kHz, and the transmit and receive sections of the device are configured to operate synchronously. Conveniently, this mode of operation is appropriate for a variety of applications, and if a 5.184 MHz master clock input is used, the default configuration results in an 8 kHz sample rate which makes this device ideal for speech and telecommunications applications.

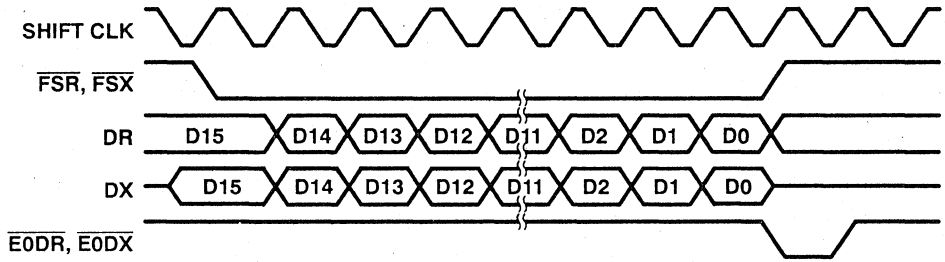
In addition to the benefit of a convenient default operating configuration, the AIC can also be programmed for a wide variety of other operating configurations. Sample rates and filter characteristics may be varied, in addition to which, numerous connections in the device may be configured to establish different internal architectures, by enabling or disabling various functional blocks.

To configure the AIC in a fashion different from the default state, the device must first be sent a serial data word with the two LSBs set to one. The two LSBs of a transmitted data word are not part of the transferred data information and are not set to one during normal operation. This condition indicates that the next serial transmission will contain secondary control information, not data. This information is then used to load various internal registers and specify internal configuration options. There are four different types of secondary control words distinguished by the state of the two LSBs of the control information transferred. Note that each secondary control word transferred must be preceded by a data word with the two LSBs set to one.

The TMS320C30 can communicate with the AIC either synchronously or asynchronously depending on the information in the control register. The operating sequence for synchronous communication with the TMS320C30 shown in Figure 19, is as follows:

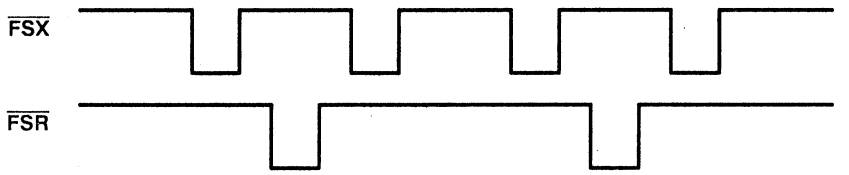
1) The $\overline{\text{FSX}}$ or $\overline{\text{FSR}}$ pin is brought low.
2) One 16-bit word is transmitted or one 16-bit word is received.
3) The $\overline{\text{FSX}}$ or $\overline{\text{FSR}}$ pin is brought high.
4) The $\overline{\text{EODX}}$ or $\overline{\text{OEDR}}$ pin emits a low-going pulse.

Figure 19. Synchronous Timing of TLC32044 to TMS320C30



For asynchronous communication, the operating sequence is similar, but $\overline{\text{FSX}}$ and $\overline{\text{FSR}}$ do not occur at the same time (see Figure 20). After each receive and transmit operation, the TMS320C30 asserts an internal receive (RINT) and transmit (XINT) interrupt, which may be used to control program execution.

Figure 20. Asynchronous Timing of TLC32044 to TMS320C30



XDS1000 Target Design Considerations

The TMS320C30 Emulator is an eXtended Development System (XDS1000) which has all the features necessary for full-speed emulation. The TMS320C30 uses a revolutionary technology to allow complete emulation via a serial scan path. If users provide a 12-pin header on their target system, realtime emulation can be performed using the TMS320C30 in their target system.

To use the emulation connector of the XDS1000, the signals shown in Figure 21. should be provided to a 12 pin header (two rows of six pins) with pin 8 cut out to provide keying. Table 3 describes the pins and signals present on the header.

Figure 21. 12 Pin Header Signals

Header Dimensions:
Pin-to-pin spacing: 0.100 inches (X,Y)
Pin width: 0.025 inches square post
Pin length:0.235 inches nominal

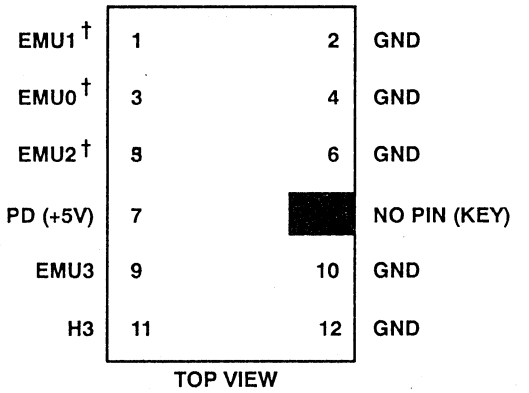


Table 3. Signal Description

Signal Name	Description
EMU0	Emulation pin 0.
EMU1	Emulation pin 1.
EMU2	Emulation pin 2.
EMU3	Emulation pin 3.
H3	TMS320C30 H3.
GND	Ground.
PD	Presence detect . It indicates that the cable is connected and target system is powered up. It should be tied to +5 volts in the target system.

In addition to the signals required at the emulation connector, the EMU4 through EMU6 signals on the TMS320C30 must also be appropiately connected to ensure proper emulation operation. The EMU4 signal must be tied to +5 volts and EMU5 and EMU6 must be left unconnected. Also, the RSV0 through RSV10 signals must be tied to +5 volts as described in the *Third-Generation TMS320 User's Guide* (literature number SPRU031).

Summary

The TMS320C30 is a high-performance 32-bit floating-point digital signal processor. Its dual parallel-interface busses and serial ports, along with a wide variety of additional support interfaces make the device an extremely flexible system-level DSP microprocessor. Using the techniques described in this report, the TMS320C30 can be used to implement sophisticated signal processing applications with the high precision and dynamic range provided by 32-bit floating-point arithmetic.

This application report has described the use of external interfaces on the TMS320C30 to connect it to memories, A/D and D/A converters, and numerous other peripheral devices, as well as the generation of wait states and other system functions.

The interfaces described in this report have all been built and tested to verify proper operation, and the techniques described can be extended to encompass design of more complex systems.

TMS320C30-IEEE Floating-Point Format Converter

Randy Restle, Regional Technology Center, Waltham, MA
Adam Cron, Digital Signal Processor Products—Semiconductor Group
Texas Instruments

Introduction

Certain applications require the exceptionally high arithmetic throughput inherent in the TMS320C30 Digital Signal Processor but must use the IEEE floating-point number format, which differs from the TMS320C30's number format. The TMS320C30 uses a 2's complement format for the mantissa and exponent. Besides making the device more compatible with analog to digital converters, it is computationally more efficient in both speed and die size than the IEEE format. Applications requiring the IEEE format can benefit from the use of a custom chip for this conversion. For this reason, a chip has been designed, built, and tested. This report describes that chip.

The TMS320C30-IEEE Floating-Point Number Format Converter is a peripheral that performs floating-point number conversions between the native format of the TMS320C30 and the Single-Precision IEEE Standard 754-1985. This conversion is performed in hardware and can convert an incoming (IEEE-formatted) or outgoing (TMS320C30-formatted) floating-point number in less than one TMS320C30 instruction cycle. Normally, the part is placed between memory and the TMS320C30.

This peripheral has two operating modes.

- Mode 1 does not pipeline any data through the chip. Instead, one wait state is automatically generated to compensate for the converter's propagation delays. This mode is equivalent in performance to equipping the TMS320C30 with a single-cycle convert instruction. In those applications where speed is of utmost importance, the pipeline mode is provided.
- Mode 2 enables the converter's built-in pipeline.

Because propagation delays through the chip reduce the access time required for TMS320C30 external memory, the pipeline mode allows conversions to take place on one data value while a previously converted value is being read, or written, by the TMS320C30. Depending on the TMS320C30 instruction cycle time and the access time of memories being used, the pipeline mode can eliminate degradation in TMS320C30 throughput entirely. However, it should be noted that values fed through the pipeline appear at the output in the next cycle. Therefore, an extra read or write (i.e., the same operation that was being performed) must be performed to flush the pipeline. Consequently, when pipeline mode is used, data values and their addresses are skewed from one another. This mode is intended for high-speed block transfer/conversion, and the address skew should be acceptable.

All control signals to and from the converter are compatible with TMS320C30 signals so that no extra circuitry is required to use this chip. In fact, it has been designed to appear as much as possible like a simple bus transceiver (e.g., SN74LS245). Consequently, it has two data buses. Data bus A (pins DA31 through DA0) should be connected directly to one of the TMS320C30's data buses and the other to memory. Its direction pin (DIR) should be tied to the read/write pin (R/$\overline{W}$), and its output enable pin ($\overline{OE}$) can be tied to either $\overline{STRB}$ or $\overline{MSTRB}$ of the TMS320C30, depending on where in the TMS320C30 memory map IEEE numbers are stored.

Key Features

This device is designed to fit into systems equipped with TMS320C30 external memory into which IEEE formatted numbers are stored. Below is a list of some specific features of the TMS320C30-IEEE Floating-Point Converter:

- Automatic wait-state generation during conversions
- Automatic interrupt generation when IEEE NaNs are encountered
- Automatic pipeline mode for single-cycle conversions
- Built-in SCOPE (i.e., JTAG) testability logic

Report Overview

- External Interfaces – Describes the external interfaces of this chip, the pinout, and pins.
- Architectural Overview – Describes the functions of the converter. Gives an overview of the TMS320C30 and IEEE Standard 754-1985 number formats and the scope of numbers that can be converted.
- Converter Operating Modes – Describes the converter's operating modes.
- Interrupts – Describes the Not a Number interrupt generated by the converter.
- Software Application Examples – Contains software application examples.
- Hardware Application Examples – Contains hardware application examples.
- JTAG/IEEE-1149.1 Scan Interface – Contains the JTAG/IEEE scan interface description.

Typographical Conventions

In this report, buses are signified with the bus name in capital letters, followed by the range of signals (bits) enclosed in parentheses and separated by a colon. For example, TI(31:0) is bus "TI", bits 31 through 0 (31 is the most significant bit, 0, the least). Table 1 shows the symbols and their corresponding meaning that are used in sections of the report concerning control logic, algorithm overview, and bit-specific conversion algorithms.

Table 1. Symbols and Meanings

Symbol	Name	Meaning
+	plus	arithmetic summation
|	pipe	logical OR
&	ampersand	logical AND
!	exclamation point	one's complement
–	minus	two's complement
^	caret	EXCLUSIVE OR

External Interfaces

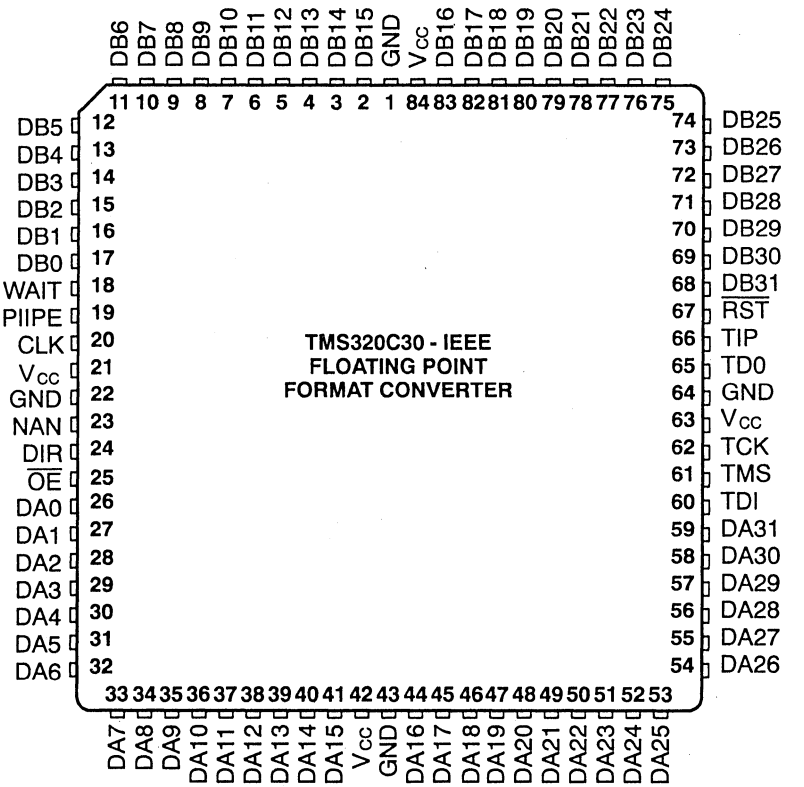
Packaging

The TMS320C30 device is housed in an 84-pin package. This pinout was chosen for efficient flow through connection to the buses. The TMS320C30-IEEE Converter's pin assignments are shown in Table 2, and the pin locations are shown in Figure 1.

Table 2. Pin Assignments

Pin	Name	Pin	Name	Pin	Name
1	GND	29	DA3	57	DA29
2	DB15	30	DA4	58	DA30
3	DB14	31	DA5	59	DA31
4	DB13	32	DA6	60	TDI
5	DB12	33	DA7	61	TMS
6	DB11	34	DA8	62	TCK
7	DB10	35	DA9	63	VCC
8	DB9	36	DA10	64	GND
9	DB8	37	DA11	65	TDO
10	DB7	38	DA12	66	TIP
11	DB6	39	DA13	67	RST
12	DB5	40	DA14	68	DB31
13	DB4	41	DA15	69	DB30
14	DB3	42	VCC	70	DB29
15	DB2	43	GND	71	DB28
16	DB1	44	DA16	72	DB27
17	DB0	45	DA17	73	DB26
18	WAIT	46	DA18	74	DB25
19	PIPE	47	DA19	75	DB24
20	CLK	48	DA20	76	DB23
21	VCC	49	DA21	77	DB22
22	GND	50	DA22	78	DB21
23	NAN	51	DA23	79	DB20
24	DIR	52	DA24	80	DB19
25	OE	53	DA25	81	DB18
26	DA0	54	DA26	82	DB17
27	DA1	55	DA27	83	DB16
28	DA2	56	DA28	84	VCC

Figure 1. Pin Locations



Pinout Description

Table 3 describes the pin functions.

Table 3. Converter Signals

Signal	Pins	Type	Description
DIR	1	Input	Direction – This pin determines what type of conversion should take place. When it is high, data on bus B is converted from IEEE to TMS320C30 format and output on bus A. When it is low, data on bus A is converted from TMS320C30 to IEEE format and output on bus B. This pin is normally tied directly to the TMS320C30 read/write pin.
$\overline{OE}$	1	Input	Output Enable (active low) – In combination with the DIR pin, this pin disables the currently driven bus (i.e., bus A or B).

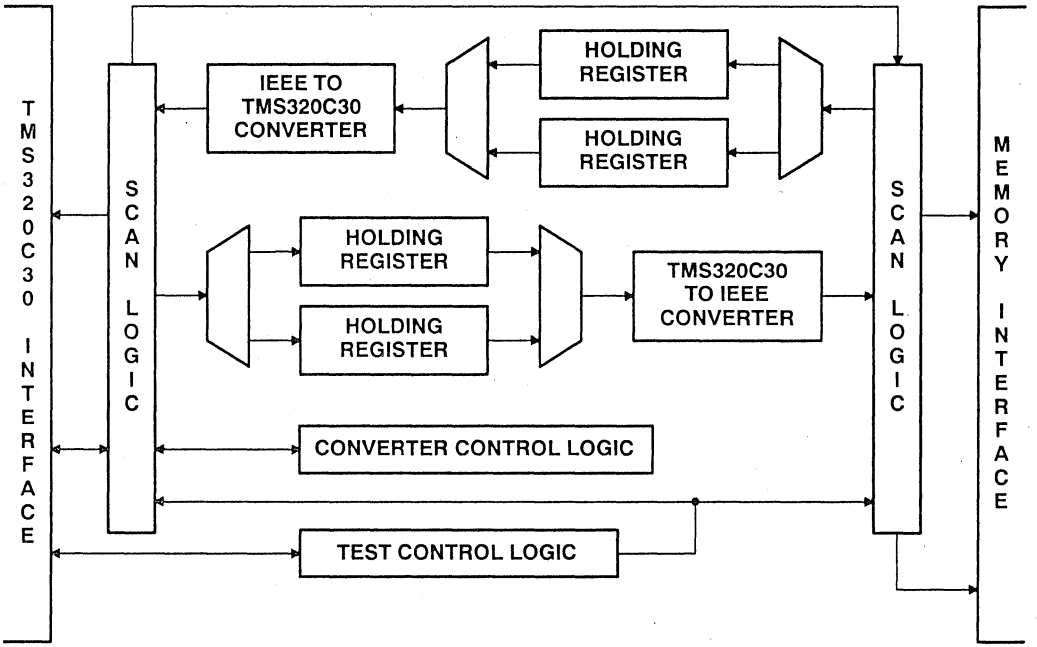
Table 3. Converter Signals (Concluded)

Signal	Pins	Type	Description
WAIT	1	Output	This pin is driven high in nonpipelined operations to signal the TMS320C30 to extend its external memory access to allow the conversion to complete. It can be tied directly to the TMS320C30 ready line. It is appropriately driven for both read and write operations, but is always low in pipelined mode of operation.
PIPE	1	Input	Pipeline Enable – When this is high, the converter is configured in pipeline mode. It must be tied low for nonpipeline mode.
CLK	1	Input	Clock – This clock is the wait-state generator and the pipeline clock. It should be connected directly to the TMS320C30 H1 clock pin.
NAN	1	Output	Not-a-Number Interrupt – This pin is driven low for 1.5 CLK cycles and signals an attempted conversion of the IEEE format: Not-a-Number. This pin can be tied directly to one of the TMS320C30 interrupt pins and can signal command or message passing in multi-processor, shared-memory-type designs.
DA(31:0)	32	Input/Output	Data Bus A – This 32-bit bus should be tied to either one of the two TMS320C30 data buses (i.e., the primary or expansion buses).
DB(31:0)	32	Input/Output	Data Bus B – This 32-bit bus is normally connected to a memory array containing IEEE-formatted data.
TCK	1	Input	Test Clock.
TMS	1	Input	Test Mode Select.
$\overline{\text{RST}}$	1	Input	Reset (active low) – This pin resets all logic on the device.
TDI	1	Input	Test Data In.
TDO	1	Output	Test Data Out.
TIP	1	Output	Test Instruction Register Parity – During instruction register scan, when paused, this output reflects instruction register even parity.

Architectural Overview

Figure 2 shows the block diagram of the converter.

Figure 2. Converter Block Diagram



Introduction

The TMS320C30 attains a peak performance of 33 MFLOPS, largely due to the floating-point format that it uses. In this format, both exponent and mantissa are represented in 2's-complement form.

In the IEEE format, the mantissa is represented in signed-magnitude form, and the exponent includes a bias (i.e., an offset). Additionally, values of numbers are not determined by the same formula. Instead, the exponent is used to flag numbers that are encoded differently. For example, if the exponent is 255, the value is considered not a number (NaN). Another exception is signaled when the exponent is zero. In this case, the mantissa is defined to be denormalized.
The TMS320C30's floating-point format is considerably simpler; most numbers can be converted to it without any loss of precision. However, some denormalized IEEE numbers are smaller than can be represented in TMS320C30 format. When these numbers are converted, they are translated to the closest TMS320C30 values. The error is less than $\pm 2^{-127}$.

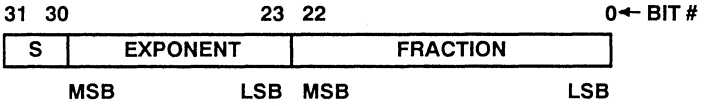
IEEE Floating-Point Format Overview

IEEE Standard 754-1985 defines formats for single-, single-extended-, double- and double-extended-precision floating-point numbers. The single-precision format fits entirely with-

in 32 bits, which is the bus width of the TMS320C30, and is the only format supported by the converter.

The format of the single-precision IEEE Standard 754-1985 is shown below:

Figure 3. Single-Precision IEEE Standard 754-1985 Format



In this format,

S is the sign bit of the mantissa (0 = positive, 1 = negative).

EXPONENT is an unsigned 8-bit field that determines the location of the binary point of the number being encoded.

FRACTION is a 23-bit field containing the fractional part of the mantissa.

LSB is the least significant bit of a field

MSB is the most significant bit of a field

The decimal value (v) of some number X is defined by one of five separate cases shown below:

Case 1: If **EXPONENT** = 255 and **FRACTION** $\neq$ 0, then v is NaN.

Case 2: If **EXPONENT** = 255 and **FRACTION** = 0, then v = $\pm$ infinity.

Case 3: If 0 < **EXPONENT** < 255, then $v = (-1)^{s}\, 2^{\mathbf{exp}-127}\, (1.\mathbf{FRAC})$

where:

S is either 0 or 1

FRAC is the decimal equivalent of **FRACTION**

EXP is the decimal equivalent of **EXPONENT**

Note that an implied 1 exists to the left of the binary point as shown above. This means the mantissa of an IEEE-encoded value has 24 bits of precision.

Case 4: If **EXPONENT** = 0 and **FRACTION** $\neq$ 0, then v is a denormalized number and $v = (-1)^{s}\, 2^{-126}\, (0.\mathbf{FRAC})$

where
S is either 0 or 1

FRAC is the decimal equivalent of **FRACTION**

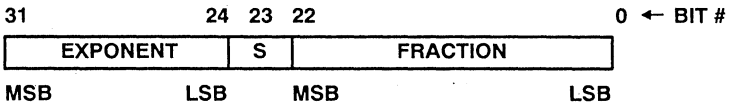
Note that an implied 0 exists to the left of the binary point as shown above. This means the mantissa of an IEEE-encoded value has 24 bits of precision.

Case 5: If **EXPONENT** = 0 and **FRACTION** = 0, then v = ± zero.

TMS320C30 Floating-Point Format Overview

TMS320C30 single-precision floating-point format uses a 2's-complement exponent and mantissa and is shown in Figure 4.

Figure 4. TMS320C30 Single-Precision Floating-Point Format



The decimal value (v) of some number X is determined as follows:

$v = \{(-2)^s + (.\mathbf{FRAC})\}\ 2^{\mathbf{exp}}$

where **S** is either 0 or 1

FRAC is the decimal equivalent of **FRACTION**

EXP is the decimal equivalent of **EXPONENT**

An alternate way of describing the TMS320C30 mantissa is as follows:

$\bar{s}$s.fraction

Note that the bit to the left of the binary point is implied and is the complement of the sign bit. This gives the TMS320C30's mantissa 24 bits of precision and not 23 bits as might be expected. For example:

The most positive TMS320C30 mantissa is

01.1111 1111 1111 1111 1111 111 = $2 - 2^{-23}$

The least positive TMS320C30 mantissa is

01.0000 0000 0000 0000 0000 000 = 1

The most negative TMS320C30 mantissa is

10.0000 0000 0000 0000 0000 000 = –2

The least negative TMS320C30 mantissa is

10.1111 1111 1111 1111 1111 111 = $-1 - 2^{-23}$

Note that zero is uniquely identified when the TMS320C30 exponent is –128.

IEEE Number Conversion

This section describes the classifications of IEEE numbers, how they are decoded, and the algorithms necessary to translate them to TMS320C30 format.

IEEE Dynamic Range

Table 4 shows the dynamic range of IEEE numbers. This chart can be used to quickly determine the case classification of an IEEE number.

Table 4. IEEE Range of Numbers

Sign	Exponent	Mantissa	Value	Type	Case
0	FF	0	not applicable	NaN	1
0	FF	0.000...000	+ infinity	+ Infinity	2A
0	FE	1.111...111	$(2-2^{-23})x2^{127}$	+ Normalized Number	3A
0	FE	1.111...110	$(2-2^{-22})x2^{127}$	+ Normalized Number	3A
0	FE	1.111...101	$(2-2^{-21}+2^{-23})x2^{127}$	+ Normalized Number	3A
0	FE	1.111...100	$(2-2^{-21})x2^{127}$	+ Normalized Number	3A
	.	.			
	.	.			
	.	.			
0	FE	1.000...000	2^{127}	+ Normalized Number	3A
0	FD	1.111...111	$(2-2^{-23})x2^{126}$	+ Normalized Number	3A
0	FD	1.111...110	$(2-2^{-22})x2^{126}$	+ Normalized Number	3A
0	FD	1.111...101	$(2-2^{-21}+2^{-23})x2^{126}$	+ Normalized Number	3A
0	FD	1.111...100	$(2-2^{-21})x2^{126}$	+ Normalized Number	3A
	.	.			
	.	.			
	.	.			
0	01	1.000...000	2^{-126}	+ Normalized Number	3A
0	00	0.111...111	$(1-2^{-23})x2^{-126}$	+ Denormalized Number	4A
0	00	0.111...110	$(1-2^{-22})x2^{-126}$	+ Denormalized Number	4A
0	00	0.111...101	$(1-2^{-21}+2^{-23})x2^{-126}$	+ Denormalized Number	4A
0	00	0.111...100	$(1-2^{-21})x2^{-126}$	+ Denormalized Number	4A
	.	.			
	.	.			
	.	.			
0	00	0.100...000	2^{-127}	+ Denormalized Number	4A
0	00	0.011...111	$(1-2^{-22})x2^{-127}$	– Denormalized Number	4B
0	00	0.011...110	$(1-2^{-21})x2^{-127}$	– Denormalized Number	4B
0	00	0.011...101	$(1-2^{-20}+2^{-22})x2^{-127}$	– Denormalized Number	4B
	.	.			
	.	.			
	.	.			
0	00	0.000...011	$(1+2^{-1})x2-148$	– Denormalized Number	4B
0	00	0.000...010	2^{-148}	– Denormalized Number	4B
0	00	0.000...001	2^{-149}	– Denormalized Number	4B
0	00	0.000...000	+ 0.0	+ Zero	5
1	00	0.000...000	– 0.0	– Zero	5
1	00	0.000...001	$-(2^{-149})$	– Denormalized Number	4D
1	00	0.000...010	$-(2^{-148})$	– Denormalized Number	4D
1	00	0.000...011	$-(1+2^{-1})x2^{-148}$	– Denormalized Number	4D
	.	.			
	.	.			
	.	.			

Table 4. IEEE Range of Numbers (Concluded)

Sign	Exponent	Mantissa	Value	Type	Case
1	00	0.011...111	$-(1-2^{-22})x2^{-127}$	– Denormalized Number	4D
1	00	0.100...000	$-(2^{-127})$	– Denormalized Number	4D
1	00	0.100...001	$-(1+2^{-22})x2^{-127}$	– Denormalized Number	4C
1	00	0.100...010	$-(1+2-21)x2-127$	– Denormalized Number	4C
1	00	0.100...011	$-(1+2^{-21}+2^{-22})x2^{-127}$	– Denormalized Number	4C
	.	.			
	.	.			
	.	.			
1	00	0.111...111	$-(1-2^{-23})x2^{-126}$	– Denormalized Number	4C
1	01	1.000...000	$-(2^{-126})$	– Normalized Number	3C
1	01	1.000...001	$-(1+2-^{23})x2^{-126}$	– Normalized Number	3B
1	01	1.000...010	$-(1+2^{-22})x2^{-126}$	– Normalized Number	3B
1	01	1.000...011	$-(1+2^{-22}+2^{-23})x2^{-126}$	– Normalized Number	3B
	.	.			
	.	.			
	.	.			
1	01	1.111...111	$-(2-2^{-23})x2^{-126}$	– Normalized Number	3B
1	02	1.000...000	$-(2^{-125})$	– Normalized Number	3C
1	02	1.000...001	$-(2+2^{-23})x2^{-125}$	– Normalized Number	3B
1	02	1.000...010	$-(2+2^{-22})x2^{-125}$	– Normalized Number	3B
1	02	1.000...011	$-(1+2^{-22}+2^{-23})x2^{-125}$	– Normalized Number	3B
	.	.			
	.	.			
	.	.			
1	FE	1.111...100	$-(2-2^{-21})x2^{127}$	– Normalized Number	3B
1	FE	1.111...101	$-(2-2^{-21}+2^{-23})x2^{127}$	– Normalized Number	3B
1	FE	1.111...110	$-(2-2^{-22})x2^{127}$	– Normalized Number	3B
1	FE	1.111...111	$-(2-2^{-23})x2^{127}$	– Normalized Number	3B
1	FF	= 0	– infinity	– Infinity	2B

IEEE-to-TMS320C30 Control Logic

The control logic that classifies incoming IEEE data in order to perform correct translation to TMS320C30 format is shown below. The form of the expressions was chosen to minimize propagation delay through the device.

The logic is simplified if the following three factors are used (refer to typographical definitions for symbols used):

```
EXPFF =  IEEE(30)   & IEEE(29)   & IEEE(28)   & IEEE(27)   &
         IEEE(26)   & IEEE(25)   & IEEE(24)   & IEEE(23)

EXP00 =  !( IEEE(30)  | IEEE(29)  | IEEE(28)  | IEEE(27)  |
         IEEE(26)   | IEEE(25)   | IEEE(24)   | IEEE(23)  )

MANT0 =  !( IEEE(21)  | IEEE(20)  | IEEE(19)  | IEEE(18)  |
         IEEE(17)   | IEEE(16)   | IEEE(15)   | IEEE(14)  |
```

IEEE(13) | IEEE(12) | IEEE(11) | IEEE(10) |
IEEE(9) | IEEE(8) | IEEE(7) | IEEE(6) |
IEEE(5) | IEEE(4) | IEEE(3) | IEEE(2) |
IEEE(1) | IEEE(0))

Then

Case 1: NaN

= EXPFF & (IEEE(22) | !MANT0)

Case 2A: positive infinity

= !IEEE(31) & EXPFF & !(IEEE(22) | !MANT0)

Case 2B: negative infinity

= IEEE(31) & EXPFF & !(IEEE(22) | !MANT0)

Case 3A: positive normalized numbers

= !IEEE(31) & !EXP00 & !EXPFF

Case 3B: negative normalized numbers with fraction ≠ 0

= IEEE(31) & !EXP00 & !EXPFF & (!MANT0 | IEEE(22))

Case 3C: negative normalized numbers with fraction = 0

= IEEE(31) & !EXP00 & !EXPFF & !(!MANT0 | IEEE(22))

Case 4A: positive denormalized numbers ≥ 2^{-127}

= !IEEE(31) & EXP00 & IEEE(22)

Case 4B: positive denormalized numbers < 2^{-127}

= !IEEE(31) & EXP00 & !IEEE(22) & !MANT0

Case 4C: negative denormalized numbers ≤ $(-1 - 2^{-23}) \times 2^{-127}$

= IEEE(31) & EXP00 & IEEE(22) & !MANT0

Case 4D: negative denormalized numbers > $(-1 - 2^{-23}) \times 2^{-127}$

= IEEE(31) & EXP00 & (IEEE(22) ^ !MANT0)

Case 5: positive and negative zero

= EXP00 & !IEEE(22) & MANT0

IEEE-to-TMS320C30 Conversion Algorithm Overview

Table 5 shows the conversion algorithms used on the sign, exponent, and mantissa fields of IEEE numbers to produce the corresponding TMS320C30 fields. These fields are broken down into bit-specific algorithms in the following section.

Table 5. Conversion Algorithms from IEEE to TMS320C30 Format

	TMS320C30		
Case	**Exponent**	**Sign**	**Fraction**
1.	e_{IEEE}	s_{IEEE}	f_{IEEE}
2A.	7Fh	s_{IEEE}	7F FFFFh
2B.	7Fh	s_{IEEE}	00 0000h
3A.	e_{IEEE} + 81h	s_{IEEE}	f_{IEEE}
3B.	e_{IEEE} + 81h	s_{IEEE}	$-f_{IEEE}$
3C.	e_{IEEE} ^ 80h	s_{IEEE}	$-f_{IEEE}$
4A.	81h	s_{IEEE}	2 x f_{IEEE}
4B.	80h	s_{IEEE}	00 0000h
4C.	81h	s_{IEEE}	2 x $-f_{IEEE}$
4D.	80h	0	00 0000h
5.	80h	0	00 0000h

Note: Fraction, above, has only 23-bits

IEEE-to-TMS320C30 Bit-Specific Conversion Algorithms

These circuits were designed by examining Table 5 and finding all possible choices for each bit. The different choices were fed into data selectors, whose addresses were derived from the case-identifying logic described in the preceding section on control logic.

For maximum performance, all data selectors were designed from NAND gates. This also permitted minimization by eliminating all NAND gates that had an input of 0 and by reducing the number of NAND inputs where a bit was always 1. However, for clarity, no minimization is shown here. Instead, that detail can be seen in the following figures.

The following bit algorithms are shown in bit descending order, starting with IEEE bit 31.

Figure 5. IEEE Bit 31 to TMS320C30 Bit 23

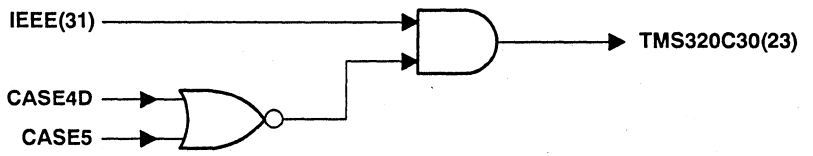
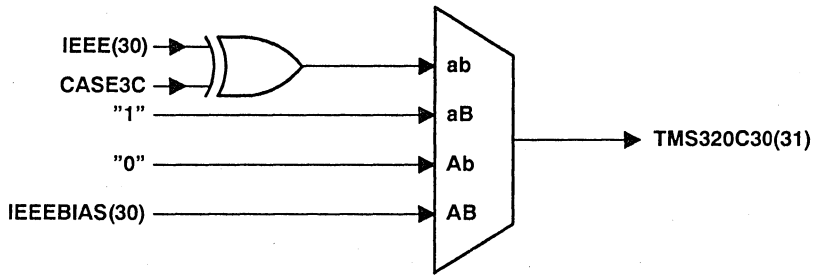
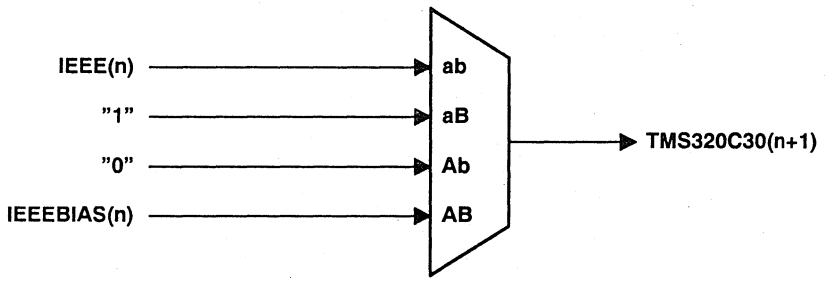


Figure 6. IEEE Bit 30 to TMS320C30 Bit 31



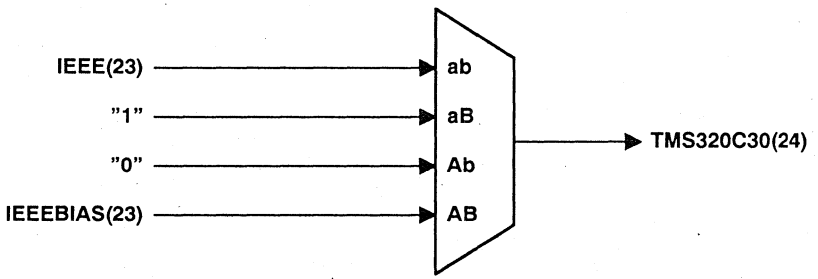
b = CASE1 | CASE2A | CASE2B | CASE3C
B = !b
A = CASE2A | CASE2B | CASE3A | CASE3B
a = !A

Figure 7. IEEE Bit n to TMS320C30 Bit n+1, Where 29 ≥ n ≥ 24



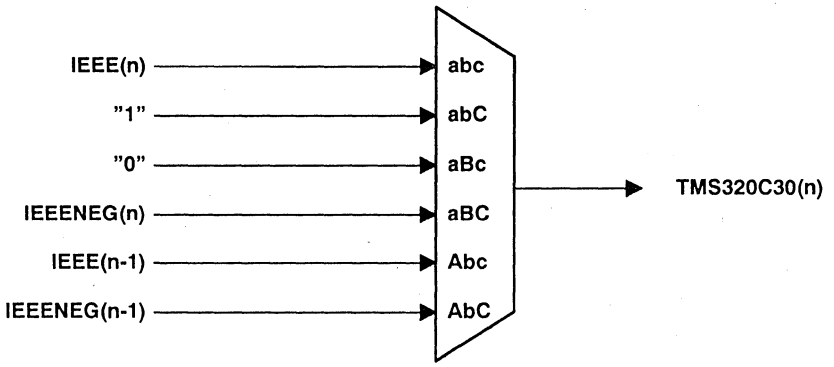
b = CASE2A | CASE2B | CASE3A | CASE3B
B = !b
a = CASE2A | CASE2B | CASE1 | CASE3C
A = !a

Figure 8. IEEE Bit 23 to TMS320C30 Bit 24



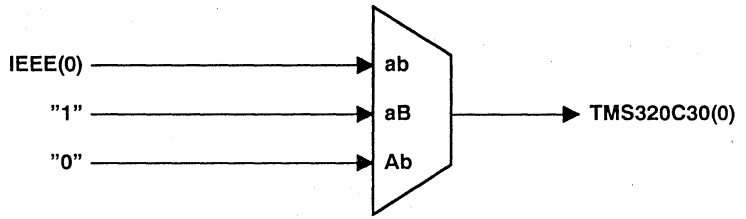
b = CASE1 | CASE3C | CASE4B | CASE4D | CASE5
B = !b
A = CASE4B | CASE4D | CASE5 | CASE3A | CASE3B
a = !A

Figure 9. IEEE Bit n to TMS320C30 Bit n, Where 22 ≥ n ≥ 1



C = CASE2A | CASE3B | CASE3C | CASE4C
c = !C
b = CASE1 | CASE2A | CASE3A | CASE4A | CASE4C
B = !b
A = CASE4A | CASE4C
a = !A

Figure 10. IEEE Bit 0 to TMS320C30 Bit 0



B = CASE2A
b = !B
A = CASE1 | CASE2A | CASE3A | CASE3B | CASE3C
a = !A

TMS320C30 Number Conversion

This section describes the classifications of TMS320C30 numbers, how they are decoded, and the algorithms necessary to translate them to IEEE format.

TMS320C30 Dynamic Range

Shown in Table 6 is the dynamic range of TMS320C30 numbers. As with Table 4, this table can be used to quickly determine case classification of a TMS320C30 number.

Table 6. TMS320C30 Range of Numbers

Exponent	Sign	Mantissa	Value	Type	Case
7F	0	1.111...111	$(2-2^{-23})x2^{127}$	Positive Number	6
7F	0	1.111...110	$(2-2^{-22})x2^{127}$	Positive Number	6
7F	0	1.111...101	$(2-2^{-21}+2^{-23})x2^{127}$	Positive Number	6
7F	0	1.111...100	$(2-2^{-21})x2^{127}$	Positive Number	6
.		.			
.		.			
.		.			
7F	0	1.000...000	2^{127}	Positive Number	6
7E	0	1.111...111	$(2-2^{-23})x2^{126}$	Positive Number	6
7E	0	1.111...110	$(2-2^{-22})x2^{126}$	Positive Number	6
7E	0	1.111...101	$(2-2^{-21}+2^{-23})x2^{126}$	Positive Number	6
.		.			
.		.			
.		.			
00	0	1.000...000	1	Positive Number	6
FF	0	1.111...111	$1-2^{-24}$	Positive Number	6
FF	0	1.111...110	$1-2^{-23}$	Positive Number	6
FF	0	1.111...101	$1-2^{-22}+2^{-24}$	Positive Number	6
.		.			
.		.			
.		.			
FF	0	1.000...000	2^{-1}	Positive Number	6
FE	0	1.111...111	$(2-2^{-23})x2^{-2}$	Positive Number	6
FE	0	1.111...110	$(2-2^{-22})x2^{-2}$	Positive Number	6
FE	0	1.111...101	$(2-2^{-21}+2^{-23})x2^{-2}$	Positive Number	6
.		.			
.		.			
.		.			
82	0	1.000...000	2^{-126}	Positive Number	6
81	0	1.111...111	$(2-2^{-23})x2^{-127}$	Positive Number	7 (note 1)
81	0	1.111...110	$(2-2^{-22})x2^{-127}$	Positivr Number	7 (note 1)
81	0	1.111...101	$(2-2^{-21}+2^{-23})x2^{-127}$	Positive Number	7 (note 1)
81	0	1.111...100	$(2-2^{-21})x2^{-127}$	Positive Number	7 (note 1)
.		.			
.		.			
.		.			
81	0	1.000...010	$(1+2^{-22})x2^{-127}$	Positive Number	7 (note 1)
81	0	1.000...001	$(1+2^{-23})x2^{-127}$	Positive Number	7 (note 1)
81	0	1.000...000	2^{-127}	Positive Number	7 (note 1)
80	0	0.111...111	(note 2)	Implied Zero	8
80	0	0.111...110	(note 2)	Implied Zero	8
80	0	0.111...101	(note 2)	Implied Zero	8
.		.			
.		.			
.		.			
80	0	0.000...001	(note 2)	Implied Zero	8

Table 6. TMS320C30 Range of Numbers (Concluded)

Exponent	Sign	Mantissa	Value	Type	Case
80	0	0.000...000	0.0	Zero	8
80	1	10.111...111	(note 2)	Implied Zero	(note 3)
80	1	10.111...110	(note 2)	Implied Zero	(note 3)
80	1	10.111...101	(note 2)	Implied Zero	(note 3)
.		.			
.		.			
.		.			
80	1	10.000...011	(note 2)	Implied Zero	(note 3)
80	1	10.000...010	(note 2)	Implied Zero	(note 3)
80	1	10.000...001	(note 2)	Implied Zero	(note 3)
80	1	10.000...000	(note 2)	Implied Zero	8
81	1	10.111...111	$(-1-2^{-23})x2^{-127}$	Negative Number	9 (note 1)
81	1	10.111...110	$(-1-2^{-22})x2^{-127}$	Negative Number	9 (note 1)
81	1	10.111...101	$(-1-2^{-21}+2^{-23})x2^{-127}$	Negative Number	9 (note 1)
.		.			
.		.			
.		.			
81	1	10.000...010	$(-2+2^{-22})x2^{-127}$	Negative Number	9 (note 1)
81	1	10.000...001	$(-2+2^{-23})x2^{-127}$	Negative Number	9 (note 1)
81	1	10.000...000	$-(2^{-126})$	Negative Number	10
82	1	10.111...111	$(-1-2^{-23})x2^{-126}$	Negative Number	11
82	1	10.111...110	$(-1-2^{-22})x2^{-126}$	Negative Number	11
82	1	10.111...101	$(-1-2^{-21}+2^{-23})x2^{-126}$	Negative Number	11
.		.			
.		.			
.		.			
FF	1	10.000...001	$-1+2^{-24}$	Negative Number	11
FF	1	10.000...000	-1	Negative Number	10
00	1	10.111...111	$(-1-2^{-23})x2^{-1}$	Negative Number	11
00	1	10.111...110	$(-1-2^{-22})x2^{-1}$	Negative Number	11
00	1	10.111...101	$(-1-2^{-21}+2^{-23})x2^{-1}$	Negative Number	11
.		.			
.		.			
.		.			
00	1	10.000...001	$-2+2^{-23}$	Negative Number	11
00	1	10.000...000	-2	Negative Number	10
01	1	10.111...111	$-2-2^{-22}$	Negative Number	11
01	1	10.111...110	$-2-2^{-21}$	Negative Number	11
01	1	10.111...101	$-2-2^{-20}+2^{-22}$	Negative Number	11
.		.			
.		.			
.		.			
7F	1	10.000...001	$(-2+2^{-23})x2^{127}$	Negative Number	11
7F	1	10.000...000	$-(2^{128})$	Negative Number	12

Notes: 1) Numbers converted to IEEE denormalized values lose one least significant bit of accuracy.

2) The TMS320C30 does not produce these numbers under normal arithmetic operations. Because the exponent of these numbers is –128, the TMS320C30 considers them zero. TMS320C30 Boolean operations are capable of producing numbers of these forms. Because of this, proper conversion to IEEE format is unclear and should be avoided. See note 3.

3) Case 8 & Case 9 are activated simultaneously. This is the only instance where the cases are not mutually exclusive. The TMS320C30 does not produce these numbers under normal arithmetic operations. Because the exponent of these numbers is –128, the TMS320C30 considers them zero. TMS320C30 Boolean operations are capable of producing numbers of these forms. Because of this, proper conversion to IEEE format is unclear. This dilemma can be resolved with minor modification to the case qualifier logic. See note 2.

TMS320C30-to-IEEE Control Logic

Conversion from TMS320C30 format to IEEE format is qualified with a different set of Boolean equations. To eliminate confusion between IEEE and TMS320C30 cases, different case numbers are used.

The logic is simplified if the following three factors are used:

```
EXP80_81 =  !C30(31) | C30(30) | C30(29) | C30(28) |
            C30(27) | C30(26) | C30(25)

EXP7F =     !C30(31) & C30(30) & C30(29) & C30(28) &
            C30(27) & C30(26) & C30(25) & C30(24)

MANT0 =     C30(22) | C30(21) | C30(20) | C30(19) |
            C30(18) | C30(17) | C30(16) | C30(15) |
            C30(14) | C30(13) | C30(12) | C30(11) |
            C30(10) | C30(9)  | C30(8)  | C30(7)  |
            C30(6)  | C30(5)  | C30(4)  | C30(3)  |
            C30(2)  | C30(1)  | C30(0)
```

Then,

Case 6: positive numbers $\geq 2^{-126}$

= !EXP80_81 & !C30(23)

Case 7: positive numbers N such that

$(2-2^{-23}) \times 2^{-127} \geq N \geq 2-127$

= EXP80_81 & C30(24) & !C30(23)

Case 8: zero

= EXP80_81 & C30(24)

Case 9: negative numbers N such that

$(-1-2^{-23})x2^{-127} \geq N \geq (-2+2^{-23})x2^{-127}$

= EXP80_81 & C30(23) & !MANT0

Case 10: negative numbers N such that

$-(2^{-126}) \geq N \geq -(2^{127})$ and whose fraction is 0

= !(EXP80_81 & !C30(24)) & !EXP7F & C30(23) & MANT0

Case 11: negative numbers N such that

$-(2^{-126}) > N > -(2^{128})$ and whose fraction $\neq 0$

= !EXP80_81 & C30(23) & !MANT0

Case 12: negative 2^{128}

= EXP7F & C30(23) & MANT0

TMS320C30-to-IEEE Conversion Algorithm Overview

Table 7 shows the conversion algorithms used on the sign, exponent, and mantissa fields of TMS320C30 numbers to produce the corresponding IEEE fields. These fields are broken down into bit-specific algorithms in the next section.

Table 7. Conversion Algorithms from TMS320C30 to IEEE Format

	IEEE		
Case	**Sign**	**Exponent**	**Fraction**
6	s_{C30}	e_{C30}+7Fh	f_{C30}
7	s_{C30}	00	(f_{C30}/2)+400000h
8	0	00	00 0000h
9	s_{C30}	00	($\bar{f}_{C30}$+1)/2+400000h
10	s_{C30}	e_{C30}+80h	00 0000h
11	s_{C30}	e_{C30}+7Fh	$\bar{f}_{C30}$+1
12	s_{C30}	FFh	00 0000h

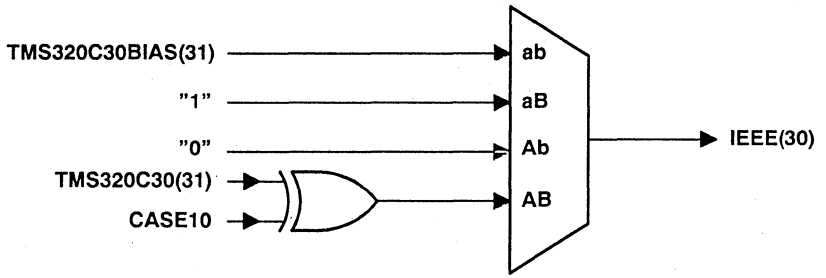
TMS320C30-to-IEEE Bit-Specific Conversion Algorithms

These circuits were designed by examining Table 7 and finding all possible choices for each bit. The different choices were fed into data selectors whose addresses were derived from the case-identifying logic described in the preceding section on TMS320C30 to IEEE control logic.

Just as in the IEEE case-identifying logic, all data selectors were designed from NAND gates for maximum performance. This also permitted minimization by eliminating all NAND gates having an input of 0 and by reducing the number of NAND inputs where a bit was always 1. However, for clarity, no minimization is shown here. Instead, that detail can be seen in the following figures.

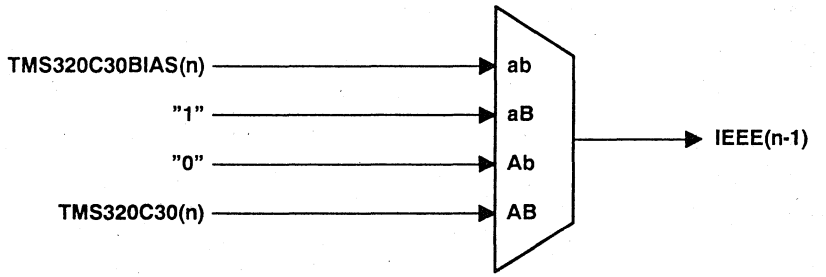
The following bit algorithms are shown in bit-descending order, starting with TMS320C30 bit 31.

Figure 11. TMS320C30 Bit 31 to IEEE Bit 30



B = CASE10 | CASE12
b = !B
a = CASE6 | CASE11 | CASE12
A = !a

Figure 12. TMS320C30 Bit n to IEEE Bit n–1, Where 31 ≥ n ≥ 24



B = CASE10 | CASE12
b = !B
a = CASE6 | CASE11 | CASE12
A = !a

Figure 13. TMS320C30 Bit 23 to IEEE Bit 31

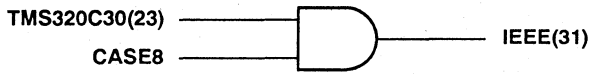
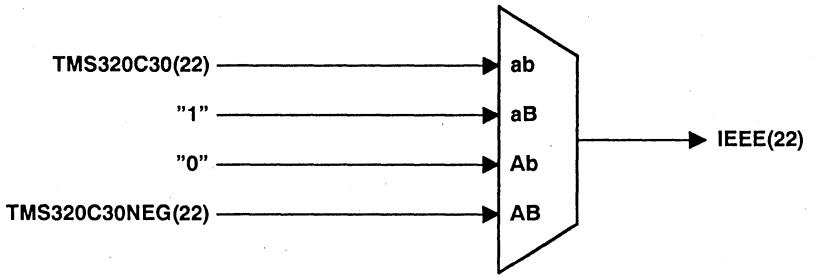
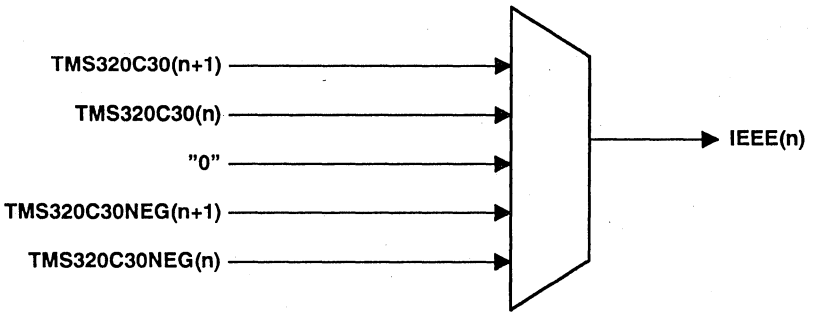


Figure 14. TMS320C30 Bit 22 to IEEE Bit 22



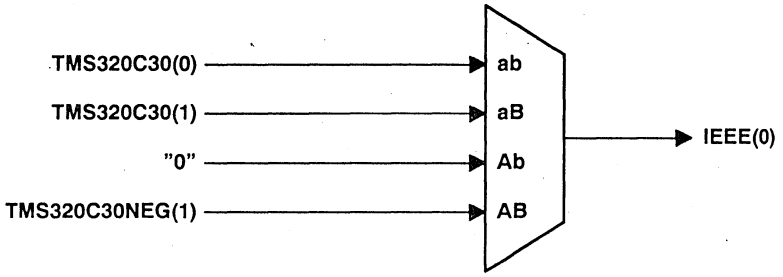
B = CASE7 | CASE9 | CASE11
b = !B
a = CASE6 | CASE7 | CASE9
A = !a

Figure 15. TMS320C30 Bit n to IEEE Bit n, Where 21 ≥ n ≥ 1



C = CASE6 | CASE9
c = !C
b = CASE6 | CASE7 | CASE11
B = !b
A = CASE11
a = !A

Figure 16. TMS320C30 Bit 0 to IEEE Bit 0:



B = CASE7 | CASE9
b = !B
a = CASE6 | CASE7 | CASE11
A = !a

Scope of Conversion

This section describes the actions taken by the converter when it converts to and from the IEEE format. When there is not a match between formats, the converter forces the translated number to the closest approximation.

IEEE-to-TMS320C30 Exceptions

The match is not exact in translating from four sets of IEEE numbers to TMS320C30 numbers. They are: NaN, ± infinity, ± zero and denormalized numbers too small to represent.

NaN (Not a Number)

The NaN format is especially useful in passing commands to another process. So that commands can be passed through the converter, NaNs are not converted. However, the bit positions of the sign and exponent bits are altered. That is, the sign bit of the IEEE number is transferred to the sign bit of the TMS320C30 format. Likewise, the exponent field is transferred. In this way, the sign of the NaN is preserved which may aid in quick detection of the code. In other words, the TMS320C30 Branch on Positive instruction (BP) or Branch on Negative instruction (BN) are effective. So that the command can be acted on quickly, a NaN interrupt is generated.

± *Infinity*

When positive or negative infinity is passed through the converter, the most positive or negative TMS320C30 number is produced.

Denormalized numbers whose magnitude $< 2^{-126}$

Half of the denormalized IEEE numbers are out of range of TMS320C30 numbers. These denormalized numbers have very small magnitudes and are therefore forced to zero when converted.

± *Zero*

The IEEE format includes representations for positive and negative zero, but the TMS320C30 format does not. The converter forces each of these numbers to the singular TMS320C30 zero format.

TMS320C30-to-IEEE Exceptions

There are two sets of TMS320C30 numbers that do not perfectly match IEEE numbers. One set consists of a single value (-2^{127}). The other consists of numbers converted to IEEE denormalized numbers.

-2^{127}

The single value, -2^{127}, is a very large negative number. When this number is translated, negative infinity is produced.

Numbers Translated to Denormalized Values

When the exponent is –127, denormalized IEEE numbers are produced, and one least significant bit of accuracy is lost. This occurs because the TMS320C30 mantissa must be right-shifted one bit in order that the exponent be increased to –126, which is the most negative exponent the IEEE format can use.

Converter Operating Modes

The converter is controlled by the TMS320C30. Conversions occur when the converter's output enable pin ($\overline{OE}$) is active (i.e., low) and the TMS320C30 performs a read or write over its primary ($\overline{STRB}$ active) or expansion ($\overline{MSTRB}$ active) buses. This requires the converter to be placed directly between the TMS320C30 and external memory. That memory is where IEEE data will be stored. If direct (i.e., no conversion wanted) access to that memory is desired, transceivers like the SN74LS245 should be added in parallel with the converter. However, doing so requires that only one data path be enabled at a time. If unused, one of the XF pins of the TMS320C30 can be dedicated to perform this selection.

During a read, data is converted from IEEE format to TMS320C30 format. During a write, data is converted from TMS320C30 format to IEEE format. This will happen if the TMS320C30 R/$\overline{W}$ or XR/$\overline{W}$ pin is tied to the converter's direction (DIR) pin. Table 8 shows how to put the converter into its two operating modes and briefly describes each mode.

Table 8. Converter Operating Modes

Mode	Pin	Description
Memory	PIPE=0	Flow-Through Conversion Enabled – In this mode, the converter essentially behaves like a simple bus transceiver, such as an SN74LS245, except with an integrated floating-point format converter. When this mode is used, conversions take two cycles. Because of this, the converter automatically generates a wait state, which will halt the TMS320C30 for one cycle until the conversion is complete.
Pipeline	PIPE=1	Converter's Pipeline Registers Enabled Internally – This mode permits single-cycle conversion. As one data value is being converted, a previously converted value is output.

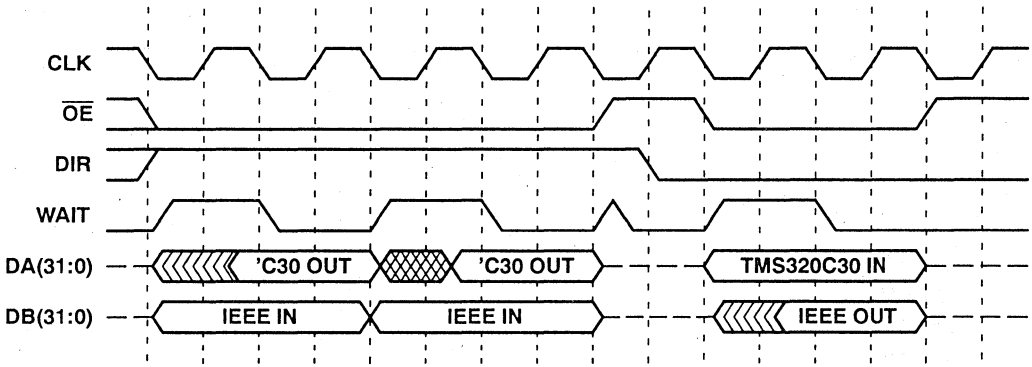
Memory Mode Operation

In this mode, one wait cycle is automatically generated during conversions from

- IEEE format to TMS320C30 format (reads)
- TMS320C30 format to IEEE format (writes)

The converter will not generate wait cycles of any other length and requires that the TMS320C30 H1 clock pin be tied to the converter's CLK pin. Figure 17 shows the timing diagram for this mode of operation.

Figure 17. Memory Mode Timing Diagram

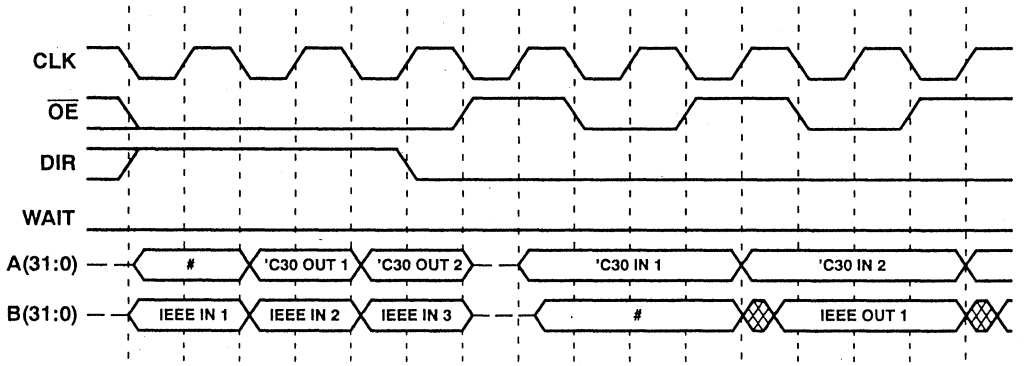


Pipelined Operation

Pipeline mode permits consecutive conversions every instruction cycle without wait cycles. However, because the pipeline has two internal stages, it takes two consecutive occurrences of the same operation (i.e., two reads or two writes) before it is filled. Therefore, the first read after a transition from a write will not provide properly converted data, and vice versa.

There is an address skew of one address when consecutive data values are converted. This should not be a major problem when blocks of memory are converted. The only added task will be to perform one extra transfer (read or write) to convert the last value remaining in the pipeline. With this exception, operation is identical to the Memory mode. Figure 18 shows a timing diagram for this mode of operation.

Figure 18. Pipeline Mode Timing Diagram



Interrupts

The converter automatically generates an interrupt whenever the conversion of an IEEE number classified as Not a Number (NaN) is attempted. The interrupt pulse is 1.5 H1 cycles wide. This is compatible with the TMS320C30 edge-triggered interrupt types. Table 9 shows this interrupt and its trigger. Note that the converter does not change the value of the NaN, but it does alter its bit positions. This assures that the sign bit of the IEEE number remains a sign bit in the TMS320C30 format. The same is true of the exponent field. The fractional field is left unchanged. If NaN is used to pass a code or command to the TMS320C30, interpretation of the code requires only the alteration of the comparison mask in software. For more information, refer to the previous subsection NaN (Not a Number).

Table 9. NaN Interrupt

Name	Function	Sources
NAN	Not a Number	IEEE CASE1: NaN

Simple Nonpipelined Conversion

If an external device (i.e., RAM, ROM, dual bus RAM, latch, etc.) contains a single-precision IEEE floating-point number and the corresponding TMS320C30 number is needed, the following TMS320C30 code will perform the required conversion:

```
EXTD    .word   0800000h        ; put address of external device here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDF     *AR0,R0         ; R0=C30 formatted number
```

The following example performs TMS320C30-to-IEEE format conversion:

```
EXTD    .word   0800000h        ; put address of external device here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        STF     R0,*AR0         ; location pointed to by AR0=IEEE formatted
*                               ;   number
```

Simple Pipelined Conversion

This example illustrates the overhead when the converter's pipeline mode is used. Since a single value will be converted, it is necessary to read the converter one extra time to flush the pipeline. Once again, assume that an external device (i.e., RAM, ROM, dual bus RAM, latch, etc.) contains a single-precision IEEE floating-point number, and the corresponding TMS320C30 number is needed.

```
EXTD    .word   0800000h        ; put address of external device here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDF     *AR0,R0         ; ignore loaded value, 1st load queues
*                               ;   pipeline
        LDF     *AR0,R0         ; R0=C30 formatted number, address is
*                               ;   immaterial
```

The following example performs TMS320C30 to IEEE format conversion:

```
EXTD    .word   0800000h        ; put address of external device here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        STF     R0,*AR0         ; value stored not correct until 2nd store
        STF     R0,*AR0         ; location pointed to by AR0=IEEE formatted
*                               ;   number
```

Pipelined Block Conversions

In the previous subsection, the pipeline was used, but not efficiently. This example shows a more typical application of pipeline mode. Again, external memory contains IEEE formatted data.

```
N       .set    03FFh           ; N = # of values to convert – 1
EXTD    .word   0800000h        ; put external address here
DADR    .word   0809800h        ; put destination address here
*
```

```
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDI     @DADR,AR1       ; load AR1 w/destination address
        LDF     *AR0++,R0       ; prime (preload) the converter's pipeline
        LDI     N,RC            ; block will be repeated N (0400h) times
        RPTB    RCR             ; specify end address of block repeat
        LDF     *AR0++,R0       ; read converted values into R0
RCR:    STF     R0,*AR1++       ; store converted values into on-chip
*                               ;   memory
```

This is more efficient:

```
N       .set    03FEh           ; N = # of values to convert - 2
EXTD    .word   0800000h        ; put external address here
DADR    .word   0809800h        ; put destination address here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDI     @DADR,AR1       ; load AR1 w/destination address
        LDF     *AR0++,R0       ; prime (preload) the converter's pipeline
        LDF     *AR0++,R0       ; read 1st converted value for 1st STF
        RPTS    N               ; repeat next instruction N-1 (03FFh)
*                               ;   times, extra loop is to store last
*                               ;   value converted
        LDF     *AR0++,R0       ; read converted values into R0
||      STF     R0,*AR1++       ; store converted values into on-chip
*                               ;   memory, 1st store will save junk
```

The following example performs TMS320C30 to IEEE format conversion:

```
N       .set    0400h           ; N equals number of values to convert
EXTD    .word   0800000h        ; put external address here
SADR    .word   0809800h        ; put source data address here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDI     @SADR,AR1       ; load AR1 w/source data address
        LDI     N,RC            ; block will be repeated N+1 (0401h) times,
*                               ;   extra loop is to store last value
*                               ;   converted
        RPTB    AC              ; specify end address of block repeat
        LDF     *AR1++,R0       ; read TMS320C30 format numbers into R0
AC:     STF     R0,*AR0++       ; store converted values into external
*                               ;   device
```

This is more efficient:

```
N       .set    03FFh           ; N equals number of values to convert - 1
EXTD    .word   0800000h        ; put external address here
SADR    .word   0809800h        ; put source data address here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDI     @SADR,AR1       ; load AR1 w/source data address
        LDF     *AR0++,R0       ; read 1st converted value for 1st STF
        RPTS    N               ; repeat next instruction N (0400h) times,
*                               ;   extra loop is to store last value
*                               ;   converted
        LDF     *AR1++,R0       ; read converted values into R0
||      STF     R0,*AR0++       ; store converted values into external
*                               ;   device
        STF     R0,*AR0++       ; store last value
```

Using TMS320C30 External Flag 0 (XF0)

As mentioned in the section on converter operating modes, one of the TMS320C30's XF pins can be tied to the converter's output enable (OE) pin to enable the data path through the converter

or to bypass it, as the case may be. The following TMS320C30 code uses the TMS320C30 XF0 pin to do this (see Hardware Applications Examples section later in this report for the hardware configuration). Nonpipelined mode is assumed.

```
N       .set    03FFh           ; N equals number of values to convert - 1
EXTD    .word   0800000h        ; put external address here
SADR    .word   0809800h        ; put source data address here
*
        LDI     @EXTD,AR0       ; load AR0 w/address of external device
        LDI     @SADR,AR1       ; load AR1 w/source data address
        LDI     2,IOF           ; XF0=output=0, select the converter
        LDF     *AR0++,R0       ; read 1st converted value for 1st STF
        RPTS    N               ; repeat next instruction N+1 (0400h) times
        LDF     *AR1++,R0       ; read converted values into R0
||      STF     R0,*AR1++       ; store converted values into on-chip
*                               ;   memory, 1st store will save junk
        LDI     6,IOF           ; XF0=output=1, deselect the converter
```

Using the TMS320C30 DMA Capability

The built-in TMS320C30 DMA controller can be used to read converted IEEE values. The TMS320C30 assembly code to set up the DMA is shown below. Non-pipelined mode is assumed.

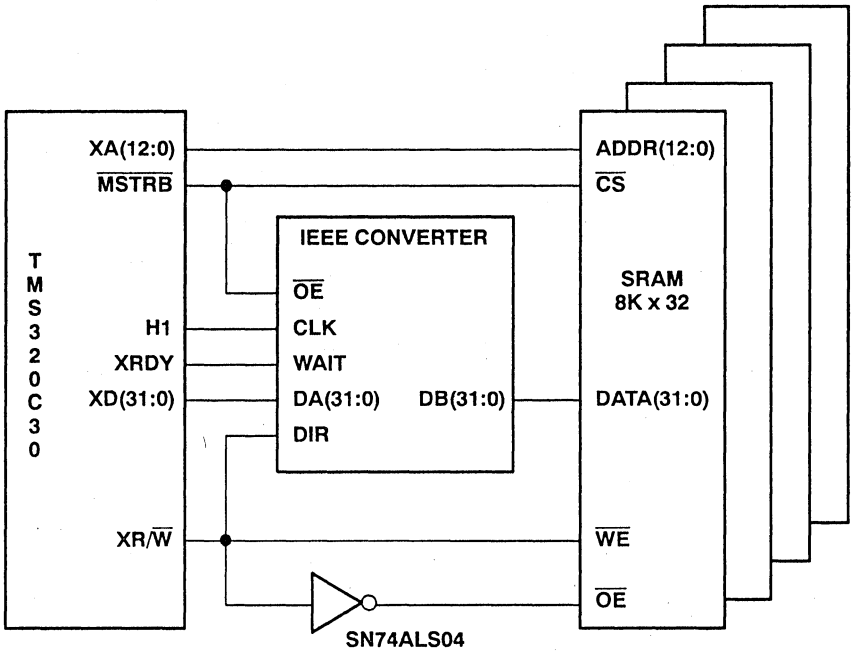
```
DMA     .word   0808000h        ; base address of DMA registers
GLBL    .word   0C53h           ; DMA global register init value
N       .set    0400h           ; N equals number of values to convert
EXTD    .word   0800000h        ; put external address here
DADR    .word   0809800h        ; put destination data address here
*
*    DMA controller setup
*
        LDI     @DMA,AR0        ; AR0 -> DMA control registers
        LDI     @EXTD,R0        ; R0 = address of IEEE data
        LDI     @DADR,R1        ; R1 = converted data destination address
        LDI     N,R2            ; R2 = DMA transfer count
        LDI     @GLBL,R3        ; R3 = DMA Global register initial value
        STI     R0,*+AR0(4)     ; DMA will transfer from external device
        STI     R1,*+AR0(6)     ; DMA will transfer to RAM block 0
        STI     R2,*+AR0(8)     ; DMA will transfer N values
        STI     R3,*AR0         ; start the DMA
```

Hardware Application Examples

IEEE Data Stored in TMS320C30 External MSTRB Memory

Below is shown an example of interfacing the converter to TMS320C30 external memory containing only IEEE formatted data. In this configuration, it is likely that the memory would be dual bus RAM to enable a second processor to share data with the TMS320C30 through this memory. Figure 19 shows an interface to a static RAM (SRAM) bank.

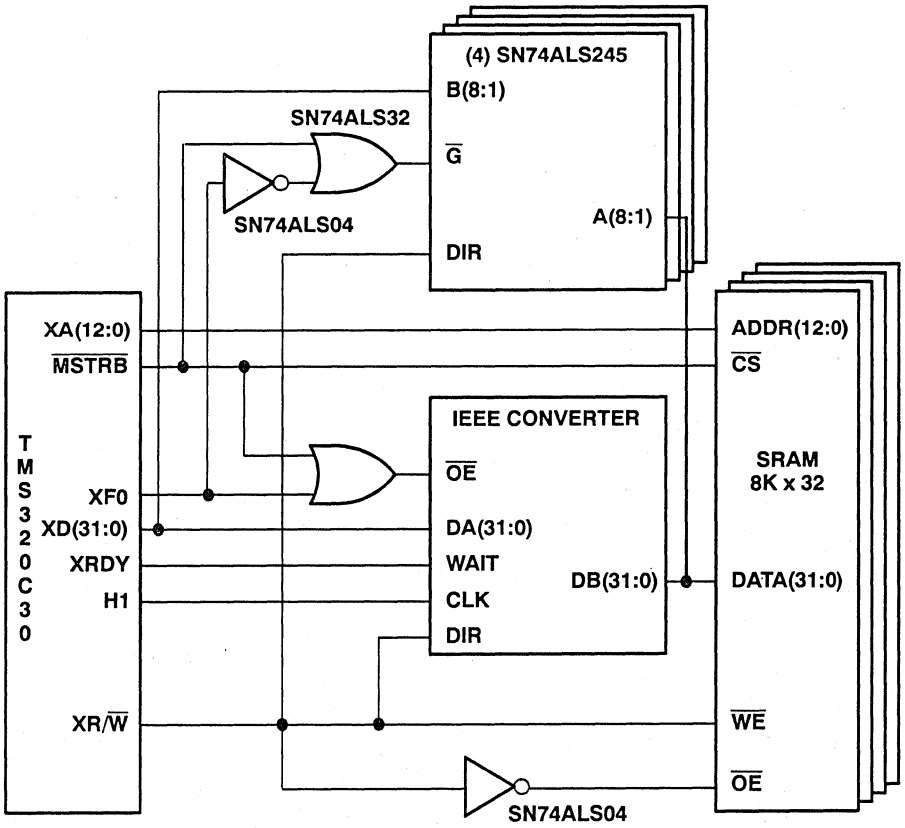
Figure 19. Interface to Static RAM



Bypassing the Converter

A previous subsection (Using TMS320C30 External Flag 0) showed TMS320C30 assembly code that used the TMS320C30 XF0 pin either to steer data through the converter or to bypass the converter for direct, or unconverted, access to that memory. Figure 20 shows a circuit that can be used with that code.

Figure 20. Steered Access to the Memory



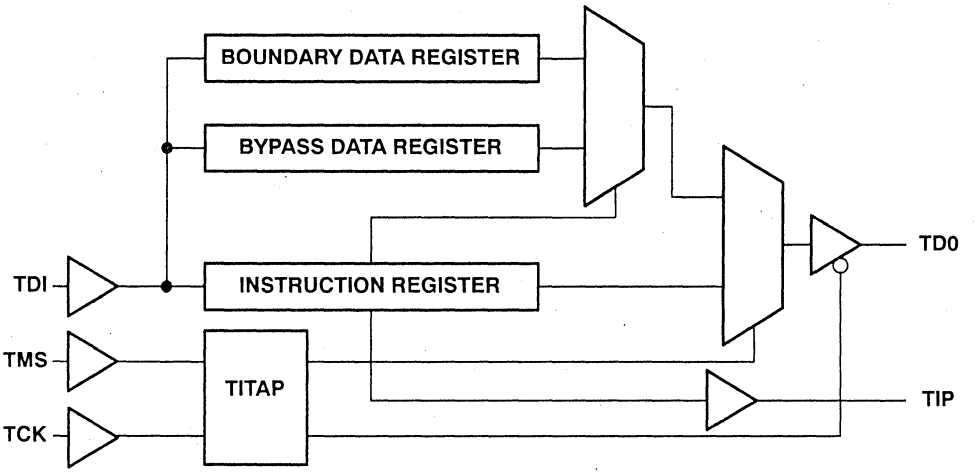
JTAG/IEEE-1149.1 Scan Interface

Integrated circuit and board-level testing is increasingly important. JTAG or IEEE-1149.1 is a standard test methodology. It is based on a 4-wire connection to a device and provides access to all I/O buffers (boundary scan) of a device. This permits stimulation and observation of internal logic. By allowing stimulation of output pins and observation of input pins, external circuitry can also be tested. If implemented completely, this can eliminate "bed of nails" test rigs.

The TMS320C30-IEEE Floating-Point Format Converter is equipped with a JTAG/ IEEE-1149.1 compatible scan interface. The internal architecture is based on Texas Instruments' SCOPEtm design specifications. This provides for boundary-scanning of the device and inclusion of an eight-bit instruction register.

Figure 21 shows the internal scan architecture and gives the naming conventions used to describe the device blocks:

Figure 21. Scan Architecture



I/O Pin Description

TCK

The TCK input clock signal is the scan clock. It typically will be generated off-board by a test controller. All tests of the device are controlled by an external controller and proceed at the scan clock (TCK) speed.

TMS

The TMS input signal is clocked in by TCK. TMS controls the test mode of the device. Using TMS and TCK, a test controller can scan registers through the device, perform tests, or place the device in a normal functional mode.

TDI

The TDI input signal is used to input serial data through the registers in the device. All data is clocked in by TCK and shifts according to the state of the test logic set up by an external test controller using TMS and TCK.

TDO

The TDO output signal is used to scan serial test data out of the device under the control of the test host. While shifting data, TDO is active-shifting data out on the falling edge of TCK. When through shifting data, TDO is tri-stated.

TIP

TIP is an output indicating good or bad parity in the instruction register. The indication defaults to good if the external controller does not check for parity. To check parity, the test controller places the device in the instruction register pause state. While in this state, the device will output the actual (i.e., hardware-determined) parity of the device's instruction register. A high logic level indicates good parity, while a low logic level indicates bad parity.

Architectural Elements

TITAP

The Texas Instruments' Test Access Port (TITAP) is a 16-state state-machine designed according to the JTAG and IEEE-1149.1 specifications. The TITAP controls the test logic and is controlled by the TMS and TCK inputs to the device from an external test host controller.

Instruction Register

The Instruction Register is eight bits in length. Table 10 lists the instructions available for this device.

Table 10. Test Instructions

msb –> lsb	Instruction
00000000	Boundary Scan
10000001	ID Register Scan
10000010	Sample Boundary Scan
00000011	Boundary Scan
00000110	Control Boundary HI-Z
10000111	Control Boundary 1/0
00001010	Read Boundary-Normal
10001011	Read Boundary-Test
00001100	Boundary Selftest
11111111	Bypass Scan
All Others	Bypass Scan

The Instruction Register is preloaded with 00000001 (msb–lsb) in the instruction register capture state of the TITAP. This is not per the JTAG/IEEE–1148.1 standards.

Boundary Scan Instruction

This instruction places the device in test mode: all function inputs and outputs are controlled by the test logic. Function inputs and outputs are sampled in the data register capture state of the TITAP, and the boundary data register is selected in the data register scan path during data register scans.

ID Register Scan Instruction

This instruction places the device in normal mode: all function inputs and outputs operate in their normal modes. The bypass data register is selected in the data register scan path during data register scans.

Sample Boundary Scan Instruction

This instruction places the device in normal mode: all function inputs and outputs operate in their normal modes. Function inputs and outputs are sampled in the data register capture state of the TITAP, and the boundary data register is selected in the data register scan path during data register scans.

Control Boundary HI-Z Instruction

This instruction places the device in test mode: all function outputs are tri-stated (if possible), while all function inputs operate in their normal mode. The bypass data register is selected in the data register scan path during data register scans.

Control Boundary 1/0 Instruction

This instruction places the device in test mode: all function inputs and outputs are controlled by the test logic. The bypass data register is selected in the data register scan path during data register scans.

Read Boundary – Normal Instruction

This instruction places the device in normal mode: all function inputs and outputs operate in their normal modes. The boundary data register retains its current state in the data register capture state of the TITAP, and the boundary data register is selected in the data register scan path during data register scans.

Read Boundary – Test Instruction

This instruction places the device in test mode: all function inputs and outputs are controlled by the test logic. The boundary data register retains its current state in the data register capture state of the TITAP, and the boundary data register is selected in the data register scan path during data register scans.

Boundary Self–Test Instruction

This instruction places the device in normal mode: all function inputs and outputs operate in their normal modes. The boundary data register contents are toggled, and the data register captures the state of the TITAP. Also, the boundary data register is selected in the data register scan path during data register scans.

Bypass Scan Instruction

This instruction places the device in normal mode: all function inputs and outputs operate in their normal modes. The bypass data register is selected in the data register scan path during data register scans.

Boundary Data Register

The boundary data register contains 70 bits and is ordered according to Figure 22.

Figure 22. Scan Path Bit Order

TDI —> DIR —> PIPE —> CLK —> OEZ —> NAN —> WAIT —>
DA31 —> DA30 —> ... —> DA1 —> DA0 —>
DB31 —> DB30 —> ... —> DB1 —> DB0 ————————> TDO

Bypass Data Register

The Bypass Data Register is one bit in length and is operated in accordance with the JTAG/ IEEE-1149.1 specifications.

Scan References

Refer to the following documents for further descriptions of the test logic of this device:

1) A Test Access Port and Boundary Scan Architecture; Technical Sub-Committee of the Joint Test Action Group (JTAG).
2) IEEE Standard 1149.1 – IEEE Standard Test Access Port and Boundary-Scan Architecture.

Part IV. Telecommunications

Implementation of a CELP Speech Coder for the TMS320C30 Using SPOX

Mark D. Grosen

Spectron Microsystems, Inc.

Introduction

Speech coders are critical to many speech transmission and store-and-forward systems. With the emergence of universal standards, it is possible to develop systems that are interoperable. Quality and bit rate for speech coders vary from toll quality at 32 kilobits/second (kbps) (CCITT ADPCM) to intelligible quality at 2.4 kbps (DOD LPC-10). Recently, a new standard for 4.8 kbps with near toll-quality has been proposed and is based on code-excited linear prediction (CELP) techniques [1,2]. Unfortunately, products based on new coding algorithms are often slow to appear because of the considerable time and effort required to develop real-time implementations.

The purpose of this article is to demonstrate how a CELP coder based on this new standard can be quickly developed using SPOX. Utilizing the power of the TMS320C30 DSP plus the ease of use provided by C and the SPOX DSP library, an efficient and portable coder can be written in a much shorter period of time than that required by conventional assembly language methods. Because of the portability of SPOX and C, the coder can also be compiled and executed on a variety of hardware platforms.

A 4.8-kbps CELP Coder

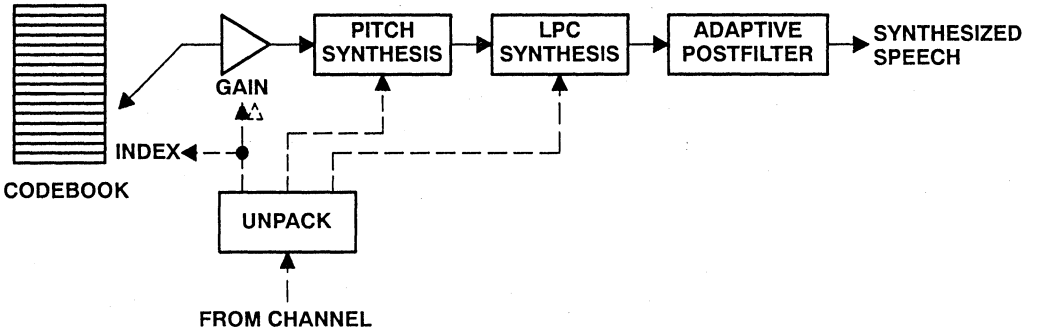
CELP coders were first introduced by Atal and Schroeder in 1984 [3]. These coders offer high quality at low bit rates, but at a high computational cost. Implementing the original systems directly required several hundred million instructions per second (MIPS). Much of the research on CELP techniques has concentrated on reducing this computational load to facilitate real-time implementations.

The proposed U. S. Federal Standard 4.8-kbps CELP coder (USFS CELP), Version 2.3, uses several techniques to reduce the complexity to a level where a one- or two-processor implementation is possible. These are the main characteristics of the coder:

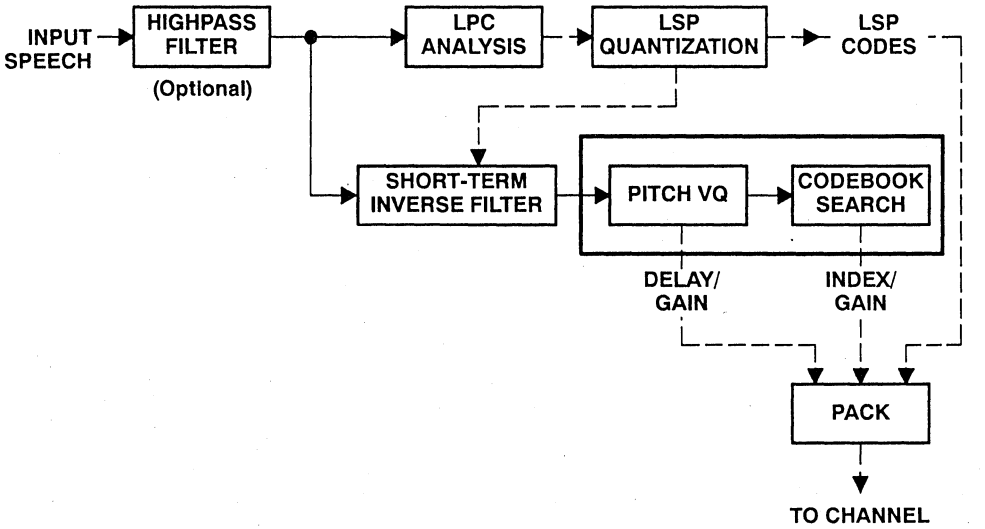
- 240-sample frame size at 8-kHz sampling rate
- Tenth-order short-term predictor
  - Calculated once per frame, open loop
  - Autocorrelation with Hamming window
  - LSP quantization
- Four subframes (60 samples)
  - One tap pitch predictor
    1) Closed loop analysis
    2) Even/odd subframe delta search method
  - 1024-element codebook
    1) Overlapped by 2 (see Pitch and Codebook Search)
    2) 75% of elements are zero

Block diagrams of the decoder and encoder are shown in Figure 1.

Figure 1. USFS CELP Decoder and Encoder Structures



DECODER



ENCODER

Bit allocations are given in Table 1 [2,4].

Table 1. 4.8-kbps CELP Parameters

	Spectrum	Pitch	Codebook
Update Parameters Bps	30 ms (240 samples) 10 LSP 1133.3	7.5 ms (60) 1 delay, 1 gain 1466.7	7.5 ms (60) 1 of 1024 index, 1 gain 2000
Remaining 200 bps reserved for expansion, error protection, and synchronization			

The standard also specifies an error protection scheme utilizing forward error-correcting Hamming code and parameter smoothing.

The major computational parts of the algorithm are the pitch search and the codebook search, both of which are performed four times per frame. An important technique to reduce the computations is the end-correction convolution technique (see Pitch and Codebook Search). This is a recursive convolution method that reduces the number of multiply-adds by an order of magnitude.

In addition, the codebook is designed to have approximately 75% of the samples equal to zero. This allows many of the convolution updates in the codebook search to be reduced to a simple shift of a vector of samples. On DSP processors with circular addressing, this shift can be replaced by using circular buffers.

To further reduce complexity, the pitch search is limited in range for every other subframe. During even-numbered subframes, the optimal pitch value is performed over the range 20 to 147 (128 values). On the odd subframes, the search is only over the range 16 from the previous pitch value. This also decreases the bit rate with a negligible effect on speech quality.

If adequate processing power is not available, you can implement an interoperable coder by using a subset of the full codebook. For example, if only the first 128 vectors from the codebook could be used, the sub-optimal coder would work with an optimal coder if the same frame structure and bit rate were used.

These techniques produce complexity estimates for the USFS CELP coder ranging from 5.3 MIPS to 16.0 MIPS for a 128-vector and 1024-vector codebook, respectively[4].

Using SPOX in Development

The computational complexity of CELP coders, even with use of the various techniques to reduce it, has made real-time implementations impractical on first- and second-generation DSPs. The recent introduction of the third-generation TMS320C30[5], however, makes it feasible to implement the USFS CELP coder with one or two processors. Furthermore, because of the general-purpose capabilities of the TMS320C30 and the availability of a C compiler and SPOX, development of a real-time coder can be significantly expedited.

In particular, SPOX provides the following functions to facilitate software development.

- C standard I/O functions
  - **printf( ), scanf( )**
  - **fopen( ), fread( ), fwrite( )**
- Stream I/O to move data efficiently
- Standard set of DSP math functions
  - Filters
  - Vector operations
  - Windows
  - Levinson-Durbin algorithm
- Processor independence

Both FORTRAN and C versions of the Version 2.3 USFS CELP coder were available as starting points for the real-time implementation. The initial development was done on a Sun worksta-

tion equipped with SPOX/SUN [6] and the usual UNIX programming tools, such as the symbolic debugger dbx. SPOX/SUN is a library of SPOX DSP math functions that can be used for developing SPOX applications on Sun workstations. The new version of the coder utilizing SPOX was checked against the existing implementation for correctness. After the new version was debugged on the workstation, the source code was recompiled employing the Texas Instruments TMS320C30 C compiler and linked with the SPOX/XDS library for the XDS1000 development system.

The same facilities for testing the code on the workstation were available on the XDS1000. A SPOX stream function (see Input/Output section) read digitized speech from a disk file. Status information was printed to the console screen. Command line arguments were used to vary the encoder's parameters such as the codebook size.

The software development process for the USFS CELP coder followed three evolutionary steps:

- C program using standard I/O
- C program using SPOX functions for faster math and I/O
- C program using SPOX and assembly language optimizations

The first step was taken because an existing C implementation was available. The C standard I/O provided by SPOX made it possible to run the application code written in C directly on the XDS1000. For example, functions (**fscanf**()) that read control information from a disk file on the Sun also worked on the XDS1000 using the PC's hard disk.

In general, it would have been easier to start with the SPOX library functions to implement some of the common operations contained in the coder. Many of the functions needed (filtering, correlation, dot-product) are in the SPOX DSP library. In this case, the C implementations of these standard vector and filter functions in the existing program were replaced with the corresponding SPOX functions. The SPOX functions, written in optimized assembly language, execute several times faster than the corresponding C functions.

The last step was needed to meet real-time constraints. XDS1000 timing capabilities allowed the identification of two time-critical sections of the code which were then rewritten in TMS320C30 assembly code. Since the interface to the SPOX math functions is open, new math functions can be written that work with SPOX data structures such as vectors and filters.

Implementation

Several major parts of the USFS CELP encoder are implemented with a mixture of C, SPOX, and TMS320C30 assembly language functions. The decoder can be easily constructed from the material presented here. An adaptive postfilter for the decoder is not described here.

The framework of the resulting encoder is shown in Figure 2. A description of the major functions performed can be found in the following sections. Appendix A provides a short summary of the SPOX functions employed in the next four sections (Input/Output, Spectrum Analysis, Filters, and Pitch and Codebook Search).

Figure 2. Structure of the Encoder Function

```
encoder(instream, outstream)
    SS_Stream    instream;
    SS_Stream    outstream;
{

    while ( SS_get(instream, SV_array(speech)) ) {

    /* Apply a high pass filter to the input speech */
        SF_apply(hpfilter, speech, speech);

    /* Find the coefficients of the short-term prediction filter */
        calculateLP(speech, invcoeffs);

    /*
     * Convert the direct form coefficients to line spectrum pairs.
     * Then quantize the LSP's and convert back to direct form.
     */
        SV_a2lsp(invcoeffs, lsps);
        quantizeLSP(lsps, qntzlsps);
        SV_lsp2a(qntzlsps, invcoeffs);

    /*
     * For each of the 4 subframes, determine the pitch prediction
     * parameters and codebook (excitation) parameters
     */
        for (i = 0; i < 4; i++) {

            genShortResidual(s[i], res[i]);/* generate short term residual */
            pitchSearch(s[i], res[i]);     /* find optimum pitch predictor */
            genFullResidual(s[i], res[i]); /* generate residual */
            codeSearch(res[i], reshat);    /* find best codebook vector */
            updateFilters(reshat);         /* update filter states */
        }
        packParams();   /* pack parameters into output array */
        SS_put(outstream, params);
    }
}
```

Input/Output

Input speech samples are obtained by employing a function (**SS_get( )**), which reads data from a named stream (**instream**). The creation of instream during program initialization determines the source of the data. During development, the easiest source is a disk file with digitized speech. When real-time testing is needed, a codec connected to a TMS320C30 serial port could be utilized. For example, instream could be created to read from standard input with the following code segment.

```
#define FRAMESIZE        240 * sizeof(Float)

instream = SS_create(DF_FILE, DF_STDIN, FRAMESIZE, NULL);
```

The output stream (**outstream**) consists of the packed frame parameters. It could also go to a disk file or a serial port by using **SS_put( )**.

Spectrum Analysis

After preconditioning the signal with a highpass filter (see the Filters section), the coefficients of the short term prediction filter can be found by using the function **calculateLP( )** shown below.

```
SV_Vector           window, rc, error, cor, gammavec;

calculateLP(s, coeffs)
    SV_Vector       s, coeffs;
{
    SV_window(s, window, s);                /* window the speech in-place */
    SV_corr(s, s, cor);                     /* autocorrelation */
    SV_autorc(cor, coeffs, rc, error);      /* Levinson-Durbin */
    SV_mul2(gammavec, coeffs);              /* bandwidth expansion */
}
```

The vector window is initialized to contain the desired window; in this case, a Hamming window is used. The autocorrelation terms are stored in the vector **cor** that has the same length as the order of the short term filter. **SV_autorc( )** uses a Levinson-Durbin type algorithm to compute the inverse filter coefficients. As a side effect, the reflection coefficients are also stored in **rc**. Finally, a 15-Hz bandwidth expansion is produced by the multiplication of the inverse filter coefficient vector by a vector (**gammavec**) consisting of the terms

$$g[i] = 0.994^i \qquad \text{for } i = 0, 1, \ldots, m-1$$

Efficient quantization is obtained by:

- Transforming the prediction coefficients into line spectrum pairs (LSPs)
- Then quantizing the LSPs

The conversions between prediction coefficients and LSPs are not currently in the SPOX library. The existing C implementation evaluates cosine values directly, which is too expensive computationally. A more efficient routine (**SV_a2lsp( )**), that employs table-lookup of cosine values, has been written utilizing the algorithm outlined in [7]. The quantized LSPs are transformed back to direct-form coefficients for use in the short-term predictor.

Filters

Three filters in the encoder can be realized by use of SPOX filter objects. The inverse filter $A(z)$ and the short term predictor $1/A(z)$ share the same filter coefficients. The former is an FIR filter and the latter an all-pole filter. The final filter is the all-pole weighting filter $W(z)$ with coefficients given by $1/A(\lambda z)$, with $\lambda = 0.8$.

During the initialization of the encoder, the filters are created with the code fragment shown below.

```
#define FILTERSIZE    11 * sizeof(Float)

    SF_Filter         invfilter, predfilter, wgtfilter;
    SV_Vector         invcoeffs, wgtcoeffs;
    SA_Array          array;

     array = SA_create(SG_CHIP, FILTERSIZE, NULL);
     invfilter = SF_create(array, NULL, NULL);
     SF_bind(invfilter, invcoeffs, NULL);

     array = SA_create(SG_CHIP, FILTERSIZE, NULL);
     predfilter = SF_create(NULL, array, NULL);
     SF_bind(predfilter, NULL, invcoeffs);
```

```
array = SA_create(SG_CHIP, FILTERSIZE, NULL);
wgtfilter = SF_create(NULL, array, NULL);
SF_bind(invfilter, NULL, wgtcoeffs);
```

Note that the inverse and prediction filters are both bound to the same coefficient vector. For each new frame of speech, this vector is updated when it is passed to **calculateLP( )**.

An important consideration is that the filters are used more than once during a frame. A different signal is filtered each time, but the state (history) of the filter must be the same. This is accomplished before each filter operation by using the

- **SF_getstate( )** function to recover a vector with the state of the filter at the end of the previous frame
- **SF_setstate( )** function to restore the filter's state

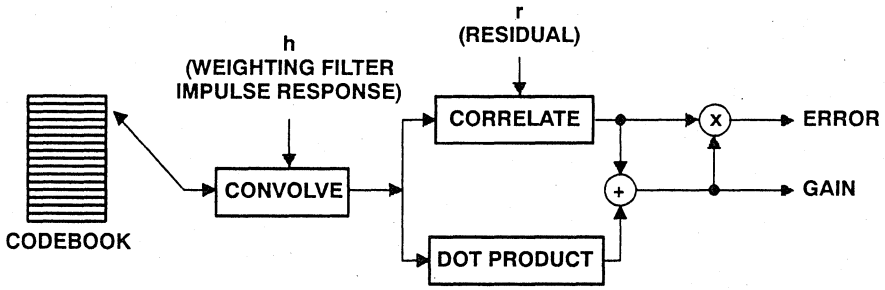
The following code segment shows how the short term prediction residual is generated for the pitch search.

```
SF_setstate(predfilter, NULL, predstate);
SV_fill(residual, 0.0);
SF_apply(predfilter, residual, residual); /* zero input of filter */

SV_sub3(residual, speech, residual);      /* speech – history */

SF_setstate(invfilter, invstate, NULL);
SF_apply(invfilter, residual, residual);  /* filter with inverse */

SF_setstate(wgtfilter, NULL, wgtstate);
SF_apply(wgtfilter, residual, residual);  /* filter with weighting */
```

Pitch and Codebook Search

After the program finds the short-term predictor and generates the corresponding residual, the pitch predictor and code book parameters are found for each of the four subframes. The pitch and codebook search functions are similar: both search over a set of values to minimize an error term. In this section, only the codebook search is illustrated (see Figure 3). Many of the functions, however, can be applied to the pitch predictor calculations.

Figure 3. Codebook Search Block Diagram



The search in Figure 3 minimizes the distance between the input vector and one of many generated vectors. The quantity being minimized is the Euclidean norm:

$$e = \| r - \hat{r} \|^2 = r^t r - 2\, r^t \hat{r} + \hat{r}^t \hat{r} \quad (1)$$

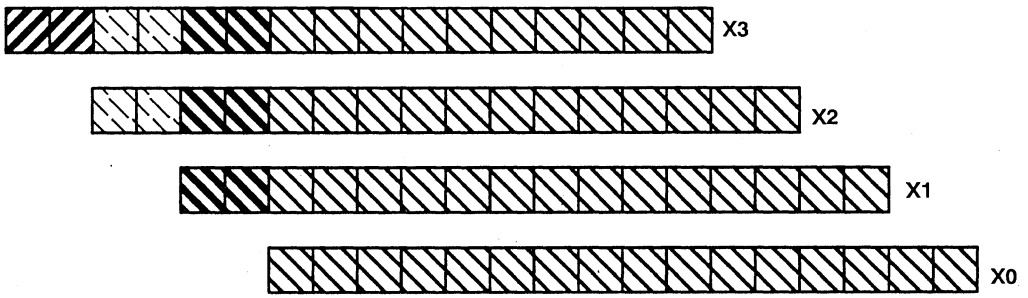
where

r = the original residual
$\hat{r}$ = the synthesized residual

It can be seen from the vector definition that only two terms need to be computed – the correlation of r and $\hat{r}$ and the energy of $\hat{r}$; this is because the energy of the original residual is invariant over all the generated residuals. It appears that there would be N convolutions and 2N dot products to perform for each sub-frame. Implemented directly, the codebook search would thus require 66 MIPS if N = 256 and a sub-frame length of 60 are specified.

Instead, the USFS CELP coder uses a specially structured codebook that greatly reduces the computational load. The biggest savings comes from the elimination of all but one of the convolutions for each subframe. The codebook is overlapped, as shown in Figure 4.

Figure 4. Structure of Overlapped Codebook



This structure permits a recursive convolution computation. The first codebook vector is convolved normally with the weighting filter. Subsequent convolutions, however, make use of the following relationships.

$$V_{i+1}(z) = z^{-1}\hat{R}_i(z) + x_{i+1}[1]H(z)$$
$$\hat{R}i + 1(z) = z^{-1}V_{i+1}(z) + x_{i+1}[0]H(z) \quad (2)$$

where $\hat{R}_i(z)$ is the Z-transform of the generated residual. Given the convolution of the previous codebook vector with the weighting filter, the convolution employing the next vector can be found with only 120 (2 × 60) multiplies and adds.

This number can be further reduced by another property of the codebook. The vectors are generated by center-clipping a gaussian noise source, which causes approximately 75% of the elements to be zero. Thus, 75% of the updates to the convolutions require no multiplications or additions; however, the convolution elements must still be shifted. The following function **update( )** implements the recursive update operation. Note that it must be called twice per codebook vector, once for each new term.

```
update(x, res, wgtimpulse)
    Float       x;
    SV_Vector   res, wgtimpulse;
{
    Float       *rptr, *rptrm1, *wptr;
    Int         len;

    len = SV_getlength(res);
    rptr = (Float *) SV_loc(res, len - 1);
    rptrm1 = rptr - 1;

    if ( x == 0.0 ) {                               /* no input, so just shift */
        for (; len > 1; len--) {
            *rptr-- = *rptrm1--;
        }
        *rptr = 0.0;
    }
    else {                                          /* update using new input */
        wptr = (Float *) SV_loc(wgtimpulse, len - 1);
        for (; len > 1; len--) {
            *rptr-- = *rptrm1-- + x * *wptr--;
        }
        *rptr = x * *wptr;
    }
}
```

Once the convolution has been determined, the corresponding error and gain can be found.

The following function calculates the error and gain terms.

```
Float error(res, reshat, gain)
    SV_Vector   res, reshat;
    Float       *gain;
{
    Float       cor, energy;

    SV_dotp(reshat, reshat, &energy);
    SV_dotp(reshat, res, &cor);
    *gain = cor / energy;
    return( *gain * cor );
}
```

The codebook search function with **update( )** and **error( )** functions is shown below. The first convolution must be calculated directly, so it is done outside of the main **for** loop. The error for each entry is compared against the current maximum; if it is greater than the maximum, this entry becomes the new best vector. The process is repeated for each of the *N* vectors.

```
SV_Vector       codebook, wgtimpulse;

codeSearch(res, reshat)
    SV_Vector res, reshat;
{
    Float       errmax, gain, err;
    Float       *cbptr;
    Int         i, best;

    findImpulse(wgtimpulse);

    SV_setbase(codebook, FIRSTVEC);

    convolve(codebook, wgtimpulse, reshat);
    errmax = error(res, reshat, &gain);
```

```
    best = 0;
    cbptr = (Float *) SV_loc(codebook, 0) - 1;

    for (i = 1; i < N; i++) {
        update(*cbptr--, reshat, wgtimpulse);
        update(*cbptr--, reshat, wgtimpulse);
        if ( (err = error(res, reshat, &gain)) > errmax ) {
            errmax = err;
            best = i;
        }
    }
}
```

After the search is completed, the gain of the best vector is recomputed and quantized. The corresponding gain index and index of the codebook element can then be readied for transmission.

Assembly Language Enhancements

The codebook and pitch searches require the largest share of the computation cycles in the encoder. One way to increase performance is to recode critical parts of these functions in assembly language. One such function is the **update( )** function described above for the recursive convolution computation.

An assembly language version of **update( )** was written to take advantage of the parallel instructions and repeat block capabilities of the TMS320C30. The assembly language function utilizes the same calling structure as the C version. The function was written using the assembly language macros provided with SPOX to work with the vector, matrix, and filter objects in the DSP library[8]. The new version of **update( )** is listed in Figure 5.

Figure 5. Update Function Written in TMS320C30 Assembly Language

```
*
* Synopsis:
*
*       Void update(x, res, wgtimpulse)
*           Float       x;
*           SV_Vector   res, wgtimpulse;
*
#include <sv30.h>

FP      .set    ar3
        .global _update
        .text

_update:
        push    FP
        ldi     sp, FP
*
*       Set the following registers by using vector object macros
*           ar0 - SV_loc(wgtimpulse, 0)
*           ar1 - SV_loc(res, 0)
*           rc  - the length of the vectors
*           r2  - x
*
        ldi     *-FP(2), ar2
        SV_get1 ar2, SV_LOC0, ar0
        ldi     *-FP(3), ar2
        SV_get2 ar2, SV_LEN|SV_LOC0, rc, ar1
*
        ldf     *-FP(4), r1                     ; x
        bzd     shift                           ; x is 0 so just shift
        subi    1, rc
        addi    rc, ar1                         ; ar1 -> res[l - 1]
        ldi     ar1, ar2                        ; ar2 -> res[i - 1]
*
*  General case when x != 0.0
*
        addi    rc, ar0                         ; ar0 -> wgt[l - 1]
        subi    2, rc                           ; set loop count

        mpyf    r1, *ar0--, r2                  ; x * wgt[i]
        addf    r2, *--ar2, r0

      rptb    lp20
        mpyf    r1, *ar0--, r2                  ; x * wgt[i]
lp20:   addf    r2, *--ar2, r0
||      stf     r0, *ar1--

         bud     end
         stf     r0, *ar1--
         mpyf    r1, *ar0, r0                        ; res[0] = x*wgt[0]
         stf     r0, *ar1
*
*  Case for x == 0.0
*
shift:   subi    2, rc                               ; loop l - 1 times
         ldf     *--ar2, r0                          ; prime the pipe

        rptb    slp
slp:    ldf     *--ar2, r0
||      stf     r0, *ar1--

        stf     r0, *ar1--                      ; final store
        ldf     0.0, r0                         ; first term = 0.0
        stf     r0, *ar1
*
end:    pop     FP
        rets
```

Performance

A complete CELP encoder was implemented as described above. Two versions were tested:

- One encompassing C and standard SPOX functions
- One having C, SPOX, and two custom TMS320C30 assembly language functions

Table 2 shows the execution times for different combinations of codebook size, processor, and implementation. To achieve near real-time performance for a codebook with 128 vectors, the codebook and pitch search functions were completely rewritten in assembly language. Each function required approximately 130 lines of assembly code.

Table 2. Timing of Various Implementations of the CELP Encoder for One Frame of Speech

Codebook Size	Sun (C/SPOX)	C30 (C/SPOX)	C30 (C/SPOX/ASM)
128	16,000 ms	88.2 ms	39.0 ms
256	24,000 ms	114.6 ms	54.3 ms

Memory requirements for the program on the TMS320C30 were approximately 14,000 words for instructions and approximately 6,000 words for data. The application code required approximately 4500 words of instructions. The SPOX operating system and DSP math functions consumed the remaining 9500 words of memory. This figure reflects many functions that are essential for easing development but unnecessary for a real-time implementation.

Once a real-time implementation has been achieved, the SPOX memory requirements can be greatly reduced by porting (or customizing) SPOX to a custom hardware implementation. In this case, the SPOX memory requirements can be reduced to approximately 4000 words, making a 12K-word implementation feasible (both data and instruction memory requirements).

These timings show that a real-time CELP coder can be implemented on a single TMS320C30. They also illustrate the power of the TMS320C30 compared to a standard microprocessor. Note that a TMS320C30 implementation has approximately 500,000 instruction cycles available in a 30-ms frame.

Version 3.0 of the USFS CELP coder has significant improvements in computational complexity, including:

- Ternary codebook to eliminate multiplications
- Shorter codebook
- Faster LSP conversion and quantization

Work to bring the SPOX implementation up to Version 3.0 is continuing. An investigation of a two-processor implementation is also being performed.

Summary

A 4.8-kbps CELP coder based on a Department of Defense-proposed standard has been implemented on a TMS320C30. Several of the functions used in the encoder were illustrated. A suboptimal implementation of the encoder using a 128-vector codebook is possible on only one TMS320C30. Work is continuing on both the algorithm and the software implementation to improve the coder's real-time performance.

With SPOX, the encoder was developed in less than one month. The resulting source (with the exception of two TMS320C30 assembly language functions) can be compiled and run on a Sun workstation, a PC, or a TMS320C30 system such as the Texas Instruments XDS1000. This represents a considerable improvement in development time and effort over previous implementation methods.

References

1) Kemp, D.P., Sueda, R. A., and Tremain, T. E., "An Evaluation of 4800 bps Voice Coders," *Proceedings of ICASSP '89*, IEEE, May 1989.
2) Campbell, J. P., Welch, V. C., and Tremain, T. E., "An Expandable Error-Protected 4800 bps CELP Coder," *Proceedings of ICASSP '89*, IEEE, May 1989.
3) Atal, B. S., and Schroeder, M. R., "Stochastic Coding of Speech at Very Low Bit Rates," *Proceedings of ICC '84*, pages 1610–1613, 1984.
4) Tremain, T. E., Campbell, J. P., and Welch, V. C., "A 4.8 kbps Code Excited Linear Predictive Coder," *Proceedings of Mobile Satellite Conference*, pages 491-496, May 1988.
5) Texas Instruments, Inc., *Third-Generation TMS320 User's Guide*, 1988.
6) Spectron MicroSystems, Inc., *SPOX/SUN User's Guide*, April 1989.
7) Soong, F. K., and Juang, B. H., "Line Spectrum Pair (LSP) and Speech Data Compression," *Proceedings of ICASSP '84*, pages 1.10.1-1.10.4, IEEE, 1984.
8) Spectron MicroSystems, Inc., *Adding Math Functions to SPOX*, March 1989.

Appendix A

The SPOX functions used in the code examples are briefly described below. Complete descriptions can be found in *Getting Started With SPOX* and the *SPOX Programming Reference Manual*. These manuals are supplied with the XDS1000. They are also available from Spectron Micro-Systems, Inc.

Stream Functions

```
SS_get – get data from a stream into an array

          Int SS_get(stream, array)
              SS_Stream   stream;
              SA_Array    array;

SS_put – put data from an array to a stream

          Int SS_put(stream, array)
              SS_Stream  stream;
              SA_Array   array;
```

Vector Functions

```
SV_autorc – perform inverse filter calculations

          Void SV_autorc(cor, inv, rc, alpha)
              SV_Vector    cor;
              SV_Vector    inv;
              SV_Vector    rc;
              SV_Vector    alpha;

SV_corr – calculate correlation of two vectors

          SV_Vector SV_corr(src1, src2, dst)
              SV_Vector    src1;
              SV_Vector    src2;
              SV_Vector    dst;

SV_dotp – calculate the dot product of two vectors

          SV_Vector SV_corr(src1, src2, result)
              SV_Vector    src1;
              SV_Vector    src2;
              Float        *result;

SV_fill – fill a vector with a value

          SV_Vector SV_fill(vector, value)
              SV_Vector    vector;
              Float        value;

SV_getlength – return the length of a vector

          Int SV_getlength(vector)
              SV_Vector    vector;

SV_loc – return the address of a vector element
```

```
Ptr SV_loc(vector, num)
    SV_Vector    vector;
    Int          num;
```

SV_mul2 – multiply elements of two vectors

```
SV_Vector SV_mul2(src, dst)
    SV_Vector    src;
    SV_Vector    dst;
```

SV_setbase – set the base of a vector

```
Void SV_setbase(vector, base)
    SV_Vector    vector;
    Int          base;
```

SV_sub3 – subtract elements of two vectors and store results in a third vector

```
SV_Vector SV_sub3(src1, src2, dst)
    SV_Vector    src1;
    SV_Vector    src2;
    SV_Vector    dst;
```

SV_window – apply a symmetric window to a vector

```
SV_Vector SV_window(src, wnd, dst)
    SV_Vector src;
    SV_Vector wnd;
    SV_Vector dst;
```

Filter Functions

SF_apply – apply a filter to a vector

```
SV_Vector SF_apply(filter, input, output)
    SF_Filter    filter;
    SV_Vector    input;
    SV_Vector    output;
```

SF_bind – bind coefficient vectors to a filter

```
Void SF_bind(filter, num, den)
    SF_Filter    filter;
    SV_Vector    num;
    SV_Vector    den;
```

SF_getstate – copy filter state arrays into vectors

```
Void SF_getstate(filter, hisinv, hisoutv)
    SF_Filter    filter;
    SV_Vector    hisinv;
    SV_Vector    hisoutv;
```

SF_setstate – copy vectors into filter state arrays

```
Void SF_setstate(filter, hisinv, hisoutv)
    SF_Filter    filter;
    SV_Vector    hisinv;
    SV_Vector    hisoutv;
```

Part V. Computers

A DSP-Based Three-Dimensional Graphics System

Nat Seshan

Digital Signal Processor Products—Semiconductor Group
Texas Instruments

This application report is based on the author's bachelor's thesis at the Massachusetts Institute of Technology.

The placement of a high-performance computational engine, such as an advanced digital signal processor, between the host processor and the video controller in a graphics system can improve performance tremendously. Several factors make the Texas Instruments TMS320C30 Digital Signal Processor well-suited to this task:

- 32-bit floating point arithmetic provides both high-resolution and large dynamic range in calculation.
- Single-cycle, 60-ns instruction execution and parallel bus access greatly improve system throughput.
- A hardware single-cycle multiplier facilitates the matrix arithmetic, which is frequently required in 3D graphics.
- The ease of programmability allows the design of flexible and expandable systems.
- Software tools, such as simulators[1], assembler/linkers[2], and high- level language debuggers/compilers[3], decrease product development time.
- In-circuit scan-path emulators[4], decrease hardware prototyping and debugging time.
- The use of a standard device lowers the overall system cost.

With the use of the TMS320C30, the host processor can request higher-level commands of the rest of the system. Instead of issuing requests for line-draws or screen clears, it can, for example, request that a 3D object be rotated 90 degrees and then be redrawn. In addition, a rendering element (usually a video controller or graphics system processor) can devote its resources solely to screen management rather than doing some portion of the computationally intensive processing. The following pages provide a description of how a 3D graphics system used the TMS320C30 to compute object transformations.

The digital signal processor resides on the TMS320C30 Application Board (C30AB) designed for the IBM PC/AT or compatible. The PC's 80x86 acts as the host processor and communicates to the C30AB through an 8-bit bus slot. Also resident on the bus is a Texas Instruments TMS34010 Software Development Board (SDB)[5,6]. The SDB contains a TMS34010 Graphics System Processor (GSP) [7], which manages the screen memory and drives the video display. Overall, this system is meant to serve as an instructional model of how a graphics system can be designed using an advanced digital signal processor.

The Potential for Graphics Pipelines

A mechanical engineer for an automobile manufacturer wants to design a robot arm for plant automation. Before building a prototype machine, he wishes to compare the ways in which various designs can pick up and assemble components. To do this,the engineer needs a CAD system capable of creating, storing, and adjusting representations of 3D objects and then rendering the images on a video display. The CAD system has four basic aspects:

1) A user interface for command entry.
2) A data management system to store objects and their screen representations.
3) One or more computational engines to perform high-speed calculations for applications such as transformations, clipping, lighting/shading, and fractal graphics.

4) A rendering engine to control the video memory and to drive the video display.

These four tasks are common to many graphics systems, whether they be intended for CAD/CAM, fractal graphics, heads-up displays in fighter aircraft, or Postscript printer control. If one or more processors are assigned to each function, the resulting pipeline will achieve greatly improved system throughput.

In a single-processor system, the CPU is directly responsible for all computations. It must write to video memory, perform all necessary computations, interface to the user, and manage all data storage and recovery. Although additions to the system, such as a video-memory controller or a floating-point coprocessor, may speed up the system, the CPU remains overly burdened as the only intelligent component of the system.

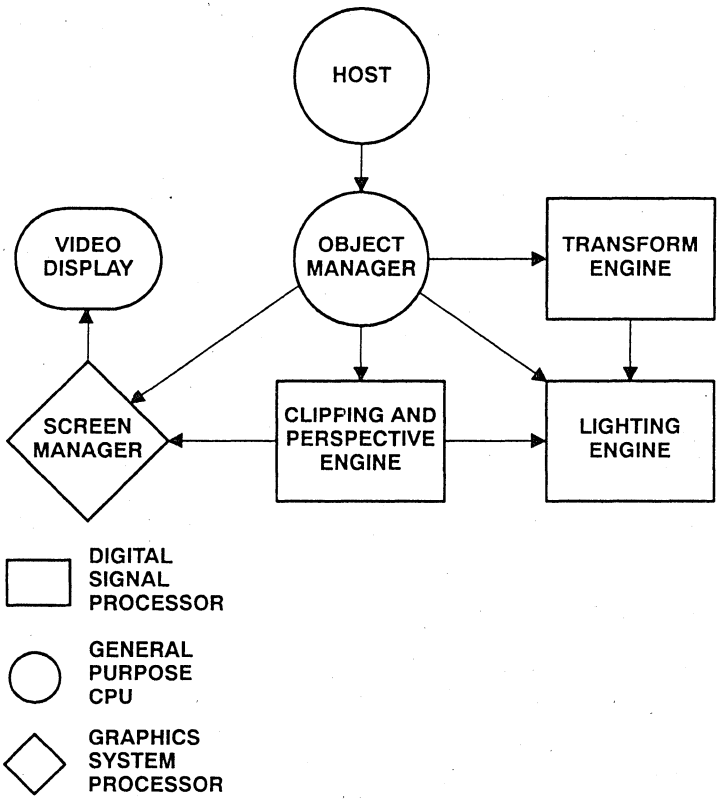
Independent Screen Management

A two-processor system can use a GSP to drive the CRT and to control the video memory. To control the display, the GSP either must interface to an analog monitor through a color palette or must directly drive a digital monitor. If the video memory is volatile, the processor needs a refresh controller that runs in parallel with other processor actions. Special hardware can be developed for screen clears and polygon fills. For flexibility of data representation, the processor should to be able to access pixels of varying bit-widths. At the instruction level, specialized operations could be created to speed pixel processing. Libraries of subroutines for windowing, drawing, and text management enable the rendering engine to execute higher-level commands. Overall, these features allow the CPU to send more powerful directives to the GSP.

A Multiprocessor Pipeline

Adding more links in the graphics pipeline can further relieve the CPU of burdensome tasks. Performance improvements result from each stage being optimized for a particular function. In addition, throughput increases with the number of stages. The pipeline may also contain multiple processors running in parallel at a particular stage to further improve the latency of that stage. Figure 1 shows a full-scale implementation of a graphics pipeline for 3D graphics.

Figure 1. A Full Scale Graphics Pipeline



In a large-scale graphics pipeline, the host processor runs the applications program. The user may be trying to use a CAD program, model the formation of galaxies, animate 3D objects, etc. The host runs these programs at the top level, provides the user interface, and communicates to all I/O devices, including mass storage systems. For numerically intensive applications it may be appropriate to have a digital signal processor as this host. For example, modeling the formation of galaxies requires numerical solutions to systems of differential equations. But even in such a case, it would be reasonable to have a more general-purpose CPU act as a user front end to the digital signal processor.

The purpose of the object manager is to communicate with the host by receiving data and transferring it to other processors in the system. It manages the global representation of all screen parameters and objects. A Reduced Instruction Set Computer (RISC) processor would be well-suited as either the host or the object manager because of its high-performance general-purpose architecture.

Because a DSP has a highly parallel architecture, a fast execution cycle time, an instruction set optimized for numerical processing, and several development tools, it would perform well as any of the computational stages in a graphics pipeline. For example, a DSP could act as a transform manager that calculates the new universal coordinates of globally stored objects according to rota-

tion, translation, and scaling commands from the object manager. Also, the DSP could act as a lighting manager that accepts parameters of environmental lighting settings from the object manager and applies them to the transformed objects. For example, the user may set ambient intensities as well as other sources of varying geometries, intensities, and colors. The lighting manager then applies these light sources to the surfaces of the objects, which may have varying degrees of specular or diffuse reflection, to compute the necessary shading.

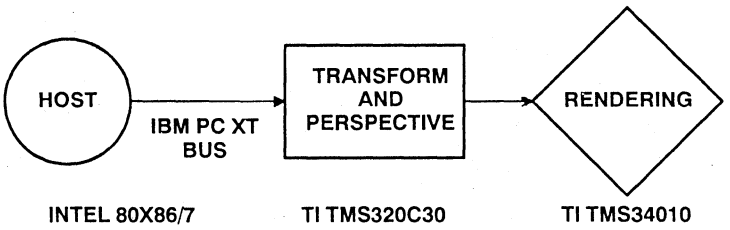
Although the perspective and clipping stage of the system is represented in Figure 1 by a single processing unit, the task may be further partitioned to several DSPs working in series. The perspective calculation takes viewing parameters from the object manager, such as direction of view, location of viewer, and zoom, and produces a two-dimensional projection for the screen. Objects that are too high, too low, or too far right or left can be clipped automatically because the resulting two-dimensional coordinates are off screen. However, clipping objects fully or partially obscured by other objects may require additional stages. Also, objects behind the viewer and those too far away for the user to recognize should be clipped appropriately.

Although digital signal processors are well-suited to be the computational stages of a graphics pipeline, a processor optimized to be a rendering engine might serve better to drive the video display and manage the video memory. Such a processor could also help with the clipping tasks described above. A z-buffer could hold the transformed z-coordinate of each pixel that is projected on to the x-y plane of the screen to facilitate hidden surface removal. A device such the Texas Instruments TMS34010 or the recently introduced TMS34020 could serve as the rendering engine in a full scale system. Both these processors have 32-bit general-purpose architectures with instruction sets and external memory interfaces optimized for graphics.

An Overview of This Implementation

The system shown in Figure 2 is not intended to be a marketable product. Rather, it is targeted toward those who have the intention of designing products in the graphics market. Firms having experience in graphics will be able to resolve the tougher issues of graphics system design without presentation of the described system. The system shown in this report illustrates an attractive option for designing a fast, reliable, portable graphics system with quick turn-around time.

Figure 2. A Simple Three-Processor Graphics Pipeline



One strength of this system is its complete use of standard, commercially available parts. In general, use of standard parts allows for faster design and manufacturing, as well as a more reliable, easier-to-support product. Even the three hardware subsystems can be found on the market:

1) The IBM PC compatible host
2) The TMS320C30 Application Board object manager and transform engine subsystem
3) The TMS34010 Software Development Board rendering subsystem

Another strength of this system is the complete use of portable software. Use of portable software often speeds design times because system software can be mostly debugged before the actual target hardware is available. All software for this system was written in Kernigan and Ritchie C. The command and rendering routine was first debugged on the PC and GSP with the intermediary stage removed. Once debugged, the computationally intensive portion of the software was ported to the DSP, which then assumed control of the GSP. The software on the TMS34010 SDB used many of the graphics routines in the TMS34010 Graphics/Math Library. These routines have been used in many other graphics systems using the TMS34010.

System Hardware

The IBM PC was chosen as the host because of its extensive support by TI development tools. In addition, a large amount of documentation is available concerning interfacing to the PC bus. The system described in this report is designed to run best on an 80386-based IBM PC compatible with an AT power supply and an 80387 floating-point coprocessor. However, either Intel 8086 or 80286 general-purpose microprocessors can also act as the host to the computational engine. The host computer sends commands to

- Load and delete objects
- Target an object for adjustment
- Adjust a particular object
- Recalculate the perspective or
- Redraw the screen.

The 80X87 floating-point coprocessor is not absolutely necessary but greatly improves the time to generate floating-point parameters for the next stage.

This graphics demonstration was the first application developed using the TMS320C30 Application Board (C30AB). Since that time, the C30AB has been included as a part of the XDS1000 emulation system for the TMS320C30 Digital Signal Processor. The TMS320C30's features include

- 60-ns single-cycle execution time (more than 33 MFLOPS)
- 2K x 32-bit dual-access RAM
- 4K x 32-bit dual-access ROM
- 64 x 32-bit instruction cache
- Two 32-bit external memory expansion buses
- Single-cycle floating-point multiply/accumulate
- Two external 32-bit memory ports

- On-chip DMA controller
- Zero-overhead loops and single-cycle branches
- Two on-chip timers and two serial ports
- Floating-point/integer and logical 32/40-bit ALU
- 16M-word memory space
- Register-based CPU
- Development tools, including a simulator, assembler/linker, optimizing C compiler, C-source debugger, and an in-circuit emulator/debugger
- On-chip scan-path emulation logic
- Low-power CMOS technology

The TMS320C30 executes commands from the 80X86 to transform objects, load objects into or delete objects from the system, and compute the projection of 3D objects on the 2D screen. When given a directive to draw the screen, it sends a command to the rendering engine to clear the current screen. Then, the TMS320C30 transfers lists of lines, points, and polygons for the next stage to render.

The TMS34010 Software Development Board (SDB) has been used in TMS34010 development support since 1987. It is configurable for a variety of monitors. The board supports the TMS34010 Graphics/Math Function Library [8] (a library of high-level routines callable from any C program). This board was slightly modified to receive commands from the C30AB as well as from the PC host. Program loaders, C compilers [9], assemblers, and C language standard I/O library support have been developed for this board, as well as for the C30AB. Both cards interface to an IBM PC through an 8-bit slot on the AT bus. The TMS34010 GSP on the SDB is an advanced high-performance CMOS 32-bit microprocessor optimized for graphics display systems. Its key features include:

- 160-ns instruction cycle time
- Fully programmable 32-bit general-purpose processor with a 128M-byte address range
- Pixel processing, X-Y addressing, and window clip/pick built into the instruction set
- Programmable pixel size with 16 boolean and 6 arithmetic pixel processing options (Raster-Ops)
- 31 general purpose 32-bit registers
- 256-byte LRU on-chip instruction cache
- Direct interfacing to both conventional DRAM and multiport video RAM
- Dedicated 8/16-bit host processor interface and HOLD/HLD interface
- Programmable CRT control (HSYNC, VSYNC, BLANK)
- Full line of hardware and software development tools, including a C compiler

The TMS34010 GSP receives commands from the TMS320C30, along with arrays of points, lines, and filled polygons to be drawn. It then uses library routines to render these images on the video display.

System Limitations

The system described here is an instructional system built in a limited development time. Aspects of the system could be optimized for speed and for memory usage. A high-speed 3D graphics system has many features that were not implemented.

This design is non-optimal in several ways. The C routines could be hand-coded to execute faster. A 32-bit host bus interface would allow word-at-a-time data transfers to the TMS320C30. The GSP could be interfaced to faster video memory. At the time of this writing, the TMS34020 second-generation graphics system processor is available. The entire TMS320C30 program could be configured to run from internal memory. Many of these optimizations were not realized because of the limited time available for developing the system.

Many operations that an advanced digital signal processor could easily perform were not designed into this system. These tasks include curved and textured surface generation, lighting, shading, and front and back clipping. For demonstrative purposes, only the endpoint transformation and perspective calculations were implemented.

Similarly, the capabilities of the GSP are clearly underutilized in this pipeline. The GSP is adept at managing multiple windows for display. It can also display text in various fonts. The presented system simply requires that the GSP manage a single graphics-only (no text) window.

Representation of Graphics Elements

Any graphics system must have a method of representing the image to be portrayed on the screen. This method requires a system that is able to store and display primitive elements. These elements could range in complexity from three coordinates describing a point to a set of parametric equations representing an irregular three-dimensional surface. However, simply defining a set of primitive drawing structures does not result in an adequate graphics data representation. The engineer designing the robot does not think of the system as several sheet-metal polygons welded together. He more likely conceives of the arm as a clamp attached to a hand, which, in turn, is attached to an arm, etc. A powerful graphics system must not only describe the primitives to be rendered on the CRT, but also how the primitives are organized or related.

Frames of reference play the central role in the organization of graphics primitives. Any set of graphics primitives rigid with respect to each other can be said to exist in the same, constant frame. When the primitives move, they move as a single unit and remain in the same orientation with respect to each other. In this system, any such set of primitives is called an object. The transformational state of any object is determined by three sets of three parameters each. These sets of the object correspond to the

- Translation
- Scale
- Rotation

Translation of an object within its frame simply amounts to moving all locations in that frame a specified distance along the x-, y-, and z-axes. Thus, each object must hold a set of translation factors, denoted in this system's software by **dx**, **dy**, and **dz** (See Listing 1 in the Appendix). Simi-

larly, **sx**, **sy**, and **sz** determine the scale of an object. These factors determine how many units of the untransformed object's coordinates are represented by one unit of the transformed object's coordinates. The three parameters shown in Appendix Listing 1 that represent all possible orientations of an object (**theta**, **phi**, and **omega**) are described in Table 1.

Table 1. Angles of Rotation

Angle	Axis Rotation is Around	Direction of Positive Rotation	Zero Value
θ	z	x to y	Positive x-axis
ω	x	y to z	Positive y-axis
ϕ	y	z to x	Positive z-axis

The Object Data Structure

Every object contains one or more sets of locations, which are referenced by the drawing primitives within the object. The **locnum** field of the object structure (see Listing 1) represents the number of locations available to be referenced by primitives within the object. This and other array sizes are kept for end points in For/Next-type loops and to allocate the appropriate space for the array contained within an object. Every **location** (see Appendix Listing 2) contains three floating-point numbers representing a coordinate in 3D space: **x**, **y**, and **z**. Their integer x-y locations on screen are also saved: **a**, **b**. To reference a location, a primitive needs only to know the index in the locs array. This allows many primitives to reference the same location.

Three different primitives were implemented to be rendered on the screen:

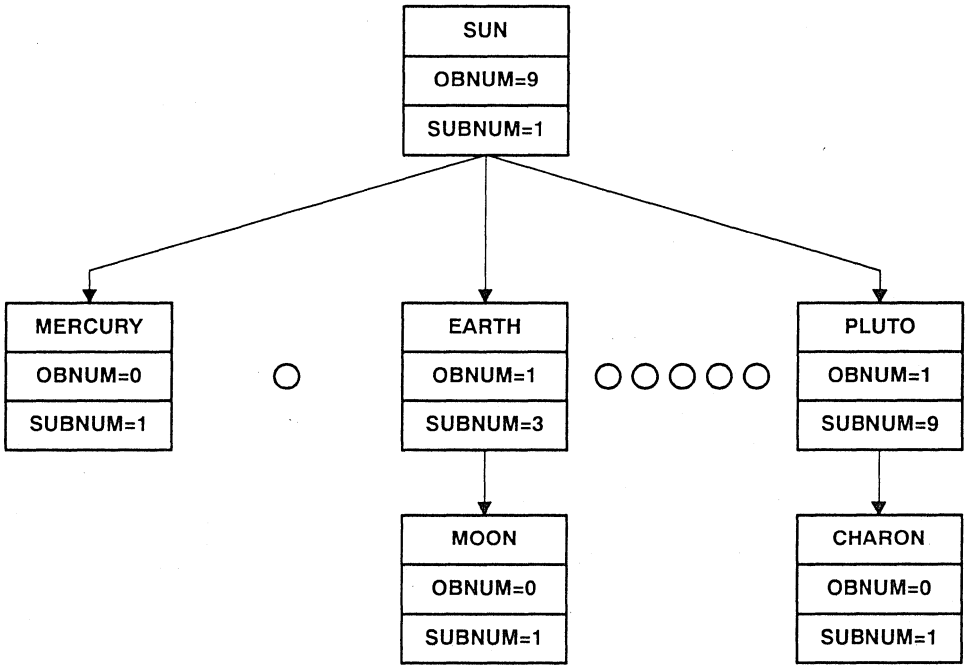
- Points
- Line segments
- Filled polygons

Points are rendered as single pixels on the screen. The **point** structure shown in Listing 3 of the Appendix contains the **color** to draw the point and the index to the location (**locn**) that is referenced by that point. The line structure in Listing 4 of the Appendix contains a **color** and two indices (**startlocn** and **endlocn**) to two end-points of the segment. Finally, the filled polygon shown in Listing 5 of the Appendix contains, in addition to the **color**, the number of vertices (**vertnum**) for the polygon, and a pointer (***vertlocn**) to an array of vertex location indices listed in the order in which they are connected). The last location in the vertex array is connected back to the first, closing the polygon.

Hierarchy

The final array contained within an object (the *parent object*) is a list of pointers to *child objects* defined with respect to the transformed frame of the parent. The number of potential internal objects, **MAXOB**, sets the static size of the array of pointers to child objects. (In this implementation, **MAXOB** = 10.) In addition, the parameter **obnum** keeps track of how many of these potential child objects are utilized. The final bookkeeping parameter is **subnum**. If **subnum** equals *n*, then the object was the *n*th object pointed to in its parent object's child-object array.

Figure 3. Hierarchical Representation of the Solar System



The solar system (Figure 3) represents a classical example of a hierarchical structure. The sun slowly revolves around the galaxy. Wherever the sun travels, the planets follow in the same frame. In turn, each planet may have satellites that revolve around them. The planet is defined with a certain offset (radius of orbit) from the sun, and the satellite is defined similarly with an offset from the planet. To describe the movement of the earth over a period of time, you need only to adjust for its revolution around the sun and the revolution of the moon around the earth. You do not need to describe the rotation of the moon around the sun because when a planet is moved, its satellites automatically move with it.

Transformation parameters are referenced to the frame of the object's parent. Thus, to fully describe a planet orbiting the sun, one must define an empty frame revolving about the sun at some offset, and then define a planet within that frame rotating about some axis. The levels of abstraction within this hierarchy give this data representation its power.

The flexibility of the **object** structure permits the system to model the viewer. The viewer is considered to be at the absolute origin of the system. At system initialization, the first object loaded is the universal object ***universe**. An appropriate choice for such an object would be a set of axes. The view is then adjusted by modifications to the parameters of the ***universe**:

dx, dy, dz - Object translation (viewing position)

sx, sy, sz - Object scale (zoom)

theta, phi, omega - Object orientation (pan)

These three sets of parameters respectively represent the position of the origin of the universe with respect to the viewer (viewing position), how much the view is magnified to the user (zoom), and where the origin is with respect to the user (pan).

Transformations

Transformations of locations in 3D space can be reduced to four-dimensional matrix arithmetic[10]. A location in space can be represented by a four-dimensional row vector ($x\,y\,z\,1$). When this vector left-multiplies any 4-by-4 transformation matrix, the resulting row vector represents the transformed point. Tables 2, 3, and 4 illustrate the 4-by-4 transformation matrices for rotation around each axis.

Table 2. Z-Axis Rotation Matrix

cos	sine	0	0
-sin	cos	0	0
0	0	1	0
0	0	0	1

Table 3. Y-Axis Rotation Matrix

cos	0	-sin	0
0	1	0	0
sin	0	cos	0
0	0	0	1

Table 4. X-Axis Rotation Matrix

1	0	0	0
0	cos	sin	0
0	-sin	cos	0
0	0	0	1

It can be shown that these matrices can be used to account for a rotation about any arbitrary axis passing through the origin. The transformation matrix shown in Table 5 corresponds to scaling a location by (**sx**, **sy**, and **sz**) and then moving it by (**dx**, **dy**, and **dz**).

Table 5. Translation and Scaling Matrix

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

The arbitrary transformation of a frame can be defined by a matrix resulting from a multiplication of a subset of the above transformation matrices. However, this multiplication is in general, not commutative. That is, rotating around the x-axis and then translating is not the same as translating and then rotating about the x-axis. By sending values for the nine parameters, the host can request the adjustment of an object. However, *this* system defines these operation as always taking place in the order below:

1) Scale object by (**sx**, **sy**, and **sz**)
2) Translate object by (**dx**, **dy**, and **dz**)
3) Rotate object around z-axis by **theta**.
4) Rotate object around x-axis by **omega**.
5) Rotate object around y-axis by **phi**.

When the matrices shown in Tables 2 through 5 are multiplied, the resulting matrix always contains $(0\ 0\ 0\ 1)$T as its final column. Thus, to denote an arbitrary transformation, you need only remember the first three columns of the composite matrix. If you were to apply the transformations in the order stated previously, the resulting equations in Table 6 would determine the element of the transformation matrix R.

Table 6. Transformation Equations

$r_{12} = s_y \sin\theta$	(2.2)
$r_{13} = s_z \sin\Omega$	(2.3)
$r_{14} = \cos\Omega\ (d_x \cos\theta - d_y \sin\theta) + d_z \sin\Omega$	(2.4)
$r_{21} = s_x(\sin\theta\ \cos\phi + \cos\theta\ \sin\Omega\ \sin\phi)$	(2.5)
$r_{22} = s_y(\cos\theta\ \cos\phi - \sin\theta\ \sin\Omega\ \sin\phi)$	(2.6)
$r_{23} = -s_z \cos\Omega\ \sin\phi$	(2.7)
$r_{24} = \sin\phi\ (\sin\Omega\ (d_x \cos\theta - d_y \sin\theta) - d_z \cos\Omega) + \cos\phi\ (d_x \sin\theta + dy \cos\theta)$	(2.8)
$r_{31} = s_x(\sin\theta\ \sin\phi - \cos\theta\ \sin\Omega\ \cos\phi)$	(2.9)
$r_{32} = s_y(\cos\theta\ \cos\phi + \sin\theta\ \sin\Omega\ \cos\phi)$	(2.10)
$r_{33} = s_z \cos\Omega\ \cos\phi$	(2.11)
$r_{34} = \cos\phi\ (\sin\Omega\ (-d_x \cos\theta + d_y \sin\theta + d_z \cos\Omega) + \sin\phi\ (d_x \sin\theta + dy \cos\theta)$	(2.12)

Note that there also exists a matrix **p[3][4]** (see Listing 1 in the Appendix) that represents the product of all the ancestral transform matrices of an object and that object's R matrix. This matrix represents the object's transformation from the absolute origin of the system.

The Host Processor's Access to Objects

The 80X86 host can exert its control over objects in the following ways:

1) Target Objects - The host can set the target object for adjustment, deletion, or insertion of a child object by either targeting the parent object or a particular child object of the currently targeted object.
2) Load and Delete Objects - The host has the ability to add objects to the system with initial transform parameters. In addition, it can remove objects from the system (including all objects within the deleted objects). When the targeted object is deleted, the new target object defaults to being the object's parent.
3) Adjust Objects - By specifying the nine transform parameters, the host can adjust an object in its parent's frame.
4) Change Perspective - To change the viewing perspective, the host must request that the ***universe** be adjusted.
5) Update Screen Representation - The host can request that the targeted object and its child objects have their location array's screen representations updated.
6) Redraw View - Once all adjustments and updates of screen coordinates are re-specified, the host can request that the view be updated.

Overall, the **object** structure serves well as a data representation for 3D graphics. A single set of locations is available to be referenced by the points, line segments, and filled polygons to be rendered on the screen. Each **object** contains parameters and matrices that specify the transformed state of the object. Thus, at any time these matrices could be applied to the original co-ordinates

loaded into the system to calculate the transformed location of the point. Therefore, as the transformation and the projection on to two-dimensional co-ordinates are done in one step, the original 3D coordinates can be retained and only the final modified two-dimensional screen representation need be updated. The point of view can simply be modified by adjusting the ***universe** as one would adjust any other object. Overall, the hierarchical **object** structure provides a powerful and flexible way to manage graphical data.

DSP Command Execution

The digital signal processor assumes the role of the object manager and keeps track of the representations. Before examining the precise manner in which the TMS320C30 processes the commands from the host, one needs to understand the underlying hardware of this subsystem. A description of the TMS320C30 Application Board can be found in the application report *TMS320C30 Application Board Functional Description*, located in this book. The report describes the avenues of communication between the C30AB and the PC over the PC's bus. An examination of how the TMS320C30 receives and processes data and commands from the 80X86/7 follows.

Initialization

As its first initialization task, the PC maps the dual-port SRAM of the C30AB into its address space by writing the 8 MSBs of address to the mapping register. It then brings the C30AB out of reset by writing a 1 to the **SWRESET** in the C30AB's control register. The PC then loads the TMS320C30 application program into the dual-port SRAM. Loader support software on the C30AB EEPROM moves the code to the proper location in the TMS320C30's address space. Finally, the PC switches the TMS320C30's memory map into run mode to start program execution. The first part of the **main** routine initializes the system (see Listing 8 in the Appendix).

For the system software to run properly, the DSP software must initialize several different items.

1) It enables the on-chip instruction cache.
2) It sets the external flag bit on the C30AB target connector to transfer control of the rendering system from the PC to the C30AB (This assumes that the PC loaded the rendering software before it started up the C30AB).
3) It configures both the primary and the expansion bus with zero software wait-states. Thus, all wait states are generated by the address-decoding PALs on the C30AB.

In addition, the linker configures

1) Primary bus SRAM as program storage
2) Expansion bus SRAM as heap memory allocation
3) Zeroth page of internal RAM as space for system constants
4) First page of internal RAM as the system stack. This configuration maximizes the potential for parallel data and instruction accesses

The initialization procedure then appropriates several local variables for system use, including

1) Two registered looping variables, **i** and **j**
2) The constant 2 PI
3) Registered pointers to the communication registers of the rendering subsystem, ***hstdata** and ***hstcntl**

The TMS320C30 initially sets the contents of these GSP registers to indicate that the computational stage does not have any requests of the rendering stage.

The TMS320C30 system software contains the global variables shown in Listing 7 of the Appendix. The dual-port SRAM pointer **dual_port** is initialized to point to the lowest location on the I/O expansion bus. This pointer points to an integer array that contains all data and command from the PC. Another pointer to the currently targeted object (***to**) is set to reference the **universe**. The ***universe** is set as its own parent with an obnum of 0, indicating no internal objects are loaded.

During the final part of initialization, the C30AB software waits for the PC to load the static ***universe** object. To understand how the PC loads objects into the system, you must comprehend the general communications protocol between the TMS320C30 and the 80X86.

Host to DSP Communication

A two-way polling scheme arbitrates access of the dual-port SRAM. The software allocates the first two words of the SRAM as **COMMAND** and **ACKNOWLEDGE** signals, respectively (see Listing 6 in the Appendix). Remember that the TMS320C30 must mask off the 24 MSBs of dual-port data to receive the proper 8-bit value. The processors poll and write to these two words in order to send requests and acknowledgments. During initialization, the TMS320C30 clears both the **COMMAND** and **ACKNOWLEDGE** locations of the dual-port SRAM. The PC graphics application software must run after this point to ensure that this phase of the initialization does not clear a command from the PC. Once the system software starts executing on both the PC and the TMS320C30, the following sequence enables the PC to send a command to the C30AB:

1) The PC waits for the dual-port SRAM to become free by polling the **ACKNOWLEDGE** word for a zero.
2) The PC loads all command parameters into the dual-port SRAM.
3) The PC then loads the appropriate command byte into **COMMAND**.
4) Once the TMS320C30 returns to its command detection loop, it acknowledges a received command by writing the same byte into the **ACKNOWLEDGE** word.
5) The PC sees that the TMS320C30 has acknowledged the command and writes 00h into COMMAND to withdraw its command. The PC thereby relinquishes control of the dual-port SRAM.
6) The TMS320C30 reads all necessary parameters into its main memory.
7) The TMS320C30, by writing a zero to the **ACKNOWLEDGE** word, indicates that the PC can request another command. This returns the sequence to step (1).

The TMS320C30 treats all of its data types as 32-bit values, but it can read only one byte of valid data from the dual-port SRAM. Thus, the TMS320C30 must mask and concatenate the bytes that the PC maps into contiguous locations to form multibyte words. In addition, since Intel and

the TMS320C30 have different standards, floating-point values from the PC must be converted before the TMS320C30 can use them.

The TMS320C30 can receive either unsigned 8-bit **chars** or unsigned 16-bit short integers from the PC. The macros shown in Listing 6 of the Appendix are used to access these data types from the dual-port SRAM. The **DPLONG** macro takes a certain location in the dual-port, finds the short integer located there, and concatenates it into a 32-bit value for the TMS320C30. The word **LONG** in the macro indicates all integers whether **char**s, **short**s, or **long**s are represented as 32-bit values by the TMS320C30.

Table 7. Comparison of Intel and TMS320C30 32-Bit Floating-Point Formats

Standard	Exponent Field Bits	Exponent Format	Sign Bit	Mantissa Field	Mantissa Format
TMS320C30	31-24	Two's Complement	23	22-0	Two's Complement
Intel	30-23	Offset Binary	31	22-0	Magnitude

Table 7 illustrates the differences between the TMS320C30 and the Intel single-precision floating-point formats. For every floating-point value that the TMS320C30 receives, it must extract the appropriate fields, convert the fields to the appropriate numerical representation, and then reassemble the fields in TMS320C30 floating-point format. The **dpfloat** routine shown in Listing 9 of the Appendix uses the union structure **fllong** shown in Listing 6 of the Appendix to allow manipulations normally available only for integers on the floating-point value. The program first concatenates the four-byte value in the dual-port SRAM into a single 32-bit integer and then converts this word to TMS320C30 format.

Computational Subsystem Software

Using the communication techniques described in the last section, the TMS320C30 processes the graphics command from the PC. After performing C30AB initialization, the program **main** enters a command detection/execution loop. For each valid value of the **COMMAND** byte, a C **case** statement executes the appropriate code. Since these routines are, in general, too long to be discussed in exhaustive detail, the rest of this section merely summarizes how they work.

When the PC wants to load an object, it first loads the initial nine floating-point transformation parameters into the dual-port SRAM. It then loads the number of

1) Locations
2) Drawn points
3) Lines
4) Filled polygons

These values are limited to 16 bits, thereby allowing for only 65,535 primitives of each type. The size of the dual-port SRAM further limits the array sizes in this implementation. Then the PC loads three floating-point parameters, (x,y, and z), for each location. The size of the dual port limits the number of locations to 377. Once these parameters are loaded into the memory, the host places the command byte for an object load into **COMMAND**. Upon reception of these parameters, the TMS320C30 allocates space for the object as a child of the current target object and also allocates

space for the location, point, and line arrays. Because the size of each polygon varies, space is allocated as each polygon is read.

After allocating global space for the new object and loading the locations, the TMS320C30 requests more data from the PC. It first requests the points, then the lines, then each polygon. The dual-port SRAM limits the primitive arrays to 2047 points and 1364 lines. In addition, each polygon is limited to 4092 vertices. The TMS320C30 makes a data request by replacing the current **COMMAND** byte that it wrote in **ACKNOWLEDGE** with 127, the flag for the PC to load more data. Although the roles of **ACKNOWLEDGE** and **COMMAND** are reversed in this case, the TMS320C30 requests data in much the same way the PC requests commands. Once the TMS320C30 completes loading the object, it selects the object as the new target object. Finally, using the equations in Table 6, the TMS320C30 calculates the initial value of the object's transformation matrix.

The target object is the object in the hierarchy selected for adjustment, deletion, or calculation of screen coordinates. The PC can either target an object's parent or one of the object's child objects. The command to target a child requires the PC to specify either the child object's sibling number or **subnum**. Thus, when selecting objects for adjustment, the PC must remember where it loaded objects into the hierarchy.

To adjust the transformation parameters of a given object, the PC simply loads the new parameters into the dual-port SRAM. The TMS320C30 adds the values of the new angles of rotation and translation factors to the previous ones. In addition, the TMS320C30 multiplies the old scaling factors by the new ones. Then, the TMS320C30 calculates the transformation matrix of the object by using the equations in Table 6. It does not recalculate screen locations, however, until this is specifically requested by the PC. The TMS320C30 can thus avoid calculating screen coordinates until all adjustments have been made.

Once the PC requests all the changes for a frame on the display, it requests recalculation of screen coordinates at each node it changed. The PC can request recalculation for a particular object and thus update its internal objects as well. This allows the TMS320C30 to avoid recalculating screen coordinates of unchanged locations. For maximum efficiency, the PC must request recalculation in the highest node that it adjusted along any particular path. Thus, in the planetary example given earlier, if, in a period of time, only Pluto and its moon Charon were moved (the other bodies miraculously standing still), only Pluto would need to be targeted for recalculation.

To calculate transformations, the TMS320C30 multiplies the object's transformation matrix by *its parent's* parent transformation matrix to obtain *its own* parent transformation matrix, **p[3][4]**. The TMS320C30 right-multiplies all locations within that object by this matrix to achieve the transformation from the absolute origin of the system. The computational engine calculates perspective by dividing the transformed x- and y-coordinate by the transformed z-coordinate so that locations farther away appear closer together. The plane $z=0$ is defined to be the plane of the screen. This also has the feature that objects behind the viewer appear upside-down in front of the viewer because the objects' z-coordinates are negative. Thus, the program running on the PC must maintain all objects in front of the viewer. Then, the TMS320C30 recursively executes this procedure for each object within the targeted object.

Unlike the recalculation of screen coordinates, the redrawing of objects is done for all objects within the system. Thus, the **draw_object** routine is called with the ***universe** as the argument. The

precise manner in which the TMS320C30 uses this program to redraw the screen is described in the TMS320C30 Drawing Routine Section found later in this report.

Summary of DSP Command Execution

The dual-port SRAM on the C30AB provides all means of communication between the PC and the TMS320C30. A two-way polling scheme arbitrates the TMS320C30's and the PC's access to this SRAM. Using this protocol, the PC can request object loading, deletion, or adjustment, but can request only modification of the object currently targeted for these changes. Also, at the host's request, the computational engine may recalculate the screen representation of all locations within the targeted object. Once all updates for a particular view are made, the PC may request a redrawing of the display. The description of the rendering subsystem, presented next, facilitates a better understanding of how the TMS320C30 requests rendering commands of the GSP.

The Rendering Subsystem

A modified version of the TMS34010 Software Development Board serves as the rendering stage of this graphics pipeline. A complete overview of this PC-based card can be found in the *TMS34010 Software Development Board User's Guide* [2]. Because only minor modifications were made to the commercially available SDB, the hardware aspects of the rendering subsystem are discussed in less detail than the computational stage. The same holds true for many software routines taken from the *TMS34010 Math/Graphics Function Library*.[8] After presenting overviews of the TMS34010 and the SDB, this section focuses on the C30AB/SDB interface and the communications protocol used for command and data transfer between the TMS320C30 and the GSP.

The TMS34010 Graphics System Processor

The TMS34010 combines the best features of general-purpose processors and graphics controllers in one powerful and flexible Graphics System Processor. Key features of the TMS34010 are its speed, high degree of programmability, and efficient manipulation of hardware-supported data types, such as pixels and two-dimensional pixel arrays.

The TMS34010's unique memory interface reduces the time needed to perform tasks such as bit alignment and masking. The 32-bit architecture supplies the large blocks of continuously-addressable memory that are necessary in graphics applications. TMS34010 system designs can take advantage of video RAM technology to facilitate applications such as high-bandwidth frame buffers; this circumvents the bottleneck often encountered when using conventional DRAMs are used in graphics systems.

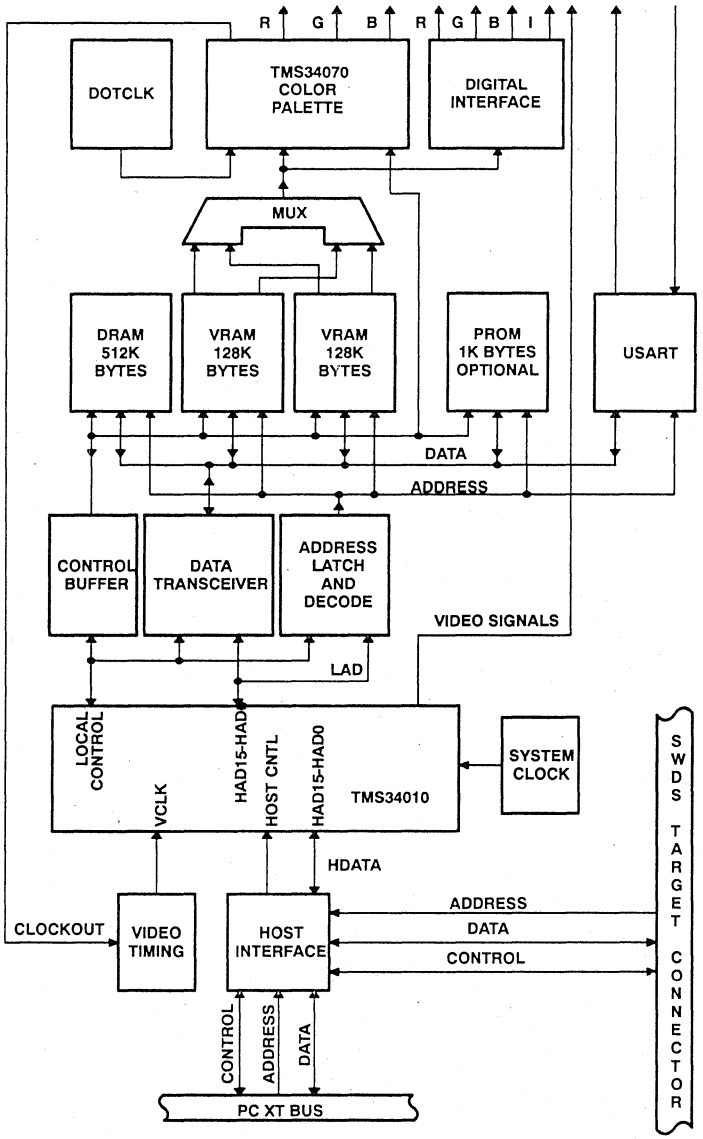
The TMS34010's instruction set includes a full complement of general-purpose instructions, as well as graphics functions from which you can construct efficient high-level functions. The instructions support arithmetic and Boolean operations, data moves, conditional jumps, plus subroutine calls and returns.

The TMS34010 architecture supports a variety of pixel sizes, frame buffer sizes, and screen sizes. On-chip functions have been carefully selected so that no functions tie the TMS34010 to a particular display resolution. This enhances the portability of graphics software and allows the TMS34010 to adapt to graphics standards such as MIT's X, CGI/CGM, GKS, NAPLPS, PHIGS, and other evolving industry and display management standards.

TMS34010 Software Development Board

Figure 4 shows the block diagram of the modified TMS34010 SDB. The graphics SDB is a single card designed around the IBM PC/XT Expansion Bus and serves as a software development tool for programmers writing application software for the TMS34010 Graphics System Processor. The development of a high-performance bit-mapped graphics display in this application report demonstrates the simplicity of hardware design using the TMS34010 SDB.

Figure 4. Modified TMS34010 Software Development Board Block Diagram



This board comes with interactive debug software. Its features include software breakpoints, software single-step and run with count. At the same time, current machine status is displayed on the top half of the host monitor.

The SDB contains 512K bytes of program RAM for the TMS34010 to execute drawing functions, application programs, and displays. Both the program RAM and the frame buffer are accessible to the host through the TMS34010's memory-mapped host port.

The frame buffer consists of eight SIP memory modules organized into four color planes. This allows 16 colors per frame from the digital monitor. The TMS34070 color palette incorporates a 12-bit color lookup table to give you a choice of 16 colors in a frame from a 4096-color palette. Furthermore, the palette incorporates a variety of unique line load features to allow the color lookup table to be reloaded on every line; this means that 16 of 4096 colors can be displayed per line.

The TMS34010 Host Interface

The GSP has two 16-bit buses: one interfaces with the video and program memory, and a second interfaces to a host processor. The host can access the GSP by writing and reading four internal memory-mapped GSP 16-bit registers:

- **HSTADRL** and **HSTADRH** together form a 32-bit pointer to a location in the GSP's address space.
- **HSTCNTL** contains several programmable fields that control host interface functions.
- **HSTDATA** buffers data that is transferred through the host interface between the GSP's local memory and the host processor.

Several signals are available for communications between the host and the GSP.

- **HD15** through **HD0** are the actual data lines.
- **HCS** is the interface select signal strobe from the host.
- **HSF1** and **HSF0** select which host register is being addressed.
- **HREAD** and **HWRITE** are, respectively, the read and write strobes from the host.

Table 8 shows how the above signals address the four host registers.

- **HLDS** and **HUDS** signals, respectively, select the low byte or the high byte of the host interface registers.
- **HRDY** informs the host when the GSP is ready to complete a transaction.
- **HINT** is the interrupt signal from the host to the GSP.

Table 8. TMS34010 Signals Controlling Host Port Interface

Host Interface Control Signals				
HCS	**HSF1 & HSF0**	**HREAD**	**HWRITE**	**Operation**
1	XX	X	X	No Operation
0	00	0	1	HSTADRL read
0	00	1	0	HSTADRL write
0	01	0	1	HSTADRH read
0	01	1	0	HSTADRH write
0	10	0	1	HSTDATA read
0	10	1	0	HSTDATA write
0	11	0	1	HSTCNTL read
0	11	1	0	HSTCNTL write

The fields in **HSTCNTL** control host interrupt processing, auto-incrementing of the host address register, and protocol in byte-at-a-time accesses to the 16-bit host port (whether the lower or the higher byte comes first). **HSTCNTL** also contains the status of interrupts from the host to the GSP and from the GSP to the host and a three-bit message word in either direction. These control bits are shown in Table 9.

Table 9. TMS34010 Host Control Register Fields

Field	Name	Purpose	Write Access
0 – 2	MSGIN	Input Message Buffer	Host Only
3	INTIN	Input Interrupt Bit	Host Only
4 – 6	MSGOUT	Output Message Buffer	GSP Only
8	INTOUT	Output Interrupt Bit	GSP Only
8	NMI	Nonmaskable Interrupt	Host Only
9	NMIN	Nonmaskable Interrupt	GSP and Host
10	Unused	Unused	Neither
11	INCW	Increment Pointer Address on Write	GSP and Host
12	INCR	Increment Pointer address on Read	GSP and Host
13	LBL	Lower Byte Last	GSP and Host
14	CF	Cache Flush	GSP and Host
15	HLT	Halt TMS34010 Processing	GSP and Host

TMS320C30 Application Board Interface

In its unmodified form, the SDB communicates to the PC host through a single transceiver. A PAL decodes the PC address into the appropriate register selection signals. The registers are mapped redundantly into blocks of PC memory address space, as shown in Table 10. The board was modified by the addition of a connector to a cable from the C30AB's target connector. The TMS320C30 sends to the modified SDB the following:

- The TMS320C30s expansion bus address
- The TMS320C30s data signals
- I/O address space access strobe
- Expansion bus read and write strobes

These signals map the GSP's host interface registers in the TMS320C30's address space (also shown in Table 10). The TMS320C30 mapping is actually replicated in four-word blocks until location 8057FFh.

Table 10. Mapping of TMS34010 Host Control Registers

Register	PC Mapping	TMS320C30 Mapping
HSTDATA0	C7000h - C7CFFh	805002h
HSTCNTL	C7D00h - C7DFFh	805003h
HSTADRL	C7E00h - C7EFFh	805000h
HSTADRH	C7F00h - C7FFFh	805001h

The modified SDB board must be able to select either the PC or the C30AB as its host. The C30AB target connector makes the two external flag bits **XF0** and **XF1** available to the SDB. The TMS320C30 can configure these flags as either input or output pins. Upon leaving reset, these pins default to inputs and remain in the high-impedance state. **XF0** is pulled low on the SDB to appear off when the TMS320C30 is in reset. After the PC loads the rendering software into the GSP, it activates the C30AB and loads the TMS320C30's software. As discussed earlier, the TMS320C30, during initialization, configures **XF0** as an output and loads it with a one. The address-decoding PALs on the SDB use this signal to select the C30AB as the SDB's host. When the TMS320C30 controls the SDB, it communicates through a full 16-bit interface to the GSP. Thus, before the integer screen coordinates are sent in two's-complement form to the GSP, they must be clipped to a range of -32,768 to 32,767. Fortunately, this range is still two orders of magnitude greater than the resolution of most monitors.

In general, the above interface is fairly straightforward. The only complication is that the designers of the GSP expected a relatively slow microcoded general-purpose processor as a host. This allows the GSP to actually assert its **HRDY** line 80 ns before it is actually ready to process a transaction. When interfacing to the TMS320C30, PALs become necessary as state machines to create the appropriate number of wait-states on host reads and writes and thus ensure proper interprocessor communication.

DSP to GSP Communication

The TMS320C30 loads all commands and data into a command buffer contained within a space not usually mapped by the SDB's C compiler configuration. This portion of GSP address space, the Shadow RAM, is normally reserved for optional PROMs. However, by writing a 1 to an RS latch in the GSP's memory space, this area becomes occupied by the topmost portion of program/data DRAM. Before the TMS320C30 starts writing to **HSTDATA** to access this memory, it configures the host address to autoincrement. Once the GSP finishes processing data in the shadow RAM, it resets the value of the address registers to point to the beginning of the shadow RAM in order to allow the TMS320C30 to properly load its next command and data.

The communication protocol between the TMS320C30 and the GSP closely resembles the protocol between the PC and the TMS320C30. The **MSGIN** and **MSGOUT** fields, respectively, replace the **COMMAND** and **ACKNOWLEDGE** words. However, rather than these fields con-

taining a particular value for a command, the value of 3 (binary 011) in either of these fields indicates that a command or an acknowledge exists. Upon reception of a command request, the GSP refers to the first location of the shadow RAM for a command word from the TMS320C30. Thus, the overall command scheme proceeds as follows:

1) The TMS320C30 waits until it sees that the **MSGOUT** field contains a 0.
2) The TMS320C30 stores all command and data into the shadow RAM.
3) The TMS320C30 writes a 3 to the **MSGIN** field and waits for acknowledgment.
4) The GSP acknowledges the reception of a command by writing a 3 to the **MSGOUT** field.
5) The TMS320C30 withdraws its request by writing a 0 to **MSGIN**.
6) The GSP reads the first word of the shadow RAM for the command and jumps to the appropriate case to process it.
7) Once the GSP is finished with all data in the shadow RAM, it resets the values of the host address registers and then writes a 0 to the **MSGOUT** bit, indicating that the TMS320C30 is free to request another command.

The TMS320C30 Drawing Routine

When the TMS320C30 receives a redraw-screen request from the PC, it sends a command to the GSP to clear the screen after the monitor has drawn the bottom line; this ensures that the last view was drawn in its entirety. The TMS320C30 then calls its **draw_object** routine with ***universe** as an argument. For each array of primitives within the **object**, the TMS320C30 sends the size of the array and the array of screen representations of the primitives themselves to the TMS34010. Thus, the TMS320C30 can request the GSP to draw arrays of points, lines, or filled polygons. Once all arrays are drawn, **draw_object** recursively executes for all child objects within the universe. In this manner, all objects defined within the system are drawn.

GSP System Initialization

Several initialization routines are provided in the *TMS34010 Math/Graphics Function Library User's Guide* [8]. The GSP executes these programs to properly configure the system before it begins its command detection loop:

- The call to **init_video** configures the graphics buffer for an NEC Multisync Monitor displaying 640 x 480 resolution.
- The **init_graphics** function initializes the graphics environment by setting up the data structures for the graphics functions and assigning default values to system parameters.
- The **init_screen** command initializes the screen. The entire frame buffer is cleared, and a color lookup table is loaded with the default color palette.
- The **init_vuport** function initializes the viewport data structures and opens viewport 0, the system, or root window.
- The **set_origin** command sets the origin of the system to the center of the screen.

Drawing Routines

Several drawing routines are also provided in the *TMS34010 Math/Graphics Function Library User's Guide* [8]:

- For each primitive in an array sent from the TMS320C30, the GSP sets the proper drawing color with the **set_color** command.
- The TMS320C30 commands the GSP to execute to the **clear_screen** before it starts to request drawing of primitives for the next view.
- The TMS320C30 requests a **wait_scan** execution from the GSP to ensure that the GSP has fully displayed the last view before drawing the current view.
- The GSP uses the **draw_point(x,y)** function to render a point on the display.
- Similarly, it uses the **draw_line(x1,y1,x2,y2)** command to draw a line. The arguments are the screen coordinates of the two end-points of the segment.
- The **fill_polygon(n, linelist, ptlist)** function takes as arguments of the number of vertices, an array of the line segments forming the sides of the polygon, and a list of screen coordinates referenced by the linelist.

Summary

The TMS34010 Software Development board provides a good rendering module for this graphics system. The support hardware has been debugged and used in industry since 1987 and thus makes a reliable rendering subsystem. The target connector to the C30AB provides access to the TMS320C30 as an alternate host. Three PALs and two transceivers allow the TMS320C30 to assume control of the GSP, once both have started running their software. The **draw_object** program on the TMS320C30 can command the GSP to draw graphics primitives. Functions in the *TMS34010 Math/Graphics Function Library User's Guide* [8] allow the GSP to initialize the monitor interface, clear the screen, ensure that an entire screen has been drawn, and draw the graphics primitives. Overall, the TMS34010 development tools provide an easy means to develop a rendering subsystem for this graphics pipeline.

Possible Improvements

Several changes may be incorporated into the system to improve performance. Some simple enhancements involve modifications of the computational subsystem's software to allow faster and more transparent command execution. Restructuring the method in which the data and command pass through the pipeline, a more complex modification, can greatly increase throughput. Additional features such as more complex primitives, lighting, windowing, and text display would require major software modifications to the system. However, any such modifications would not need to change the communication protocols or the command detection loops significantly. Finally, although the TMS320C30 represents the state-of-the-art in digital signal processing, the host processor and the rendering engine may be improved.

Computational Subsystem Software

The drawing routine currently sends the primitive arrays of an object one at a time to the GSP. Instead, it should send all primitive arrays for all objects to be redrawn in a single pass. The GSP should then process the contents of this stack of commands and data.

Currently, as soon as the PC finishes requesting objects adjustments, it must request recalculations of the screen coordinates of location arrays. The screen_object routine must operate on all

objects that have been adjusted directly or indirectly by having their ancestors adjusted. Instead, this routine should be called once with the ***universe** as the argument. The **object** structure should contain a flag that is set when an object is adjusted and reset when it is drawn. Thus, the new **screen_object** procedure would recursively search down the hierarchy of objects until it encounters an object that has been adjusted and then should recalculate all the screen coordinates for it and those of its internal objects. Upon completion, it should search the rest of the hierarchy for adjusted objects. Thus, the host would have to request only adjustment, targeting, and draw commands. Screen representations would be automatically recalculated whenever a draw command is executed.

Rendering Subsystem Software

Rendering subsystem drawing routines could be improved by designing functions coded to handle the primitive arrays rather than individual programming elements. These functions may be able to fit in the GSP's instruction cache and improve execution time.

Improved Data Flow

One problem consistent at all stages of the system is the method of buffering. A single buffer usually contains all data and commands to be transferred from one stage to the next. Thus, during command execution one processor may wait for the other to relinquish control of the command buffer.

The first of two methods to improve the dual-port SRAM connecting the PC and the DSP is to divide the SRAM into two buffers. The PC writes the current command to one buffer, while the TMS320C30 processes commands and data stored in the other. This prevents contention for the dual-port SRAM. The particular buffer which each processor controls is swapped on each command request. Second, adding three more 4K x 8 dual-port SRAMS in parallel would allow the PC to communicate to the TMS320C30 with full 32-bit wide words. Thus, the masking and concatenation necessary to receive larger data types would become unnecessary. On the original design the potential addition of these RAMs consumed a prohibitive amount of board space. Full word size is possible only if space constraints are eased.

The splitting of the command buffer between the TMS320C30 and the GSP allows the GSP to draw the current screen while the TMS320C30 sends the primitive arrays for the next. Similarly, two display buffers allow one buffer to be displayed on the monitor while the GSP draws the next view to the other.

Computational Features

The DSP is suited to perform many other types of computational features. Because these functions are more complex, they were not implemented in the limited design time available. This system truncates objects that are too high, too low, too far right, or too far left by using the GSP's drawing routines that automatically clip coordinates outside the screen boundaries. However, the system cannot determine whether one object is in front of another and draw the objects appropriately. Functions to do this hidden-surface removal require complex algorithms to determine whether

one 3D surface obscures another. Simpler routines could be made to clip objects that are too far away to see or objects that are behind the viewer.

A lighting feature would allow appropriate factors of light intensity and reflection to determine the shading of surfaces. Lighting may be ambient (equal everywhere) or come from several possible source geometries. Reflections could either be diffuse and scatter light equally in all directions, or be specular like those off any shiny surface. With these parameters, the TMS320C30 can compute the appropriate shading of a given pixel. In this scenario, the GSP is reduced to drawing single points with a given color. Thus, any lighting function would slow rendering time.

More complex primitives can be produced by using the TMS320C30 to generate arrays of pixels representing solutions to equations. The PC could dispatch a command to draw a primitive based on a particular type of equation (such as the parametric equations representing a sphere) and then load the appropriate parameters for that equation. The DSP would generate the appropriate set of pixels for that object and send it to the GSP as arrays of points.

Rendering Features

The *TMS34010 Math/Graphics Function Library* [8] permits the user to create and select various windows for display. Once a window is selected the DSP can run the existing system software within that window. Thus, the host would also need to be able to direct the DSP to tell the GSP how to manipulate its windows. The Library also enables the GSP to print text on the screen. This feature also would not be very difficult to implement.

A More Advanced Host

A more advanced host could be a high-speed RISC processor such as SPARC. This unit could communicate with the DSP at faster rates, so command transfers would consume less time. In addition, SPARC is a 32-bit machine, which could allow word transfers between host and DSP in a single instruction.

A More Advanced Rendering Engine

The TMS34010's performance as a rendering engine could be improved. If the GSP could be ready to complete a transaction when the **HRDY** line is asserted and not some period of time later, the C30AB to SDB interface would be more straightforward and not require as many wait states. This problem is corrected in the second-generation GSP TMS34020, which was not available at the time of the design of this system. In addition, the TMS34020 also allows the host to transparently access the GSP's bus while the GSP continues processor functions.

Conclusion

Despite its shortcomings, this system still demonstrates the dataflow in a graphics pipeline using a digital signal processor as a computational element. One main benefit of the digital signal

processor is the availability of development tools such as C compilers, assembler/linkers, software development boards, and in-circuit emulators that accelerate design time. The TMS320C30 also provides speeds comparable to many bit-slice processors that require programmers to develop extensive microcode routines. The hardware multiplier, floating-point capability, RISC architecture, and parallel bus access facilitate fast, precise graphics calculations. Overall, a digital signal processor provides an attractive option to the graphics system designer interested in making high-performance systems with quick turnaround time.

References

1) *TMS320C30 Simulator User's Guide* (literature number SPRU017), Texas Instruments, 1989.
2) *Third-Generation TMS320 User's Guide* (literature number SPRU031), Texas Instruments, 1988.
3) *TMS320C30 C Compiler User's Guide* (literature number SPRU034), Texas Instruments, 1988.
4) *TMS320C30 Assembly Language Tools User's Guide* (literature number SPRU035), Texas Instruments, 1989.
5) *TMS34010 Software Development Board Schematics* (literature number SPVU003), Texas Instruments, 1986.
6) *TMS34010 Software Development Board User's Guide* (literature number SPVU002A), Texas Instruments, 1987.
7) *TMS34010 User's Guide* (literature number SPVU001A), Texas Instruments, 1988.
8) *TMS34010 Math/Graphics Function Library User's Guide* (literature number SPVU006), Texas Instruments, 1987.
9) *TMS34010 C Compiler Reference Guide* (literature number SPVU005A), Texas Instruments, 1986.
10) Foley, J.D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison Wesley, 1984.

Appendix A

Graphics Programs

Listing	Name
1	TMS320C30 C Structure Representing an Object
2	TMS320C30 C Structure Representing a Location
3	TMS320C30 C Structure Representing a Point
4	TMS320C30 C Structure Representing a Line
5	TMS320C30 C Structure Representing a Filled Polygon
6	TMS320C30 Communications Macros
7	TMS320C30 Global Variables
8	TMS320C30 Main Command Execution Loop
9	TMS320C30 Floating-Point Conversion Routine
10	TMS320C30 Object Loading Routine
11	TMS320C30 Screen Coordinate Calculation Routine
12	TMS320C30 Transformation Matrix Evaluation Routine
13	TMS320C30 Object Deletion Routine
14	TMS320C30 Request for Additional Data in Object Load
15	TMS320C30 Object Drawing Routine
16	TMS34010 Point Structure
17	TMS34010 Line Structure
18	TMS34010 Color Array
19	TMS34010 Color Palette
20	TMS34010 Main Command Execution Routine
21	PC Object Loading Data Structure
22	PC Communications Macros
23	PC Global Variables
24	PC Targeted Object Adjustment Routine
25	PC Routine to Set Parameters for an Object Load
26	PC Routine to Target Parent of Current Target Object
27	PC Routine to Target a Child of Current Target Object
28	PC Routine to Redraw Screen
29	PC Routine to Load the Primitives of a Wireframe Cube
30	PC Main Routine to Draw a "Planetary System of" Cubes

```
*************************************************************************

-->Listing 1: TMS320C30 C Structure Representing an Object

struct object
{
   struct object *parent;/* object within who's frame the object is defined */
   long subnum;                              /* sibling number of object   */
   long locnum;                              /* number of locations        */
   long ptnum;                               /* number of points           */
   long lnnum;                               /* number of lines            */
   long pgnum;                               /* number of polygons         */
   long obnum;                               /* number of daughter objects */
   float sx;      float sy;       float sz;  /* scale factors              */
   float dx;      float dy;       float dz;  /* offsets                    */
   float theta;          /* angle of rotation around z-axis (x to y)       */
   float phi;            /* angle of rotation around x-axis (y to z)       */
   float omega;          /* angle of rotation around y-axis (z to x)       */
   float r[3][4];        /* matrix formed by scale, the offset, then rotate */
   float p[3][4];        /* ascending product of all ancestral r matrices  */
   loc      *locs;       /* pointer to location array                      */
   point    *points;                  /* pointer to point array           */
   line     *lines;                   /* pointer to line array            */
   polygon  *polygons;                /* pointer to polygon array         */
   struct object *objects[MAXOB];     /* pointer to array of              */
                                      /*pointers to child objects         */
};

*************************************************************************


*************************************************************************

-->Listing 2: TMS320C30 C Structure Representing a Location

typedef struct
{
   float x;     float y;     float z;              /* world  coordinates */
   long a;      long b;                            /* screen coordinates */
} loc;

*************************************************************************
```

```
*************************************************************************

-->Listing 3: TMS320C30 C Structure Representing a Point

typedef struct
{
   long color;
   long locn;                    /* number of location in location array */
} point;

*************************************************************************


*************************************************************************

-->Listing 4: TMS320C30 C Structure Representing a Line

typedef struct
{
   long color;
   long startlocn;                                  /* start loc number */
   long endlocn;                                    /* end loc number   */
} line;

*************************************************************************


*************************************************************************

-->Listing 5: TMS320C30 C Structure Representing a Filled Polygon

typedef struct
{
   long color;
   long vertnum;                        /* number of vertices            */
   long *vertlocn;                      /* array of vertices loc numbers */
} polygon;

*************************************************************************
```

```
*********************************************************************

-->Listing 6: TMS320C30 Communications Macros

/*-------------------------------------------------------------------*/
/*              COMMUNICATIONS MACROS TO GSP                         */
/*-------------------------------------------------------------------*/
#define CTLFREE      0x0800
#define CTLREQ       0x0803
#define CTLACK       0x0833
#define CTLWITH      0x0830
#define HOSTCNTL     (*hstcntl & 0x00FFF)
/*-------------------------------------------------------------------*/
/*              MAXIMUM NUMBER OF INTERNAL OBJECTS                   */
/*-------------------------------------------------------------------*/
#define MAXOB        10
/*-------------------------------------------------------------------*/
/*              PC COMMUNICATION LOCATIONS                           */
/*-------------------------------------------------------------------*/
#define COMMAND      (*dual_port & 0x0FF)
#define ACKNOWLEDGE  dual_port[1]
/*-------------------------------------------------------------------*/
/*              DATA RECOVERY FROM THE DUAL PORT                     */
/*-------------------------------------------------------------------*/
#define DP0(a)       dual_port[a]
#define DP1(a)       dual_port[a + 1]
#define DP2(a)       dual_port[a + 2]
#define DP3(a)       dual_port[a + 3]
#define DPLONG(a)    ((long) ((DP1(a) & 0x00FF) << 8 | (DP0(a) & 0x00FF)))

*********************************************************************


*********************************************************************

-->Listing 7: TMS320C30 Global Variables

long k,l;                          /* temporary and looping variables     */
struct object *universe, *to, *no; /* universe, target object, next object */
unsigned long *dual_port;          /* dual port SRAM                      */
union                              /* variable to construct a c30 format  */
{                                  /*  float from intel format allowing   */
float          f;                  /*  bit manipulation on a float        */
unsigned long  i;
} fllong;

*********************************************************************
```

```
*********************************************************************

-->Listing 8: TMS320C30 Main Command Execution Loop

void main()
{
    register float twopi = 6.283185308;
    register long i, j;
    register long *hstdata = (long *) 0x805002; /* 340 host data register   */
    register long *hstcntl = (long *) 0x805003; /* 340 host control register*/
    dual_port = (unsigned long *) 0x804000;
    asm("       OR      0800h,ST");/* enable cache                          */
    asm("       LDI     02h,IOF"); /* set XF0 and assume control of 340SDB */
   /* set for zero internal wait states on both buses                       */
    *((unsigned long *) 0x808060) = 0;
    *((unsigned long *) 0x808064) = 0x1000;
    *hstcntl     = CTLFREE;/* turn off any request to TMS34010              */
    *dual_port   = 0;      /* turn off any request from the PC              */
    ACKNOWLEDGE = 0;       /* turn off any acknowlegement to the PC         */
   /* allocate space for the internal object                                */
    universe = (struct object *) malloc (sizeof(struct object));
    to         = universe;         /* target universe                      */
    to->subnum = 0;                /* set universe sibling number to 0     */
    to->parent = to;               /* universal object is its own parent   */
    while(COMMAND != 1);           /* first command must be a load object  */
    ACKNOWLEDGE = 1;               /* acknowledge that c30 is ready        */
    while(COMMAND != 0);           /* wait for pc to withdraw request      */
    load_object();                 /* load universe                        */
    ACKNOWLEDGE = 0;               /* show that dual port is free          */
    matrix();                      /* calculate transformation matrix      */
    for(;;)                        /* infinite loop for PC command detection*/
    {
        while(COMMAND == 0);       /* wait for PC to request service       */
        j = COMMAND;               /* save command                         */
        ACKNOWLEDGE = j;           /* acknowledge request                  */
        while(COMMAND != 0);       /* wait for PC to withdraw request      */
        switch (j)                 /* execute requested command number     */
        {
           case 1:                 /* LOAD A DAUGHTER OBJECT               */
               if (to->obnum == MAXOB) break; /* abort if > maximum objects */
               j = ++to->obnum;    /* increase number of daughter objects  */
                                   /* allocate space for new object        */
            to->objects[j] = (struct object *) malloc (sizeof(struct object));
               no = to->objects[j]; /* next object is daughter object      */
               no->subnum = j;     /* set sibling number of next object    */
               no->parent = to;    /* assign current object as no's paret  */
               to = no;            /* target daughter object               */
               load_object();      /* load daughter object                 */
               ACKNOWLEDGE = 0;    /* show that dual port is free          */
               matrix();           /* calculate transform matrix           */
               break;

            case 2:                /* TARGET A DAUGHTER OBJECT             */
               j = DPLONG(2);      /* get daughter object number to target */
```

```
    ACKNOWLEDGE = 0;    /* show that dual port is free          */
    if (j > to->obnum) break; /* can only target existing object */
    to = to->objects[j]; /* target daughter object              */
    break;

  case 3:               /* TARGET PARENT OBJECT                 */
    ACKNOWLEDGE = 0 ;   /* show that dual port is free          */
    to = to->parent;    /* set targeted object to parent        */
    break;

  case 4:               /* DELETE TARGETED OBJECT               */
    ACKNOWLEDGE = 0;    /* show that request dual port is free  */
    if (to == universe) break;/* don't allow deletion of universe*/
    j = to->subnum + 1; /* get number of next sibling           */
    no = to->parent;    /* set next object to parent            */
    delete_object(to);  /* delete current object                */
    to = no;            /* target parent object                 */
    l = to->obnum;      /* find total number of siblings        */

   /* decrement sibling number on all younger siblings          */
    for(i = j; i <= l; ++i) --to->objects[i]->subnum;
    --to->obnum;   /* decrement total number of daughter objects */
    break;

 case 5:                /* ADJUST TARGETED OBJECT               */
    to->sx    *= dpfloat(2); /* adjust scales                   */
    to->sy    *= dpfloat(6);
    to->sz    *= dpfloat(10);
    to->dx    += dpfloat(14); /* adjust offsets                */
    to->dy    += dpfloat(18);
    to->dz    += dpfloat(22);
    to->theta += dpfloat(26); /* adjust angles                 */
    to->phi   += dpfloat(30);
    to->omega += dpfloat(34);
    ACKNOWLEDGE = 0;          /* show that dual port is free   */

   /* keep angles in the (0,2pi) range)                         */
    to->theta = fmod(to->theta, twopi);
    to->phi   = fmod(to->phi  , twopi);
    to->omega = fmod(to->omega, twopi);
    matrix(to);               /* recalculate transform matrix  */
    break;

 case 6:                      /* DRAW UNIVERSE                 */
    ACKNOWLEDGE = 0;          /* show that dual port is free   */

    while(HOSTCNTL != CTLFREE); /* wait for 340 to be free     */
    *hstdata = 4;             /* enter command for a screen clear */
    *hstcntl = CTLREQ;        /* request service from 340      */
    while(HOSTCNTL != CTLACK); /* wait for acknowledgement     */
    *hstcntl = CTLWITH;       /* withdraw request              */

    draw_object(universe);    /* draw universe                 */

    while(HOSTCNTL != CTLFREE); /* wait for 340 to be free     */
    *hstdata = 6;             /* enter command for a scanline  */
    *hstcntl = CTLREQ;        /* request service from 340      */
    while(HOSTCNTL != CTLACK); /* wait from acknowledgement    */
    *hstcntl = CTLWITH;       /* withdraw request              */
    break;

  case 7:                     /* CALCULATE SCREEN COORDINATES  */
   /* +++WARNING+++ the PC user must execute a screen command to */
   /* screen all objects that have been adusted since the last   */
   /* draw before the next draw. However, if an object is        */
   /* screened all daughter objects are as well.                 */
    ACKNOWLEDGE = 0;          /* show that dual port is free   */
    screen_object(to);        /* calcuclate screen coordinates */
    break;

  default:
    ACKNOWLEDGE = 0;          /* show that dual port is free */
    break;
  }
 }
}
```

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

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

-->Listing 9: TMS320C30 Floating-Point Conversion Routine

float dpfloat (a)
register unsigned long a;           /* offset from start of dual port SRAM */
{
  register unsigned long sign;
  unsigned long mant, ex;

  a =   (DP3(a) << 24               /* concatenate 4-byte value            */
      | (DP2(a) & 0x00FF) << 16
      | (DP1(a) & 0x00FF) << 8
      | (DP0(a) & 0x00FF));
  sign = (a & 0x80000000) >> 8;     /* extract and reposition sign bit     */
  ex   =  ((a & 0x7F800000)         /* extract exponent                    */
        - 0x3f800000) << 1;         /* converts to 2's complement          */
  if (sign)
  {
    mant = (- a) & 0x007FFFFF;      /* takes 2's complement of mantissa    */
    if (mant == 0) ex  -= 0x01000000; /* checks for input mantissa of -2   */
  }
  else mant = a & 0x007FFFFF;       /* otherwise leave mantissa alone      */
  a = sign + mant + ex;             /* reconstruct floating-point fields   */
  fllong.i = a;
  return fllong.f;                  /* return reconstructed float          */
}

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

```
*************************************************************************

-->Listing 10: TMS320C30 Object Loading Routine

void load_object()
{
   register long i,j;                    /* temporary and looping variables */
   register struct object *o;            /* pointer to target object        */
   register loc     *temploc;            /* temporary location pointer      */
   register line    *templn;             /* temporary line pointer          */
   polygon          *temppg;             /* temporary polygon pointer       */
   point   *temppt;                      /* temporary point pointer         */
   long lc   = DPLONG(2);                /* number of coordinate locations  */
   long pt   = DPLONG(4);                /* number of points                */
   long ln   = DPLONG(6);                /* number of lines                 */
   long pg   = DPLONG(8);                /* number of polygons              */

    o = to;                        /* set target object as object for loading /

  /* initialize primitive numbers and transform parameters                  */
   o->locnum = lc;
   o->ptnum  = pt;
   o->lnnum  = ln;
   o->pgnum  = pg;
   o->obnum  = -1;
   o->sx     = dpfloat(10);    o->sy  = dpfloat(14);   o->sz    = dpfloat(18);
   o->dx     = dpfloat(22);    o->dy  = dpfloat(26);   o->dz    = dpfloat(30);
   o->theta  = dpfloat(34);    o->phi = dpfloat(38);   o->omega = dpfloat(42);

  /* ALLOCATE SPACE FOR OBJECT PRIMITIVES                                   */
   o->locs   = (loc *) malloc (sizeof (loc) * lc);
   o->points     = (point   *) malloc (sizeof (point  ) * pt);
   o->lines    = (line  *) malloc (sizeof (line ) * ln);
   o->polygons   = (polygon *) malloc (sizeof (polygon) * pg);

  /* LOAD UPTO 377 LOCATIONS PER OBJECT                                     */
   for (i = 0, j=46; i < lc; ++i, j += 12)
   {
       temploc    = &(o->locs[i]);              /* save temporary location */
       temploc->x = dpfloat(j);                 /* load world coordinates  */
       temploc->y = dpfloat(j + 4);
       temploc->z = dpfloat(j + 8);
   }

  /* LOAD UPT 2047 POINTS PER OBJECT                                        */
   if (pt)
   {
       more_data();
       for (i = 0, j=2 ; i < pt; ++i, j += 4)
       {
           temppt       = &(o->points[i]); /* set temporary point location */
           temppt->color = DPLONG(j);      /* get point color              */
           temppt->locn  = DPLONG(j + 2);  /* get point location           */
       }
   }

  /* LOAD UPTO 1364 LINES                                                   */
   if (ln)
   {
       more_data();
       for (i = 0, j=2 ; i < ln; ++i, j += 6)
       {
           templn            = &(o->lines[i]);    /* set temporary line    */
           templn->color     = DPLONG(j);         /* get color             */
           templn->startlocn = DPLONG(j + 2);     /* get starting location */
           templn->endlocn   = DPLONG(j + 4);     /* get ending location   */
       }
   }

  /* LOAD ONE POLYGON AT A TIME                                             */
   if (pg)
   {
       for (i = 0; i < pg; ++i)
       {
           more_data();
           temppg          = &(o->polygons[i]);  /* set temporary polygon  */
           temppg->color   = DPLONG(2);          /* get color              */
           l               = DPLONG(4);          /* get number of verteces */
           temppg->vertnum  = l;                 /* set number of verteces */

          /* allocate space for vertex location list                        */
           temppg->vertlocn =  (long *) malloc (sizeof (long) * l);


           for (k = 0, j = 6;  k < l; ++k, j += 2)         /* load verteces */
           {
               temppg->vertlocn[k] = DPLONG(j);      /* set vertex location */
           }
       }
   }
}

*************************************************************************
```

```
*************************************************************************

-->Listing 11: TMS320C30 Screen Coordinate Calculation Routine

void screen_object(o)
register struct object *o;
{
   register long i,j;                  /* temporary and looping variables   */
   register loc *temploc;              /* temporary location pointer        */
   register struct object *tempob;     /* temporary object pointer          */
   register float x,y;                 /*  co-ordinate floating point values */
   float z,d;                          /*  and perspective constant          */

   tempob = o->parent;              /* set temporary object to parent object  */

  /* COMPUTE PARENT MATRIX                                                   */
  /* if object is universe set parent matrix to transform matrix r           */
   if (o == universe)
   {
       for(i = 0; i < 3; ++i) for(j = 0; j < 4; ++j)  o->p[i][j] = o->r[i][j];
   }

  /* otherwise p matrix is product of r matrix and parent's p matrix         */
   else for(i = 0; i < 3; ++i)
   {
      o->p[i][0] = o->r[0][0] * tempob->p[i][0] + o->r[1][0] * tempob->p[i][1]
                 + o->r[2][0] * tempob->p[i][2];
      o->p[i][1] = o->r[0][1] * tempob->p[i][0] + o->r[1][1] * tempob->p[i][1]
                 + o->r[2][1] * tempob->p[i][2];
      o->p[i][2] = o->r[0][2] * tempob->p[i][0] + o->r[1][2] * tempob->p[i][1]
                 + o->r[2][2] * tempob->p[i][2];
      o->p[i][3] = o->r[0][3] * tempob->p[i][0] + o->r[1][3] * tempob->p[i][1]
                 + o->r[2][3] * tempob->p[i][2] + tempob->p[i][3];
   }

  /* COMPUTE SCREEN COORDINATES                                              */
   j = o->locnum;                                 /* get number of locations */
   for (i = 0; i < j; ++i)
   {
       temploc = &(o->locs[i]);                     /* set temporary location */

      /* save global coordinates                                             */
       x = temploc->x;          y = temploc->y;             z = temploc->z;

      /* calculate z value, add offset of 5, and invert for perspective      */
    d = 1/(x * o->p[2][0] + y * o->p[2][1] + z * o->p[2][2] + o->p[2][3] + 10);

      /* calculate transformed x and y, add perspective, and scale to screen*/
       k = (long) ((x * o->p[0][0] + y * o->p[0][1]
                  + z * o->p[0][2] +     o->p[0][3]) * d * 200);
       l = (long) ((x * o->p[1][0] + y * o->p[1][1]
                  + z * o->p[1][2] +     o->p[1][3]) * d * 200);

      /* clip to a 16 bit integer                                            */
       if      (k > 32000) k =  32000; else if (k < -32000) k = -32000;
       if      (l > 32000) l =  32000; else if (l < -32000) l = -32000;

      /* set screen coordinates                                              */
       temploc->a = k;                 temploc->b = l;
   }

  /* screen all internal objects                                             */
   j = o->obnum;
   for (i = 0; i <= j; ++i) screen_object(o->objects[i]);

*************************************************************************

*************************************************************************

-->Listing 12: TMS320C30 Transformation Matrix Evaluation Routine

matrix()
{
   register float        cost, sint;           /* transform temporary     */
   float                 coso, sino, cosp, sinp;/* variables              */
   register struct object *o;

   o = to;
   cost = cos(o->theta);
   sint = sin(o->theta);
   coso = cos(o->omega);
   sino = sin(o->omega);
   cosp = cos(o->phi);
   sinp = sin(o->phi);
   o->r[0][0] = o->sx * cost * coso;
   o->r[0][1] = - o->sy * sint * coso;
   o->r[0][2] = o->sz * sino;
   o->r[0][3] = (o->dx * cost - o->dy * sint) * coso + o->dz * sino;
   o->r[1][0] = o->sx * (sint * cosp + cost * sino * sinp);
   o->r[1][1] = o->sy * (cost * cosp - sint * sino * sinp);
   o->r[1][2] = - o->sz * coso * sinp;
   o->r[1][3] =   ((o->dx * cost - o->dy * sint) * sino - o->dz * coso) * sinp
              + (o->dx * sint + o->dy * cost) * cosp;
   o->r[2][0] = o->sx * (sint * sinp - cost * sino * cosp);
   o->r[2][1] = o->sy * (cost * sinp + sint * sino * cosp);
   o->r[2][2] = o->sz * coso * cosp;
   o->r[2][3] =   ((- o->dx * cost + o->dy * sint) * sino + o->dz * coso)
              * cosp           +  (o->dx * sint + o->dy * cost) * sinp;
}

*************************************************************************
```

```
*************************************************************************

-->Listing 13: TMS320C30 Object Deletion Routine

void delete_object (o)
register struct object *o;
{
   register long i, j;                          /* temporary, looping variables */

   free (o->locs);                              /* delete location array        */
   free (o->points);                            /* delete point array           */
   free (o->lines);                             /* delete line array            */
   j = o->pgnum;                                /* get number of polygons       */
   for (i = 0; i <= j; ++i) free (o->polygons[i].vertlocn); /* delete           */
   for (i = 0; i <= j; ++i) free (o->polygons);             /*  polygons        */
   j = o->obnum;                             /* get number of daughter objects */
   for (i = 0; i <= j;++i) delete_object(o->objects[i]); /* delete objects      */
   free (o);                                            /* delete object       */

*************************************************************************


*************************************************************************

-->Listing 14: TMS320C30 Request for Additional Data in Object Load

void more_data()
{
          ACKNOWLEDGE = 127;          /* request more data                  */
          while(COMMAND != 127);      /* wait for more data                 */
          ACKNOWLEDGE = 1;            /* restore old acknowledge            */
          while(COMMAND != 0);        /* wait for PC to resume old command */


*************************************************************************
```

```
*************************************************************************

-->Listing 15: TMS320C30 Object Drawing Routine

void draw_object (o)
register struct object *o;
{
   register long    i;                          /* temporary, looping variable */
   register loc     *temploc;                   /* temporary location pointer  */
   point            *temppt;                    /* temporary point pointer     */
   register line    *templn;                    /* temporary line pointer      */
   polygon          *temppg;                    /* temporary point pointer     */
   register long *hstdata = (long *) 0x805002; /* 340 host data register      */
   register long *hstcntl = (long *) 0x805003; /* 340 host control register   */
   register j = o->lnnum;                       /*-temporary, looping variable */

  /* DRAW ANY LINES                                                          */
   if (j)
   {
       while (HOSTCNTL != CTLFREE);             /* wait till 340 is free       */
       *hstdata = 123;                          /* send command to draw object */
       *hstcntl = CTLREQ;                       /* request service from 340    */
       *hstdata = j;                            /* send number of lines        */
       for(i=0; i < j; ++i)                     /* send lines                  */
       {
           templn  = &(o->lines[i]);                    /* save line pointer */
           *hstdata = templn->color;                    /* send color        */
           *hstdata = o->locs[templn->startlocn].a;     /* send start        */
           *hstdata = o->locs[templn->startlocn].b;     /*   coordinates     */
           *hstdata = o->locs[templn->endlocn].a;       /* send end          */
           *hstdata = o->locs[templn->endlocn].b;       /*   coordinates     */
       }
       while(HOSTCNTL != CTLACK);    /* wait for 340 to acknolwedge request */
       *hstcntl = CTLWITH;           /* withdraw request                    */
   }

  /* DRAW ANY POINTS                                                         */
   j = o->ptnum;                     /* get number of points                */
   if (j)
   {
       while (HOSTCNTL != CTLFREE);             /* wait till 340 is free       */
       *hstdata = 1;                            /* send command to draw object */
       *hstcntl = CTLREQ;                       /* request service from 340    */
       *hstdata = j;                            /* send number of points       */
       for(i=0; i < j; ++i)                     /* send points                 */
       {
           temppt = &(o->points[i]);              /* save point pointer       */
           *hstdata = temppt->color;              /* send color               */
           *hstdata = o->locs[temppt->locn].a;  /* send screen coordinates    */
           *hstdata = o->locs[temppt->locn].b;
       }
       while(HOSTCNTL != CTLACK);    /* wait for 340 to acknolwedge request */
       *hstcntl = CTLWITH;           /* withdraw request                    */
   }
```

```
/* DRAW ANY POLYGONS                                                    */
 l = o->pgnum;
 if (l)
 {
    for(i = 0; i < l; ++i)            /* draw polygons                  */
    {
         temppg = &(o->polygons[i]);  /* wait till 340 is free          */
         j = temppg->vertnum;         /* send command to draw object    */
         while (HOSTCNTL != CTLFREE); /* request service from 340       */
         *hstdata = 5;                /* send number of points          */
         *hstcntl = CTLREQ;           /* send points                    */
         *hstdata = temppg->color;    /* send color                     */
         *hstdata = j;                /* send number of verteces        */

        /* send point connect list (0,1 , 1,2 , 2,3 .... j-2,j-1 , j-1,0 */
         *hstdata = 0;
         for(k = 1; k < j; ++k)
         {
             *hstdata = k;              *hstdata = k;
         }
         *hstdata = 0;

        /* send vertex location list                                    */
         for(k = 0; k < j; ++k)
         {
             temploc = &(o->locs[temppg->vertlocn[k]]); /* save point    */
             *hstdata = temploc->a;       *hstdata = temploc->b;
         }
         while(HOSTCNTL != CTLACK); /* wait for 340 to acknolwedge request*/
         *hstcntl = CTLWITH;       /* withdraw request                 */
    }
 }

/* DRAW ANY DAUGHTER OBJECTS                                            */
 j = o->obnum;                                  /* get daughter objects */
 for (i = 0; i <= j; ++i) draw_object(o->objects[i]);
```

```
**********************************************************************

**********************************************************************
```

-->Listing 16: TMS34010 Point Structure

```
typedef struct                                                  /* POINT */
{
   short color;  /* point color                                        */
   short x;      /* x co-ordinate                                      */
   short y;      /* y co-ordinate                                      */
} point;
```

```
**********************************************************************
```

```
**********************************************************************
```

-->Listing 17: TMS34010 Line Structure

```
typedef struct                                                  /* LINE */
{
    short color; /* line color                                         */
    short x1;    /* x co-ordinate of starting point                    */
    short y1;    /* y co-ordinate of starting point                    */
    short x2;    /* x co-ordinate of end point                         */
    short y2;    /* y co-ordinate of end point                         */
} line;
```

```
**********************************************************************

**********************************************************************
```

-->Listing 18: TMS34010 Color Array

```
long color[16] = {
   CC0, CC1, CC2, CC3, CC4, CC5, CC6, CC7, CC8, CC9, CC10, CC11, CC12,
   CC13, CC14, CC15};
```

```
**********************************************************************

**********************************************************************
```

*-->Listing 19: TMS34010 Color Palette

```
short mypalet[16] = {
   0x0000, 0xF000, 0x00F0, 0xF0F0, 0x0F00, 0xFF00, 0x0FF0, 0xFFF0,
   0x0AF0, 0x0900, 0xFA70, 0xF4A0, 0x17B0, 0x6660, 0x9990, 0xBBB0 };
```

```
**********************************************************************
```

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

-->Listing 20: TMS34010 Main Command Execution Routine

main()
{
   register line *templn;              /* temporary line pointer            */
   register point *temppt;             /* temporary point pointer           */
   register short tempint;             /* temporary integer                 */
   register short i;                   /* looping variable                  */
   line  *lines;                       /* pointer to line array             */
   point *points;                      /* pointer to point array            */
   short *hstadrh, *hstadrl, * hstctll, *number, *pgnum, *pointer, adrl, adrh;
   *((short *) 0x04000000) = 0x0001;  /* turn on shadow ram                 */
   *((short *) 0xC00000B0) &= 0x7FFF; /* enable cache                       */
   hstctll = (short *) 0xC00000F0;    /* host control register low byte     */
   hstadrh = (short *) 0xC00000E0;    /* host address register high word    */
   hstadrl = (short *) 0xC00000D0;    /* host address register low word     */
   pointer = (short *) 0xFFF00000;    /* pointer to beginning of shadow ram */
   lines  = (line  *) (0xFFF00020);   /* starting point of line array       */
   points = (point *) (0xFFF00020);   /* starting point of point array      */
   pgnum  = (short *) (0xFFF00020); /*location of number of polygon verteces*/
   number = (short *) (0xFFF00010);   /* number of primitives to draw       */
   adrl = (short) ((long) pointer)  & 0x0000FFFF);
   adrh = (short) (((long) pointer) >> 16) & 0x0000FFFF)
   init_video(1);    /* configure for a NEC MULTISYNC, non-interlaced, 60Hz */
   init_grafix();                      /* initialize graphics environment   */
   init_screen();                      /* initialize screen                 */
   init_vuport();                      /* initialize viewing window         */
   set_origin(320,240);                /* place origin at center of screen  */
   *hstadrh = adrh;                    /* reset start data address          */
   *hstadrl = adrl;
   *hstctll = 0;                       /* turn off any command to the 340   */
   for (;;)
   {
       while (*hstctll != 0x0003);     /* wait for request from the C30     */
       *hstctll = 0x0030;              /* acknowledge request               */
       while (*hstctll != 0x0030);     /*wait for c30 to load data & withdraw*/
       switch (*pointer)               /* decode command                    */
       {
           case 123:                                       /* DRAW LINES    */
               tempint = *number;                          /* get number    */
               for (i = 0; i < tempint; ++i)               /*   of lines    */
               {
                   templn = &(lines[i]);                   /*set line point*/
                   set_color1(color[templn->color]);       /* set color     */
                   draw_line( templn->x1,                  /* draw line     */
                              templn->y1,
                              templn->x2,
                              templn->y2);
               }
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               break;
           case 1:                                         /* DRAW POINTS   */
               tempint = *number;                          /* get number of*/
               for (i=0; i < tempint; ++i)                 /*     points    */
               {
                   temppt = &(points[i]);                  /* save point    */
                   set_color1(color[temppt->color]);       /* set colors    */
                   draw_point( temppt->x,                  /* draw point    */
                               temppt->y);
               }
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               break;
           case 3:                                  /* SET SCREEN BACKGROUND */
               new_screen(color[*number],mypalet); /* clear screen          */
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               break;
           case 4:                                  /* SET BACKGROUND BLACK  */
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               new_screen(0,mypalet); /* clear screen                      */
               break;
           case 5:                                  /* DRAW A FILLED POLYGON */
               set_color1(color[*number]);         /* set polygon color     */
               tempint = *pgnum;                   /* get number of verteces*/
               fill_polygon(tempint,               /* fill polygon          */
                            (short *) (pointer + 3),
                            (short *) (pointer + 3 + (tempint << 1)));
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               break;
           case 6:                                  /* WAIT FOR COMPLETE SCREEN RESCAN */
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               wait_scan(0);          /*wait till scan reaches top of screen*/
               wait_scan(479);    /*wait till scan reaches bottom (line 479)*/
               break;
           default:
               *hstadrh = adrh;       /* reset start data address          */
               *hstadrl = adrl;
               *hstctll = 0;          /* turn off any command to the 340   */
               break;
       }
   }
}

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

```
*************************************************************************

-->Listing 21: PC Object Loading Data Structure

typedef struct
{
   short ptnum;                          /* number of points (location)    */
   short dtnum;                          /* number of drawn dots           */
   short lnnum;                          /* number of lines                */
   short pgnum;                          /* number of filled polygons      */
   float sx;     float sy;   float sz;   /* scale factors                  */
   float dx;     float dy;   float dz;   /* offset factors                 */
   float theta;  float phi;  float omega; /* angles of rotation             */
} trans;

*************************************************************************


*************************************************************************

-->Listing 22: PC Communications Macros

#define DATASHORT(a) *((unsigned short *) (dual_port + a))
#define DATAFLOAT(a) *((float *) (dual_port + a))
#define COMMAND      *dual_port
#define ACKNOWLEDGE  *((unsigned char *) 0xE0008001)

*************************************************************************


*************************************************************************

-->Listing 23: PC Global Variables

char *dual_port;                    /* dual port sram connecting to C30 SWDS*/
trans *data;

*************************************************************************
```

```
*************************************************************************

-->Listing 24: PC Targeted Object Adjustment Routine

void adjust_object(sx, sy, sz, dx, dy, dz, theta, phi, omega)
double sx, sy, sz, dx, dy, dz, theta, phi, omega;
{
   while(ACKNOWLEDGE != 0);
   DATAFLOAT(2)  = sx;     DATAFLOAT(6)  = sy;   DATAFLOAT(10) = sz;
   DATAFLOAT(14) = dx;     DATAFLOAT(18) = dy;   DATAFLOAT(22) = dz;
   DATAFLOAT(26) = theta;  DATAFLOAT(30) = phi;  DATAFLOAT(34) = omega;
   COMMAND = 5;
   while(ACKNOWLEDGE != 5);
   COMMAND = 0;
}

*************************************************************************


*************************************************************************

-->Listing 25: PC Routine to Set Parameters for an Object Load

void set_parameters(sx, sy, sz, dx, dy, dz, theta, phi, omega)
double sx, sy, sz, dx, dy, dz, theta, phi, omega;
{
   while (ACKNOWLEDGE  != 0);          /* wait for C30 to be free          */
   data->sx    = sx;      data->sy    = sy;    data->sz    = sz;
   data->dx    = dx;      data->dy    = dy;    data->dz    = dz;
   data->theta = theta;   data->phi   = phi;   data->omega = omega;
}

*************************************************************************


*************************************************************************

-->Listing 26: PC Routine to Target Parent of Current Target Object

void target_parent()
{
   while(ACKNOWLEDGE  != 0);           /* wait for C30 to be free          */
   COMMAND = 3;                        /* command to target parent object  */
   while(ACKNOWLEDGE  != 3);           /* wait for C30 to acknowlege request*/
   COMMAND = 0;                        /* withdraw request                 */
}

*************************************************************************
```

```
*********************************************************************

-->Listing 27: PC Routine to Target a Child of Current Target Object

void target_child(x)
int x;
{
      while(ACKNOWLEDGE  != 0);      /* wait for C30 to be free           */
      DATASHORT(2) = x;              /* target 1st daughter object        */
      COMMAND = 2;                   /* command to target daughter object */
      while(ACKNOWLEDGE  != 2);      /* wait for C30 to acknowlege request*/
      COMMAND = 0;                   /* withdraw request                  */
}

*********************************************************************


*********************************************************************

-->Listing 28: PC Routine to Redraw Screen

void draw_object()
{
      while(ACKNOWLEDGE  != 0);      /* wait for C30 to be free           */
      COMMAND = 7;                   /* command to compute screen co-ords */
      while(ACKNOWLEDGE  != 7);      /* wait for C30 to acknowlege request*/
      COMMAND = 0;                   /* withdraw request                  */
      while(ACKNOWLEDGE  != 0);      /* wait for C30 to be free           */
      COMMAND = 6;                   /* command to draw screen            */
      while(ACKNOWLEDGE  != 6);      /* wait for C30 to acknowlege request*/
      COMMAND = 0;                   /* withdraw request                  */
}

*********************************************************************
```

```
*********************************************************************

-->Listing 29: PC Routine to Load the Primitives of a Wireframe Cube

void cube(c)
long c;
{
   data->ptnum = 8;                  /* number of points (cube verteces)  */
   data->dtnum = 0;                  /* no dots                           */
   data->lnnum = 12;                 /* twelve lines     (cube edges)     */
   data->pgnum = 0;                  /* no filled polygons                */
/* ----X COORDINATE----  ----Y COORDINATE----   ----Z COORDINATE----      */
   DATAFLOAT(46)  =  1;  DATAFLOAT(50)  =  1;   DATAFLOAT(54)  =  1;
   DATAFLOAT(58)  =  1;  DATAFLOAT(62)  = -1;   DATAFLOAT(66)  =  1;
   DATAFLOAT(70)  =  1;  DATAFLOAT(74)  = -1;   DATAFLOAT(78)  = -1;
   DATAFLOAT(82)  =  1;  DATAFLOAT(86)  =  1;   DATAFLOAT(90)  = -1;
   DATAFLOAT(94)  = -1;  DATAFLOAT(98)  =  1;   DATAFLOAT(102) =  1;
   DATAFLOAT(106) = -1;  DATAFLOAT(110) = -1;   DATAFLOAT(114) =  1;
   DATAFLOAT(118) = -1;  DATAFLOAT(122) = -1;   DATAFLOAT(126) = -1;
   DATAFLOAT(130) = -1;  DATAFLOAT(134) =  1;   DATAFLOAT(138) = -1;
   COMMAND = 1;                      /* command to load object            */
   while (ACKNOWLEDGE != 1);         /* wait for C30 to acknowledge request*/
   COMMAND = 0;                      /* withdraw request                  */
   while (ACKNOWLEDGE != 127);       /* wait for C30 request lines        */
   COMMAND = 127;                    /* command to load lines             */
/* LINE COLOR-------   START POINT-------   ENDPOINT----------            */
   DATASHORT(2)  = c;  DATASHORT(4)  = 0;  DATASHORT(6)  = 1;
   DATASHORT(8)  = c;  DATASHORT(10) = 1;  DATASHORT(12) = 2;
   DATASHORT(14) = c;  DATASHORT(16) = 2;  DATASHORT(18) = 3;
   DATASHORT(20) = c;  DATASHORT(22) = 3;  DATASHORT(24) = 0;
   DATASHORT(26) = c;  DATASHORT(28) = 4;  DATASHORT(30) = 5;
   DATASHORT(32) = c;  DATASHORT(34) = 5;  DATASHORT(36) = 6;
   DATASHORT(38) = c;  DATASHORT(40) = 6;  DATASHORT(42) = 7;
   DATASHORT(44) = c;  DATASHORT(46) = 7;  DATASHORT(48) = 4;
   DATASHORT(50) = c;  DATASHORT(52) = 0;  DATASHORT(54) = 4;
   DATASHORT(56) = c;  DATASHORT(58) = 1;  DATASHORT(60) = 5;
   DATASHORT(62) = c;  DATASHORT(64) = 2;  DATASHORT(66) = 6;
   DATASHORT(68) = c;  DATASHORT(70) = 3;  DATASHORT(72) = 7;
   while (ACKNOWLEDGE  != 1);        /* wait for C30 to resume loading    */
   COMMAND = 0;                      /* show no requests                  */
}

*********************************************************************
```

```
*************************************************************************

-->Listing 30: PC Main Routine to Draw a "Planetary System of" Cubes

main()
{
   register int x;
   dual_port = (char *)  0xE0008000; /* location of dual port sram         */
   data      = (trans *) 0xE0008002; /* location of object data            */
   COMMAND = 0;
   set_parameters(.0001,.0001,.0001,0.,0.,0.,0.,0.,0.);
   cube(3);
   set_parameters(.4,.4,.4,0.,0.,8.,0.,0.,0.);
   cube(2);
   set_parameters(.2,.2,.2,0.,5.,0.,0.,0.,0.);
   cube(6);
   target_parent();
   set_parameters(.2,.2,.2,0.,-5.,0.,0.,0.,0.);
   cube(4);
   target_parent();
   target_parent();
   set_parameters(.3,.3,.3,0.,0.,6.,0.,0.,0.);
   cube(5);
   target_parent();
   set_parameters(.3,.3,.3,0.,6.,0.,0.,0.,0.);
   cube(1);
   target_parent();
   set_parameters(.3,.3,.3,0.,0.,-6.,0.,0.,0.);
   cube(5);
   target_parent();
   set_parameters(.3,.3,.3,0.,-6.,0.,0.,0.,0.);
   cube(1);
   target_parent();
   for(x = 0; x < 1000; ++x)
   {
       adjust_object(1.00926,1.00926,1.00926,0.,0.,0.,0.,0.,.2);
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(2);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(3);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(4);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.2,0.);
       target_parent();
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,-.4);
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,.4,0.,0.);
       target_parent();
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,.4,0.,0.);
       target_parent();
       target_parent();
       screen_object();
       draw_screen();
   }
   for(x =0; x < 1100; ++x)
   {
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.005,.2);
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(2);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(3);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(4);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,.25,0.);
       target_parent();
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,0.,-.4);
       target_child(0);
       adjust_object(1.,1.,1.,0.,0.,0.,.3,0.,0.);
       target_parent();
       target_child(1);
       adjust_object(1.,1.,1.,0.,0.,0.,.3,0.,0.);
       target_parent();
       target_parent();
       screen_object();
       draw_screen();
   }

*************************************************************************
```

Part VI. Tools

The TMS320C30 Applications Board Functional Description

Tony Coomes—Software Development Systems
Nat Seshan—Digital Signal Processor Products
Semiconductor Group
Texas Instruments

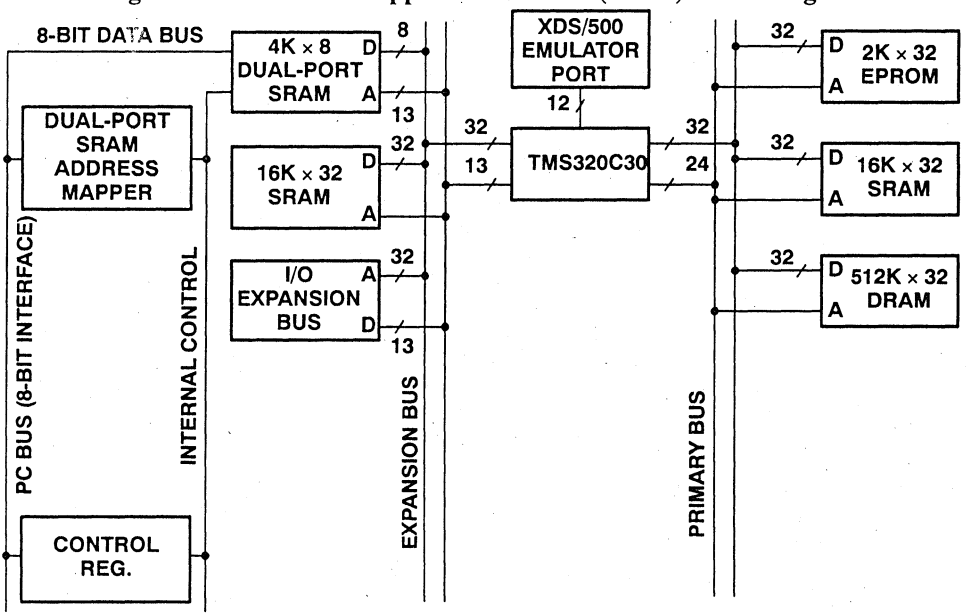
Introduction

This report describes the architecture of the TMS320C30 Applications Board (APPB), which is part of the TMS320C30 XDS1000 Development System. The XDS1000 is an in-circuit emulation tool for TMS320C30 hardware/software system development. The APPB was designed with two goals: to provide a basic platform for software development and to provide a variety of interfaces to the TMS32C30. There are four key interfaces used on the APPB:

1) SRAM
2) EPROM
3) Dual-port SRAM
4) DRAM

The SRAM and EPROM interfaces on the APPB are quite simple; thus, this report focuses on the dual-port SRAM and the DRAM interfaces. Figure 1 shows a basic block diagram of the APPB.

Figure 1. TMS320C30 Applications Board (APPB) Block Diagram



The APPB features include the following:

- TMS320C30/host communications via a designated, relocatable 4K-byte dual-bus SRAM memory block.
- 16K-words (64K-bytes) zero wait-state SRAM on the TMS320C30 primary bus (STRB).
- 2K-words of one wait-state EPROM for interrupt and reset vectors on the TMS320C30 primary bus.
- 16K-words (64K-bytes) zero wait-state SRAM on the TMS320C30 expansion bus (MSTRB). The SRAM can be selected in either one of two 8K-word banks.

- I/O expansion bus.
- 512K-words of DRAM on the TMS320C30 primary bus.
- Emulation port.
- IBM PC, PC/XT, PC/AT support.

The remainder of this document describes each interface in more detail.

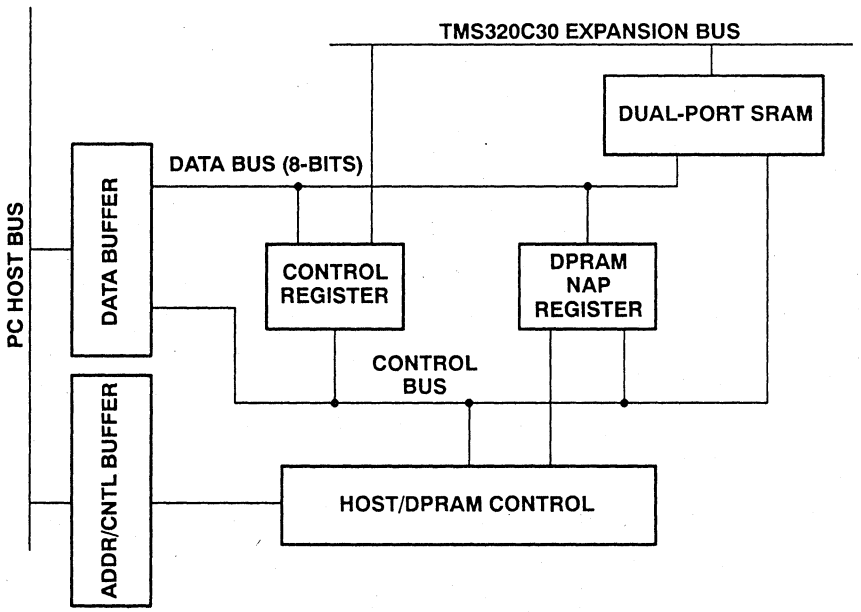
Host/TMS320C30 Interface

The host/TMS320C30 interface is composed of two basic blocks, the dual-port SRAM and the control logic. The control logic consists of address decoding, a read/write control register, and a write-only mapping register. The control registers are mapped into the host I/O space as shown in Table 1. Figure 2 is a block diagram of the host interface.

Table 1. Host I/O Memory Locations for Control Registers

Host I/O Memory Locations	Contents
0330 – 0337	Semaphores (LSB is the only valid bit)
0338	Dual-port SRAM mapping register Q
0339	Control register R

Figure 2. Host Interface Block Diagram



One of the major problems in developing an application for a PC is finding a block of memory that does not conflict with other memory-mapped cards. To ease this problem, the dual port SRAM interface has been designed to be relocatable on 4K-byte boundries throughout the lower 1M-bytes of host memory space. A software example of how to map the dual-port SRAM into this space is given later in this report.

Writing a value to a hardware mapping register on the APPB relocates the dual-port SRAM. When a host memory access is generated, the value in the mapping register is compared to host address bits A12–A19. If they match, a dual-port SRAM access is allowed. To ensure PC and PC/XT compatibility, the dual-port SRAM can be located only in the lower 1M-bytes of host memory.

The APPB contains one general-purpose control register. This register is broken into two four-bit nibbles. The lower nibble can be read from and written to by the host and read by the TMS320C30. The upper nibble can be read from and written to by the TMS320C30 and read by the host. The lower nibble of the control register is cleared by any reset to or from the host PC. The upper nibble of the control register is cleared by any reset to the TMS320C30. The names of the APPB control register bits and host/TMS320C30 access capabilities are given in Table 2. Table 3 gives the control register bit definitions.

Table 2. APPB General-Purpose Control Register Bits

Bit	Name	Host Access	C30 Access
0	CINT	Write/Read	Read only
1	XINTCLR	Write/Read	Read only
2	DPSEL	Write/Read	Read only
3	SWRESET	Write/Read	Read only
4	XINT	Read only	Write/Read
5	CINTCLR	Read only	Write/Read
6	MBANK	Read only	Write/Read
7	MSWAP	Read only	Write/Read

Table 3. APPB General-Purpose Control Register Bit Definitions

Bit	Name	Function
0	CINT	Clears and disables interrupts from the TMS320C30 to the host (XINT). XINTCLR must be set to 1 before the TMS320C30 can generate an interrupt to the host. The host clears and reenables XINT by writing 0, then 1 to XINTCLR. On reset, XINTCLR is read as a 0.
1	XINTCLR	Interrupt (INT0) to the TMS320C30. The host may interrupt the TMS320C30 by setting this bit to 1. The TMS320C30 clears and re-enables the CINT by writing 0, then 1 to CINTCLR. The host cannot generate an interrupt to the TMS320C30 while CINTCLR = 0. On reset, CINT is read as a 0.
2	DPSEL	Dual-port SRAM select. When this bit is set to 1, the dual-port SRAM is memory-mapped in the 4K-byte space of the host PC specified by the 8-bit value in register Q. When DPSEL = 0, the dual-port SRAM will not be mapped in the host PC's address space. On reset, DPSEL is read as a 0.
3	SWRESET	TMS320C30 SWDS soft reset. SWRESET = 0 resets the TMS320C30 SWDS. SWRESET must be set to 1 to take the SWDS out of the reset state. On reset (power on), SWRESET is read as a 0.
4	XINT	Interrupt to the host PC. The TMS320C30 may interrupt the host by setting this bit to 1. The host clears and re-enables XINT by writing 0, then 1 to XINTCLR. The TMS320C30 cannot generate an interrupt to the host while XINTCLR = 0. On reset, XINT is read as a 0.
5	CINTCLR	Clears and disables interrupts from the the host to the TMS320C30 (CINT). CINTCLR must be set to 1 before the host can generate an interrupt to the TMS320C30. The TMS320C30 clears and re-enables CINT by writing 0, then 1 to CINTCLR. On reset, CINTCLR is read as a 0.
6	MBANK	Memory bank select. The 16K-word bank of memory on the TMS320C30 parallel I/O Bus (SRAM space 1) is mapped as two overlapping banks of 8K-words each. MBANK = 0 selects the lower 8K-words, MBANK = 1 selects the upper 8K-words. On reset, MBANK is read as a 0.
7	MSWAP	Memory Swap. The MSWAP bit is used to swap the address map for EPROM and SRAM space 0. MSWAP = 0 maps the EPROM at 000000h–003FFFh and SRAM space 0 at F00000h–F03FFFh. MSWAP = 1 maps the EPROM at F00000h–F03FFFh and SRAM space 0 at 00000h–003FFFh. On reset, MSWAP is read as a 0.

The last portion of the control section contains the dual-port SRAM semaphore registers. Semaphore registers are used to coordinate communications between the host and the TMS320C30. Note that these semaphores do not provide hardware protection of the memory array. Instead, they provide a basic means (via software control) to ensure that data can be accessed from both sides of the dual-port SRAM without being corrupted. A software example that uses the semaphores is presented later in this report.

SRAM and EPROM Interfaces

There are two SRAM interfaces on the APPB: one on the primary bus and one on the expansion bus. Both are implemented with eight 16K-bit × 4, 25-ns SRAMs that provide zero wait-state TMS320C30 operation at 32 MHz. The interfaces are quite simple and consist of a set of address buffers, termination resisters, and a PAL for address decode on the primary bus. Note that the TMS320C30 address lines are routed to various components scattered around the board and then to the primary bus expansion. To prevent line reflections on the SRAM addresses, buffers have been used to isolate the SRAM.

There are two special features on the APPB that apply to the SRAM:

1) You can swap the memory address ranges of the EPROM and the SRAM on the primary bus by setting or clearing the MSWAP bit previously described in Table 3.
2) There are two 8K-word pages of memory on the expansion bus.

By swapping the EPROM and SRAM, you can load in your own interrupt and reset vectors. Otherwise, you would have to remove the EPROMs and reprogram them with your own defined interrupt/reset vectors. The following code segment sets/clears the MSWAP bit.

```
#define EPROM           0               /* select EPROM */
#define SRAM            1               /* select SRAM */

sel_mswap(mem_type)
int mem_type;
{
   char *cntlreg = (char *)0x00805FF7; /* pointer to control reg */

  if (mem_type)     *cntlreg |= 0x80;    /* set MSWAP to 1 select SRAM */
     else           *cntlreg &= 0x7F;    /* set MSWAP to 0 select EPROM */
}
```

There are 16K-words of SRAM on the expansion bus; however, the TMS320C30 can directly access only 8K-words. Instead of wasting the unaddressable 8K-words, you can use a bank addressing bit (MBANK) in the APPB control register to select between the lower and upper 8K-word segments.

The following code segment selects the current bank of memory.

```
#define BANK0           0               /* select lower 8K */
#define BANK1           1               /* select upper8K */

sel_mbank(bank)
int bank;
{
   char *cntlreg = (char *)0x00805FF7; /* pointer to control reg */

    if (bank)       *cntlreg |= 0x40;     /* select bank 1 */
      else          *cntlreg &= 0xBF;     /* select bank 0 */
}
```

The APPB supports 2K-words of one wait-state EPROM on the primary bus for a boot loader and operating system support. As stated earlier, this EPROM is remappable.

DRAM Interface

The APPB provides a DRAM expansion module that is connected to the TMS320C30 primary bus. Historically, DRAM interfaces to DSP devices have not been popular because of interface

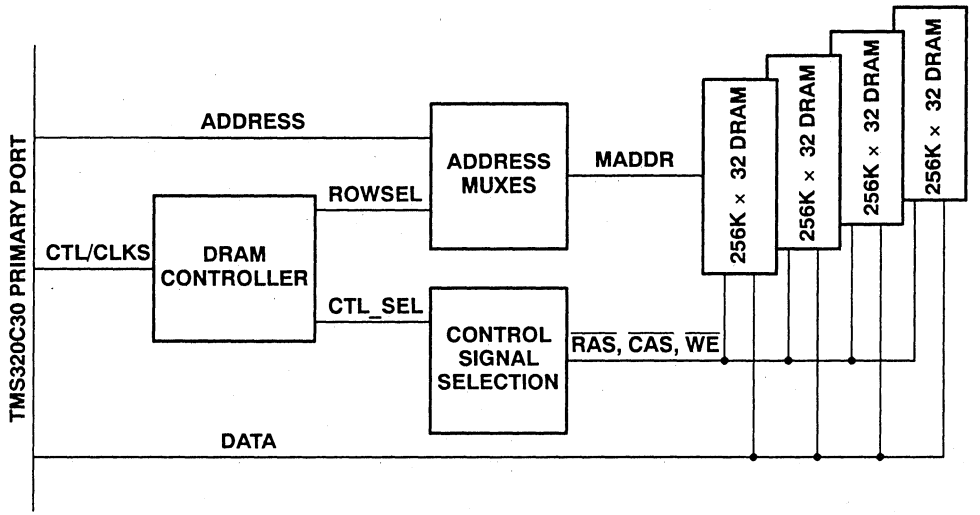
difficulty and limited processor address space. The TMS320C30 supplies solutions to both of those issues with its memory interface and 16M-words address space. Two areas of the TMS320C30 memory interface are most useful for DRAM design:

- Use of bank mode
- The ability to do continous reads while in a bank without deasserting the $\overline{\text{STRB}}$ signal

When you use these two features, it is quite simple to design a medium-speed interface to page-mode DRAMs.

The TMS320C30 DRAM module consists of four banks of memory, each bank 256K × 32 bits, that provide 1M-word (4M-bytes) of medium speed storage for the TMS320C30 (see Figure 3). The bank-switch function on the TMS320C30 provides fast page-mode access on back-to-back read cycles within a DRAM page. All address and control lines to the memory array are buffered and series-terminated for good signal quality. The memory array uses CAS-before-RAS refresh to reduce component count. There is no onboard refresh timer; instead, SDACK0 from the host PC provides a refresh request every 12–16 μs. The DRAM access/cycle times are summarized in Table 4.

Figure 3. TMS320C30 Bank Addressing



In Table 4, these definitions are assumed:

Access Time – Number of clocks from $\overline{\text{STRB}}$ active to data clocked into the TMS320C30.
Cycle time – Number of clocks between two back-to-back cycles (includes DRAM $\overline{\text{RAS}}$ precharge on non-page-mode cycles).

Table 4. TMS320C30 DRAM Access and Cycle Times

Mode	Access Time (clks)	Cycle Time (clks)
Read	3	5
Read (page mode)	3/2†	2
Write	3	4

† First page-mode access takes 3 clocks; the following accesses take 2 clocks each.

The four banks of DRAM are mapped into the TMS320C30 memory space at the address locations shown in Table 5.

Table 5. DRAM Bank Memory Locations in the TMS320C30 Memory Space

DRAM Memory Bank No.	TMS320C30 Memory Location
0 ($\overline{RAS0}$,$\overline{CAS0}$)	400000H–43FFFFH
1 ($\overline{RAS1}$,$\overline{CAS1}$)	440000H–47FFFFH
2 ($\overline{RAS2}$,$\overline{CAS2}$)	480000H–4BFFFFH
3 ($\overline{RAS3}$,$\overline{CAS3}$)	4C0000H–4FFFFFH

Memory decode for the DRAM module is performed in two steps:

1) The APPB main card provides a memory select to decode the board range of 400000H–4FFFFFh.
2) Bank decode is then provided on the DRAM module through TMS320C30 address bits A18 and A19.

The DRAM controller consists of a pair of registered PALs, several SSI gates, and a delay line (used to time DRAM row/column address multiplexing). DRAM timing is generated from PAL UE5 (see schematics in Appendix C), while address decoding and special refresh control are provided by PAL UD5. Both PALs are clocked off of a delayed H1 clock. The DRAM controller looks for every opportunity to generate page-mode cycles to the DRAM. The TMS320C30 leaves $\overline{STRB}$ low for back-to-back reads; the DRAM controller looks for this condition and cycles CAS while holding RAS low (i.e., DRAM page-mode access). When $\overline{STRB}$ goes high, the DRAM controller will take both RAS and CAS high to prepare for a new access. For proper operation, the TMS320C30 primary bus control register (refer to the Primary Bus Control Register subsection in the *Third-Generation TMS320 User's Guide*) must be set to operate off of the external ready signal and use a maximum bank size of 512 words (refer to the the Programmable Bank Switching subsection of the *Third-Generation TMS320 User's Guide*).

Figures 4 through 6 show the timing for the various DRAM cycles.

Figure 4. Page-Mode Read-Cycle Timing Diagram

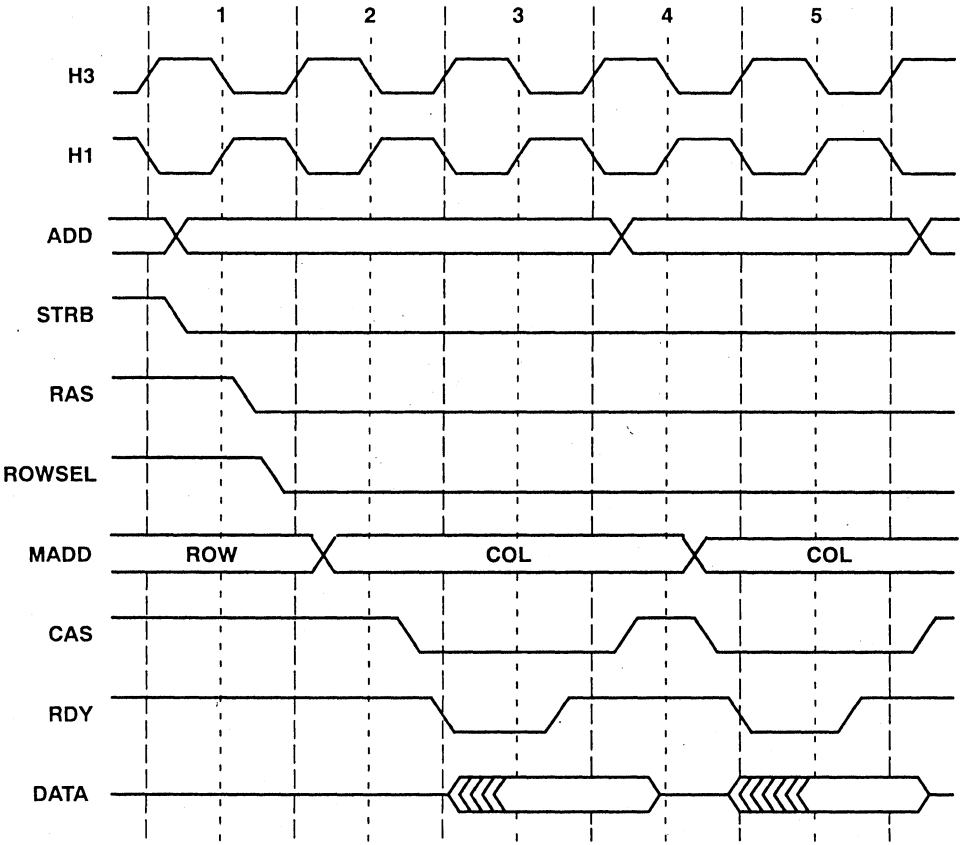


Figure 5. Single Write-Cycle Timing Diagram

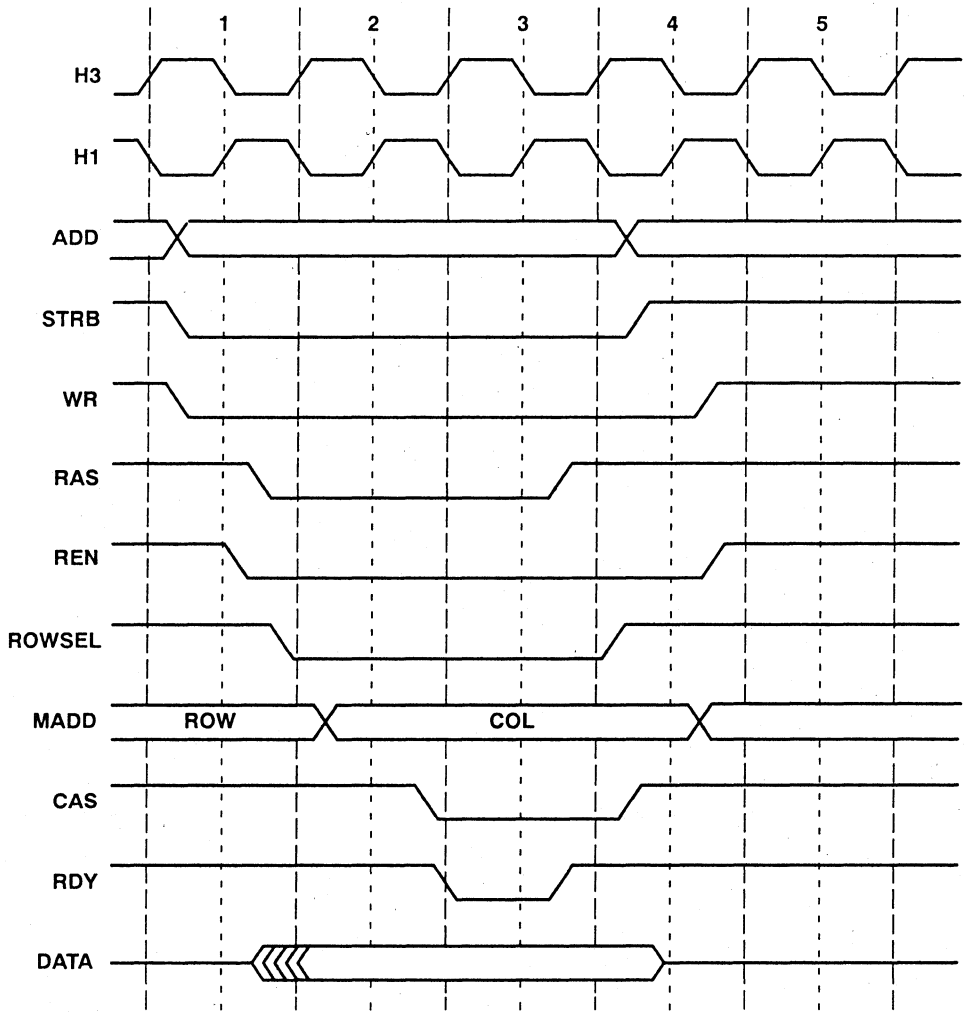
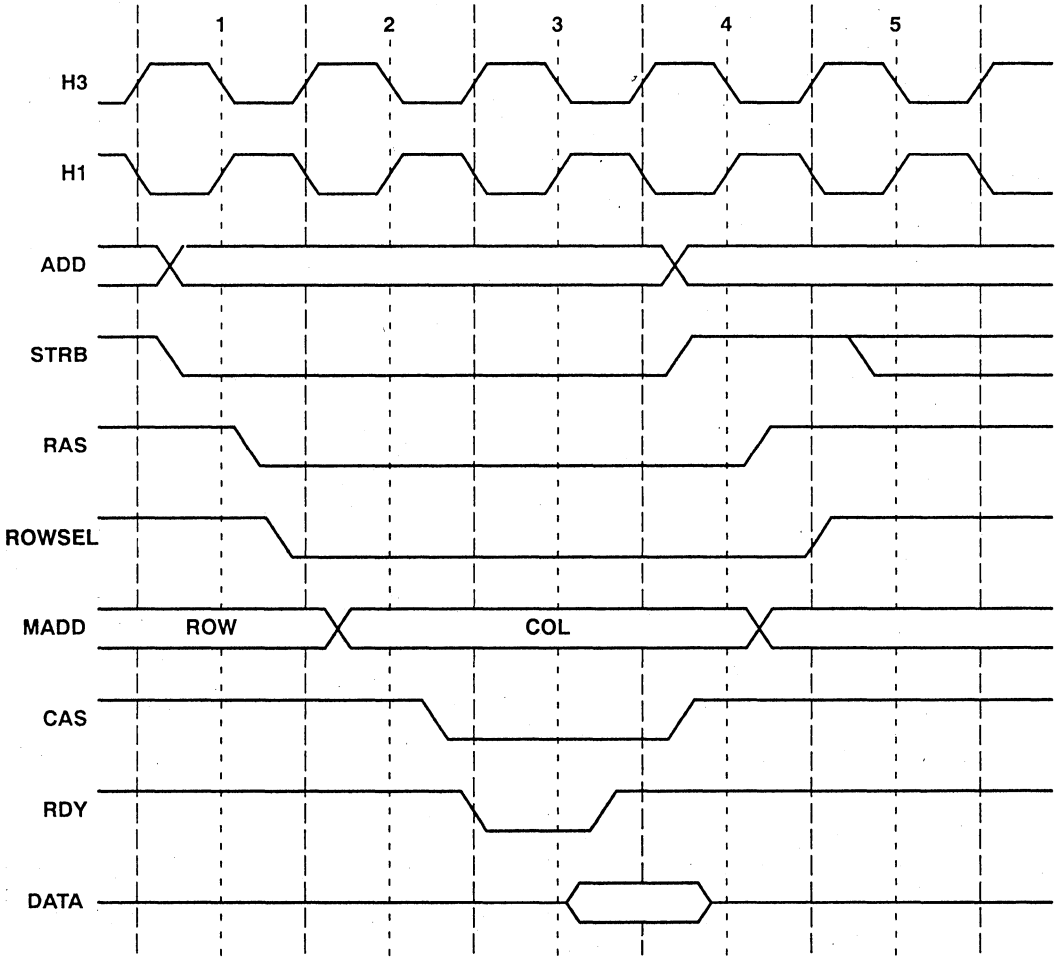


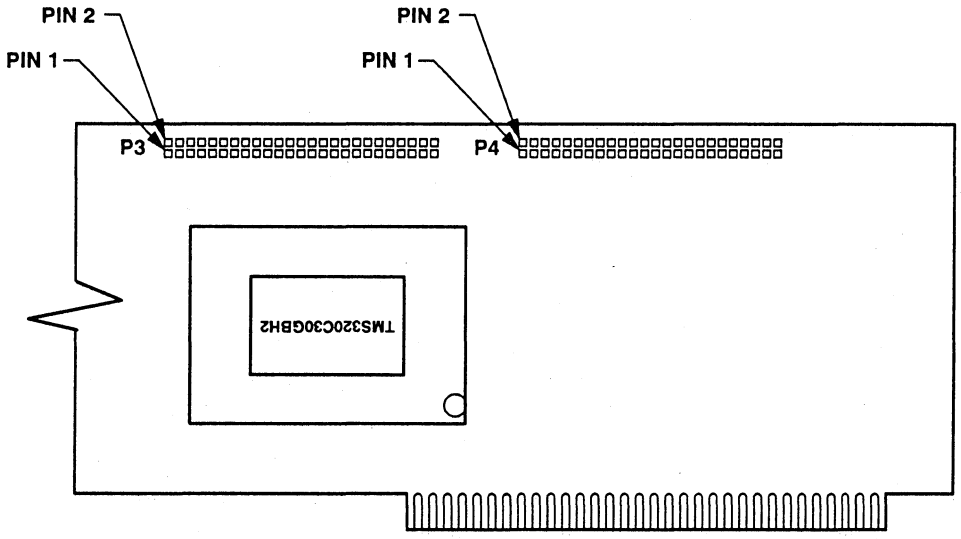
Figure 6. Single Read-Cycle Timing Diagram



Expansion Interface

The APPB's two expansion connectors contain the signals from the TMS320C30 expansion port, serial ports, flag pins, etc. Each 50-pin connector (P3 and P4 of Figure 7) is composed of a dual row of 25 pins located on 0.1-inch centers. These expansion connectors provide easy connection to other hardware via standard 50-wire flat ribbon cable. Figure 6 shows the orientation of the connectors. See schematic sheet 7 of Appendix C for pinout details.

Figure 7. TMS320C30 Applications Board



Dual-Port SRAM Interface

All communications between the TMS320C30 and the host occur through the dual-port SRAM, which is 4K-bytes deep, with 8 dedicated semaphore registers. On the host side, the dual-port memory array is memory-mapped, while the semaphores are I/O-mapped. On the TMS320C30 side, the dual-port SRAM is located on the expansion bus with the memory array mapped from 0x00804000–0x00804FFF and the semaphores mapped from 0x00805FF8–0x00805FFF. The host can directly access the dual-port SRAM without having to compensate for byte-wide access limitations. However, as the TMS320C30 can do only 32-bit accesses, the upper 24 bits of a data word are undefined. The TMS320C30 must therefore format data written to and read from the dual-port SRAM. A software example is given later in this report.

While dual-port SRAMs provide an excellent means for multiprocessor communications, a certain amount of software overhead is required to coordinate data flow. As might be expected, there are numerous methods for coordinating data flow. This application report presents a set of primitives that have been developed to form a basic communications protocol. The primitives are written entirely in C and have been tested on the XDS1000 with the simple test routine provided. Remember that there are numerous ways to do a communications protocol. The method shown in this report is not the best for all applications; it is simply a method that makes good use of the capability of the dual-port SRAM.

The following are basic ideas of the communications protocol developed for this applications report.

1) The dual-port memory is broken into eight equal segments. The first segment is used only for control structures and command passing. The remaining seven segments are used entirely for data passing. Segment size is set to 512 bytes. The number and size of segments can be changed at compile time if desired.

2) Each of the seven data segments is totally independent from any other data segment. However, only one processor can own a particular segment at any given time. The TMS320C30 and host can simultanously access the dual-port SRAM as long as both are not trying to access the same segment.
3) The host is the master; the TMS320C30 is the slave. The TMS320C20 polls the dual-port control segment to determine if the host has deposited a command. If a command is present, the TMS320C30 executes the command and then returns to polling.
4) Only the first semaphore register is used in the dual-port. Each processor uses this semaphore to gain access to the control segment. Access to the seven data memory segments are coordinated via the control structures, not the semaphores.
5) There are seven control structures in the control segment, one for each data segment. Each control structure consists of 22 bytes and are defined as follows:

Byte	Name	Definition
0	pflag	Buffer present (i.e., being used)
1	command	Command to execute
2	buf_stat	Status of the data buffer
3	nc	Reserved
4–7	count	Number of 32–bit words to transfer
8–11	addr	TMS320C30 to read/write data
12–21	message	Ten bytes reserved for message passing

Appendix A contains routines for the communication primitives used by the host and the TMS320C30. Appendix A1 contains routines for the PC side, Appendix A2 routines for the TMS320C30 side. Note that the routines on both sides have the same names and perform essentially the same function. Appendix A3 contains a memory map and description (TMS320C30 view). After the code has been compiled, use the following sequence to execute the test program:

1) Reset the XDS/1000:

```
 xreset   [RETURN]
c30reset  [RETURN]
```

2) Get into the emulator and load the TMS320C30 dual-port code.

```
emu30           [RETURN]        ; load emulator
xr                              ; reset the c30
lo              'file name'     ; load the object file
xd                              ; execute disconnect
[esc]                           ; escape to main menu
q 'yes'                         ; quit emulator
```

At this point, your dual bus code should be executing and waiting for a host input.

3) Execute host dual-port code.

```
'file name'
```

The host code will then print the numbers 0 through 25 to the screen.

Conclusion

This report has provided basic functional details of the TMS320C30 APPB. Because of their complexity, the DRAM and dual-port SRAM interfaces have been discussed. The features of the TMS320C30 allow it to encompass a wide range of interfaces. The TMS320C30 bank-switch mode and continuous strobe signal on back-to-back read cycles overcome traditional DSP/DRAM problems of interface difficulty and limited processor address space. A set of communications primitives routines to use with dual-port SRAM have been provided in Appendix A. These routines are written in C for ease of understanding and modification to meet individual needs.

Appendix A

TMS320C30 Application Board Routines, Memory Map and Description

```
/*************************************************************************/
/*                                                                       */
/*   APPENDIX A1                                                         */
/*                                                                       */
/*   TMS320C30 APPLICATION BOARD ROUTINES - PC SIDE                      */
/*                                                                       */
/*   Texas Instruments Inc.                                              */
/*   10/25/89                                                            */
/*                                                                       */
/*   Functions:                                                          */
/*                                                                       */
/*        int APPB_reset()        Reset APPB                             */
/*        int APPB_dpinit()       Intialize APPB.                        */
/*        int APPB_getsem()       Get access to semaphore bit N          */
/*        int APPB_relsem()       Release access to semaphore bit N      */
/*        int APPB_getctlblk()    Get a control block in DPRAM           */
/*        int APPB_relctlblk()    Release control block in DPRAM         */
/*        int APPB_getmemblk()    Get a block of memory from DPRAM       */
/*        int APPB_putmemblk()    Put a block of memory to DPRAM         */
/*                                                                       */
/*   All code was compiled with Microsoft C compiler version 5.1 using the  */
/*   large model. If small model is used, then pointers used to access the  */
/*   dual port SRAM would have to be declared and used as 'far' pointers     */
/*   (i.e. 32-bit pointer). Under the large model, all pointers are          */
/*   defaulted to 32 bits.                                               */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>

/*************************************************************************/
/*                                                                       */
/*   Constant definitions for the TMS320C30 Applications Board.          */
/*                                                                       */
/*************************************************************************/

#define     outport     outp
#define     inport      inp
#define     SEM_BASE    0x0330
#define     MAP_REG     0x0338
#define     CTL_REG     0x0339

#define     CINT        0x01
#define     XINTCLR_    0x02
#define     DPSEL       0x04
#define     SWRESET_    0x08
#define     XINT        0x10
#define     CINTCLR_    0x20
#define     MBANK       0x40
#define     MSWAP       0x80

#define     DPRAM_CTL   0xC9000000
#define     DPRAM_SEG   0xC9
#define     DPRAM_MEMBASE  0xC9000200

#define     DPRAM_SIZE     0x1000
#define     DPRAM_BLKS     7
#define     DPRAM_BLK_SIZE 512
#define     NUM_SEMS       8
#define     MAX_SEM_TIME   10000

#define     BUF_EMPTY      0
#define     BUF_FULL       1

#define     NOP            0x00
#define     HOST_MEM_WR    0x80
#define     HOST_MEM_RD    0x81

typedef     unsigned char  UCHAR;
typedef     unsigned short UINT;
typedef     unsigned long  ULONG;

typedef     struct
            {
                UCHAR   pflag;
                UCHAR   command;
                UCHAR   buf_stat;
                UCHAR   nc;
                ULONG   count;
                ULONG   addr;
                UCHAR   message[10];
            }DPCNTL;
```

```
/*************************************************************************/
/*                                                                       */
/*   Test program.                                                       */
/*                                                                       */
/*   Sequence:                                                           */
/*                                                                       */
/*   1)  Write a block of memory to the dual port.                       */
/*   2)  Read back the block of data from the dual port.                 */
/*                                                                       */
/*************************************************************************/

main()
{
   UINT  semnum[DPRAM_BLKS];
   int i;
   ULONG   memarray[25],mem2array[25];

   APPB_dpint();


   for(i=0;i<25;i++) {memarray[i] = (ULONG)i; mem2array[i] = 0UL;}

   if(APPB_putmemblk(25UL,memarray,0x00809900))
       printf("failed memory write\n");

   if(APPB_getmemblk(25UL,0x00809900,mem2array))
       printf("failed memory read\n");

   for(i=0;i<25;i++)  printf("value read %d\n",mem2array[i]);

   exit(0);
}
```

```
/*************************************************************************/
/*                                                                       */
/*   APPB_reset(),PC side                                                */
/*                                                                       */
/*   Reset APPB.                                                         */
/*                                                                       */
/*   Sequence:                                                           */
/*                                                                       */
/*   1)  Clear control register.                                         */
/*   2)  Set SWRESET_ to 1.                                              */
/*                                                                       */
/*************************************************************************/
int APPB_reset()
    {
       outport(CTL_REG,0);
       outport(CTL_REG,SWRESET_);
       return(0);
    }

/*************************************************************************/
/*                                                                       */
/*   APPB_dpint(), PC side                                               */
/*                                                                       */
/*   Sequence:                                                           */
/*                                                                       */
/*   1)  Set DPRAM semaphores to 1 (free).                               */
/*   2)  Set DPRAM mapping register.                                     */
/*   3)  Set DPRAM global enable bit to 1.                               */
/*                                                                       */
/*************************************************************************/
int APPB_dpint()
    {
       int i;
       UINT  semaddr = SEM_BASE;
       UCHAR *dpram  = (UCHAR *)DPRAM_CTL;

       for(i=0;i<8;i++)    outport(semaddr++,1);
       outport(MAP_REG,DPRAM_SEG);
       outport(CTL_REG,DPSEL | SWRESET_);
       return(0);
    }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_getsem(), PC side                                               */
/*                                                                       */
/*  Attempts to gain access of semaphore 'semnum'.                       */
/*  Return a 0 if successful, a -1 if failed.                            */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Write 0 to semaphore.                                            */
/*  2)  Decrement timoute, check for timeout = 0, or semaphore = 0.      */
/*  3)  Return pass/fail.                                                */
/*                                                                       */
/*************************************************************************/

int APPB_getsem(semnum)
      UINT  semnum;
   {
      UINT  semaddr = SEM_BASE + semnum;
      UINT  timeout = MAX_SEM_TIME;

      outport(semaddr,0);
      while( --timeout && (inport(semaddr) & 1));

      if(timeout) return(0);
      else return(-1);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_relsem(), PC side                                               */
/*                                                                       */
/*  Release semaphore at 'semnum'.                                       */
/*  Return a 0 if successful, a -1 if failed.                            */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Write 1 to semaphore.                                            */
/*  2)  Decrement timeout, check for timeout = 0, or semaphore = 1.      */
/*  3)  Return pass/fail.                                                */
/*                                                                       */
/*************************************************************************/

int APPB_relsem(semnum)
      UINT  semnum;
   {
      UINT  semaddr = SEM_BASE + semnum;
      UINT  timeout = MAX_SEM_TIME;

      outport(semaddr,1);
      while( --timeout && !(inport(semaddr) & 1));

      if(timeout) return(0);
      else return(-1);
   }
```

```
/************************************************************************/
/*                                                                      */
/*  APPB_getctlblk(), PC side                                           */
/*                                                                      */
/*  Find unused block of memory in the dual port.                       */
/*  Return a 0 if successful, a -1 if failed.                           */
/*                                                                      */
/*  Sequence                                                            */
/*                                                                      */
/*  1)  Search control structures for free block of memory.             */
/*  2)  If block free, set semnum to block index, return 0.             */
/*  3)  Else, return -1 (failed to find block).                         */
/*                                                                      */
/************************************************************************/

int APPB_getctlblk(semnum)
      UINT  *semnum;
   {
      int i;
      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;

      if(APPB_getsem(0))  return(-1);

      for(i=0;i<DPRAM_BLKS;i++)
          if(!dpctl[i].pflag)
          {
            dpctl[i].pflag   = 1;
            dpctl[i].command = NOP;
            dpctl[i].buf_stat = BUF_EMPTY;
            *semnum = i;
            if(APPB_relsem(0)) return(-1);
            else return(0);
          }

      APPB_relsem(0); return(-1);
   }
```

```
/************************************************************************/
/*                                                                      */
/*  APPB_relctlblk(), PC side                                           */
/*                                                                      */
/*  Release block of memory in the dual port.                           */
/*  Return a 0 if successful, a -1 if failed.                           */
/*                                                                      */
/*  Sequence                                                            */
/*                                                                      */
/*  1)  Null out the control structure.                                 */
/*  2)  Return.                                                         */
/*                                                                      */
/************************************************************************/
int APPB_relctlblk(semnum)
      UINT  semnum;
   {
      int i;
      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;

      if(APPB_getsem(0))  return(-1);
      dpctl[semnum].pflag    = 0;
      dpctl[semnum].command  = NOP;
      dpctl[semnum].buf_stat = BUF_EMPTY;
      if(APPB_relsem(0)) return(-1);
      else return(0);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_putmemblk(), PC side                                            */
/*                                                                       */
/*  Write block of memory to the dual port.                              */
/*  Return a 0 if successful, a -1 if failed.                            */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Find free block of dual port to write memory.                    */
/*  2)  Write the memory.                                                */
/*  3)  Write memory parameters to control block.                        */
/*                                                                       */
/*************************************************************************/

int APPB_putmemblk(cnt,src,dst)
      ULONG cnt;
      ULONG *src;
      ULONG dst;
   {

      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;
      ULONG  *dpram;
      UINT    dpblk;
      int     i;

      if(APPB_getctlblk(&dpblk))  return(-1);

      dpram = (ULONG*)(DPRAM_MEMBASE + (dpblk * DPRAM_BLK_SIZE));

      for(i=0;i<cnt;i++)
      *dpram++ = *src++;

      if(APPB_getsem(0))  return(-1);

      dpctl[dpblk].command  = HOST_MEM_WR;
      dpctl[dpblk].buf_stat = BUF_FULL;
      dpctl[dpblk].count    = cnt;
      dpctl[dpblk].addr     = dst;

      if(APPB_relsem(0))  return(-1);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_getmemblk(), PC side                                            */
/*                                                                       */
/*  Read block of memory to the dual port.                               */
/*  Return a 0 if successful, a -1 if failed.                            */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Find free block of dual port for memory.                         */
/*  2)  Write memory parameters to control block.                        */
/*  3)  Wait for TMS320C30 to put requested memory into the dual port.   */
/*  4)  Read data from the dual port.                                    */
/*  5)  Release block of dual port memory.                               */
/*                                                                       */
/*************************************************************************/
int APPB_getmemblk(cnt,src,dst)
      ULONG cnt;
      ULONG src;
      ULONG *dst;
   {

      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;
      ULONG  *dpram;
      UINT    dpblk;
      int     i;
      UINT  timeout = MAX_SEM_TIME;

      if(APPB_getctlblk(&dpblk))  return(-1);

      dpram = (ULONG*)(DPRAM_MEMBASE + (dpblk * DPRAM_BLK_SIZE));

      if(APPB_getsem(0)) return(-1);

      dpctl[dpblk].command  = HOST_MEM_RD;
      dpctl[dpblk].buf_stat = BUF_EMPTY;
      dpctl[dpblk].count    = cnt;
      dpctl[dpblk].addr     = src;

      while( --timeout )
      {
          if(!APPB_getsem(0) && (dpctl[dpblk].buf_stat == BUF_FULL)) break;
          if(APPB_relsem(0))  return(-1);
      }

      if(APPB_relsem(0) || !timeout) return(-1);

      for(i=0;i<cnt;i++)
          *dst++ = *dpram++;

      if(APPB_relctlblk(dpblk)) return(-1);

   }
```

Appendix A2. TMS320C30 Applications Board Routines—TMS320C30 Side

```
/************************************************************************/
/*                                                                      */
/*  APPENDIX A2                                                         */
/*                                                                      */
/*  TMS320C30 APPLICATION BOARD ROUTINES - TMS320C30 SIDE               */
/*                                                                      */
/*  Texas Instruments Inc.                                              */
/*  10/20/89                                                            */
/*                                                                      */
/*  Functions:                                                          */
/*                                                                      */
/*       int APPB_dpinit()       Intialize APPB.                        */
/*       int APPB_getsem()       Get access to semaphore bit N          */
/*       int APPB_relsem()       Release access to semaphore bit N      */
/*       int APPB_getctlblk()    Get a control block in DPRAM           */
/*       int APPB_relctlblk()    Release control block in DPRAM         */
/*       int APPB_getmemblk()    Get a block of memory from DPRAM       */
/*       int APPB_putmemblk()    Put a block of memory to DPRAM         */
/*       int APPB_getlong()      Read a long int from the DPRAM         */
/*       int APPB_getcommand()   Read a command and parameters from DPRAM */
/*                                                                      */
/*  All code was compiled with TMS320C30 C compiler version 2.1, using the */
/*  small model.                                                        */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                                                                      */
/*    Constant definitions for the TMS320C30 Applications Board.        */
/*                                                                      */
/************************************************************************/
#define         SEM_BASE        0x00805FF8
#define         CTL_REG         0x00805FF7

#define         CINT            0x01
#define         XINTCLR_        0x02
#define         DPSEL           0x04
#define         SWRESET_        0x08
#define         XINT            0x10
#define         CINTCLR_        0x20
#define         MBANK           0x40
#define         MSWAP           0x80

#define         DPRAM_CTL       0x00804000
#define         DPRAM_MEMBASE   0x00804200
#define         DPRAM_SIZE      0x1000
#define         DPRAM_BLKS      7
#define         DPRAM_BLK_SIZE  512
#define         NUM_SEMS        8
#define         MAX_SEM_TIME    10000

#define         BUF_EMPTY       0
#define         BUF_FULL        1

#define         NOP             0x00
#define         HOST_MEM_WR     0x80
#define         HOST_MEM_RD     0x81

typedef         unsigned char   UCHAR;
typedef         unsigned short  UINT;
typedef         unsigned long   ULONG;

typedef         struct
                {
                    UCHAR   pflag;
                    UCHAR   command;
                    UCHAR   buf_stat;
                    UCHAR   nc;
                    UCHAR   count[4];
                    UCHAR   addr[4];
                    UCHAR   message[10];
                }DPCNTL;

typedef         struct
                {
                    UCHAR   mblk;
                    UCHAR   mcmd;
                    ULONG   mcnt;
                    ULONG   maddr;
                }MPARMS;
```

```
/************************************************************************/
/*                                                                      */
/*   Test program, TMS320C30 side.                                      */
/*                                                                      */
/*   Sequence:                                                          */
/*                                                                      */
/*   1)  Initialize the the dual port SRAM.                             */
/*   2)  Poll dual port for commands.                                   */
/*   3)  Execute commands as encountered.                               */
/*                                                                      */
/************************************************************************/
main()
{
      int i;
      MPARMS  mparms;

      APPB_dpint();

      for(;;)
      {

          APPB_getcommand(&mparms);
          switch(mparms.mcmd)
          {
              case NOP:   break;

              case HOST_MEM_WR:
                  APPB_getmemblk(mparms.mcnt,mparms.maddr,mparms.mblk);
              break;

              case HOST_MEM_RD:
                  APPB_putmemblk(mparms.mcnt,mparms.maddr,mparms.mblk);
              break;

              default:    break;

          }
      }
}
```

```
/************************************************************************/
/*                                                                      */
/*   APPB_dpint(), TMS320C30 side.                                      */
/*                                                                      */
/*   Sequence:                                                          */
/*                                                                      */
/*   1)  Set DRAM semaphores to 1 (free).                               */
/*   2)  Clear entire dual port RAM.                                    */
/*                                                                      */
/************************************************************************/
int APPB_dpint()
    {
       int i;
       UCHAR *semaddr = (UCHAR *)SEM_BASE;
       UCHAR *dpram   = (UCHAR *)DPRAM_CTL;

       for(i=0;i<8;i++)    *semaddr++ = 1;
       for(i=0;i<DPRAM_SIZE;i++) *dpram++ = 0;

       return(0);
    }
```

```
/************************************************************************/
/*                                                                      */
/*  APPB_getsem(), TMS320C30 side                                       */
/*                                                                      */
/*  Attempts to gain access of semaphore 'semnum'                       */
/*                                                                      */
/*  Sequence                                                            */
/*                                                                      */
/*  1)  Write 0 to semaphore.                                           */
/*  2)  Wait till read a 0.                                             */
/************************************************************************/

int APPB_getsem(semnum)
      UINT  semnum;
   {
      UCHAR *semaddr = (UCHAR *)(SEM_BASE + semnum);

      *semaddr = 0; while(*semaddr & 1UL); return(0);
   }
```

```
/************************************************************************/
/*                                                                      */
/*  APPB_relsem(), TMS320C30 side                                       */
/*                                                                      */
/*  Release semaphore at 'semnum'                                       */
/*                                                                      */
/*  Sequence                                                            */
/*                                                                      */
/*  1)  Write 1 to semaphore.                                           */
/*  2)  Wait till read 1.                                               */
/************************************************************************/

int APPB_relsem(semnum)
      UINT  semnum;
   {
      UCHAR *semaddr = (UCHAR *)(SEM_BASE + semnum);

      *semaddr = 1; while(!(*semaddr & 1UL)); return(0);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_getctlblk(), TMS320C30 side.                                    */
/*                                                                       */
/*  Find unused block of memory in the dual port.                        */
/*  Return a 0 if successful, a -1 if failed.                            */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Search control structures for free block of memory.              */
/*  2)  If block free, set semnum to block index, return 0.              */
/*  3)  Else,  return -1 (failed to find block).                         */
/*                                                                       */
/*************************************************************************/
int APPB_getctlblk(semnum)
      UINT  *semnum;
   {
      int i;
      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;

      APPB_getsem(0);

      for(i=0;i<DPRAM_BLKS;i++)
          if(!(dpctl[i].pflag & 1UL))
          {
            dpctl[i].pflag    = 1;
            dpctl[i].command  = NOP;
            dpctl[i].buf_stat = BUF_EMPTY;
            *semnum = i;
            APPB_relsem(0);   return(0);
          }

      APPB_relsem(0); return(-1);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_relctlblk(), TMS320C30 side.                                    */
/*                                                                       */
/*  Release block of memory in the dual port.                            */
/*  Return a 0 if successful, a -1 if failed.                            */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Null out the control structure.                                  */
/*  2)  Return.                                                          */
/*                                                                       */
/*************************************************************************/
int APPB_relctlblk(semnum)
      UINT  semnum;
   {
      int i;
      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;

      APPB_getsem(0);
      dpctl[semnum].pflag    = 0;
      dpctl[semnum].command  = NOP;
      dpctl[semnum].buf_stat = BUF_EMPTY;
      APPB_relsem(0);  return(0);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_putmemblk(), TMS320C30 side.                                    */
/*                                                                       */
/*  Move block of data to dual port.                                     */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Move data to the dual port.                                      */
/*  2)  Set dual port buffer status to BUF_FULL.                         */
/*                                                                       */
/*************************************************************************/
int APPB_putmemblk(cnt,src,dpblk)
      ULONG cnt;
      ULONG *src;
      UINT  dpblk;
   {

      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;
      UCHAR  *dpram;
      ULONG   temp;
      int     i,j;

      dpram = (UCHAR *)(DPRAM_MEMBASE + (dpblk * DPRAM_BLK_SIZE));

      for(i=0;i<cnt;i++)
      { temp = *src++; for(j=0;j<32;j+=8) *dpram++ = temp >> j; }

      APPB_getsem(0);
      dpctl[dpblk].buf_stat = BUF_FULL;
      APPB_relsem(0); return(0);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*  APPB_getmemblk(), TMS320C30 side.                                    */
/*                                                                       */
/*  Move block of data from dual port.                                   */
/*                                                                       */
/*  Sequence                                                             */
/*                                                                       */
/*  1)  Move data from the dual port.                                    */
/*  2)  Release block of dual port memory.                               */
/*                                                                       */
/*************************************************************************/
int APPB_getmemblk(cnt,dst,dpblk)
      ULONG cnt;
      ULONG *dst;
      UINT  dpblk;
   {

      DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;
      UCHAR  *dpram;
      ULONG   temp;
      int     i,j;

      dpram = (UCHAR *)(DPRAM_MEMBASE + (dpblk * DPRAM_BLK_SIZE));

      for(i=0;i<cnt;i++)
      {
          temp = 0UL;
          for(j=0;j<32;j+=8) temp |= ((*dpram++) &0x000000ff) << j;
          *dst++ = temp;
      }

      APPB_relctlblk(dpblk);  return(0);
   }
```

```
/*************************************************************************/
/*                                                                       */
/*   APPB_getlong(), TMS320C30 side.                                     */
/*                                                                       */
/*   Get a long word of data from the dual port.                         */
/*************************************************************************/
int APPB_getlong(src,dst)
        ULONG *src;
        ULONG *dst;
    {
        int j;
        *dst = 0UL;
        for(j=0;j<32;j+=8) *dst |= ((*src++) & 0x000000ff) << j;
        return(0);
    }
```

```
/*************************************************************************/
/*                                                                       */
/*   APPB_getcommand(), TMS320C30 side.                                  */
/*                                                                       */
/*   Search the dual port control structures for commands.               */
/*                                                                       */
/*   Sequence                                                            */
/*                                                                       */
/*   1)  Get access to dual port semaphore 0.                            */
/*   2)  If at end of control structures, reset current_blk.             */
/*   3)  Search control structures for a command.                        */
/*   4)  If found, format parameters, return.                            */
/*   5)  Else, search to the end of list, return.                        */
/*                                                                       */
/*************************************************************************/
int APPB_getcommand(mparms)
      MPARMS  *mparms;
    {
       DPCNTL *dpctl = (DPCNTL *)DPRAM_CTL;
       static int  current_blk = -1;

       APPB_getsem(0);

       if(current_blk >= DPRAM_BLKS) current_blk = -1;

       while(current_blk++ < DPRAM_BLKS)
       {
           if(dpctl[current_blk].pflag & 1UL)
           {
               mparms->mcmd = dpctl[current_blk].command & 0x000000ff;
               mparms->mblk = current_blk;
               APPB_getlong(&dpctl[current_blk].count,&mparms->mcnt);
               APPB_getlong(&dpctl[current_blk].addr,&mparms->maddr);
               APPB_relsem(0); return(0);
           }
       }

       APPB_relsem(0); mparms->mcmd = NOP; return(0);
    }
```

APPENDIX A3. Memory Map and Description (TMS320C30 View)

Listed below is a summary of the APPB memory map.

000000 –	003FFF	EPROM (Boot EPROM/remappable)
004000 –	3FFFFF	Unused
400000 –	4FFFFF	DRAM space
400000 –	43FFFF	256K-word DRAM minimum configuration
440000 –	47FFFF	256K-word DRAM minimum configuration
480000 –	4BFFFF	256K-word DRAM option bank 2
4C0000 –	4FFFFF	256K-word DRAM option bank 3
500000 –	7FFFFF	Unused
800000 –	801FFF	SRAM space 1 (16K-byte zero wait-state SRAM)
802000 –	805FFF	Reserved by TI
804000 –	805FFF	I/O Devices
804000 –	804FFF	4K-byte dual-port SRAM
805000 –	805FF6	I/O Expansion Bus
805FF7		Control Register R
805FF8 –	805FFF	dual-port RAM Semaphores (D0 only)
806000 –	807FFF	Reserved by TI
808000 –	8097FF	Memory mapped Peripherals
809800 –	809BFF	RAM Block 0
809C00 –	809FFF	RAM Block 1
80A000 –	EFFFFF	Unused
F00000 –	F03FFF	SRAM space 0 (16K-byte zero wait-state SRAM, remappable)
F00800 –	FFFFFF	Unused

Appendix B

Modules

Appendix	Name
B1	Module U5 – TMS320C30 Software Development Board
B2	Module U6 – TMS320C30 Software Development Board
B3	Module RAMDEC – TMS320C30 Software Development Board
B4	Module RDYEN – TMS320C30 Software Development Board
B5	Module RAMCONTROL – TMS320C30 SWDS DRAM Module
B6	Module RAMDEC – TMS320C30 SWDS DRAM Module

Appendix B1. TMS320C30 Software Development Board

```
Module U5
title'
DWG NAME            TMS320C30 SOFTWARE DEVELOPMENT BOARD
DWG #       2554377
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        NAT SESHAN
DATE        10/01/88'

XSUC8 device 'P2018';

SA0         Pin 1;
SA1         Pin 2;
SA2         Pin 3;
SA3         Pin 4;
SA4         Pin 5;      "PC XT ADDRESS LINES – INPUTS
SA5         Pin 6;
SA6         Pin 7;
SA7         Pin 8;
SA8         Pin 9;
SA9         Pin 10;
NSMEMW      Pin 11;     "PC XT MEMORY WRITE STROBE
GND         Pin 12;
NSMEMR      Pin 13;     "PC XT MEMORY READ STROBE – INPUT
NSIOW       Pin 14;     "PC XT IO WRITE STROBE – INPUT
NSGBA       Pin 15;     "SDB READ STROBE – OUTPUT
NPQ         Pin 16;     "DUAL-PORT ADDRESS RANGE STROBE – INPUT
XAEN        Pin 17;     "PC XT BUS TRANSACTION DISABLE – INPUT
NRG         Pin 18;     "SDB CONTROL REGISTER R ENABLE – OUTPUT
NQG         Pin 19;     "SDB DUAL-PORT ADDRESS LATCH ENABLE – OUTPUT
NDPSEML     Pin 20;     "DUAL-PORT SEMAPHORE SELECT – OUTPUT
NDPCEL      Pin 21;     "DUAL-PORT SRAM CHIP ENABLE – OUTPUT
SGAB        Pin 22;     "HOST DATA BUS INPUT ENABLE – OUTPUT
NSIOR       Pin 23;     "PC XT IO READ STROBE – INPUT
VCC         Pin 24;

SA = [SA9, SA8, SA7, SA6, SA5, SA4, SA3, SA2, SA1 ,SA0];
 X = .X.;

equations

            !NQG        = !XAEN & (SA == ^h338);
            !NRG        = !XAEN & (SA == ^h339);
            !NDPSEML    = !XAEN & SA9 & SA8 & !SA7 & !SA6 & SA5 & SA4 & !SA3
                          & !NSIOW
                          # !XAEN & SA9 & SA8 & !SA7 & !SA6 & SA5 & SA4 & !SA3
                          & !NSIOR;
```

```
!NDPCEL    = !XAEN & !NPQ;
SGAB       = !NSIOW & !XAEN
             # !NSMEMW & !XAEN ;
!NSGBA     = !XAEN & !NSIOR & (SA == ^h339)
             # !XAEN & !NSIOR & SA9 & SA8 & !SA7 & !SA6 & SA5
             & SA4 & !SA3
             # !XAEN & !NSMEMR & !NPQ;

end U5
```

Appendix B2. Module U6

```
Module U6
title'
DWG NAME    TMS320C30 SOFTWARE DEVELOPMENT BOARD
DWG #       2554377
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        NAT SESHAN
DATE        10/01/88'

XSUF10      Device    'P20L8';

CIOA0       Pin 1;
CIOA1       Pin 2;
CIOA2       Pin 3;
CIOA3       Pin 4;
CIOA4       Pin 5;
CIOA5       Pin 6;
CIOA6       Pin 7;
CIOA7       Pin 8;
CIOA8       Pin 9;
CIOA9       Pin 10;
CIOA10      Pin 11;
GND         Pin 12;
CIOA11      Pin 13;
CIOA12      Pin 14;
TIOW        Pin 15;
NSRANGE     Pin 16;
CIORNW      Pin 17;
NFR         Pin 18;
NFG         Pin 19;
NDPMEMGR    Pin 20;
NDPSEMGR    Pin 21;
TIOR        Pin 22;
NCIOSTRB    Pin 23;
VCC         Pin 24;

            X = .X.;
            C = .C.;

            CIOA = [CIOA12,CIOA11,CIOA10,CIOA9,CIOA8,
                    CIOA7,CIOA6,CIOA5,CIOA4,CIOA3,CIOA2,CIOA1,CIOA0];

equations

            !NSRANGE    = !NCIOSTRB & !CIOA12
                          # !NCIOSTRB & (CIOA >= ^h1FF7);
            !NDPMEMGR   = !NCIOSTRB & !CIOA12;
            !NDPSEMGR   = !NCIOSTRB & (CIOA >= ^h1FF8);
```

```
!NFG        = !NCIOSTRB & !CIORNW & (CIOA == ^h1FF7);
!NFR        = !NCIOSTRB &  CIORNW & (CIOA == ^h1FF7);
!TIOR       = NCIOSTRB
              # (CIOA >= ^h1FF7)
              # !CIOA12
              # !CIORNW;

!TIOW       = NCIOSTRB
              # (CIOA >= ^h1FF7)
              # !CIOA12
              # CIORNW;
```

test_vectors

([CIOA, NCIOSTRB, CIORNW] –>
 [TIOR, TIOW, NSRANGE, NFG, NFR, NDPMEMGR, NDPSEMGR]);

READ OR WRITE TO A SEMAPHORE

[^h1FF8, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FF9, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FFA, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FFB, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FFC, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FFD, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FFE, 0, X] –> [0, 0, 0, 1, 1, 1, 0];
[^h1FFF, 0, X] –> [0, 0, 0, 1, 1, 1, 0];

WRITE TO F REGISTER

[^h1FF7, 0, 0] –> [0, 0, 0, 0, 1, 1, 1];

READ FROM F REGISTER

[^h1FF7, 0, 1] –> [0, 0, 0, 1, 0, 1, 1];

NCIOSTRB DISABLED

[X , 1, X] –> [0, 0, 1, 1, 1, 1, 1];

EXTERNAL READS

[^b1000000000000, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000001, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000010, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000011, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000100, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000101, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000110, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000000111, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001000, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001001, 0, 1] –> [1, 0, 1, 1, 1, 1, 1];

```
[^b1000000001010, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001011, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001100, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001101, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001110, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^b1000000001111, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF0, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF1, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF2, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF3, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF4, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF5, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
[^h1FF6, 0, 1] -> [1, 0, 1, 1, 1, 1, 1];
EXTERNAL IO WRITES
[^b1000000000000, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000001, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000010, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000011, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000100, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000101, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000110, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000000111, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001000, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001001, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001010, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001011, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001100, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001101, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001110, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^b1000000001111, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF0, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF1, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF2, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF3, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF4, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF5, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
[^h1FF6, 0, 0] -> [0, 1, 1, 1, 1, 1, 1];
test_vectors
([CIOA12, NCIOSTRB, CIORNW] ->
[TIOR, TIOW, NSRANGE, NFG, NFR, NDPSEMGR, NDPMEMGR]);
DUAL-PORT SRAM READ OR WRITE
[0, 0, X] -> [0, 0, 0, 1, 1, 1 ,0];
end U6
```

Appendix B3. Module RAMDEC

```
module RAMDEC
title'
DWG NAME            TMS320C30 SOFTWARE DEVELOPMENT BOARD
DWG #       2554377
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        TONY COOMES
DATE        10/01/88'

XSUB4       device      'P16L8';

a12         Pin 1;      "c30 address inputs
a13         Pin 2;
a14         Pin 3;
a15         Pin 4;
a16         Pin 5;
a17         Pin 6;
a18         Pin 7;
a19         Pin 8;
a20         Pin 9;
a21         Pin 11;
a22         Pin 13;
a23         Pin 14;
m_swap      Pin 15;     "sram/eprom swap bit
vss         Pin 10;

memen       Pin 18;     "dram expansion select
sram        Pin 17;     " sram select
eprom       Pin 16;     "eprom select
busen       Pin 12;     "eprom/dram data buffer select
vcc         Pin 20;

madd = [a23,a22,a21,a20,a19,a18,a17,a16,a15,a14,a13,a12];

equations

            "On reset the eprom and sram maps are swapped
            "           m_swap = 0          m_swap = 1
            "sram       F00000–F03FFF       000000–003FFF
            "eprom      000000–003FFF       F00000–F03FFF

            sram    = !(((madd >= ^h000) & (madd <= ^h003) & m_swap)
                      # ((madd >= ^hF00) & (madd <= ^hF03) & !m_swap));

            eprom   = !(((madd >= ^h000) & (madd <= ^h003) & !m_swap)
                      # ((madd >= ^hF00) & (madd <= ^hF03) & m_swap));

            memen   = !((madd >= ^h400) & (madd <= ^h4FF));

            busen   = !(!eprom # !memen);
```

```
test_vectors
([madd,  m_swap ] -> [sram, eprom, memen, busen])
 [^h000,  1   ] -> [ 0,   1,   1,    1 ];
 [^h000,  0   ] -> [ 1,   0,   1,    0 ];
 [^h004,  1   ] -> [ 1,   1,   1,    1 ];
 [^hF00,  1   ] -> [ 1,   0,   1,    0 ];
 [^hF00,  0   ] -> [ 0,   1,   1,    1 ];
 [^hFF0,  1   ] -> [ 1,   1,   1,    1 ];
 [^hF00,  1   ] -> [ 1,   0,   1,    0 ];
 [^h400,  0   ] -> [ 1,   1,   0,    0 ];
 [^h4CF,  1   ] -> [ 1,   1,   0,    0 ];
 [^h800,  1   ] -> [ 1,   1,   1,    1 ];
end RAMDEC
```

Appendix B4. Module RDYEN

```
module RDYEN
title'
DWG NAME            TMS320C30 SOFTWARE DEVELOPMENT BOARD
DWG #       2554377
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        TONY COOMES
DATE        10/01/88'

XSUC3       device      'P16R4';

clk         Pin 1;
busen       Pin 2;      "eprom/dram data bus enable
eprom       Pin 3;      "eprom select
strb        Pin 4;      "c30 strobe
rd_wr       Pin 5;      "c30 read/write
bhiz        Pin 7;      "dram expansion bus hold
oe          Pin 11;
vss         Pin 10;

dat_rd      Pin 19;     "data read enable
dat_wr      Pin 18;     "data write enable
prdy        Pin 17;     "eprom ready
epromcs     Pin 12;     "eprom chip select
vcc         Pin 20;

c = .C.;

equations

"note: bhiz is active for 1 TMS320C30 clock cycle at the end of a dram
"      access. This provides the necessary turn off time between
"      dram/eprom accesses.

dat_rd      =     !(!busen  & !strb & rd_wr & bhiz);

dat_wr      =     (!busen  & !strb & !rd_wr & bhiz);

epromcs     =     !(!busen & rd_wr & !strb & !eprom & bhiz);

prdy        :=    !(!busen & !strb & rd_wr & prdy & !eprom & bhiz);
```

```
test_vectors
([clk, strb, busen, rd_wr, eprom, oe, bhiz  ] -> prdy)
 [ c,  1,    1,    1,    1,    0,   1  ] ->  1;
 [ c,  0,    0,    1,    0,    0,   0  ] ->  1;
 [ c,  0,    0,    1,    0,    0,   1  ] ->  0;
 [ c,  0,    0,    1,    0,    0,   1  ] ->  1;
 [ c,  0,    0,    1,    0,    0,   1  ] ->  0;
 [ c,  1,    0,    1,    0,    0,   1  ] ->  1;
 [ c,  1,    0,    1,    0,    0,   1  ] ->  1;
test_vectors
([strb, busen, rd_wr, eprom, bhiz  ] -> [dat_rd, dat_wr, epromcs])
 [ 1,    1,    1,    1,     1   ] -> [ 1,    0,     1  ];
 [ 0,    0,    1,    1,     1   ] -> [ 0,    0,     1  ];
 [ 0,    0,    0,    1,     1   ] -> [ 1,    1,     1  ];
 [ 0,    1,    1,    1,     1   ] -> [ 1,    0,     1  ];
 [ 1,    0,    1,    1,     1   ] -> [ 1,    0,     1  ];
 check eprom
 [ 1,    0,    1,    0,     1   ] -> [ 1,    0,     1  ];
 [ 0,    0,    1,    0,     1   ] -> [ 0,    0,     0  ];
 [ 0,    0,    1,    0,     0   ] -> [ 1,    0,     1  ];
 [ 0,    0,    0,    0,     1   ] -> [ 1,    1,     1  ];
 [ 0,    1,    1,    0,     1   ] -> [ 1,    0,     1  ];
 [ 1,    0,    1,    1,     1   ] -> [ 1,    0,     1  ];

end RDYEN
```

Appendix B5. Module RAMCONTROL

```
Module RAMCONTROL
title'
DWG NAME    320C30 SWDS DRAM MODULE
DWG #       2554397
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        TONY COOMES
DATE        10/01/88'

XDUE5       device      'P16R8';

clk         Pin 1;
refreq_     Pin 2;      "refresh request
strb_       Pin 3;      "c30 strobe
rd          Pin 4;      "c30 read/write
memen_      Pin 5;      "memory board chip select
oe_         Pin 11;     "pal output enable
vss         Pin 10;

s0          Pin 19;     "state variable
refclr      Pin 18;     "refresh clear
casen       Pin 17;     "column address strobe
ren         Pin 16;     "write strobe
rasen       Pin 15;     "row address strobe
mrdy        Pin 14;     "dram ready strobe
busact      Pin 13;     "dram bus active
s1          Pin 12;     "state variable
vcc         Pin 20;

"define machine states
"[refclr,rasen,casen,mrdy,busact,s0,s1];

idle    =   ^b1111111;
ras0    =   ^b1011111;
cas0    =   ^b1000111;
cas1    =   ^b1011101;
whld    =   ^b1111110;
trp     =   ^b1111001;
ref1    =   ^b0101111;
ref2    =   ^b0001111;
ref3    =   ^b0011111;
ref4    =   ^b1111101;

refreq  =   !refreq_;   "convert to positive logic
strb    =   !strb_;
memen   =   !memen_;
oe      = !oe_;

c = .C.;
```

```
c = .C.;
output = [refclr,rasen,casen,mrdy,busact,s0,s1];
equations
ren      :=    !(!rd & !strb_);   high on read, low on writes
state_diagram output
state idle:
        case  ( refreq &  strb &  memen)      :ref1;    "ref has 1st priority
              ( refreq &  strb & !memen)     :ref1;
              ( refreq & !strb &  memen)     :ref1;
              ( refreq & !strb & !memen)     :ref1;
              (!refreq &  strb &  memen)     :ras0;
              (!refreq &  strb & !memen)     :idle;
              (!refreq & !strb &  memen)     :idle;
              (!refreq & !strb & !memen)     :idle;
        endcase;
state   ras0:
        goto cas0;
state   cas0:                          "cycle cas on page mode reads
        case  rd                              :cas1;
              !rd                             :whld;
        endcase;
state   cas1:                          "cycle cas on page mode reads
        case  strb & !refreq                  :cas0;
              strb &  refreq                  :trp ;
              !strb & !refreq                 :trp ;
              !strb &  refreq                 :trp ;
        endcase;
state   whld:                          "wait for refreq or !strb
        case  strb & !refreq                  :whld;
              strb &  refreq                  :ref1;
              !strb & !refreq                 ;idle;
              !strb &  refreq                 :ref1;
             endcase;
state   trp:                           "cas,ras high
        case  refreq                          :ref1;
              !refreq                         :idle;
        endcase;
state   ref1:                          "cas,refclr low
        goto ref2;
state   ref2:                          "ras low
        goto ref3;
```

```
state     ref3:                    "cas high
          goto ref4;

state     ref4:                    "ras high
          goto idle;

test_vectors  "page mode read, ref, page mode read
([clk,refreq ,strb , rd,memen ,  oe ]->[output,ren])
 [ c,  0,    0,   1,   0,    1  ]->[idle ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[ras0 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas0 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas1 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas0 ,  1 ];
 [ c,  1,    1,   1,   1,    1  ]->[cas1 ,  1 ];
 [ c,  1,    1,   1,   1,    1  ]->[trp  ,  1 ];
 [ c,  1,    1,   1,   1,    1  ]->[ref1 ,  1 ];
 [ c,  1,    1,   1,   1,    1  ]->[ref2 ,  1 ];
 [ c,  1,    1,   1,   1,    1  ]->[ref3 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[ref4 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[idle ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[ras0 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas0 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas1 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas0 ,  1 ];
 [ c,  0,    1,   1,   1,    1  ]->[cas1 ,  1 ];
 [ c,  0,    0,   1,   1,    1  ]->[trp  ,  1 ];
 [ c,  0,    0,   1,   0,    1  ]->[idle ,  1 ];

test_vectors  "write cycle
([clk,refreq ,strb , rd, memen, oe  ]->[output,ren])
 [ c,  0,    0,   0,   0,    1  ]->[idle ,  1 ];
 [ c,  0,    1,   0,   1,    1  ]->[ras0 ,  0 ];
 [ c,  0,    1,   0,   1,    1  ]->[cas0 ,  0 ];
 [ c,  0,    1,   0,   1,    1  ]->[whld ,  0 ];
 [ c,  0,    1,   0,   1,    1  ]->[whld ,  0 ];
 [ c,  0,    1,   0,   1,    1  ]->[whld ,  0 ];
 [ c,  0,    0,   0,   1,    1  ]->[idle ,  1 ];
 [ c,  0,    0,   1,   0,    1  ]->[idle ,  1 ];

 "write cycle /ref
 [ c,  0,    0,   0,   0,    1  ]->[idle ,  1 ];
 [ c,  0,    1,   0,   1,    1  ]->[ras0 ,  0 ];
 [ c,  1,    1,   0,   1,    1  ]->[cas0 ,  0 ];
 [ c,  1,    1,   0,   1,    1  ]->[whld ,  0 ];
 [ c,  1,    1,   0,   1,    1  ]->[ref1 ,  0 ];
 [ c,  1,    1,   0,   1,    1  ]->[ref2 ,  0 ];
 [ c,  1,    0,   0,   0,    1  ]->[ref3 ,  1 ];
 [ c,  0,    0,   1,   0,    1  ]->[ref4 ,  1 ];
 [ c,  0,    0,   1,   0,    1  ]->[idle ,  1 ];

end RAMCONTROL
```

Appendix B6. Module RAMDEC

```
module RAMDEC
title'
DWG NAME            320C30 SWDS DRAM MODULE
DWG #       2554397
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        TONY COOMES
DATE        10/01/88'

XDUD5       device      'P16R4';

clk         Pin 1;
refclr      Pin 2;      "clear refresh stat
a18         Pin 3;      "c30 address 18
a19         Pin 4;      "c30 address 19
memen       Pin 5;      "dram board memory enable
strb        Pin 6;      "c30 strobe
mux         Pin 7;      "address mux
oe          Pin 11;     "pal output enable
vss         Pin 10;

ras0        Pin 17;     "ras select 0
ras1        Pin 16;     "ras select 1
ras2        Pin 15;     "ras select 2
ras3        Pin 14;     "ras select 3
rowsel      Pin 13;     "row address select
vcc         Pin 20;

c = .C.;

equations

ras0 := !(!refclr # (!a19 & !a18 & !memen & !strb));
ras1 := !(!refclr # (!a19 &  a18 & !memen & !strb));
ras2 := !(!refclr # ( a19 & !a18 & !memen & !strb));
ras3 := !(!refclr # ( a19 &  a18 & !memen & !strb));

rowsel = mux;
```

```
test_vectors  "page mode read, ref, page mode read
([clk,refclr, memen, strb, a19, a18, oe]->[ras0, ras1, ras2, ras3])
[ c,  1,    1,   1,   0,  0, 0 ]->[ 1,   1,   1,   1 ];
[ c,  1,    0,   0,   0,  0, 0 ]->[ 0,   1,   1,   1 ];
[ c,  1,    0,   0,   0,  1, 0 ]->[ 1,   0,   1,   1 ];
[ c,  1,    0,   0,   1,  0, 0 ]->[ 1,   1,   0,   1 ];
[ c,  1,    0,   0,   1,  1, 0 ]->[ 1,   1,   1,   0 ];
[ c,  1,    1,   0,   1,  1, 0 ]->[ 1,   1,   1,   1 ];
[ c,  1,    0,   1,   1,  1, 0 ]->[ 1,   1,   1,   1 ];
[ c,  0,    0,   1,   1,  1, 0 ]->[ 0,   0,   0,   0 ];
[ c,  1,    0,   1,   1,  1, 0 ]->[ 1,   1,   1,   1 ];
[ c,  0,    0,   0,   1,  1, 0 ]->[ 0,   0,   0,   0 ];
[ c,  1,    0,   0,   1,  1, 0 ]->[ 1,   1,   1,   0 ];

test_vectors "rowsel
(mux -> rowsel)
 1  ->  1;
 0  ->  0;

end RAMDEC
```

Appendix C

TMS320C30 Application Board Schematics

Appendix C1. TMS320C30 Software Development Schematics

NOTES, UNLESS OTHERWISE SPECIFIED:

1. ALL AS, ALS, LS DEVICES ARE PREFIXED WITH AN SN74
2. VCC IS APPLIED TO PIN 8 OF ALL 8-PIN IC's, PIN 14 OF ALL 14-PIN IC's, PIN 16 OF ALL 16-PIN IC's, PIN 20 OF ALL 20-PIN IC's, ETC.
3. GROUND IS APPLIED TO PIN 4 OF ALL 8-PIN IC's, PIN 7 OF ALL 14-PIN IC's, PIN 8 OF ALL 16-PIN IC's, PIN 10 OF ALL 20-PIN IC's, ETC.
4. DEVICE TYPE, PIN NUMBERS, AND REFERENCE DESIGNATOR OF GATES ARE SHOWN AS FOLLOWS:

00 AND 04 = DEVICE TYPES
1, 2, AND 3 = PIN NUMBERS
U06 AND U07 = REFERENCE DESIGNATORS

5. RESISTANCE VALUES ARE IN OHMS.
6. RESISTORS ARE 1/4 WATT, 5%.
7. CAPACITANCE VALUES ARE IN MICROFARADS.

REVISIONS			
REV	DESCRIPTION	DATE	APPROVED

PAGE	CONTENTS
1	TITLE PAGE
2	PC XT CONNECTORS, BUFFERS, AND TRANSCEIVERS
3	DUAL PORT SRAM AND EXPANSION BUSS CONTROL CIRCUITRY
4	EXPANSION BUSS 16K x 32 SRAM
5	TMS320C30 BUSSES
6	TMS320C30 TIMER SIGNALS, 12-PIN INTERFACE, PULL-UP PACKAGES
7	ADDRESS BUSS AND CONTROL SIGNAL BUFFERS
8	DATA BUSS TRANSCEIVES
9	TARGET BOARD CONNECTORS, MEMORY EXPANSION BOARD CONNECTORS
10	PARALLEL BUSS MEMORY EXPANSION
11	PARALLEL BUSS CONTROL CIRCUITRY AND EPROMS
12	DECOUPLING CAPACITORS

COMPUTER GENERATED DRAWING ; DO NOT REVISE MANUALLY

PARTS LIST

QTY	ITEM NO	PART OR IDENTIFYING NUMBER	NOMENCLATURE OR DESCRIPTION	NOTES
2554375	7119			

NEXT ASSY: 2554375 | USED ON: 7119
APPLICATION

REVISION STATUS OF SHEETS										
REV										
SH	1	2	3	4	5	6	7	8	9	10
REV										
SH	11	12	13	14	15	16	17	18	19	20
REV										
SH	21	22	23	24	25	26	27	28	29	30
REV										
SH	31	32	33	34	35	36	37	38	39	40

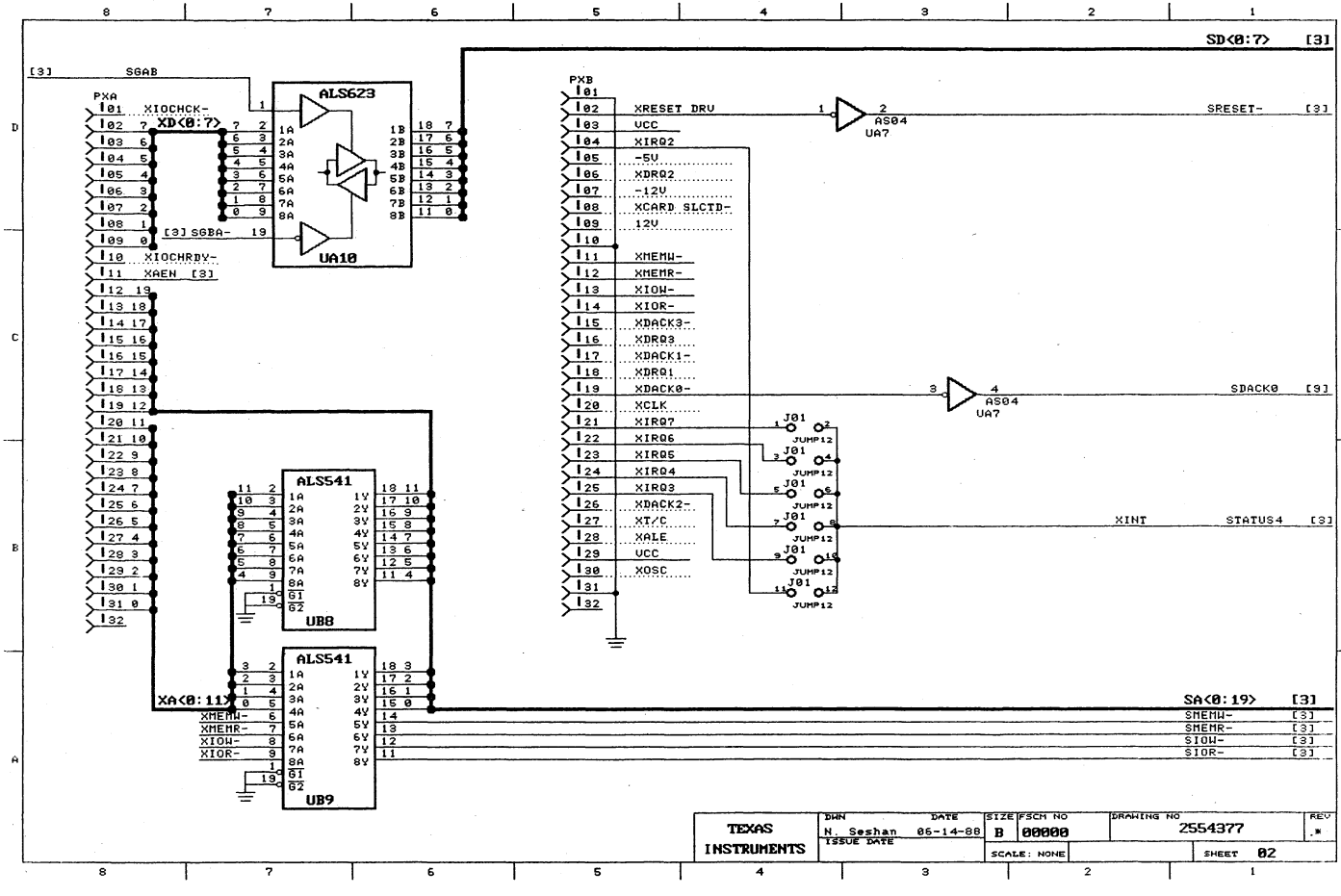
DRN	N. Seshan	DATE 06-15-88
CHK		DATE
ENGR	N. Seshan	DATE 06-15-88
APVD ENGR	Tony Coomes	DATE
QA		DATE
APVD MFG		DATE
RLSE		DATE

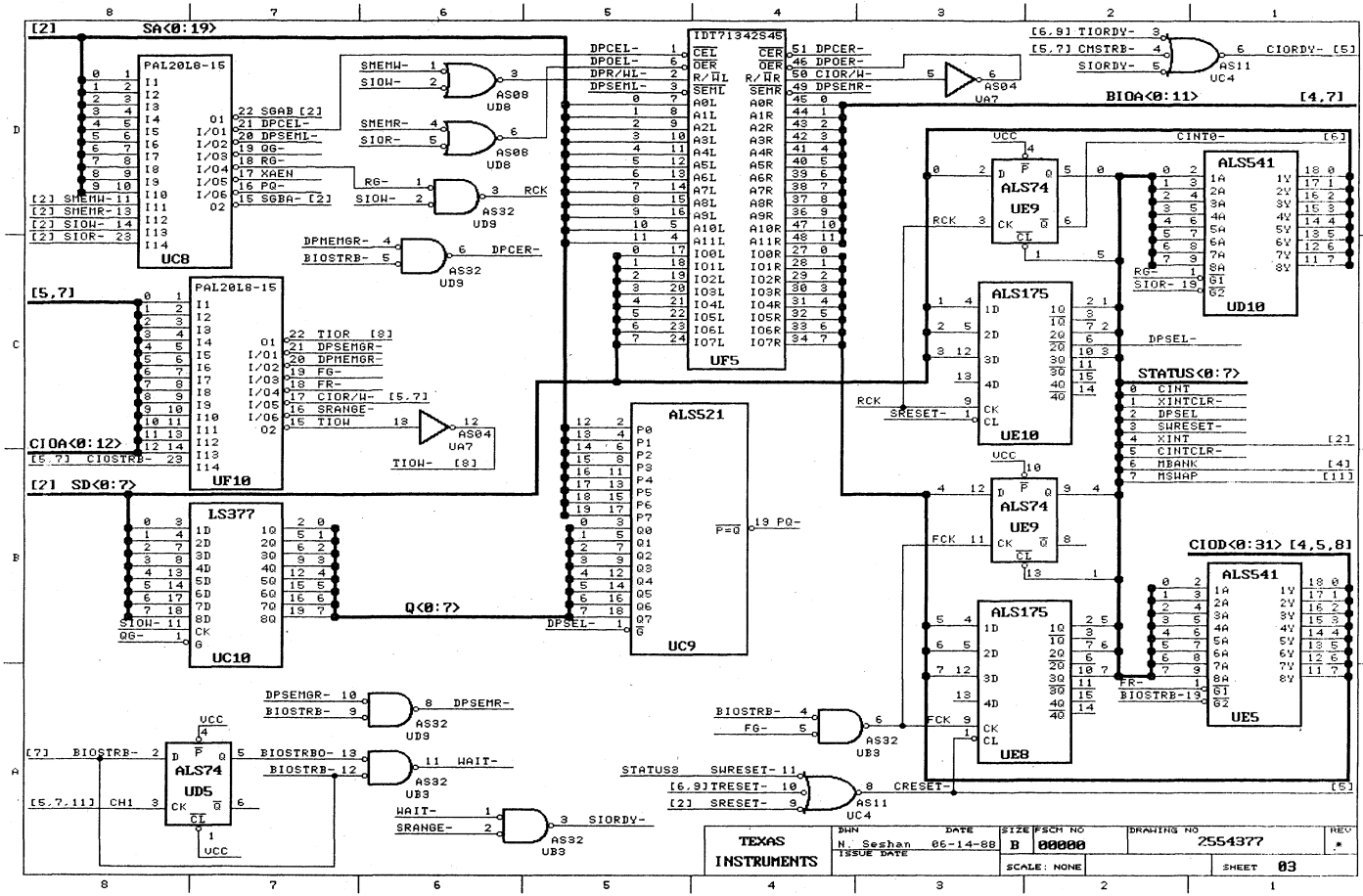
TEXAS INSTRUMENTS
Data Systems Group

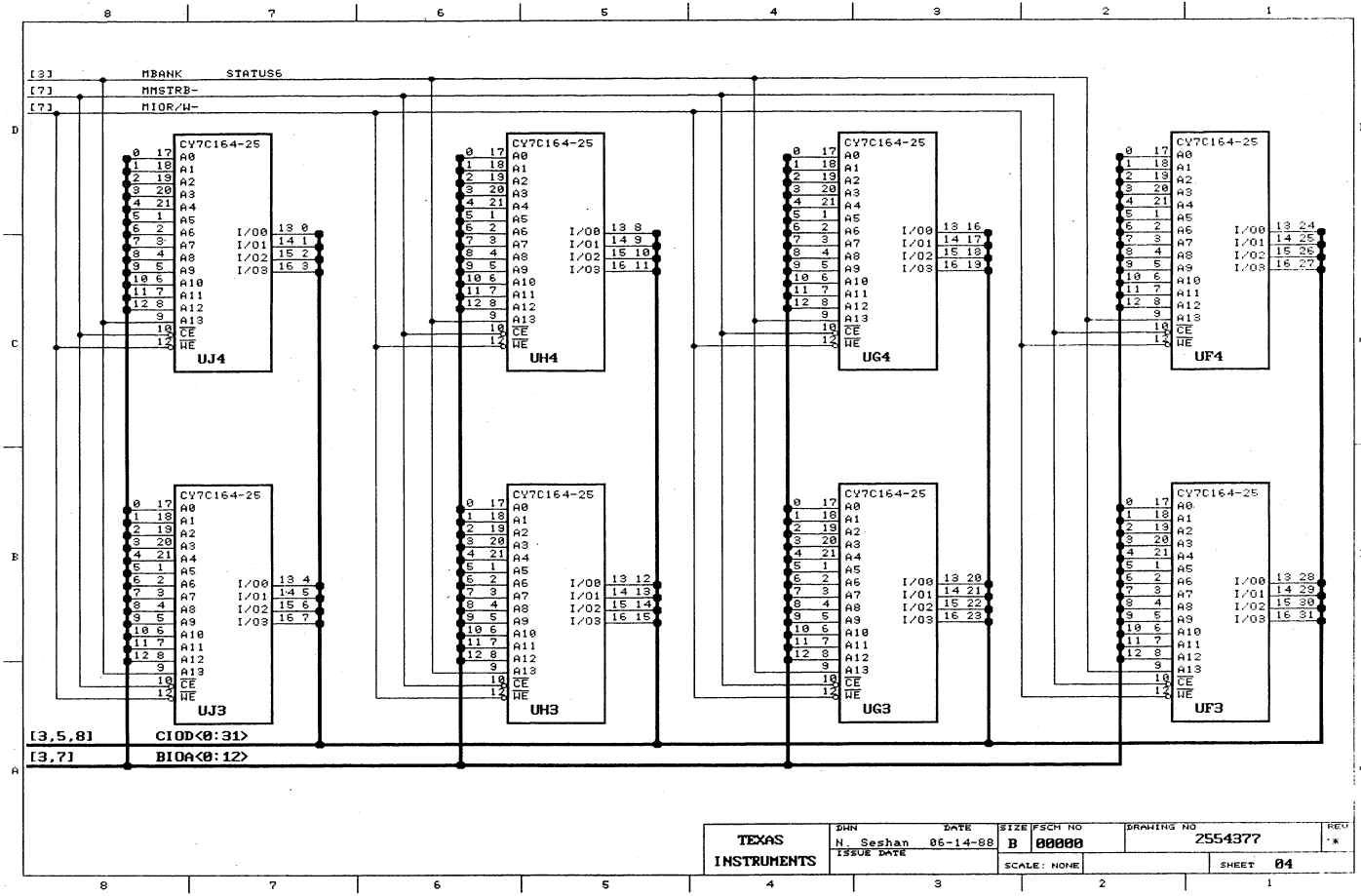
TMS320C30 SOFTWARE DEVELOPMENT BOARD

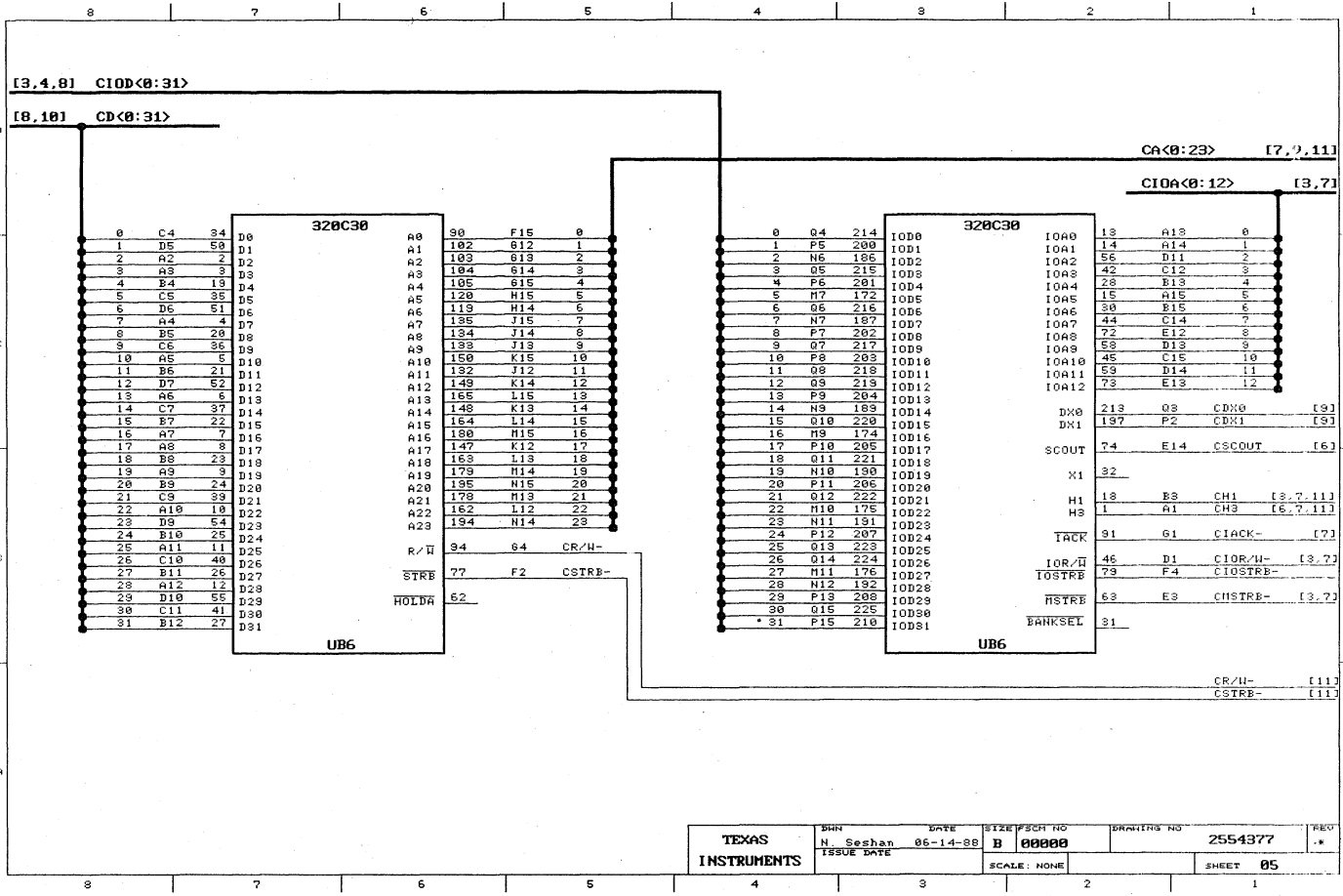
SIZE	FSCM NO	DRAWING NO	REV
B	00000	2554377	.N

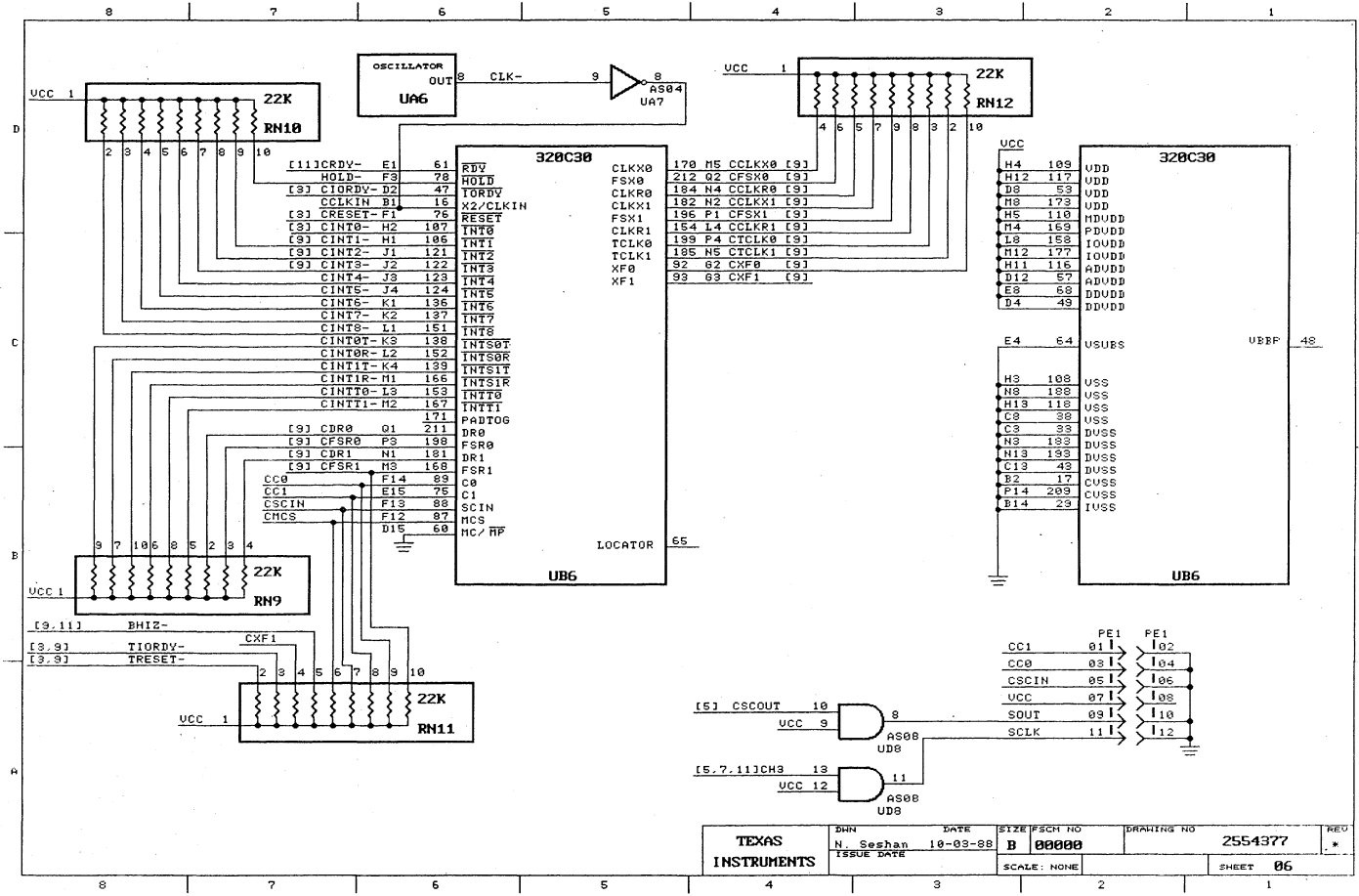
SCALE NONE | SHEET 1 OF 12

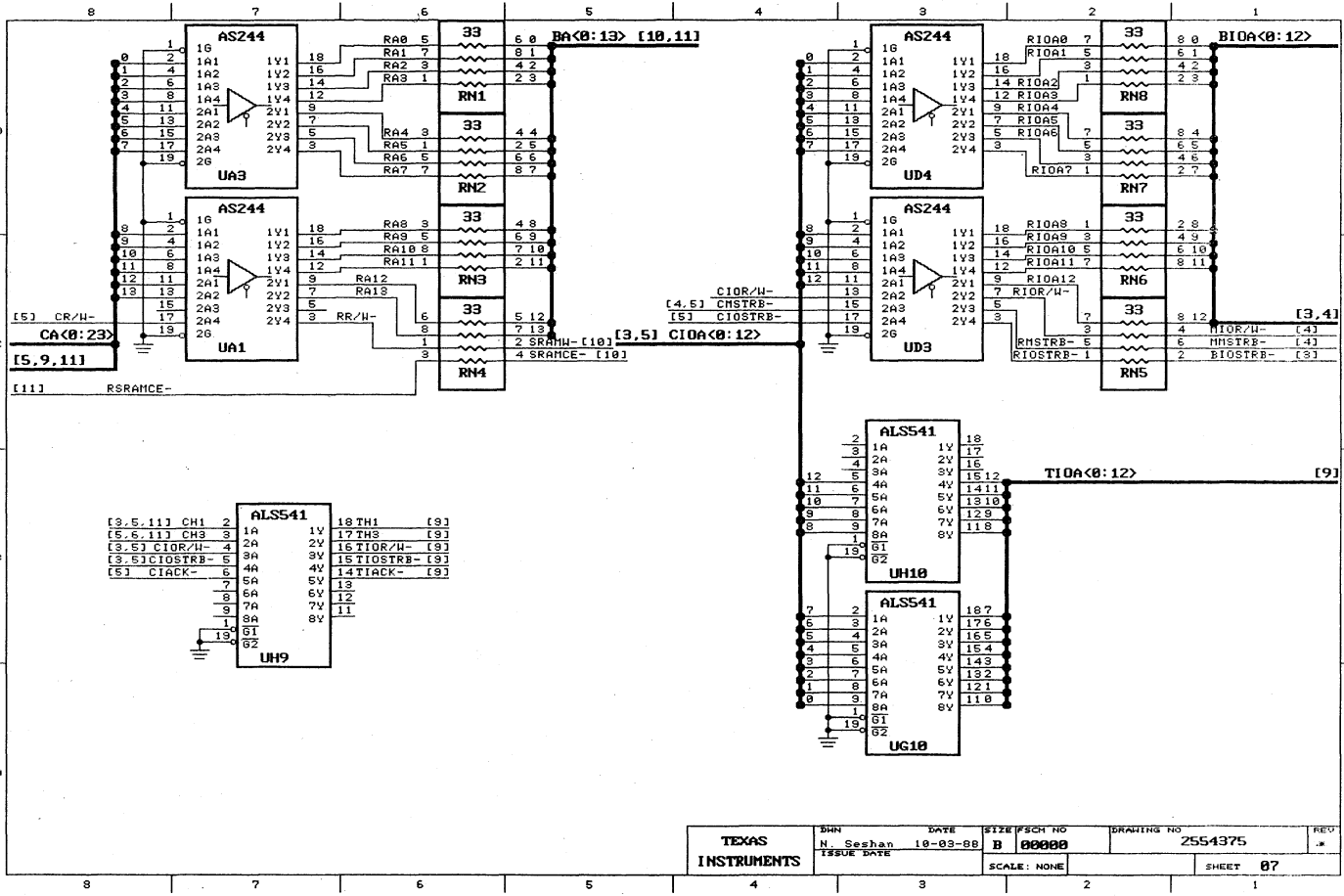

SD<0:7> [3]
[3] SGAB
PXA
XIOCHCK-
XD<0:7>
[3] SGBA-
XIOCHRDY-
XAEN [3]
ALS623
UA10
PXB
XRESET DRV
VCC
XIRQ2
-5V
XDRQ2
-12V
XCARD SLCTD-
12V
XMEMW-
XMEMR-
XIOW-
XIOR-
XDACK3-
XDRQ3
XDACK1-
XDRQ1
XDACK0-
XCLK
XIRQ7
XIRQ6
XIRQ5
XIRQ4
XIRQ3
XDACK2-
XT/C
XALE
VCC
XOSC
AS04
UA7
SRESET- [3]
SDACK0 [9]
XINT
STATUS4 [3]
JUMPER
ALS541
UB8
UB9
XA<0:11>
SA<0:19> [3]
SMEMW- [3]
SMEMR- [3]
SIOW- [3]
SIOR- [3]
TEXAS INSTRUMENTS
N. Seshan 06-14-88
B
00000
2554377
SCALE: NONE
SHEET 02

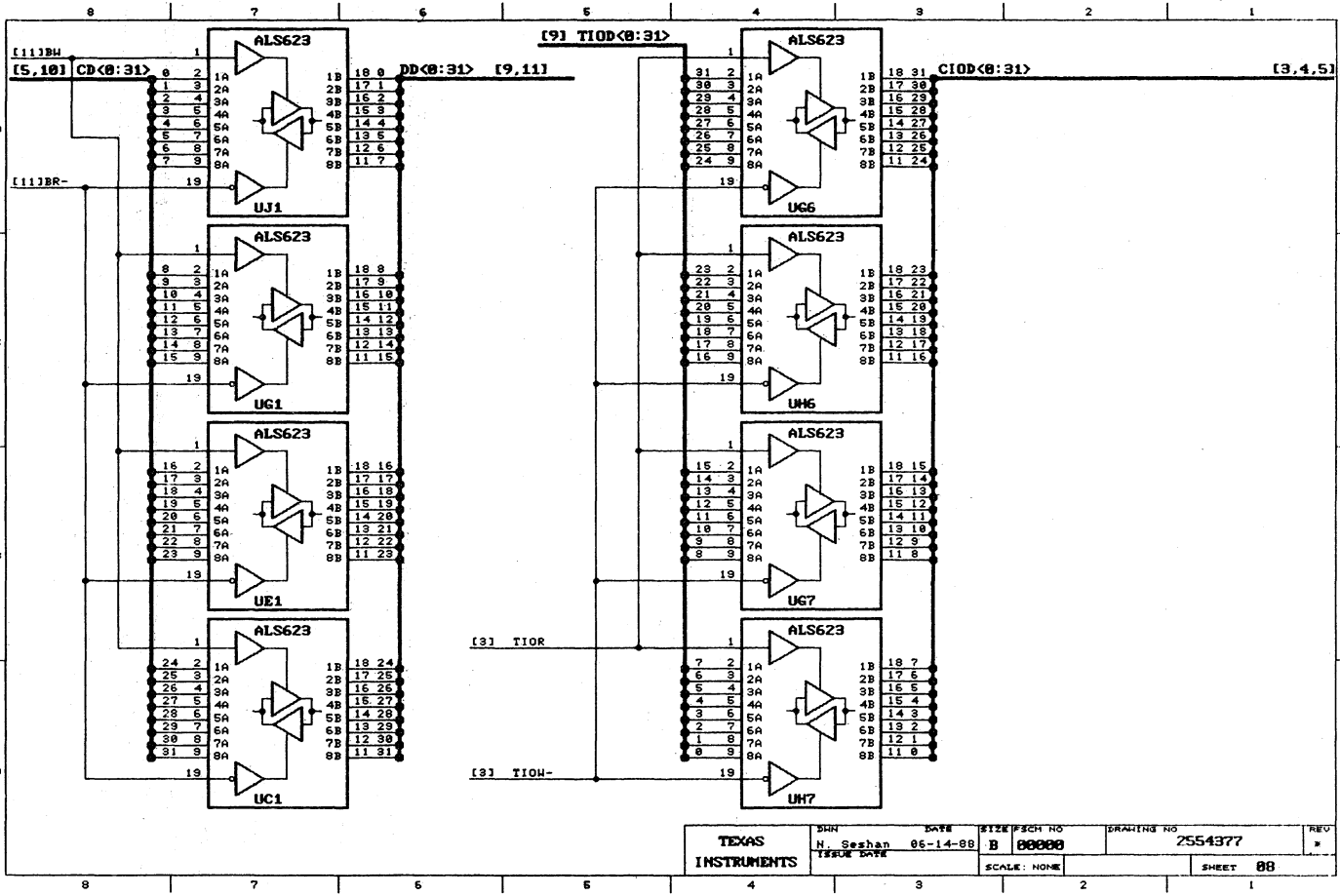

SA<0:19>
PAL20L8-15
UC8
PAL20L8-15
UF10
IDT71342S45
UF5
ALS521
UC9
LS377
UC10
ALS74
UD5
ALS74
UE9
ALS74
UE9
ALS175
UE10
ALS175
UE8
ALS541
UD10
ALS541
UE5
AS08
AS32
AS04
AS11
BIOA<0:11>
CIOA<0:12>
SD<0:7>
Q<0:7>
CIOD<0:31>
STATUS<0:7>
TEXAS INSTRUMENTS
N. Seshan
06-14-88
2554377
SCALE: NONE
SHEET 03

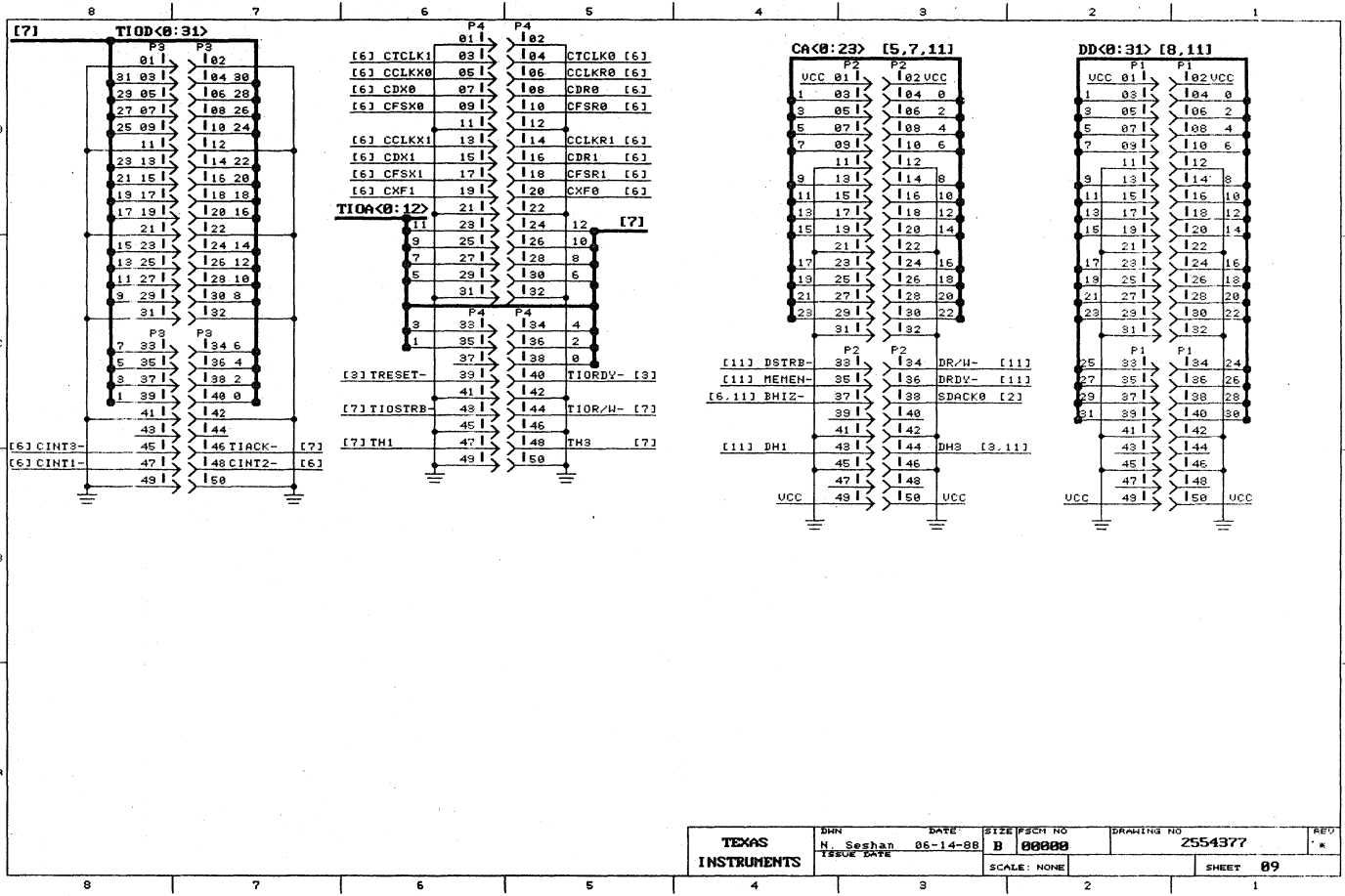

[3] MBANK STATUS6
[7] MMSTRB-
[7] MIOR/W-
CY7C164-25
UJ4
UH4
UG4
UF4
UJ3
UH3
UG3
UF3
I/O0
I/O1
I/O2
I/O3
[3,5,8] CIOD<0:31>
[3,7] BIOA<0:12>
TEXAS INSTRUMENTS
N. Seshan 06-14-88
B 00000
2554377
SCALE: NONE
SHEET 04

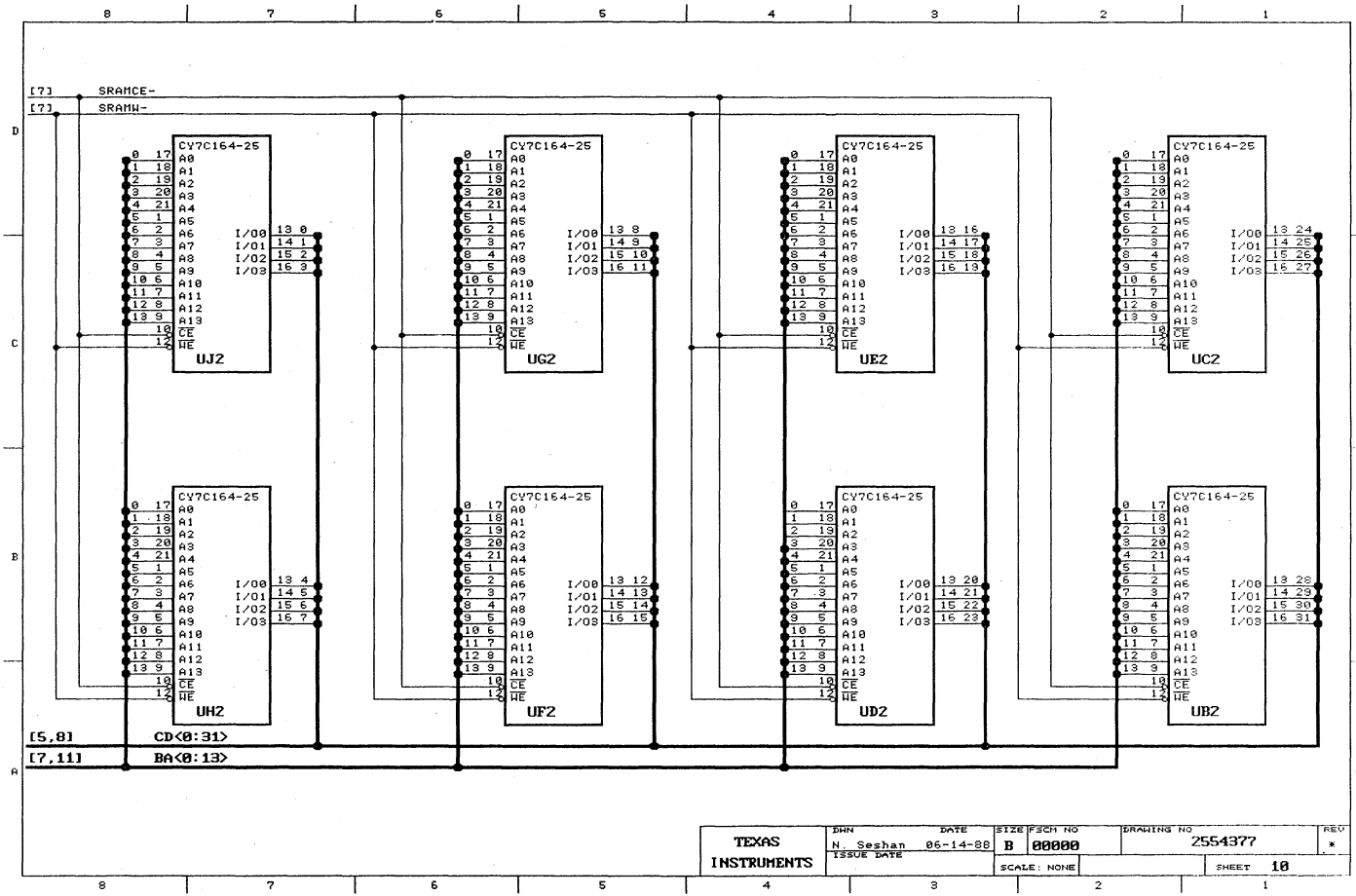

[3,4,8] CIOD<0:31>
[8,10] CD<0:31>
CA<0:23> [7,9,11]
CIOA<0:12> [3,7]
320C30
UB6
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15 D16 D17 D18 D19 D20 D21 D22 D23 D24 D25 D26 D27 D28 D29 D30 D31
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16 A17 A18 A19 A20 A21 A22 A23
R/W
STRB
HOLDA
CR/W-
CSTRB-
IOD0 IOD1 IOD2 IOD3 IOD4 IOD5 IOD6 IOD7 IOD8 IOD9 IOD10 IOD11 IOD12 IOD13 IOD14 IOD15 IOD16 IOD17 IOD18 IOD19 IOD20 IOD21 IOD22 IOD23 IOD24 IOD25 IOD26 IOD27 IOD28 IOD29 IOD30 IOD31
IOA0 IOA1 IOA2 IOA3 IOA4 IOA5 IOA6 IOA7 IOA8 IOA9 IOA10 IOA11 IOA12
DX0 DX1
SCOUT
X1
H1 H3
IACK
IOR/W
IOSTRB
MSTRB
BANKSEL
CDX0 [9]
CDX1 [9]
CSCOUT [6]
CH1 [3,7,11]
CH3 [6,7,11]
CIACK- [7]
CIOR/W- [3,7]
CIOSTRB- [3,7]
CMSTRB- [3,7]
CR/W- [11]
CSTRB- [11]
TEXAS INSTRUMENTS
DWN N. Seshan DATE 06-14-88
ISSUE DATE
SIZE B
FSCM NO 00000
DRAWING NO 2554377
REV .*
SCALE: NONE
SHEET 05

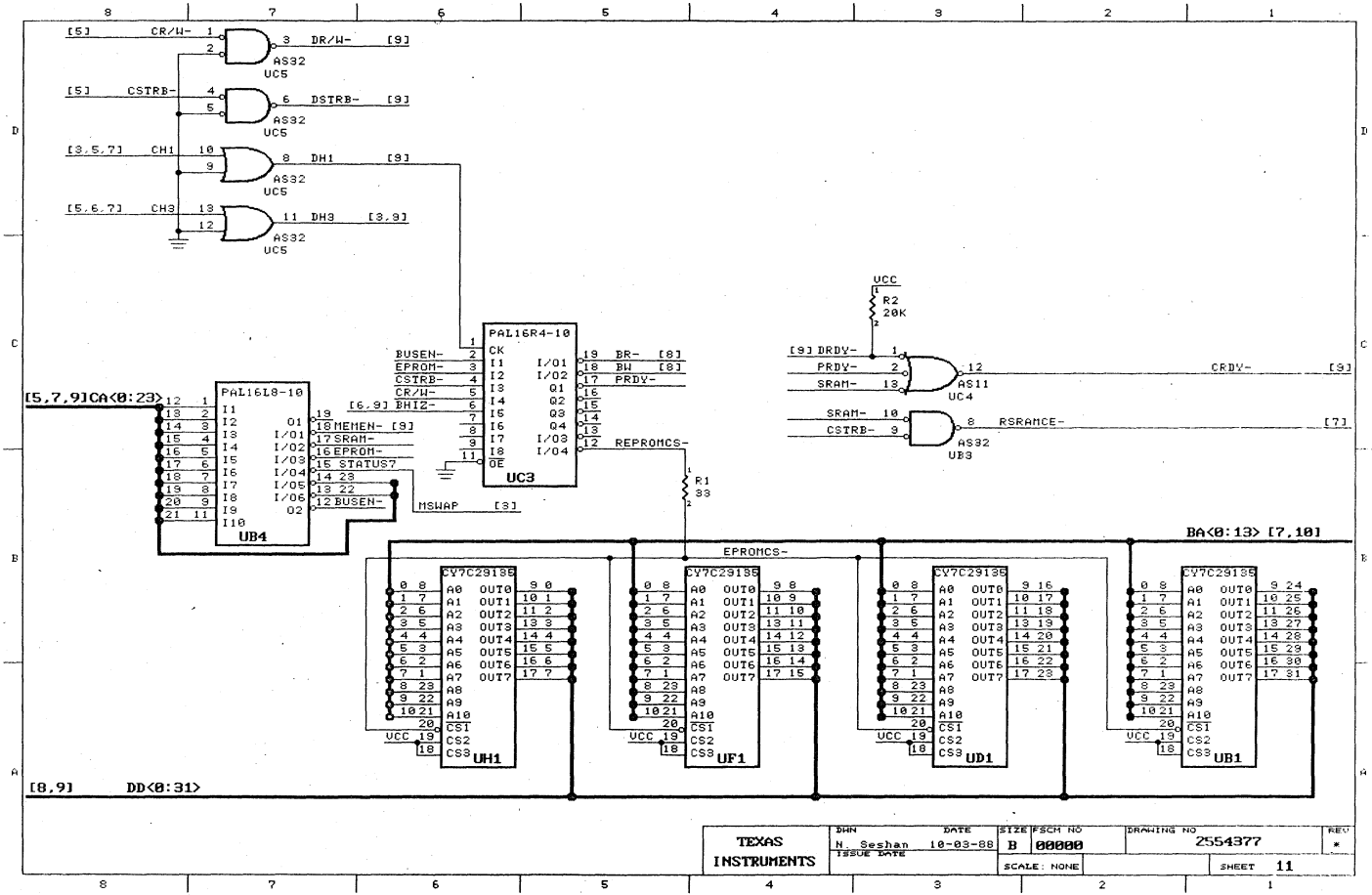

OSCILLATOR
OUT
UA6
CLK-
AS04
UA7
VCC
22K
RN10
RN12
RN9
RN11
320C30
UB6
RDY
HOLD
IORDY
X2/CLKIN
RESET
INT0
INT1
INT2
INT3
INT4
INT5
INT6
INT7
INT8
INTS0T
INTS0R
INTS1T
INTS1R
INTT0
INTT1
PADTOG
DR0
FSR0
DR1
FSR1
C0
C1
SCIN
MCS
MC/MP
CLKX0
FSX0
CLKR0
CLKX1
FSX1
CLKR1
TCLK0
TCLK1
XF0
XF1
LOCATOR
VDD
VSS
DVSS
CVSS
IVSS
MDVDD
PDVDD
IOVDD
ADVDD
DDVDD
VSUBS
VBBP
[11]CRDY-
HOLD-
[9] CIORDY-
CCLKIN
[3] CRESET-
[3] CINT0-
[9] CINT1-
[9] CINT2-
[9] CINT3-
CINT4-
CINT5-
CINT6-
CINT7-
CINT8-
CINT0T-
CINT0R-
CINT1T-
CINT1R-
CINTT0-
CINTT1-
[9] CDR0
[9] CFSR0
[9] CDR1
[9] CFSR1
CC0
CC1
CSCIN
CMOS
[9,11] BHIZ-
[3,9] TIORDY-
[3,9] TRESET-
CXF1
[5] CSCOUT
[5,7,11]CH3
AS08
UD8
PE1
SOUT
SCLK
TEXAS INSTRUMENTS
DRN
N. Seshan
DATE
10-03-88
ISSUE DATE
SIZE
B
FSCM NO
00000
DRAWING NO
2554377
REV
*
SCALE: NONE
SHEET
06

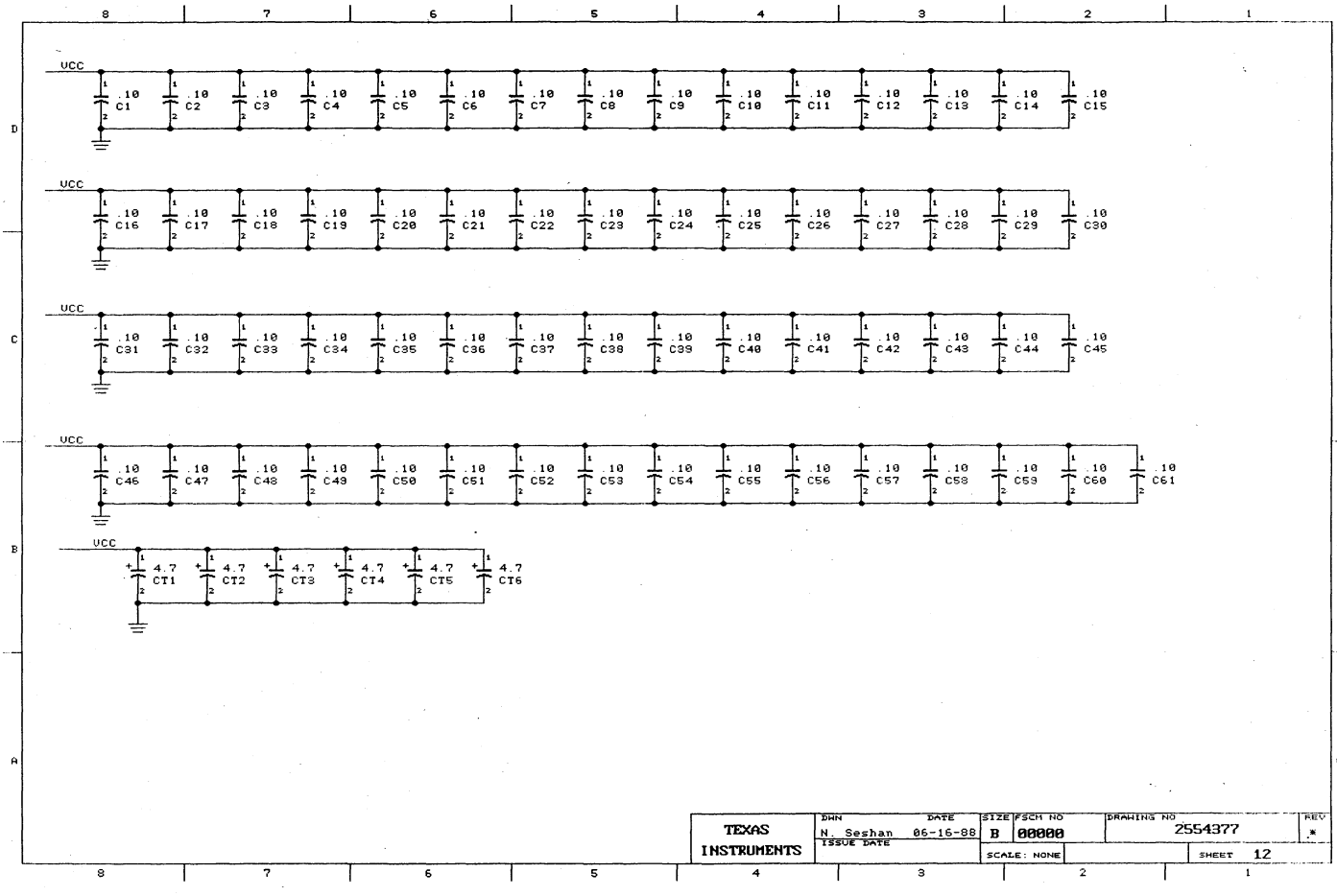

AS244
UA3
UA1
UD4
UD3
RN1
RN2
RN3
RN4
RN8
RN7
RN6
RN5
BA<0:13> [10,11]
BIOA<0:12>
[3,4]
[3,5] CIOA<0:12>
CA<0:23>
[5,9,11]
[5] CR/W-
[11] RSRAMCE-
CIOR/W-
[4,5] CMSTRB-
[5] CIOSTRB-
RR/W-
SRAMW- [10]
SRAMCE- [10]
RIOA0
RIOA1
RIOA2
RIOA3
RIOA4
RIOA5
RIOA6
RIOA7
RIOA8
RIOA9
RIOA10
RIOA11
RIOA12
RIOR/W-
RMSTRB- [4]
RIOSTRB- [3]
MIOR/W- [4]
MMSTRB- [4]
BIOSTRB- [3]
ALS541
UH9
UH10
UG10
TIOA<0:12>
[9]
[3,5,11] CH1
[5,6,11] CH3
[3,5] CIOR/W-
[3,5]CIOSTRB-
[5] CIACK-
TH1
TH3
TIOR/W-
TIOSTRB-
TIACK-
TEXAS INSTRUMENTS
N. Seshan
10-03-88
B
00000
2554375
SCALE: NONE
SHEET 07


[11]BW
[5,10] CD<0:31>
[11]BR-
[9] TIOD<0:31>
DD<0:31> [9,11]
CIOD<0:31> [3,4,5]
[3] TIOR
[3] TIOW-
ALS623
UJ1
UG1
UE1
UC1
UG6
UH6
UG7
UH7
TEXAS INSTRUMENTS
N. Seshan 06-14-88
B
00000
2554377
SCALE: NONE
SHEET 08


[7] TIOD<0:31>
[6] CINT3-
[6] CINT1-
TIACK- [7]
CINT2- [6]
[6] CTCLK1
[6] CCLKX0
[6] CDX0
[6] CFSX0
[6] CCLKX1
[6] CDX1
[6] CFSX1
[6] CXF1
TIOA<0:12>
CTCLK0 [6]
CCLKR0 [6]
CDR0 [6]
CFSR0 [6]
CCLKR1 [6]
CDR1 [6]
CFSR1 [6]
CXF0 [6]
[7]
[3] TRESET-
TIORDY- [3]
[7] TIOSTRB-
TIOR/W- [7]
[7] TH1
TH3 [7]
CA<0:23> [5,7,11]
[11] DSTRB-
[11] MEMEN-
[6,11] BHIZ-
DR/W- [11]
DRDY- [11]
SDACK0 [2]
[11] DH1
DH3 [3,11]
VCC
DD<0:31> [8,11]
TEXAS INSTRUMENTS
DWN N. Seshan
DATE 06-14-88
ISSUE DATE
SIZE B
FSCM NO 00000
DRAWING NO 2554377
REV *
SCALE: NONE
SHEET 09


[7] SRAMCE-
[7] SRAMW-
CV7C164-25
UJ2
UG2
UE2
UC2
UH2
UF2
UD2
UB2
[5,8] CD<0:31>
[7,11] BA<0:13>
TEXAS INSTRUMENTS
DWN N. Seshan DATE 06-14-88
ISSUE DATE
SIZE B
FSCM NO 00000
DRAWING NO 2554377
REV *
SCALE: NONE
SHEET 10


[5] CR/W- 1 2 AS32 UC5 3 DR/W- [9]
[5] CSTRB- 4 5 AS32 UC5 6 DSTRB- [9]
[3,5,7] CH1 10 9 AS32 UC5 8 DH1 [9]
[5,6,7] CH3 13 12 AS32 UC5 11 DH3 [3,9]
PAL16R4-10 UC3
BUSEN- EPROM- CSTRB- CR/W- [6,9] BHIZ-
CK I1 I2 I3 I4 I5 I6 I7 I8 OE
I/O1 I/O2 O1 O2 O3 O4 I/O3 I/O4
19 BR- [8] 18 BW [8] 17 PRDY- REPROMCS-
MSWAP [3]
R1 33
UCC R2 20K
[9] DRDY- PRDY- SRAM- AS11 UC4 12 CRDY- [9]
SRAM- CSTRB- AS32 UB3 8 RSRAMCE- [7]
[5,7,9] CA<0:23>
PAL16L8-10 UB4
I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 O1 I/O1 I/O2 I/O3 I/O4 I/O5 I/O6 O2
18 MEMEN- [9] 17 SRAM- 16 EPROM- 15 STATUS? 12 BUSEN-
EPROMCS-
BA<0:13> [7,10]
CY7C291B6 UH1 UF1 UD1 UB1
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 CS1 CS2 CS3
OUT0 OUT1 OUT2 OUT3 OUT4 OUT5 OUT6 OUT7
VCC
[8,9] DD<0:31>
TEXAS INSTRUMENTS
DWN N. Seshan DATE 10-03-88 ISSUE DATE
SIZE B FSCH NO 00000 DRAWING NO 2554377 REV *
SCALE: NONE SHEET 11


VCC
.10 C1
.10 C2
.10 C3
.10 C4
.10 C5
.10 C6
.10 C7
.10 C8
.10 C9
.10 C10
.10 C11
.10 C12
.10 C13
.10 C14
.10 C15
VCC
.10 C16
.10 C17
.10 C18
.10 C19
.10 C20
.10 C21
.10 C22
.10 C23
.10 C24
.10 C25
.10 C26
.10 C27
.10 C28
.10 C29
.10 C30
VCC
.10 C31
.10 C32
.10 C33
.10 C34
.10 C35
.10 C36
.10 C37
.10 C38
.10 C39
.10 C40
.10 C41
.10 C42
.10 C43
.10 C44
.10 C45
VCC
.10 C46
.10 C47
.10 C48
.10 C49
.10 C50
.10 C51
.10 C52
.10 C53
.10 C54
.10 C55
.10 C56
.10 C57
.10 C58
.10 C59
.10 C60
.10 C61
VCC
4.7 CT1
4.7 CT2
4.7 CT3
4.7 CT4
4.7 CT5
4.7 CT6
TEXAS INSTRUMENTS
DWN N. Seshan
DATE 06-16-88
ISSUE DATE
SIZE B
FSCM NO 00000
DRAWING NO 2554377
REV *
SCALE: NONE
SHEET 12

NOTES: UNLESS OTHERWISE SPECIFIED:

1. ALL AS, ALS, LS DEVICES ARE PREFIXED WITH AN SN74
2. VCC IS APPLIED TO PIN 8 OF ALL 8-PIN IC's, PIN 14 OF ALL 14-PIN IC's, PIN 16 OF ALL 16-PIN IC's, PIN 20 OF ALL 20-PIN IC's, ETC.
3. GROUND IS APPLIED TO PIN 4 OF ALL 8-PIN IC's, PIN 7 OF ALL 14-PIN IC's, PIN 8 OF ALL 16-PIN IC's, PIN 10 OF ALL 20-PIN IC's, ETC.
4. DEVICE TYPE, PIN NUMBERS, AND REFERENCE DESIGNATOR OF GATES ARE SHOWN AS FOLLOWS:

00 AND 04 = DEVICE TYPES
1, 2, AND 3 = PIN NUMBERS
U06 AND U07 = REFERENCE DESIGNATORS

5. RESISTANCE VALUES ARE IN OHMS.
6. RESISTORS ARE 1/4 WATT, 5%
7. CAPACITANCE VALUES ARE IN MICROFARADS.

REVISIONS			
REV	DESCRIPTION	DATE	APPROVED

COMPUTER GENERATED DRAWING ; DO NOT REVISE MANUALLY

PARTS LIST

-1 QTY	ITEM NO	PART OR IDENTIFYING NUMBER	NOMENCLATURE OR DESCRIPTION	NOTES

REVISION STATUS OF SHEETS										
REV	A	A	*	*	*	*	*	*		
SH	1	2	3	4	5	6	7	8	9	10
REV										
SH	11	12	13	14	15	16	17	18	19	20
REV										
SH	21	22	23	24	25	26	27	28	29	30
REV										
SH	31	32	33	34	35	36	37	38	39	40

2554395 | 7119

NEXT ASSY | USED ON

APPLICATION

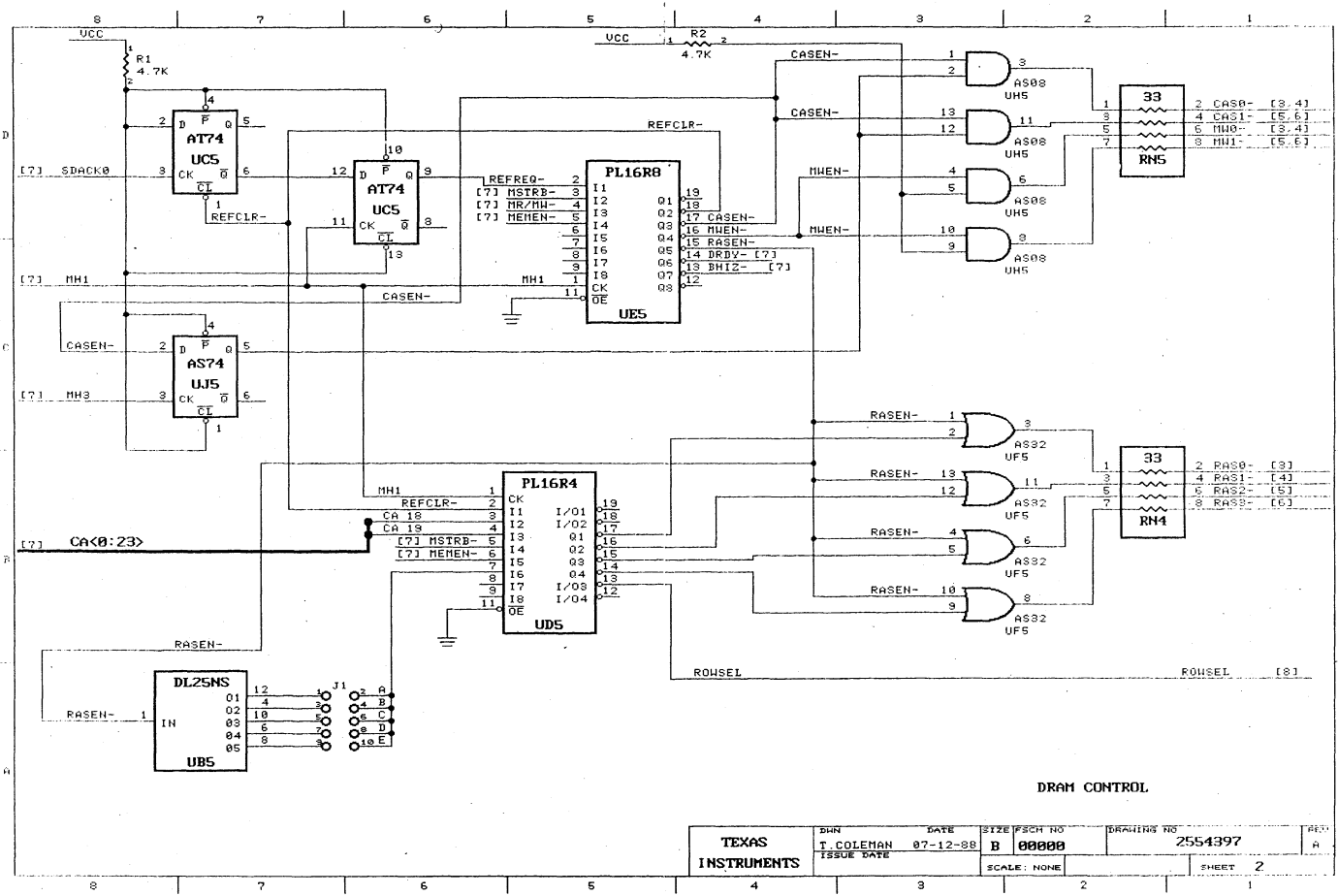
DWN	T.COLEMAN	DATE 07-12-88
CHK		DATE
ENGR	T.COOMES	DATE 07-12-88
APVD ENGR		DATE
QA		DATE
APVD MFG		DATE
RLSE		DATE

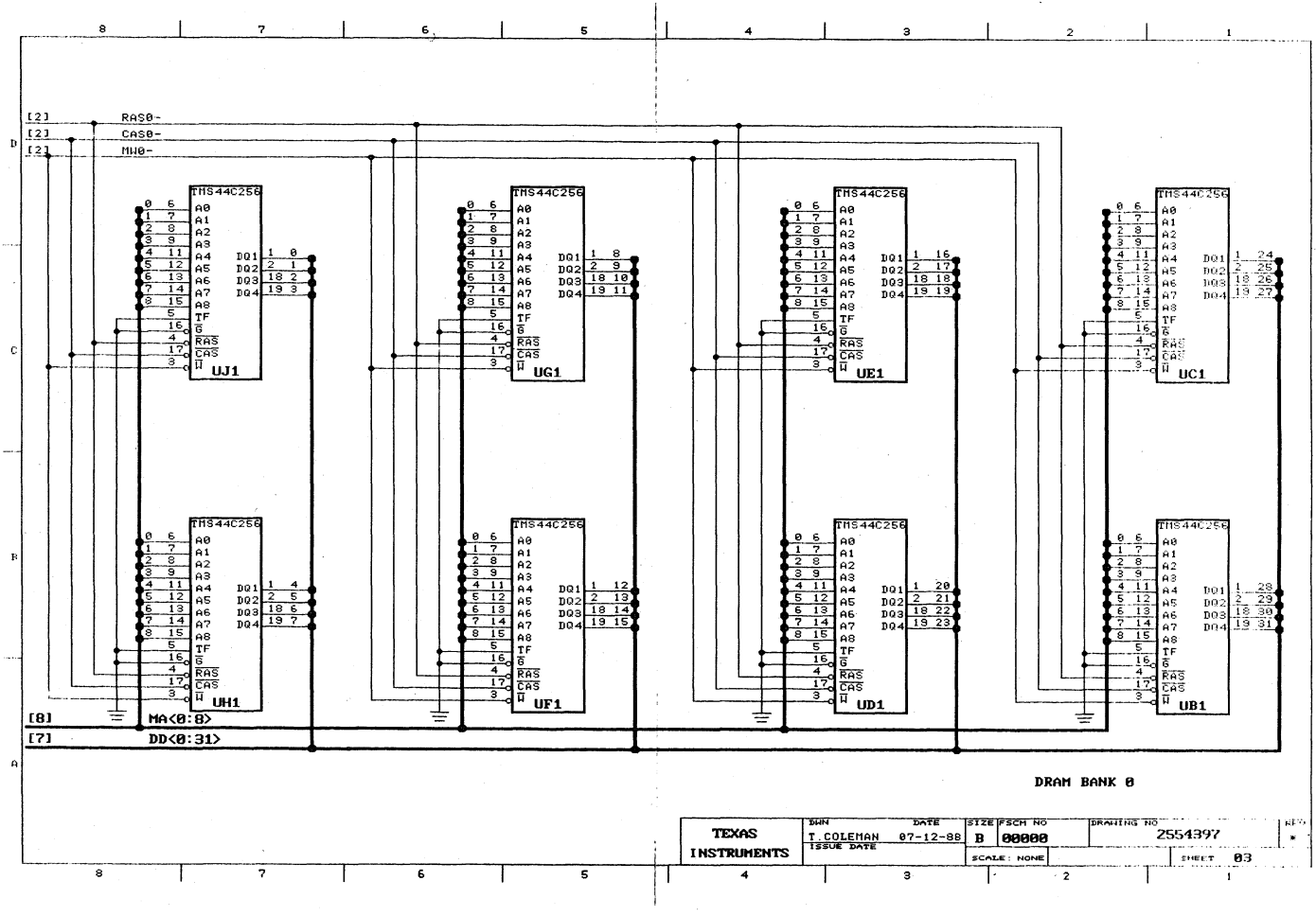
TEXAS INSTRUMENTS
Data Systems Group

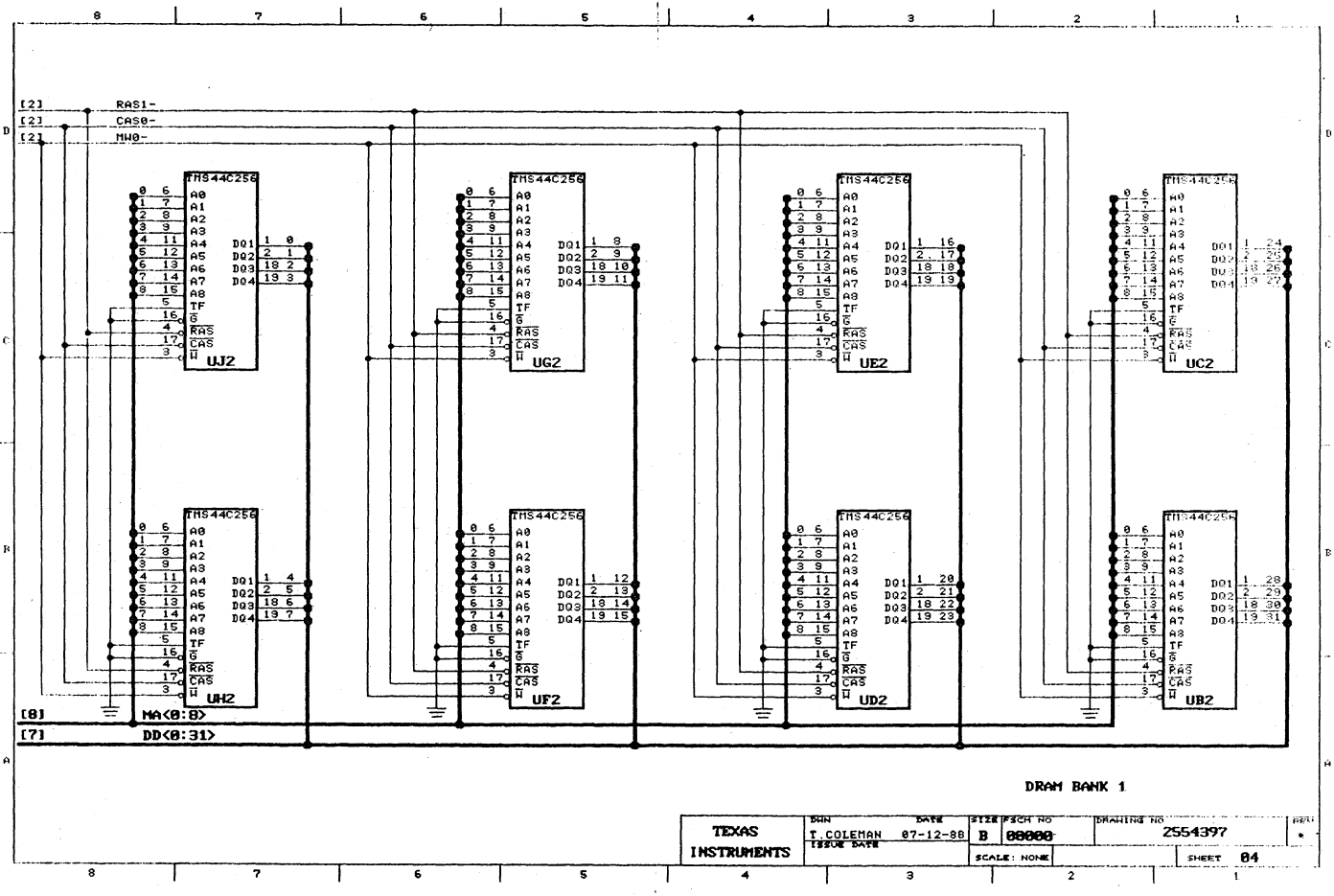
320C30 SWDS DRAM MODULE

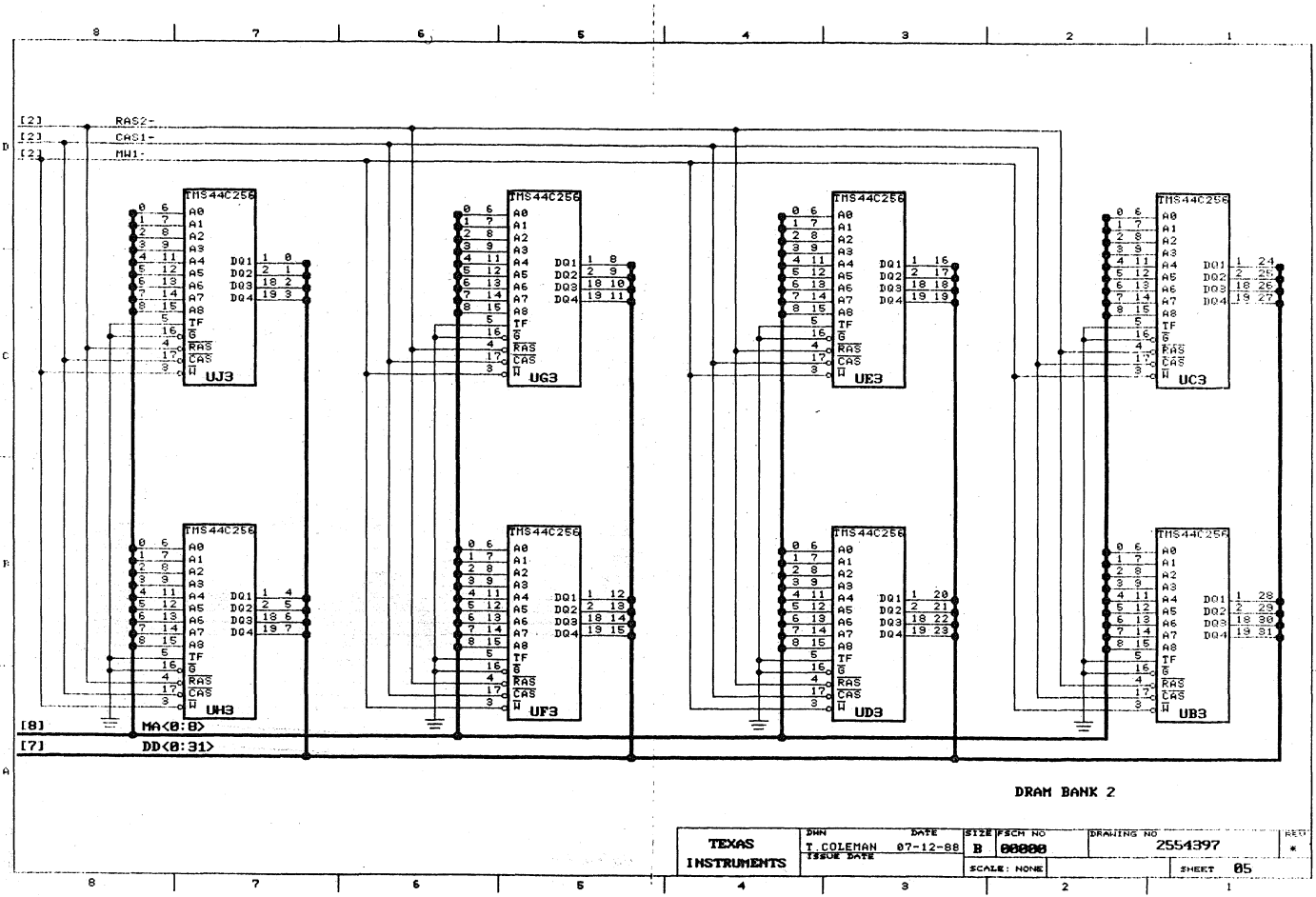
SIZE	FSCM NO	DRAWING NO	REV
B	00000	2554397	H

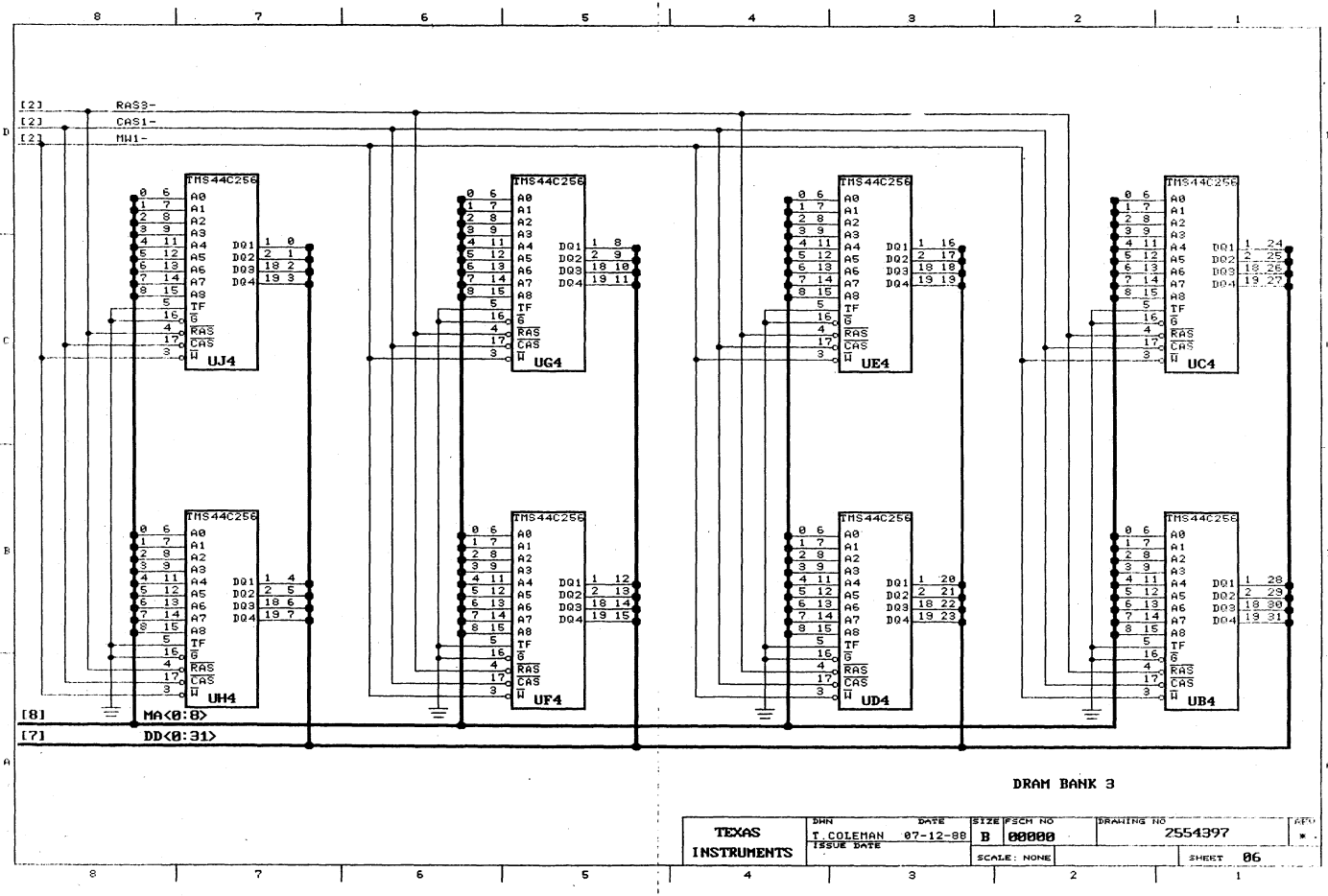
SCALE NONE | SHEET 1 OF 8

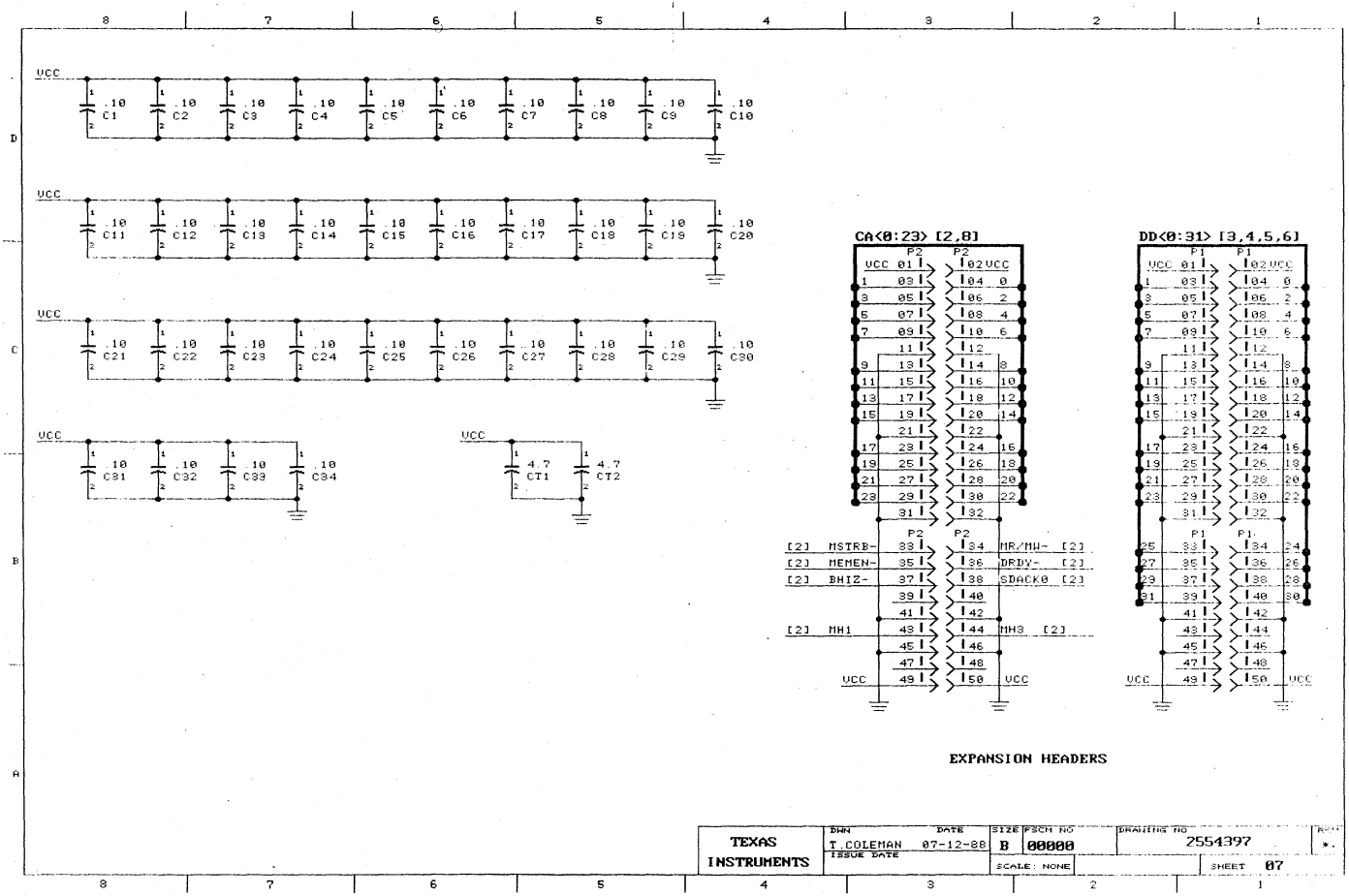

VCC
R1
4.7K
R2
4.7K
AT74
UC5
AT74
UC5
AS74
UJ5
PL16R8
UE5
PL16R4
UD5
DL25NS
UB5
J1
AS08
UH5
AS32
UF5
33
RN5
33
RN4
CASEN-
REFCLR-
MWEN-
RASEN-
ROWSEL
[7] SDACK0
[7] MH1
[7] MH3
[7] CA<0:23>
REFRE0-
[7] MSTRB-
[7] MR/MW-
[7] MEMEN-
MH1
CASEN-
REFCLR-
CA 18
CA 19
[7] MSTRB-
[7] MEMEN-
RASEN-
ROWSEL
[8]
14 DRDY- [7]
18 BHIZ- [7]
2 CAS0- [3,4]
4 CAS1- [5,6]
6 MW0- [3,4]
8 MW1- [5,6]
2 RAS0- [3]
4 RAS1- [4]
6 RAS2- [5]
8 RAS3- [6]
DRAM CONTROL
TEXAS INSTRUMENTS
T.COLEMAN
07-12-88
B
00000
2554397
A
SCALE: NONE
SHEET 2

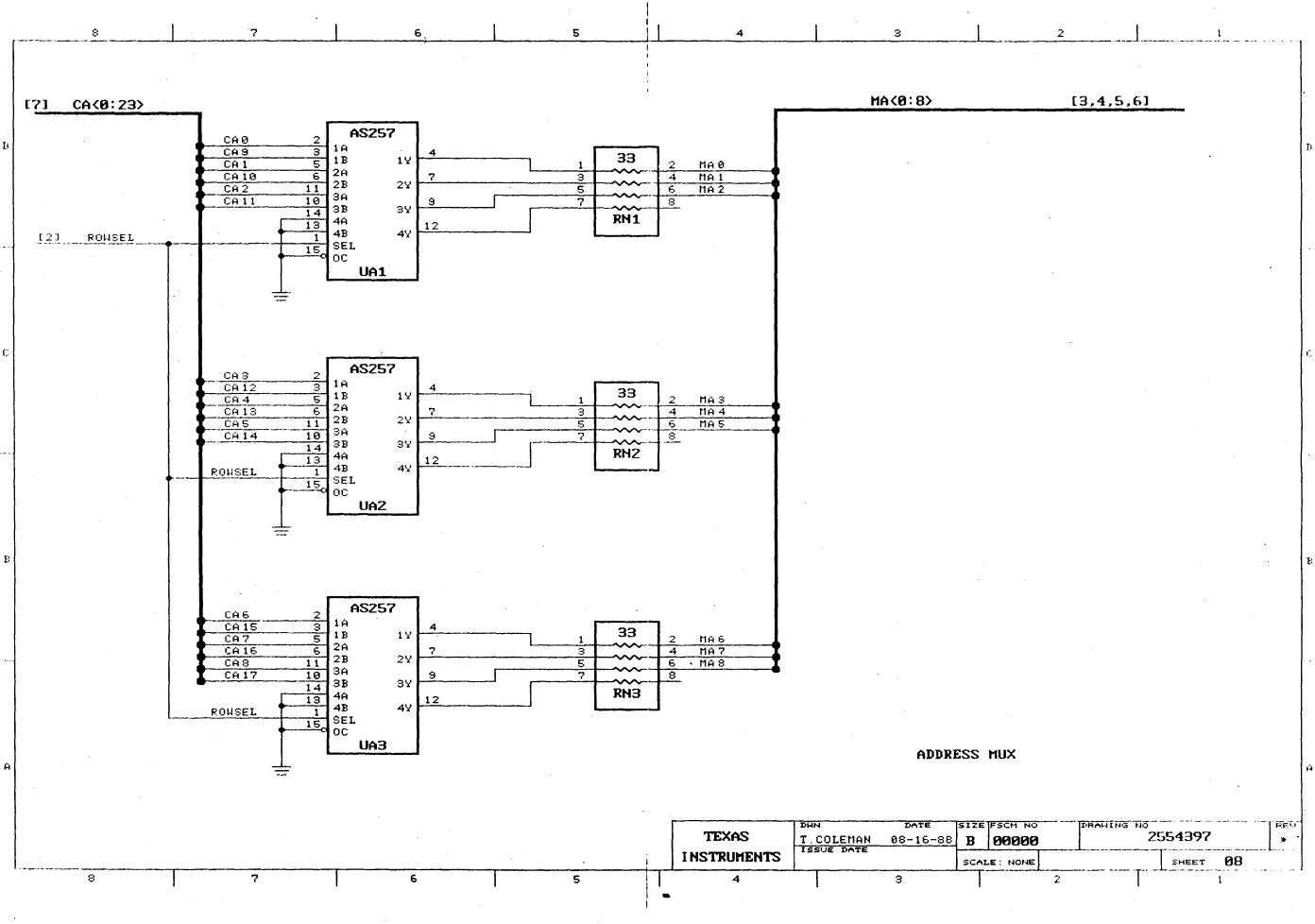

[2] RAS0-
[2] CAS0-
[2] MW0-
TMS44C256
UJ1
UG1
UE1
UC1
UH1
UF1
UD1
UB1
[8] MA<0:8>
[7] DD<0:31>
DRAM BANK 0
TEXAS INSTRUMENTS
T.COLEMAN 07-12-88
B
00000
2554397
SCALE: NONE
SHEET 03


[2] RAS1-
[2] CAS0-
[2] MW0-
TMS44C256
UJ2
UG2
UE2
UC2
UH2
UF2
UD2
UB2
[8] MA<0:8>
[7] DD<0:31>
DRAM BANK 1
TEXAS INSTRUMENTS
T.COLEMAN
07-12-88
SIZE B
SCALE: NONE
SHEET 04


[2] RAS2-
[2] CAS1-
[2] MW1-
TMS44C256
UJ3
UG3
UE3
UC3
UH3
UF3
UD3
UB3
[8] MA<0:8>
[7] DD<0:31>
DRAM BANK 2
TEXAS INSTRUMENTS
T.COLEMAN 07-12-88
SIZE B
DRAWING NO 2554397
SCALE: NONE
SHEET 05


[2] RAS3-
[2] CAS1-
[2] MW1-
TMS44C256
UJ4
UG4
UE4
UC4
UH4
UF4
UD4
UB4
[8] MA<0:8>
[7] DD<0:31>
DRAM BANK 3
TEXAS INSTRUMENTS
T.COLEMAN 07-12-88
B 00000
2554397
SCALE: NONE
SHEET 06


VCC
.10 C1
.10 C2
.10 C3
.10 C4
.10 C5
.10 C6
.10 C7
.10 C8
.10 C9
.10 C10
.10 C11
.10 C12
.10 C13
.10 C14
.10 C15
.10 C16
.10 C17
.10 C18
.10 C19
.10 C20
.10 C21
.10 C22
.10 C23
.10 C24
.10 C25
.10 C26
.10 C27
.10 C28
.10 C29
.10 C30
.10 C31
.10 C32
.10 C33
.10 C34
4.7 CT1
4.7 CT2
CA<0:23> [2,8]
DD<0:31> [3,4,5,6]
P2
P1
[2] MSTRB-
[2] MEMEN-
[2] BHIZ-
[2] MH1
MR/MW- [2]
DRDY- [2]
SDACK0 [2]
MH3 [2]
EXPANSION HEADERS
TEXAS INSTRUMENTS
DWN
DATE
T.COLEMAN 07-12-88
ISSUE DATE
SIZE
B
FSCM NO
00000
DRAWING NO
2554397
SCALE: NONE
SHEET 07


[7] CA(0:23)
MA(0:8)
[3,4,5,6]
AS257
UA1
UA2
UA3
33
RN1
RN2
RN3
[2] ROWSEL
ROWSEL
CA0
CA9
CA1
CA10
CA2
CA11
CA3
CA12
CA4
CA13
CA5
CA14
CA6
CA15
CA7
CA16
CA8
CA17
MA0
MA1
MA2
MA3
MA4
MA5
MA6
MA7
MA8
ADDRESS MUX
TEXAS INSTRUMENTS
T.COLEMAN 08-16-88
B 00000
2554397
SCALE: NONE
SHEET 08

TMS320 Bibliography

Since the TMS32010 was disclosed in 1982, the TMS320 family has received an ever-increasing amount of recognition. The number of outside parties contributing to the extensive development support offered by Texas Instruments is rapidly growing. Many technical articles are being written about TMS320 applications in the field of digital signal processing.

The following articles and papers have been published since 1982 regarding the Texas Instruments TMS320 Digital Signal Processors. Readers who are interested in gaining further information about these processors and their applications may obtain copies of these articles/papers from their local or university library.

The articles are broken down into 12 different application categories. Articles in each category are in reverse chronological order (most recent first). Articles having the same publication date are shown in alphabetical order by authors name.

The application categories are:

1) General Purpose DSP
2) Graphics/Imaging
3) Instrumentation
4) Voice/Speech
5) Control
6) Military
7) Telecommunications
8) Automotive
9) Consumer
10) Industrial
11) Medical
12) Development Support

General Purpose DSP

1) R. Chassaing, "A Senior Project Course in Digital Signal Processing with the TMS320," *IEEE Transactions on Education*, USA, Volume 32, Number 2, pages 139–145, May 1989.
2) P.E. Papamichalis, C.S. Burrus, "Conversion of Digit-Reversed to Bit-Reversed Order in FFT Algorithms," *Proceedings of ICASSP 89*, USA, pages 984–987, May 1989.
3) P.E. Papamichalis, "Application, Progress and Trends in Digital Signal Processing," *Proceedings of Mikroelktronik Conference*, Baden-Baden, March 1989.
4) R. Chassaing, "Adaptive Filtering with the TMS320C25 Digital Signal Processor," *Proceedings of 1989 ASEE Conference,* USA, pages 215–217, 1989.
5) P.E. Papamichalis, R. Simar, Jr., "The TMS320C30 Floating-Point Digital Signal Processor," *IEEE Micro Magazine*, USA, pages 13–29, December 1988.
6) K. Rogers, "The Real-Time Thing (Digital Signal Controller)," *Electronic Engineering Times*, USA, Number 506, page 85, October 1988.
7) P.E. Papamichalis, "Impact of DSP Devices on Fast Algorithms," *Proceedings of the 1988 IEEE DSP Workshop*, USA, September 1989.

8) G. Umamaheswari, C. Eswaran, A. Jhunjhunwala, "Signal Processing with a Dual-Bank Memory," *Microprocessor Microsystems,* Great Britain, Volume 12, Number 4, pages 206–210, May 1988.
9) G. Castellini, P. Luigi, E. Liani, L. Pierucci, F. Pirri, S. Rocchi, "A Multiprocessor Structure Based on Commercial DSP," *Proceedings of ICASSP 88,* USA, Volume V, page 2096, April 1988.
10) M.R. Civanlar, R.A. Nobakht, "Optimal Pulse Shape Design Using Projections onto Convex Sets," *Proceedings of ICASSP 88,* USA, Volume D, p. 1874, April 1988.
11) L.J. Eriksson, M.C. Allie, C.D. Bremigan, R.A. Greiner, "Active Noise Control Using Adaptive Digital Signal Processing," *Proceedings of ICASSP 88,* USA, Volume A, page 2594, April 1988.
12) G. Mirchandani, D.D. Ogden, "Experiments in Partitioning and Scheduling Signal Processing Algorithms for Parallel Processing," *Proceedings of ICASSP 88,* USA, Volume D, page 1690, April 1988.
13) P. Papamichalis, "FFT Implementation on the TMS320C30," *Proceedings of ICASSP 88,* USA, Volume D, page 1399, April 1988.
14) A.C. Rotger-Mora, "An N-Dimensional SIMD Ring Architecture for Implementing Very Large Order Adaptive Digital Filters," *Proceedings of ICASSP 88,* USA, Volume V, page 2140, April 1988.
15) J. Santos, J. Parera, M. Veiga, "A Hypercube Multiprocessor for Digital Signal Processing Algorithm Research," *Proceedings of ICASSP 88,* USA, Volume D, page 1698, April 1988.
16) R. Simar, A. Davis, "The Application of High-Level Languages to Single-Chip Digital Signal Processors," *Proceedings of ICASSP 88,* USA, Volume D, page 1678, April 1988.
17) K. Bala, "Running on Embedded Power. (Dedicated 32-Bit Microprocessors Used in New Microcontrollers)(Technology Trends: Microprocessors and Peripherals)," *Electronic Engineering Times,* USA, Number 478, page 34, March 1988.
18) J. Cooper, "DSP Chip Speeds VME Transfer," *ESD: Electronic Systems Design,* USA, Volume 18, Number 3, pages 47,48,50,51, March 1988.
19) L. Vieira de Sa, F. Perdigao, "A Microprocessing System for the TMS32020," *Microprocessing Microprogramming,* Netherlands, Volume 23, Number 1–5, pages 221–225, March 1988.
20) G. Wade, "Offset FFT and Its Implementation on the TMS320C25 Processor," *Microprocessing Microsystems,* Great Britain, Volume 12, Number 2, pages 76–82, March 1988.
21) R. Chassaing, "Digital Broadband Noise Synthesis by Multirate Filtering Using the TMS320C25," *Proceedings of 1988 ASEE Conference,* USA, pages 394–397, 1988.
22) R. Chassaing, "A Senior Project Course on Applications in Digital Signal Processing with the TMS320," *Proceedings of 1988 ASEE Conference,* USA, pages 354–359, 1988.
23) L.N. Bohs, R.C. Barr, "Real-Time Adaptive Sampling with the Fan Method," *Proceedings of the Ninth Annual Conference of the IEEE Engineering in Medicine and Biology Society,* USA, Volume 4, pages 1850–1851, November 1987.
24) T. Kimura, Y. Inabe, T. Hayashi, K. Uchimura, K. Hamazato, "Dual-Chip SLIC Using VLSI Technology," *Conference Record of GLOBECOM Tokyo '87,* Volume 3, pages 1766–1770, November 1987.

25) W.S. Gass, R.T. Tarrant, T. Richard, B.I. Pawate, M. Gammel, P.K. Rajasekaran, R.H. Wiggins, C.D. Covington, "Multiple Digital Signal Processor Environment for Intelligent Signal Processing," *Proceedings of the IEEE*, USA, Volume 75, Number 9, pages 1246–1259, September 1987.
26) L. Johnson, R. Simar, Jr., "A High Speed Floating Point DSP," *Conference Record of MIDCON/87*, USA, pages 396–399, September 1987.
27) K.S. Lin, G.A. Frantz, R. Simar, Jr., "The TMS320 Family of Digital Signal Processors," *Proceedings of the IEEE*, USA, Volume 75, Number 9, pages 1143–1159, September 1987.
28) S.L. Martin, "Wave of Advances Carry DSPs To New Horizons. (Digital Signal Processing)," *Computer Design*, USA, Volume 26, Number 17, pages 69–82, September 1987.
29) C. Murphy, A. Coats, J. Conway, P. Colditz, P. Rolfe, "Doppler Ultrasound Signal Analysis Based on the TMS320 Signal Processor," *27th Annual Scientific Meeting of the Biological Engineering Society*, Great Britain, Volume 10, Number 2, pages 127–129, September 1987.
30) G.S. Kang, L.J. Fransen, "Experimentation With An Adaptive Noise-Cancellation Filter," *IEEE Transactions on Circuits and Systems*, USA, Volume CAS-34, Number 7, pages 753–758, July 1987.
31) R. Chassaing, "Applications in Digital Signal Processing with the TMS320 Digital Signal Processor in an Undergraduate Laboratory," *Proceedings of the 1987 ASEE Annual Conference*, USA, Volume 3, pages 1320–1324, June 1987.
32) D.W. Horning, "An Undergraduate Digital Signal Processing Laboratory," *Proceedings of the 1987 ASEE Annual Conference*, USA, Volume 3, pages 1015–1020, June 1987.
33) D. Locke, "Digitising In The Gigahertz Range," *IEE Colloguium on Advanced A/D Conversion Techniques*, Great Britain, Digest Number 48, 10/1–4, April 1987.
34) S. Orui, M. Ara, Y. Orino, E. Sazuki, H. Makino, "Realization of IIR Filter using the TMS320," *Resident Reports of Kogakuin University*, Japan, Number 62, pages 195–204, April 1987.
35) R. Simar, T. Leigh, P. Koeppen, J. Leach, J. Potts, D. Blalock, "A 40 MFLOPS Digital Signal Processor: The First Supercomputer on a Chip," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396–0, Volume 1, pages 535–538, April 1987.
36) R. Simar, "TMS320: Texas Instruments Family of Digital Signal Processors," *Proceedings of SPEECH TECH 87*, USA, pages 42–47, April 1987.
37) G.Y. Tang, B.K. Lien, "A Multiple Microprocessor System For General DSP Operation," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396–0, Volume 2, pages 1047–1050, April 1987.
38) L. Vieira de Sa, "Second MicroProcessor Enhances TMS32020 System," *EDN: Electronic Design News*, USA, Volume 32, Number 9, pages 230–232, April 1987.
39) T.J. Moir, T.G. Vishwanath, D.R. Campbell, "Real-Time Self-Tuning Deconvolution Filter and Smoother," *International Journal of Control*, Great Britain, Volume 45, Number 3, pages 969–985, March 1987
40) R. Simar, M. Hames, "CMOS DSP Packs Punch of a Supercomputer," *EDN: Electronic Design News*, USA, Volume 35, Number 7, pages 103–106, March 1987.

41) S. Sridharan, "On Improving the Performance of Digital Filters Designed Using the TMS32010 Signal Processor," *Journal of Electrical and Electronic Engineers of Australia, Australia*, Volume 7, Number 1, pages 80–82, March 1987.
42) R. McCammon, "Software Routine Probes TMS32010 Code," *EDN: Electronic Design News,* USA, Volume 32, Number 4, pages 200,202, February 1987.
43) J. Prado, R. Alcantara, "A Fast Square-Rooting Algorithm Using A Digital Signal Processor," *Proceedings of IEEE*, USA, Volume 75, Number 2, pages 262–264, February 1987.
44) T.G. Vishwanath, D.R. Camppbell, T.J. Moir, "Real-Time Implementation Using a TMS32010 Microprocessor," *IEEE Transactions on Industrial Electronics*, USA, Volume 1E-34, Number 1, pages 115–118, February 1987.
45) R Chassaing, "Applications in Digital Signal Processing with the TMS320 Digital Signal Processor in an Undergraduate Laboratory," *Proceedings of 1987 ASEE Conference*, USA, pages 1320–1324, 1987.
46) R.M. Sovacool, "EPROM Enhances TMS32020 Mu C's Memory," *EDN: Electronic Design News,* USA, Volume 32, Number 1, page 231, 1987.
47) F. Kocsis, F. Marx, "Fast DFT Modules For The TMS32010 Digital Signal Processor," *Meres and Automation, Hungary*, Volume 35, Number 1, pages 6–11, 1987.
48) Y.V.V.S. Murty, W.J. Smolinski, "Digital Filters for Power System Relaying," *International Journal of Energy Systems*, USA, Volume 7, Number 3, pages 125–129, 1987.
49) S. Wang, "The TMS32010 High Speed Processor and Its Applications," *Mini-Micro Systems*, China, Volume 8, Number 3, pages 24–32, 1987.
50) G.A. Frantz, K.S. Lin, J.B. Reimer, J. Bradley, "The Texas Instruments TMS320C25 Digital Signal Microcomputer," *IEEE Microelectronics,* USA, Volume 6, Number 6, pages 10–28, December 1986.
51) P. Renard, "A/D Converters: The Advantage of a Mixture of Techniques," *Mesures, France*, Volume 51, Number 16, pages 80–81, December 1986.
52) M. Ara, E. Suzuki, "Design of Real Time Filter Using DSP," *Resident Reports of Kogakuin University*, Japan, Number 61, pages 115–127 October 1986.
53) J. Reidy, "Connection of a 12-Bit A/D Converter to Fast DSPs," *Electronik, Germany*, Volume 35, Number 22, pages 132–134, October 1986.
54) G.R. Steber, "Implementation of Adaptive Filters on the TMS32010 DSP Microcomputer," *Proceedings of IECON 86*, Catalog Number 86CH2334–1, Volume 2, pages 653–656, September/October 1986.
55) D. Collins, M.A. Rahman, "Digital Filter Design Using The TMS320 Digital Signal Processor," *Proceedings of EUSIPCO-86*, Volume 1pages 163–166, September 1986.
56) R. Simar, Jr., J.B. Reimer, "The TMS320C25: A 100 ns CMOS VLSI Digital Signal Processor," *1986 Workshop on Applications of Signal Processing to Audio and Acoustics*, September 1986.
57) J. Dudas, A. Stipkovits, E. Simonyi, "On The recursive Momentary Discrete Fourier Transform," *Proceedings of EUSIPCO-86*, Volume 1, pages 303–306, September 1986.
58) E. Feder, "Digital Signal Processor – General Purpose or Dedicated?," Electronics Industry, France, Number 111, pages 74–82, September 1986.
59) K. Herberger, "The Use of Signal Processors For Simulating Data Circuits," *Proceedings of EUSIPCO-86*, Volume 2, pages 1109–1112, September 1986.

60) K. Kassapoglou, P. Hulliger, "Implementation of Recursive Least Squares Identification Algorithm on The TMS320," *Proceedings of EUSIPCO-86*, Volume 2, pages 1263–1266, September 1986.
61) G. Lucioni, "General Processor Application; CAD Tool For Filter Design," *Proceedings of EUSIPCO-86,* Volume 2, pages 1335–1338, September 1986.
62) R. Schapery, "A 10-MIP Digital Signal Processor From Texas Instruments," *Conference Record of Midcon 86,* USA, 1/2/1–11, September 1986.
63) "DSP Microprocessors," *Inf. Elettronica*, Italy, Volume 14, Number 7–8, pages 21–28,
64) R.L. Barnes, S.H. Ardalan, "Multiprocessor Architecture For Implementing Adaptive Digital Filters," *Conference Record of ICC-86*, Catalog Number 86CH2314–3, Volume 1, pages 180–185, June 1986.
65) A.D.E. Brown, "EPROMS Simplify TMS32010 Memory System," *EDN: Electronic Design News*, USA, Volume 31, Number 13, page 230, June 1986.
66) T. Kolehamainen, T. Saramaki, M. Renfors, Y. Neuvo, "Signal Processor Implementation of Computationally Efficient FIR Filter Structures – Theory and Practice," *2nd Nordic Symposium on VLSI in Computers and Communications,* 10 pages, June 1986.
67) T.G. Marshall Jr.,"Transform Methods For Developing Parallel Algorithms For Cyclic-Block Signal Processing," *Conference Record of ICC-86*, Catalog Number 86CH2314–3, Volume 1, pages 288–294, June 1986.
68) S. Abiko, M. Hashizume, Y. Matsushita, K. Shinozaki, T. Takamizawa, C. Erskine, S. Magar, "Architecture and Applications of a 100-ns CMOS VLSI Digital Signal Processor," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243–4, Volume 1, pages 393–396., April 1986.
69) T.P. Barnwell, "Algorithm Development and Multiprocessing Issues for DSP Chips," *Proceedings of Speech Technology 86,* April 1986.
70) W. Gass, "TMS32020 – The Quick and Easy Solution to DSP Problems," *Proceedings of Speech Technology 86*, April 1986.
71) M. Hashizume, S. Abiko, Y. Matsushita, K. Shinozaki,T. Takamizawa, S. Magar, J. Reimer, "A 100-ns CMOS VLSI Digital Signal Processor Using Double Level Metal Structure," *Semiconductor Group 1986 Technical Meeting,* April 1986.
72) R.E. Morley, A.M. Engebretson, and J.G. Trotta, "A Multiprocessor Digital Signal Processing System for Real-Time Audio Applications," *IEEE Transactions on Acoustics, Speech and Signal Processing*, USA, Volume ASSP-34, Number 2, April 1986.
73) S.G. Smith, A. Fitzgerald, P.B. Denyer, D. Renshaw, N.P. Wooten, R. Creasey, "A Comparison of Micro-DSP And Silicon Compiler Implementations of a Polyphase-Network Filter Bank," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243–4, Volume 3, pages 2207–2210, April 1986.
74) J. Reimer, M. Hames, "Next Generation CMOS Chip Stakes High-Performance Claim on 10-MIPS DSP Operations," *Electronic Design*, USA, Volume 34, Number 8, pages 141–146, April 1986.
75) W.W. Smith, "Playing to Win: Product Development with the TMS320 Chip," *Speech Technology Magazine*, March/April 1986.
76) D. Essig, C. Erskine, E. Caudel, and S. Magar, "A Second-Generation Digital Signal Processor," *IEEE Journal of Solid-State Circuits*, USA, Volume SC-21, Number 1, pages 86–91, February 1986.

77) W.K. Anakwa, T.L. Stewart, "TMS320 Microprocessor-Based System For Signal Processing," *Proceedings of the ISMM International Symposium*, pages 64–65, February 1986.
78) M. Omenzetter, "Universal Signal Processors Offers High Data Throughput," *Electronik*, Germany, Volume 35, Number 4, pages 71–77, February 1986.
79) P.F. Regamey, "Matched Filtering Using a Signal Microprocessor TMS320," *Mitt. AGEN*, Switzerland, Number 42, pages 31–35, February 1986.
80) "TI Set To Show 2nd-Generation DSP," *Electronics*, USA, pages 23–24, February 3, 1986.
81) "TI Preps CMOS Versions of Signal-Processor Chips," *Electronics Engineering Times*, USA, page 6, February 3, 1986.
82) D. Wilson, "Digital Signal Processing Moves on Chip," *Digital Design*, USA, Volume 16, Number 2, pages 33–34, February 1986.
83) "TI Chip Heads for Fast Lane of Digital Signal Processing," *Electronics*, USA, page 9, January 27, 1986.
84) R.D. Campbell and S.R. McGeoch, "The TMS32010 Digital Signal Processor – An Educational Viewpoint," *International Journal for Electrical Engineering Education*, Great Britain, Volume 23, Number 1, pages 21–31, January 1986.
85) P. Eckelman, "The Cascadable Signal Processor For Digital Signal Processing," *Electronics Industry*, Germany, Volume 17, Number 10, pages 26–27, 1986.
86) R. Cook, "Digital Signal Processors," *High Technology*, USA, Volume 5, Number 10, pages 25–30, October 1985.
87) C.F. Howard, "A High-Level Approach to Digital Processing Design," *Proceedings of MILCOMP/85*, USA, October 1985.
88) H.E. Lee, "Versatile Data-Acquisition System Based on the Commodore C-64/C-128 Microcomputer," *Proceedings of the Symposium of Northeastern Accelerator Personnel*, USA, Volume 57, Number 5, pages 983–985, October 1985.
89) N.K. Riedel, D.A. McAninch, C. Fisher, and N.B. Goldstein, "A Signal Processing Implementation for an IBM PC-Based Workstation," *IEEE Micro*, USA, Volume 5, Number 5, pages 52–67, October 1985.
90) K.E. Marrin, "VLSI and Software Move DSP Into Mainstream," *Computer Design*, USA, Volume 24, Number 9, pages 69–72, September 1985.
91) "Signal Processor ICs: Highly Integrated ICs Making DSP More Attractive," *Electronics Engineering Times*, USA, pages 37–38, September 2, 1985.
92) K.E. Marrin, "VLSI and Software Move DSP Techniques into Mainstream," *Computer Design*, USA, September 1985.
93) "High-Speed Four-Channel Input Board," *Electronics Weekly*, USA, Number 1277, p. 31, July 24, 1985.
94) "4-Channel Analog-Input Board Puts Signal-Processing on VMF Bus," *EDN: Electronic Design News*, USA, Volume 30, Number 17, page 74, July 1985.
95) R.H. Cushman, "Third-Generation DSPs Put Advanced Functions On-Chip," *EDN: Electronic Design News*, USA, July 1985.
96) W.W. Smith, Jr., "Agile Development System, Running on PCs, Builds TMS320-Based FIR Filter," *Electronic Design*, USA, Volume 33, Number 13, pages 129–138, June 6, 1985.

97) S. Magar, S.J. Robertson, and W. Gass, "Interface Arrangement Suits Digital Processor to Multiprocessing," *Electronic Design*, USA, Volume 33, Number 5, pages 189–198, March 7, 1985.
98) G. Kropp, "Signal Processor Offers Multiprocessor Capability," *Elektronik*, Germany, Volume 34, Number 6, pages 53–58, March 1985.
99) S. Magar, D. Essig, E. Caudel, S. Marshall and R. Peters, "An NMOS Digital Signal Processor with Multiprocessing Capability," *Digest of IEEE International Solid-State Circuits Conference*, USA, February 1985.
100) "TI 'Shiva' Chip Outlined," *Electronics Engineering Times*, USA, page 15, February 18, 1985.
101) S. Magar, E. Caudel, D. Essig, and C. Erskine, "Digital Signal Processor Borrows from P to Step up Performance, *Electronic Design*, USA, Volume 33, Number 4, pages 175–184, February 21, 1985.
102) C. Erskine, S. Magar, E. Caudel, D. Essig, and A. Levinspuhl, "A Second-Generation Digital Signal Processor TMS32020: Architecture and Applications," *Traitement de Signal*, France, Volume 2, Number 1, pages 79–83, January–March 1985.
103) S. Baker, "TI 'Shiva' Chip Outlined," *Electronic Engineering Times*, USA, Number 317, page 15, February 1985.
104) S. Baker, "Silicon Bits," *Electronic Engineering Times*, USA, Number 316, page 42, February 1985.
105) H. Bryce, "Board Arrives For Digital Signal Processing on the VMEbus," *Electronic Design*, USA, Volume 33, Number 2, page 266, 1985.
106) K. Marrin, "VME-Compatible DSP System Incorporates TMS320 Chip," *EDN: Electronic Design News*, USA, Volume 30, Number 2, page 122, January 1985.
107) C. Erskine and S. Magar, "Architecture and Applications of A Second-Generation Digital Signal Processor," *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, USA, 1985.
108) D.P. Morgan and H.F. Silverman, "An Investigation into the Efficiency of a Parallel TMS320 Architecture: DFT and Speech Filterbank Applications," *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, USA, Volume 4, pages 1601–1604, 1985.
109) P. Harold, "VME Bus Meeting Sparks Change in Standard, New Products," *EDN: Electronic Design News*, USA, Volume 29, Number 26, page 18, December 1984.
110) W. Loges, "A Code Generator Sets up the Automatic Controller Program for the TMS320," *Elektronik*, Germany, Volume 33, Number 22, pages 154–158, November 1984.
111) H. Volkers, "Fast Fourier Transforms with the TMS320 as Coprocessor," *Elektronik*, Germany, Volume 33, Number 23, pages 109–112, November 1984.
112) Keun-Ho Ryoo, "On the Recent Digital Signal Processors," *Journal of South Korean Institute of Electrical Engineering*, South Korea, Volume 33, Number 9, pages 540–549, September 1984.
113) D. Wilson, "Editor's Comment," *Digital Design*, USA, Volume 14, Number 9, page 14, September 1984.
114) "Signal Processors Will Squeeze Into One Chip, Says TI's French," *Electronics*, USA, Volume 57, Number 9, pages 14,20, May 1984.

115) S. Mehrgardt, "32-Bit Processor Produces Analog Signals," *Elektronik*, Germany, Volume 33, Number 7, pages 77–82, April 1984.
116) S. Magar, "Signal Processing Chips Invite Design Comparisons," *Computer Design*, USA, Volume 23, Number 4, pages 179–186, April 1984.
117) S. Mehrgardt, "General-Purpose Processor System for Digital Signal Processing," *Elektronik*, Germany, Volume 33, Number 3, pages 49–53, February 1984.
118) T. Durham, "Chips: Familiarity Breeds Approval," *Computing*, Great Britain, page 26, January 1984.
119) J. Bradley and P. Ehlig, "Applications of the TMS32010 Digital Signal Porcessor and Their Tradeoffs," *Midcon/84 Electronic Show and Convention*, USA, 1984.
120) J. Bradley and P. Ehlig, "Tradeoffs in the Use of the TMS32010 as a Digital Signal Processing Element," *Wescon/84 Conference Record*, USA, 1984.
121) E. Fernandez, "Comparison and Evaluation of 32-Bit Microprocessors," *Mini/Micro Southeast Computer Conference and Exhibition*, USA, 1984.
122) D. Garcia, "Multiprocessing with the TMS32010," *Wescon/84 Conference Record*, USA, 1984.
123) S. Magar, "Architecture and Applications of a Programmable Monolithic Digital Signal Processor – A Tutorial Review," *Proceedings of IEEE International Symposium on Circuits and Systems*, USA, 1984.
124) D. Quarmby (Editor), "Signal Processor Chips," *Granada*, England 1984.
125) R. Steves, "A Signal Processor with Distributed Control and Multidimensional Scalability," *Proceedings of IEEE National Aerospace and Electronics Conference*, USA, 1984.
126) V. Vagarshakyan and L. Gustin, "On A Single Class of Continuous Systems – A Solution to the Problem on the Diagnosis of Output Signal Characteristics Recognition Procedures," *IZV. AKAD. NAUK ARM. SSR, SER. TEKH. NAUK, USSR*, Volume 37, Number 3, pages 22–27, 1984.
127) J. So, "TMS320 – A Step Forward in Digital Signal Processing," *Microprocessors and Microsystems*, Great Britain, Volume 7, Number 10, pages 451–460, December 1983.
128) J. Elder and S. Magar, "Single-Chip Approach to Digital Signal Processing," *Wescon/83 Electronic Show and Convention*, USA, November 1983.
129) M. Malcangi, "VLSI Technology for Signal Processing. III," *Elettronica Oggi*, Italy, Number 11, pages 129–138, November 1983.
130) P. Strzelcki, "Digital Filtering," *Systems International*, Great Britain, Volume 11, Number 11, pages 116–117, November 1983.
131) W. Loges, "Digital Controls Using Signal Processors," *Elektronik, Germany*, Volume 32, Number 19, pages 51–54, September 1983.
132) "TI's Voice Chip Makes Debut," *Computerworld*, USA, Volume 17, Number 15, page 91, April 1983.
133) L. Adams, "TMS320 Family 16/32-Bit Digital Signal Processor, An Architecture for Breaking Performance Barriers," *Mini/Micro West 1983 Computer Conference and Exhibition*, USA, 1983.
134) R. Blasco, "Floating-Point Digital Signal Processing Using a Fixed-Point Processor," *Southcon/83 Electronics Show and Convention*, USA, 1983.

135) R. Dratch, "A Practical Approach to Digital Signal Processing Using an Innovative Digital Microcomputer in Advanced Applications," *Electro '83 Electronics Show and Convention*, USA, 1983.
136) C. Erskine, "New VLSI Co-Processors Increase System Throughput," *Mini/Micro Midwest Conference Record*, USA, 1983.
137) L. Kaplan, "Flexible Single Chip Solution Paves Way for Low Cost DSP," *Northcon/83 Electronics Show and Convention*, USA, 1983.
138) L. Kaplan, "The TMS32010: A New Approach to Digital Signal Processing," *Electro '83 Electronics Show and Convention*, USA, 1983.
139) S. Mehrgardt, "Signal Processing with a Fast Microcomputer System," *Proceedings of EUSIPCO-83 Second European Signal Processing Conference*, Netherlands, 1983.
140) L. Morris, "A Tale of Two Architectures: TI TMS 320 SPC VS. DEC Micro/J–11," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1983.
141) L. Pagnucco and D. Garcia, "A 16/32 Bit Architecture for Signal Processing," *Mini/Micro West 1983 Computer Conference and Exhibition*, USA, 1983.
142) J. Potts, "A Versatile High Performance Digital Signal Processor," *Ohmcon/83 Conference Record*, USA, 1983.
143) J. Potts, "New 16/32-Bit Microcomputer Offers 200-ns Performance," *Northcon/83 Electronics Show and Convention*, USA, 1983.
144) R. Simar, "Performance of Harvard Architecture in TMS320," *Mini/Micro West 1983 Computer Conference and Exhibition*, USA, 1983.
145) K. McDonough, E. Caudel, S. Magar, and A. Leigh, "Microcomputer with 32-Bit Arithmetic Does High-Precision Number Crunching," *Electronics*, USA, Volume 55, Number 4, pages 105–110, February 1982.
146) K. McDonough and S. Magar, "A Single Chip Microcomputer Architecture Optimized for Signal Processing," *Electro/82 Conference Record*, USA, 1982.
147) L. Kaplan, "Signal Processing with the TMS320 Family," *Midcon/82 Conference Record*, USA, 1982.
148) S. Magar, "Trends in Digital Signal Processing Architectures," *Wescon/82 Conference Record*, USA, 1982.

Graphics/Imaging

1) J.A. Lindberg, "Color Cell Compression Shrinks NTSC Images," *ESD: Electronic Systems Design Magazine*, USA, Volume 17, Number 10, pages 91–96, October 1987
2) S. Ganesan, "A Digitial Signal Processing Microprocessor Based Workstation For Myoelectric Signals," *Fifth International Conference on System Engineering*, USA, Catalog Number 87CH2480–2, pages 427–438, September 1987.
3) JU. Pokovny, O. Skoloud, "Digitisation of a Video Signal From a Television For a Microcomputer," *Sdelovaci Tech.*, Czechoslovakia, Volume 35, Number 6, pages 207–211, June 1987.
4) M.E. Bukaty, "A Vehicle Identification System For Surveillance Applications," *Topical Meeting on Machine Vision*. Technical Digest Series, USA, Volume 12, pages 106–109, March 1987.

5) K.N. Ngan, A.A. Kassim, H.S. Singh, "Parallel Image-Processing System Based on THe TMS32010 Digital Signal Processor," *IEE Proceedings in Electronics*, Great Britain, Volume 134, Number 2, pages 119–124, (March 1987.
6) K.N. Ngan, A.A. Kassim, H. Singh, "A TMS32010-Based Fast Parallel Vison Processor," *Proceedings of the International Workshop on Industrial Applications of Machine Vision and Machine Intelligence*, Catalog Number 87TH0166–9, pages 156–161, February 1987.
7) P. Bellamah, "Hardware-Software Increases Video Storage Capacity," *PC Week*, USA, Volume 4, Number 4, page 15, January 27 1987.
8) J.M. Younse, "Motion Detection Using the Statistical Properties of a Video Image," *Proceedings of SPIE International Society of Optical Engineering*, USA, Volume 697, pages 233–243, August 1986.
9) T. Gehrels, B.G. Marsden, R.S. McMillan, J.V. Scotti, "Astrometry With a Scanning CCD," *Astronomy Journal*, USA, Volume 91, Number 5, pages 1242–1248, May 1986.
10) S. Srinivasan, A.K. Jain, T.M. Chin, "Cosine Transform Block Codec For Images Using TMS32010," *IEEE International Symposium on Circuits and Systems*, USA, Catalog Number 86CH2255–8, Volume 1, pages 299–302, May 1986.
11) D.M. Holburn and I.D. Sommerville, "A High-Speed Image Processing System Using the TMS32010," *Software and Microsystems*, Great Britain, Volume 4, Number 5–6, pages 102–108, October–December 1985.
12) C. D. Crowell and R. Simar, "Digital Signal Processor Boosts Speed of Graphics Display Systems," *Electronic Design*, USA, Volume 33, Number 7, pages 205–209, March 1985.
13) J. Reimer and A. Lovrich, "Graphics with the TMS32020," *WESCON/85 Conference Record*, USA, 1985.
14) H. Megal and A. Heiman, "Image Coding System – A Single Processor Implementation," *MILCOM/85 IEEE Military Communications Conference Record*, USA, 1985.
15) G. Gaillat, "The CAPITAN Parallel Processor: 600 MIPS for Use in Real Time Imagery," *Traitement de Signal*, France, Volume 1, Number 1, pages 19–30, October–December 1984.

Instrumentation

1) G.R. Halsall, D.R. Burton, M.J. Lalor, C.A. Hobson, "A Novel Real-Time Opto-Electronic Profilometer Using FFT Processing," *Proceedings of ICASSP 89*, USA, pages 1634–1637, May 1989.
2) A.J. Pratt, R.E. Gander, B.R. Brandell, "Real-Time Median Frequency Estimator," *Proceedings of the Ninth Annual Conference of the IEEE Engineering in Medicine and Biology Society*, USA, Volume 4, pages 1840–1841, November 1987.
3) D.Y. Cheng, A. Gersho, "A Fast Codebook Search Algorithm For Nearest-Neighbor Pattern Matching," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243–4, Vol 1, pages 265–268, April 1986.
4) Y. Chikada, M. Ishiguro, H. Hirabayashi, M. Morimoto, K. Morita, T. Kanazawa, H. Iwashita, K. Nakazima, S. Ishikawa, T. Takahashi, K. Handa, T.Kazuga, S. Okumura, T. Miyazawa, K. Miura, S. Nagasawa, "A Very Fast FFT Spectrum Analyzer For Radio

Astronomy," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243–4, Volume 4, pages 2907–2910, April 1986.
5) R.C. Wittenberg, "Four Microprocessors Power Multifunction Analyzer," *Electronic Engineering Times*, USA, Number 306, page 30, November 1984.
6) D. Lee, T. Moran, and R. Crane, "Practical Considerations for Estimating Flaw Sizes from Ultrasonic Data," *Materials Evaluation*, Volume 42, Number 9, pages 1150–1158, August 1984.
7) S. Magar, R. Hester, and R. Simpson, "Signal-Processing c Builds FFT-Based Spectrum Analyzer," *Electronic Design*, USA, Volume 30, Number 17, pages 149–154, August 1982.

Voice/Speech

1) A. Aktas, H. Hoge, "Multi-DSP and VQ-ASIC Based Acoustic Front-End for Real-Time Speech Processing Tasks," *Proceedings of EUROSPEECH 89*, pages 586–589, September 1989.
2) D. Bergmann, D. Boillon, F. Bonifacio, R. Breitschadel, "Experimental Speech Input/Output System," *Proceedings of ICASSP 89*, USA, pages 1138–1141, May 1989.
3) J. DellaMorte, P.E. Papamichalis, "Full-Duplex Real-Time Implementation of the FED-STD-1015 LPC-10e Standard V.52 on the TMS320C25," *Proceedings of SPEECH TECH 89*, pages 218–221, May 1989.
4) B.I. Pawate, G.R. Doddington, "Implementation of a Hidden Markov Model-Based Layered Grammar Recognizer," *Proceedings of ICASSP 89*, USA, pages 801–804, May 1989.
5) P.E. Papamichalis, "High Quality Speech Coding: Some Recent Algorithms," *Proceedings of SPEECH TECH 89*, pages 329–333, May 1989.
6) J.C. Ventura, "Digital Audio Gain Control for Hearing Aids," *Proceedings of ICASSP 89*, USA, pages 2049–2052, May 1989.
7) N. Matsui, H. Ohasi, "DSP-Based Adaptive Control of a Brushless Motor," *IEEE Industry Application Society Annual Meeting*, USA, October 1988.
8) A. Albarello, R. Breitschaedel, A. Ciaramella, E. Lenormand, "Implementation of an Acoustic Front-End For Speech Recognition," *CSELT Technical Report*, Italy, Volume 16, Number 5, pages 455–459, August 1988.
9) D. Curl, "Voice Over Data Means More For Your Money," *Communications*, Great Britain, Volume 5, Number 8, pages 27–29, August 1988.
10) H. Hanselman, H. Henrichfreise, H. Hostmann, A. Schwarte, "Hardware/Software Environment for DSP-Based Multivariable Control," *12th. IMACS World Congress*, July 1988.
11) J.B. Attili, M. Savic, J.P. Campbell, Jr., "A TMS32020-Based Real Time Text-Independent, Automatic Speaker Verification System," *Proceedings of ICASSP 88*, USA, Volume S, page 599, April 1988.
12) D. Chase, A. Gersho, "Real-Time VQ Codebook Generation Hardware For Speech Processing," *Proceedings of ICASSP 88*, USA, April 1988.
13) T. Kohonen, K. Torkkola, M. Shozaki, J. Kangas, O. Venta, "Phonetic Typewriter for Finnish and Japanese," *Proceedings of ICASSP 88*, USA, Volume S, page 607, April 1988.

14) I. Lecomte, M. Lever, L. Lelievre, M. Delprat, A. Tassy, "Medium Band Radio Communications," Proceedings of ICASSP 88, USA, April 1988.
15) J.B. Reimer, K.S. Lin, "TMS320 Digital Signal Processors in Speech Applications," Proceedings of SPEECH TECH '88, April 1988.
16) M. Smmendorfer, D. Kopp, H. Hackbarth, "A High-Performance Multiprocessor System for Speech Processing Applications," *Proceedings of ICASSP 88*, USA, Volume V, page 2108, April 1988.
17) P. Vary, K. Hellwig, R. Hoffmann, R.J. Sluyter, C. Garland, M. Russo, "Speech Codec for the European Mobile Radio System," *Proceedings of ICASSP 88*, USA, Volume S, page 227, April 1988.
18) A. Hunt, "A Speaker-Independent Telephone Speech Recognition System: The VCS TeleRec," *Speech Technology*, USA, Volume 4, Number 2, pages 80–82, March–April 1988.
19) R.A. Sukkar, J.L. LoCicero, J.W. Picone, "Design and Implementation of a Robust Pitch Detector Based on a Parallel Processing Technique," *IEEE Journal of Selected Areas of Communications*, USA, Volume 6, Number 2, pages 441–451, February 1988.
20) A.Z. Baraniecki, "Digital Coding of Speech Algorithms and Architecture," *Proceedings of IECON '87*, November 1987.
21) G.R. Steber, "Audio Frequency DSP Laboratory on a Chip-TMS32010," *Proceedings of IECON '87*, Volume 2, pages 1047–1051, November 1987.
22) S.H. Kim, K.R. Hong, H.B. Han, W.H. Hong, "Implementation of Real Time Adaptive Lattice Predictor on Digital Signal Processor," *Proceedings of TENCON 87*, South Korea, Volume 3, pages 1131–1135, August 1987.
23) J.B. Reimer, M.L. McMahan, W.W. Anderson, "Speech Recognition For a Low Cost System Using a DSP," *Digest of Technical Papers for 1987 International Conference on Consumer Electronics*, June 1987.
24) A. Ciaramella, G. Venuti, "Vector Quantization Firmware For an Acoustical Front-End Using the TMS32020," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 4, pages 1895–1898, April 1987.
25) G.A. Frantz, K.S. Lin, "A Low Cost Speech System Using the TMS320C17," *Proceedings of SPEECH TECH '87*, pages 25–29, April 1987.
26) Z. Gorzynski, "Realtime Multitasking Speech Application on the TMS320," *Microprocessors and Microsystems*, Great Britain, Volume 11, Number 3, pages 149–156, April 1987.
27) P. Papamichalis, D. Lively, "Implementation of the DOD Standard LPC-10/52E on the TMS320C25," *Proceedings of SPEECH TECH '87*, pages 201–204, April 1987.
28) B.I. Pawate, M.L. McMahan, R.H. Wiggins, G.R. Doddington, P.K. Rajasekaran, "Connected Word Processor on a Multiprocessor System," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 2, pages 1151–1154, April 1987.
29) S. Roucos, A. Wilgus, W. Russell, "A Segment Vocoder Algorithm For Real-Time Implementation," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 4, pages 1949–1952, April 1987.
30) H. Yeh, "Adaptive Noise Cancellation For Speech With a TMS32020," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 2, pages 1171–1174, April 1987.

31) R. Conover, D. Gustafson, "VLSI Architecture For Cepstrum Calculations," *1987 IEEE Region 5 Conference*, USA, Catalog Number 87CH2383-8, pages 63–64, March 1987.
32) K. Field, A. Derr, L. Cosell, C. Henry, M. Kasner, J. Tiao, "A Single Board Multrate APC Speech Coding Terminal," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 2, pages 960–963, April 1987.
33) H. Brehm, W. Stammler, "Description and Generation of Sperically Invariant Speech-Model Signals," *Signal Processing, Netherlands*, Volume 12, Number 2, pages 119–141, March 1987.
34) A.Z. Baraniecki, "Digital Coding of Speech Algorithms and Architectures," *Proceedings of IECON '87*, Volume 2, pages 977–984, 1987.
35) B. Flocon, P. Lockwood, J. Sap, L. Sauter, "MARIPA: Speaker Independent Recognition of Speech on IBM-PC," *Eighth International Conference on Pattern Recognition*, Catalog Number 86CH2342-4, pages 893–895, October 1986.
36) M.T. Reilly, "A Hybridized Linear Prediction Code Speech Synthesizer," *Conference Records for MILCOM 86*, USA, Catalog Number 86CH2323-4, Volume 2, 32.5/1–5, October 1986.
37) K. Torkkola, H. Riittinen, T. Kohonen, "Microprocessor-Based Word Recognizer For a Large Vocabulary," *Eighth International Conference on Speech Recognition Proceedings*, Catalog Number 86CH2342-4, pages 814–816, October 1986.
38) C.H. Lee, D.Y. Cheng, D.A. Russo et al, "An Integrated Voice-Controlled Voice Messaging System," *Proceedings of Speech Technology 86*, April 1986.
39) Kun-Shan Lin and G.A. Frantz, "A Survey of Available Speech Hardware for Computer Systems," *Proceedings of Speech Technology 86*, April 1986.
40) L.R. Morris, "Software Engineering for an IBM PC/TI-SPEECH Realtime Digital Speech Spectrogram Production System," *Proceedings of Speech Technology 86*, April 1986.
41) K. Torkkola, H. Riittinen, "A Microprocessor-Based Recognition System For Large Vocabularies," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 1, pages 333–337, April 1986.1)
42) Z. Gorzynski, "Real Time Software Engineering on the TMS320: Application in a Pitch Detector Implementation," *International Conference on Speech Input/Output; Techniques and Applications*, Conference Publication Number 258, pages 270–275, March 1986.
43) S. Ganesan, M.O. Ahmad, "A Real Time Speech Signal Processor," *Proceedings of the ISMM Internal Symposium*, pages 46–49, February 1986.
44) L. Gutcho, "DECtalk-a Year Later," *Speech Technology*, Volume 3, Number 1, pages 98–102, August–September 1985.
45) B. Bryden, H.R. Hassanein, "Implementation of a Hybrid Pitch-Excited/Multipulse Vocoder for Cost-Effective Mobile Communications," *Proceedings of Speech Technology 85*, April 1985.
46) M. McMahan, "A Complete Speech Application Development Environment," *Proceedings of SPEECH TECH 85*, pages 293–295, April 1985.
47) H. Hassanein and B. Bryden, "Implementation of the Gold-Rabiner Pitch Detector in a Real Time Environment Using an Improved Voicing Detector," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1985.

48) K. Lin and G. Frantz, "Speech Applications with a General Purpose Digital Signal Processor," *IEEE Region 5 Conference Record*, USA, March 1985.
49) K. Lin and G. Frantz, "Speech Applications Created by a Microcomputer," *IEEE Potentials*, USA, December 1985.
50) M. Malcangi, "Programmable VLSI's for Vocal Signals," *Electronica Oggi*, Italy, Number 10, pages 103–113, October 1984.
51) V. Kroneck, "Conversing with the Computer," *Elektrotechnik*, Germany, Volume 66, Number 20, pages 16–18, October 1984.
52) P.K. Rajasekaran and G.R. Doddington, "Real-Time Factoring of the Linear Prediction Polynomial of Speech Signals," *Digital Signal Processing – 1984: Proceedings of the International Conference*, pages 405–410, September 1984.
53) M. Hutchins and L. Dusek, "Advanced ICs Spawn Practical Speech Recognition," *Computer Design*, USA, Volume 23, Number 5, pages 133–139, May 1984.
54) E. Catier, "Listening Cards or Speech Recognition," *Electronique Industrielle*, France, Number 67, pages 72–76, March 1984.
55) O. Ericsson, "Special Processor Did Not Meet Requirements – Built Own Synthesizer," *Elteknik Aktuell Elektronik*, Sweden, Number 3, pages 32–36, February 1984.
56) H. Strube, "Synthesis Part of a 'Log Area Ratio' Vocoder Implemented on a Signal-Processing Microcomputer," *IEEE Transactions on Acoustics, Speech and Signal Processing*, USA, Volume ASSP-32, Number 1, pages 183–185, February 1984.
57) B. Bryden and H. Hassanein, "Implementation of Full Duplex 2.4 Kbps LPC Vocoder on a Single TMS320 Microprocessor Chip," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1984.
58) M. Dankberg, R. Iltis, D. Saxton, and P. Wilson, "Implementation of the RELP Vocoder Using the TMS320," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1984.
59) A. Holck and W. Anderson, "A Single-Processor LPC Vocoder," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1984.
60) N. Morgan, "Talking Chips," McGraw-Hill, 1984.
61) A. Kumarkanchan, "Microprocessors Provide Speech to Instruments," *Journal of Institute of Electronic and Telecommunication Engineers*, India, Volume 29, Number 12, December 1983.
62) L. Dusek, T. Schalk, and M. McMahan, "Voice Recognition Joins Speech on Programmable Board," *Electronics*, USA, Volume 56, Number 8, pages 128–132, April 1983.
63) J.R. Lineback, "Voice Recognition Listens For Its Cue," *Electronics*, USA, Volume 56, Number 1, page 110, January 1983.
64) D. Daly and L. Bergeron, "A Programmable Voice Digitizer Using the TI TMS320 Microcomputer," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1983.
65) W. Gass, "The TMS32010 Provides Speech I/O for the Personal Computer," *Mini/Micro Northeast Electronics Show and Convention*, USA, 1983.
66) A. Holck, "Low-Cost Speech Processing with TMS32010," *Midcon/83 Conference Record*, USA, 1983.

67) H. Strube, R. Wilhelms, and P. Meyer, "Towards Quasiarticulatory Speech Synthesis in Real Time," *Proceedings of EUSIPCO-83 Second European Signal Processing Conference*, Netherlands, 1983.
68) T. Schalk and M. McMahan, "Firmware-Programmable μc Aid Speech Recognition," *Electronic Design*, Volume 30, Number 15, pages 143–147, July 1982.

Control

1) I. Ahmed, "16-Bit DSP Microcontroller Fits Motion Control System Application," *PCIM*, October 1988.
2) D. Bursky, "Merged Resources Solve Control Headaches," *Electronic Design*, USA, pp 157–159, October 1988.
3) I. Ahmed, "Implementation of Self Tuning Regulators with TMS320 Family of Digital Signal Processors," *MOTORCON '88*, pages 248–262, September 1988.
4) D. Dunnion, M. Stropoli, "Design a Hard-Disk Controller with DSP Techniques," *Electronic Design*, USA, pages 117–121, September 1988.
5) R. van der Kruk, J. Scannell, "Motion Controller Employs DSP Technology," *PCIM*, September 1988.
6) S.W. Yates, R.D. Williams, "A Fault Tolerant Multiprocessor Controller For Magnetic Bearings," *IEEE Micro*, USA, Volume 8, Number 4, page 6, August 1988.
7) I. Garate, R.A. Carrasco, A.L. Bowden, "An Integrated Digital Controller For Brushless AC Motors Using a DSP Microprocessor," *Third International Conference on Power Electronics and Variable-Speed Drive*, Conference Publication Number 291, Conference Publication Number 291, pages 249–252, July 1988.
8) J.M. Corliss, R. Neubert, "DSP Keeps Keep Disk Drive on Track," *Computer Design*, USA, pages 60–65, June 1988.
9) Y.V.V.S. Murty, W.J. Smolinski, S. Sivakumar, "Design of a Digital Protection Scheme For Power Transformers Using Optimal State Observers," *IEE Proc. C, Generation Transmission, Distribution*, Great Britain, Volume 135, Number 3, pages 224–230, May 1988.
10) R.D. Jackson, D.S. Wijesundera, "Direct Digital Control of Induction Motor Currents," *IEE Colloquim on 'Microcomputer Instrumentation and Control Systems in Power Electronics*, Great Britain, Digest Number 61, 1/1–3, April 1988.
11) A. Lovrich, G. Troullinos, R. Chirayil, "An All Digital Automatic Gain Control," *Proceedings of ICASSP 88*, USA, Volume D, page 1734, April 1988.
12) K. Bala, "Running on Imbedded Power," *Electronics Engineering Times*, USA, March 1988.
13) I. Ahmed, S. Meshkat, "Using DSPs in Control," *Control Engineering*, February 1988.
14) M. Babb (Editor), "Solving Control Problems With Specialized Processors," *Control Engineering*, February 1988.
15) S. Meshkat, "High-Level Motion Control Programming Using DSPs," *Control Engineering*, February 1988.
16) S. Meshkat, I. Ahmed, "Using DSPs in AC Induction Motor Drives," *Control Engineering*, February 1988.

17) J. Tan, N. Kyriakopoulos, "Implemmention of a Tracking Kalman Filter on a Digital Signal Processor," *IEEE Transactions of Industrial Electronics*, USA, Volume 35, Number 1, pages 126–134, February 1988.
18) H. Hanselman, "LQG-Control of a Highly Resonant Disc Drive Head Positioning Actuator," *IEEE Transactions on Industrial Electronics*, USA, Volume 35, Number 1, pages 100–104, February 1988.
19) I. Ahmed, "DSP Architectures for Digital Control Systems," *SATECH 1988*, 1988.
20) S. Meskat, "Advanced Motion Control Systems," *Intertec Communications* – Ventura, CA., 1988.
21) I. Ahmed, S. Lundquist, "DSPs Tame Adaptive Control," *Machine Design*, USA, Volume 59, Number 28, pages 125–129, November 1987.
22) B.K. Bose, P.M. Szczesny, "A Microcomputer-Based Control and Simulation of an Advanced IPM Synchronous Machine Drive System For Electric Vehicle Propulsion," *Proceedings of IECON '87*, Volume 1, pages 454–463, November 1987.
23) Y. Dote, M. Shinojima, R.G. Hoft, "Digital Signal Processor (DSP)-Based Novel Variable Structure Control For Robotic Manipulator," *Proceedings of IECON '87*, Volume 1, pages 175–179, November 1987.
24) J.P. Pratt, S. Gruber, "A Real-Time Digital Simulation of Synchronous Machines: Stability Consiferations and Implementation," *IEEE Transactions on Industrial Electronics*, USA, Volume 1E-34, Number 4, pages 483–493, November 1987.
25) I. Ahmed, "Deadbeat Controllers and Observers with the TMS320," *MOTORCON '87*, pages 22–33, September 1987.
26) I. Ahmed, S. Lindquist, "Digital Signal Processors: Simplifying High-Performance Control," *Machine Design*, September 1987.
27) R.D. Ciskowski, C.H. Liu, H.H. Ottesen, S.U. Rahman, "System Identification: An Experimental Verification," *IBM Journal of Research Developments*, Volume 31, Number 5, pages 571–584, September 1987.
28) J.A. Taufiq, R.J. Chance, C.J. Goodman, "On-Line Implementation of Optimised PWM Schemes For Traction Inverter Drives," *International Conference of 'Electric Railway Systems For a New Century*, Conference Publication Number 279, September 1987.
29) Y. Dote, M. Shinojima, H. Yoshimura, "Microprocessor-Based Novel Variable Structure Control For Robot Manipulator," *Proceedings of the 10th. IFAC World Congress*, July 1987.
30) H. Hanselmann, A. Schwarte, "Generation of Fast Target Processor Code From High Level Controller Descriptions," Presented at 10th. IFAC World Congress, July 1987.
31) E. Debourse, "Emergence of DSPs in Machine-Tool Axes Control Systems: Application of Distributed Interpolation Concepts," *Proceedings of the International Workshop on Industrial Automation*, February 1987.
32) C. Chen, "The Mathematical Model and Computer Simulation of an LCI Drive," *Electrical Machinery Power Systems*, USA, Volume 13, Number 3, pages 195–206, 1987.
33) R.D. Ciskowski, C.H. Liu, H.H. Ottesen, S.U. Rahman, "System Identification: An Experimental Verification," *IBM Journal Research Development*, USA, September 1987.
34) H. Hanselmann, "Implementation of Digital Controllers – A Survey," *Automatica*, Volume 23, Number 1, pages 7–32, 1987.

35) H. Henrichfreise, W. Moritz, H. Siemensmeyer, "Control of a Light, Elastic Manipulation Device," *Conference on Applied Motion Control*, 1987.
36) M.C. Stich, "Digital Servo Algorithm For Disk Actuator Control," *Conference on Applied Motion Control*, pages 35–41, 1987.
37) T. Takeshita, K. Kameda, H. Ohashi, N. Matsui, "Digital Signal Processor Based High Speed Current Control of Brushless Motor," *Electronic Engineering*, Japan, USA, Volume 106, Number 6, pages 42–49, November–December 1986.
38) R. Lessmeier, W. Schumacher, W. Leonard, "Microprocessor-Controlled AC-Servo Drives With Synchronous or Induction Motors: Which is Preferable?," *IEEE Transactions On Industry Applications*, USA, September/October 1986.
39) R. Alcantara, J. Prado, C Guegen, "Fixed-Point Implementation of the Fast Kalman Algorithm: Using the TMS32010 Microprocessor," *Proceedings of EUSIPCO-86*, Volume 2, pages 1335–1338, September 1986.
40) B. Nowrouzian, M.H. Hamza, "DC Motor Control Using a Switched-Capacitor Circuit," *Proceedings of the IASTED International Symposium on High Technology in the Power Industry*, pages 352–356, August 1986.
41) N. Matsui, T. Takeshita, "Digital Signal Processor-Based Controllers For Motors," *SICE*, July 1986.
42) H. Hanselmann, "Using Digital Signal Processors For Control," *Proceedings of EICON*, 1986.
43) H. Hanselman, W. Moritz, "High Banwidth Control of the Head Positioning Mechanism in a Winchester Disc Drive," *Proceedings of IECON*, pages 864–869, 1986.
44) R. Cushman, "Easy-to-Use DSP Converter ICs Simplify Industrial-Control Tasks," *Electronic Design*, USA, Volume 29, Number 17, pages 218–228, August 1984.
45) W. Loges, "Signal Processor as High-Speed Digital Controller," *Elektronik Industrie*, Germany, Volume 15, Number 5, pages 30–32, 1984.
46) W. Loges, "Higher-Order Control Systems with Signal Processor TMS320," *Elektronik*, Germany, Volume 32, Number 25, pages 53–55, December 1983.

Military

1) V. Lazzari, Quacchia, M. Sereno, E. Turco, "Implementation of a 16 Kbit/s Split Band-Adaptive Predictive Codec For Digital Mobile Radio Systems," *CSELT Technical Reports*, Italy, Volume 16, Number 5, pages 443–447, August 1988.
2) P. Papamichalis, J. Reimer, "Implementation of the Data Encryption Standard Using the TMS32010," *Digital Signal Processing Applications*, 1986.

Telecommunications

1) S. Casale, R. Russo, G.C. Bellina, "Optimal Architectural Solution Using DSP Processors for the Implementation of an ADPCM Transcoder," *Proceedings of GLOBECOM '89*, pages 1267–1273, November 1989.
2) A. Lovrich and J.B. Reimer, "A Multi-Rate Transcoder," *Transactions on Consumer Electronics*, USA, November 1989.
3) J.L. Dixon, V.K. Varma, N.R. Sollenberger, D.W. Lin, "Single DSP Implementation of a 16 Kbps Sub-Band Speech Coder for Portable Communications," *Proceedings of ICASSP 89*, USA, pages 184–187, May 1989.

4) J.L. So, "Implementation of an NIC (Nearly Instantaneous Companding) 32 Kbps Transcoder Using the TMS320C25 Digital Signal Processor," *Proceedings of GLOBECOM 88*, Section 43.4, November 28 – December 1, 1988.
5) V. Lazzari, Quacchia, M. Sereno, E. Turco, "Implementation of a 16 Kbit/s Split Band-Adaptive Predictive Codec For Digital Mobile Radio Systems," *CSELT Technical Reports*, Italy, August 1988.
6) V. Del Bello, "Signal Processor For Telephone Functions," *Elettronica Oggi*, Italy, Number 63, pages 155–157, June 1988.
7) N. Tamaki, "Studies on Subscriber Line Equalizer Using Decision Feedback Equalizing Circuit," *Transactions of the Institue of Information Communication English B.*, Japan, Volume J71B, Number 5, pages 616–625, May 1988.
8) A. Charbonnier, J-P. Petit, "Sub-Band ADPCM Coding for High Quality Audio Signals," *Proceedings of ICASSP 88*, USA, Volume A, page 2540, April 1988.
9) D. Chase, A. Gersho, "Real-Time VQ Codebook Generation Hardware for Speech Processing," *Proceedings of ICASSP 88*, USA, Volume 3, pages 1730–1733, April 1988.
10) V.K. Jain, S.S. Skrzypkowiak, R.B. Heathcock, "TMS320C25 Based Enhanced ADPCM Transcoder," *Proceedings of ICASSP 88*, USA, Volume S, page 635, April 1988.
11) T.C. Jedrey, N.E. Lay, W. Rafferty, "An All Digital 8-DPSK TCM Modem for Land Mobile Satellite Communications," *Proceedings of ICASSP 88*, USA, Volume D, page 1722, April 1988.
12) A, Lovrich, G. Troullinos, R. Chirayil, "An All Digital Automatic Gain Control," *Proceedings of ICASSP 88*, USA, Volume D, page 1734, April 1988.
13) P. Voros, "High-Quality Sound Coding Within 2x64 Kbit/s Using Instantaneous Dynamic Bit-Allocation," *Proceedings of ICASSP 88*, USA, Volume A, page 2536, April 1988.
14) H.P. Widmer, R. Keung, "HF Data Communication For Extremely Low SNR and High Interference Level," *Fourth International Conference on HF Radio Systems and Techniques*, Conference Publication Number 284, pages 33–37, April 1988.
15) W.B. Michael, P.D. Hill, "Performance Evaluation of a Real-Time TMS32010-Based Adaptive Noise Canceller (ANC)," *IEEE Transactions on Acousticical Speech Signal Processing*, USA, Catalog Number 86CH2255-8, Volume 3, pages 892–895, March 1988.
16) H. Ando, M. Nakaya, H. Hona, I. Iizuka, Y. Horiba, "A DSP Line Equalizer VLSI for TCM Digital Subscriber-Line Transmission," *IEEE Journal of Solid-State Electronics*, USA, Volume 23, Number 1, pages 118–123, February 1988.
17) N. Tamaki, "Studies on an Adaptive Line Equalizer For Subscriber Loops," *Transactions of the Institute of Electronic Information Communication English B.*, Japan, Volume J71B, Number 2, pages 172–180, February 1988.
18) A. Ayerbe Garcia, J.M. Guell Rabasso, A.L. Villen, J.A. Martinez Ayuso, "ADPCM-32 Kbit/S Coder/Decoder For Telephone Channels," *Mundo Electron.*, Spain, Number 178, pages 103–109, November 1987.
19) M. Ishikawa, Y. Tanaka, T Kimura, "An Adaptive Line Equalizer VLSI Using Digital Signal Processing," *IEEE Journal Solid-State Circuits*, USA, Volume 23, Number 3, pages 830–835, November 1987.

20) O. Matsubara, K. Yabuta, E. Sato, H. Takatori, "A Switched Capacitor Line Equalizer For Digital Subscriber Loop Transmission," *Conference Record of GLOBECOM Tokyo '87*, Japan, Volume 3, pages 1746–1751, November 1987.
21) Mills, J.D., V.P. Telang, C.E. Rohrs, "A Data and Voice System For The General Service Telephone Network," *Proceedings of IECON '87*, Volume 2, pages 1143–1148, November 1987.
22) C. Nuthalapati, "A FET Processor Based Phase Noise Measurement System For Radar ATE," *Proceedings of AUTOTESTCON '87*, Catalog Number 87CH2510-6, pages 47–51, November 1987.
23) N. Tamaki, S. Sugimoto, F. Mano, "A Line Terminating Circuit Using the DSP Technique," *Conference Records of GLOBECOM Tokyo '87*, Japan, Volume 3, pages 1731–1735, November 1987.
24) G.J. Saulnier, W.A. Haskins, P. Das, "Tone Jammer Suppression in a Direct Sequence Spread Sprectrum Receiver Using Adaptive Lattice and Transversal Filters," *Conference Records of MILCOM 87*, USA, Volume 1, pages 123–127, October 1987.
25) R.N. Bera, K.S. Rattan, "Real-Time Simulation of the Unmanned Research Vehicle Using Multi-Rate Sampling," *Fifth International Conference on System Engineering*, USA, Catalog Number 87CH2480-2, pages 573–578, September 1987.
26) D. Shear, "Design and Build a Transponder Using DSP Tools. (A Related Article on the Functional Capability of the Acoustic Transponder)," *EDN: Electronic Design News*, USA, Volume 32, Number 18, pages 137–148, September 1987.
27) S.H. Kim, K.R. Hong, H.B. Han, W.H. Hong, "Implementation of Real Time Adaptive Lattice Predictor on Digital Signal Processor," *Proceedings of TENCON 87*, South Korea, Volume 3, pages 1131–1135, August 1987.
28) W. Mattern, "Multifrequency Communications Channel Decoder With The Type TMS32010 Signal Processor Circuit," *Electronics Industry*, France, Number 127, Supplement Number 13, pages 29–33, June 1987.
29) A. Ciaramella, G. Venuti, "Vector Quantization Firmware For an Acoustical Front-End Using the TMS32020," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 4, pages 1895–1898, April 1987.
30) M.J. Pettitt, D. Remedios, A.W. Davis, A. Hadjifotiou, S. Wright, "A Coherent Transmission System Using DFB Lasers and Phase Diversity Reception," *IEE Colloqium on 'High Capacity Fibre Optic Systems'*, Great Britain, Digest Number 23, 9/1–5, February 1987.
31) G.J. Saulnier, K. Yum, P. Das, "The Suppression of Tone-Jammers Using Adaptive Lattice Filtering," *IEEE International Conference on Communications '87*, USA, Volume 2, pages 869–873, June 1987.
32) H.H. Lu, D. Hedberg, B. Fraenkel, "Implementation of High-Speed Voiceband Data Modems Using The TMS320C25," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 4, pages 1915–1918, April 1987.
33) S. Ono, N. Kondoh, M. Kobayashi, M. Hata, "A New Automatic Equalizer For Digital Subscriber Loops," *Electronics and Communications in Japan*, Part 1, Japan, Volume 70, Number 4, pages 93–102, April 1987.

34) J.M. Perl, M. Aharoni, "TMS32010 Implementation of an Improved Kineplex Type HF Modem," *Proceedings of MELECON*, Catalog Number 87CH2425–7, pages 135–140, March 1987.
35) R. Schwarze, W. Tobergte, "Digital Signal Processors in Data Transmission," *Electronik*, Germany, Volume 36, Number 3, pages 73–78, February 1987.
36) J.I. Statman, E.R. Rodenmich, "Parameter Estimation Based on Doppler Frequency Shifts," *IEEE Transactions on Aerospace and Electronic Systems*, USA, Volume AES-23, Number 1, pages 31–39, January 1987.
37) R. Komiya, K. Yoshida, N. Tamaki, "The Loop Coverage Between TCM and Echo Canceller Under Various Noise Considerations," *IEEE Transactions on Communications*, USA, Volume COM-34, Number 11, pages 1058–1067, November 1986.
38) I. Ahmed, A. Lovrich, "Adaptive Line Enhancer Using the TMS320C25," *Conference Records of Northcon/86*, USA, 14/3/1–10, September/October 1986.
39) H. Brehm, W. Stammler, M. Warner, "Design of a Highly Flexible Digital Simulator For Narrowband Fading Channels," *Proceedings of EUSIPCO-86*, Volume 2, pages 1113–1116, September 1986.
40) J.M. Perl, A. Bar, J. Cohen, "TMS-320 Implementation of a x2400 BPS V.26 Modem," *Proceedings of EUSIPCO-86*, Volume 2, pages 1121–1124, September 1986.
41) C.R. Spitzer, "All-Digital Jets Are Taking Off," *IEEE Spectrum*, USA, Volume 23, Number 9, pages 51–56, September 1986.
42) D. Boudreau, "2400 BPS TMS32010 Modem Implementation For Mobile Satellite Applications," *Proceedings of the Thirteenth Biennial Symposium on Communications*, Canada, Volume B-3, pages 1–4, June 1986.
43) R. Chirayil, A. Lovrich, G. Troullinos, "2400 BPS Modem Implementation Using a General Purpose DSP," *Digest of Technical Papers for 1986 International Conference on Consumer Electronics*, pages 110–111, June 1986.
44) D. Hanke, K. Wilhelm, H. Meyer, "Development and Application of In-Flight Simulator For Flying Qualities Research at DFVLR," *Proceedings of NAECON 1986*, USA, Catalog Number 86CH2307-7, Volume 2, pages 490–498, May 1986.
45) P.D. Hill, W.B. Mikhael, "Performance Evaluation of a Real-Time TMS32010-Based Adaptive Noise Filter," *Proceedings of 1986 IEEE International Symposium on Circuits and Systems*, USA, Volume 36, Number 3, pages 411–412, May 1986.
46) S.M. Kuo, M.A. Rodriquez, "Implementation of an Adaptive Frequency Sampling Line Enhancer," *Proceedings of 1986 IEEE International Symposium on Circuits and Systems*, USA, Catalog Number 86CH2255-8, Volume 3, pages 896–899, May 1986.
47) G. Troullinos, J. Bradley, "Split-Band Modem Implementation Using The TMS32010 Digital Signal Processor," *Conference Records of Electro/86 and Mini/Micro Northeast*, USA, 14/1/1–21, May 1986.
48) R. Vemula, E. Lee, "A Microprocessor-Based Noise Cancellor For The Cockpit," *Proceedings of NAECON 1986*, USA, Catalog Number 86CH2307-7, Volume 4, pages 1323–1327, May 1986.
49) C.R. Cole, A. Haoui, P.L. Winship, "A High-Performance Digital Voice Echo Canceller on a SINGLE TMS32020," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 1, pages 429–432, April 1986.

50) F.L. Kitson, K.A. Zeger, "A Real-Time ADPCM Encoder Using Variable Order Prediction (Speech)," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 2, pages 825–828, April 1986.
51) G. Mirchandani, R.C. Gaus, Jr., L.K. Bechtel, "Performance Characteristics of a Hardware Implementation of The Cross-Talk Resistant Adaptive Noise Canceller," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 1, pages 93–96, April 1986.
52) G.S. Muller, C.K. Pauw, "Acoustic Noise Cancellation," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 2, pages 913–916, April 1986.
53) J. Rothweiler, "Performance of a Real Time Low rate Voice Codec," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 4, pages 3039–3042, April 1986.
54) P.J. Wilson, J.M. Puetz, A.V. McCree, D.T. Wang, "An Integrated Voice Codec and Echo Canceller Implemented in a Single DSP Processor," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 2, pages 1333–1336, April 1986.
55) G. Albertengo, S. Benedetto, E. Biglieri, "A DSP Application: An Adaptive Echo Canceller," *Proceedings of the IASTED International Symposium: Modelling, Identification, and Control, MIC '86*, pages 69–72, February 1986.
56) A.W. Davis, S. Wright, M.J. Pettitt, J.P. King, K. Richards, "Coherent Optical Receiver For 680 Mbit/S Using Phase Diversity," *Electron. Lett.*, Great Britain, Volume 22, Number 1, pages 9–11, January 1986.
57) C.R. Cole, A. Haoui, P.L. Winship, "A High-Performance Digital Voice Echo Canceller on a Single TMS32020," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1986.
58) H. Hanselman, W. Moritz, "High Bandwidth Control of the Head Positioning Mechanism in a Winchester Disc Drive," *Proceedings of IECON*, 1986.
59) Sleeper Product – "A Combo Voice/Data I/O Card – Awakens Interest," *Electronic Engineering Times*, USA, pages 80–81, November 11, 1985.
60) R. Chjirayil, P. Ehlig, J. Bradley, G. Troullinos, "Modem Implementation Using The TMS32010," *Proceedings of the National Communications Forum*, 1985, Volume 39, pages 711–715, September 1985.
61) P. Ehlig, "DSP Chip Adds Multitasking Telecomm Capability to Engineering Workstation," *Electronic Design*, USA, Volume 33, Number 10, pages 173–184, May 2, 1985.
62) W.J. Christmas, "A Microprocessor-Based Digital Audio Coder and Decoder," *International Conference on Digital Processing of Signals in Communications*, Number 62, pages 22–26, April 1985.
63) J. Reimer, M. McMahan and M. Arjmand, "ADPCM on a TMS320 DSP Chip," *Proceedings of SPEECH TECH 85*, pages 246–249, April 1985.
64) P. Mock, "Add DTMF Generation and Decoding to DSP– P Designs," *Electronic Design*, USA, Volume 30, Number 6, pages 205–213, March 1985.
65) V. Milutinovic, "4800 Bit/s Microprocessor-Based CCITT Compatible Data Modem," *Microprocessing and Microprogramming*, Volume 15, Number 2, pages 57–74, February 1985.1)
66) G. Corsini and P. Terreni, "A Radar Echo Simulator Based on P TMS320," *Proceedings of MELECON/85 IEEE Mediterranean Electrotechnical Conference* (Sponsors: Mayor

of Madrid, Ministers of Industry and Energy, Spanish National Telephone Co.), USA, Volume 2, pages 327–330, 1985.

67) T. Fjallbrant, "A TMS320 Implementation of a Short Primary Block ATC-System with Pitch Analysis," *International Conference on Digital Processing of Signals in Communications*, Number 62, pages 93–96, 1985.

68) D.P. Kelly and J.L. Melsa, "Syllabic Companding and 32 Kb/s ADPCM Performance," *IEEE International Conference on Communications*, USA, Volume 1, pages 414–417, 1985.

69) A. Vaghar, and V. Milutinovic, "An Analysis of Algorithms for Microprocessor Implementation of High-Speed Data Modems," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, Volume 4, pages 1656–1659, 1985.

70) J. Perl, "Channel Coding in a Self-Optimizing HF Modem," *International Zurich Seminar on Digital Communications; Applications of Source Coding, Channel Coding and Secrecy Coding: Proceedings*, pages 101–106, 1984.

71) R. Chirayil, P. Ehlig, "Integrating Low to Medium and High Speed Modems," *Mini-Micro Southwest-84 Computer Conference and Exhibition*, September 1984,

Automotive

1) Kun-Shan Lin, "Trends of Digital Signal Processing in Automotive," *International Congress on Transportation Electronic (CONVERGENCE '88)*, October 1988.

2) K.E. Beck, M.M. Hahn, "A Real-Time Combustion Analysis Instrument," *SAE Technical Paper Series*, USA, February–March 1988.

3) M. Payne, "Do Not Disturb: Lotus in Action. (Lotus Racing Cars Use of an Active Suspension System)," *Electronics Weekly*, USA, i20 1394, page 12, January 1988.

4) D.A. Williams, S. Oxley, "Application of the Digital Signal Processor to an Automotive Control System," *6th. International Conference on Automotive Electronics*, October 1987.

5) C.M. Anastasia, G.W. Pestana, "A Cylinder Pressure Sensor for Closed Loop Engine Control," *SAE Technical Paper Series*, February 1987.

Consumer

1) A. Lovrich and J.B. Reimer, "A Multi-Rate Transcoder," *Digest of Technical Papers for 1989 International Conference on Consumer Electronics*, June 7–9 1989.

2) G.A. Frantz, J.B. Reimer, and R.A. Wotiz, "Julie, The Application of DSP to a Product," *Speech Tech Magazine*, USA, September 1988.

3) J.B. Reimer and G.A. Frantz, "Customization of a DSP Integrated Circuit for a Customer Product," *Transactions on Consumer Electronics*, USA, August 1988.

4) J.B. Reimer, P.E. Nixon, E.B. Boles, and G.A. Frantz, "Audio Customization of a DSP IC," *Digest of Technical Papers for 1988 International Conference on Consumer Electronics*, June 8–10 1988.

5) H. Mitschke, "Video Recorder: Picture From A Store," *Funschau*, West Germany, Number 9, pages 56–58, April 1988.

6) J.B. Reimer, P.E. Nixon, E.B. Boles, G.A. Frantz, "Audio Customization of a DSP IC," *Digest of Technical Papers for 1987 International Conference on Consumer Electronics*, June 1987.

7) G.R. Steber, "Audio Frequency DSP Laboratory on a Chip-TMS32010," *Proceedings of IECON '87*, Volume 2, pages 1047–1051, November 1987.

Industrial

1) R.C. Chance, T.A. Taufiq, "A TMS32010 Based Near Optimized Pulse Width Modulated Waveform Generator," *Third International Conference on Power Electronics & Variable Speed Drives*, Conference Publication Number 291, July 1988.
2) G. Anwar, R. Horowitz, M. Tomizuka, "Implementation of a MRAC for a Two Axis Direct Drive Robot Manipulator Using a Digital Signal Processor," *American Control Conference*, pages 658–660, June 1988.
3) D.E. Luttrell, T.A. Dow, "Control of Precise Positioning System with Cascaded Colinear Actuators," *American Control Conference*, pages 121–126, June 1988.
4) Y.V.V.S. Murty, W.J. Smolinski, S. Sivakumar, "Design of a Digital Protection Scheme For Power Transformers Using Optimal State Observers," *IEE Proc. C, Generation Transmission, Distribution*, Great Britain, Volume 135, Number 3, pages 224–230, May 1988.
5) M. Ruscio, M. Santoro, M. Adorni, A. Chiaravalloti, "Digital Control System For The Coordinated Boiler/Turbine Control in the ENEL Piombino Power Station," *Elettrotencia*, Italy, Volume 75, Number 3, pages 253–258, March 1988.
6) J.A. Taufiq, R.J. Chance, C.J. Goodman, "On-Line Implementation of Optimised PWM Schemes For Traction Inverter Drives," *International Conference of 'Electric Railway Systems For a New Century'*, Great Britain, Conference Publication Number 279, pages 63–67, September 1987.
7) Y. Dote, M. Shinojima, H. Hoshimura, "Microprocessor-Based Novel Variable Structure Control for Robot Manipulator," *Proceedings of the 10th. IFAC World Congress*, July 1987.
8) H. Henrichfriese, W. Moritz, H. Siemensmeyer, "Control of a Light, Elastic Manipulation Device," *Conference on Applied Motion Control*, pages 57–66, 1987.
9) Y. Wang, M. Andrews, S. Butner, G. Beni, "Robot-Controller System," *15th Annual Symposium on Incremental Motion Control Systems & Devices*, pages 17–26, June 1986.
10) R. Cushman, "Easy-to-Use DSP Converter ICs Simplify Industrial-Control Tasks," *Electronic Design*, USA, Volume 29, Number 17, pages 218–228, August 1984.
11) P. Rojek and W. Wetzel, "Multiprocessor Concept for Industrial Robots: Multivariable Control with Signal Processors," *Elektronik*, Germany, Volume 33, Number 16, pages 109–113, August 1984.
12) G. Farber, "Microelectronics-Developmental Trends and Effects on Automation Techniques," *Regelungstechnik Praxis*, Germany, Volume 24, Number 10, pages 326–336, October 1982.

Medical

1) F.S. Schlindwein, D.H. Evans, "A Real-Time Autoregressive Spectrum Analyzer for Doppler Ultrasound Signals," *Ultrasound in Medicine and Biology*, Volume 15, Number 3, pages 263–272, 1989
2) N. Dillier, "Programmable Master Hearing Aid With Adaptive Noise Reduction Using A TMS32020," *Proceedings of ICASSP 88*, USA, Volume A, page 2508, April 1988.

3) P.B. Knapp, H.S. Lusted, "A Real-Time Digital Signal Processing System for Bioelectric Control of Music," *Proceedings of ICASSP 88*, USA, Volume A, page 2556, April 1988.
4) R.B. Knapp, B. Townshend, "A Real-Time Digital Signal Processing System for an Auditory Prosthesis," *Proceedings of ICASSP 88*, USA, Volume A, page 2493, April 1988.
5) L.R. Morris, P.B. Barszczewski, "Design and Evolution of a Pocket-Sized DSP Speech Processing System for a Cochlear Implant and Other Hearing Prosthesis Applications," *Proceedings of ICASSP 88*, USA, Volume A, page 2516, April 1988.
6) T.J. Sullivan, S.M. Natarajan, "VLSI Based Design of a Battery-Operated Digital Hearing Aid," *Proceedings of ICASSP 88*, USA, Volume A, page 2512, April 1988.
7) A.J. Pratt, R.E. Gander, B.R. Brandell, "Real-Time Median Frequency Estimator," *Proceedings of the Ninth Annual Conference of the IEEE Engineering in Medicine and Biology Society*, USA, November 1987.
8) H. Xu, Y.H. Liang, L.G. Zhou, "The Real-Time Realization of Fetal ECG Heart Rate Monitor by Adaptive System," *Proceedings of the Ninth Annual Conference of the IEEE Engineering in Medicine and Biology Society,* USA, (Catalog Number 87CH2513-0), Volume 3, pages 1662–1663, November 1987.
9) L.R. Morris, page Braszczewski, J. Bosloy, "Algorithm Selection and Software Time/Space Optimisation for a DSP Micro-Based Speech Processor For a Multi-Electrode Cochlear Implant," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 2, pages 972–975, April 1987.
10) K.C. McGill, K.L. McMillan, "A Smart Trigger For Real-Time Spike Classification," *Proceedings of the Eighth Annual Conference of the IEEE Engineering in Medicine and Biology Society*, USA, Catalog Number 86CH2368-9, Volume 1, pages 275–278, November 1986.
11) C. Murphy, page Rolfe, "Application of the TMS320 Signal Processor For The Real-Time Processing of The Doppler Ultasound Spectra," *Proceedings of the Eighth Annual Conference of the IEEE/Engineering in Medicine and Biology*, USA, Catalog Number 86CH2368-9, Volume 2, pages 1175–1178, November 1986.
12) "Innovations: Digital Hearing Aid," *IEEE Spectrum*, USA, 22 December 1985.
13) A. Casini, G. Castellini, P.L. Emiliani, and S. Rocchi, "An Auxiliary Processor for Biomedical Signals Based on a Signal Processing Chip," *Digital Signal Processing – 1984: Proceedings of the International Conference*, pages 228–232, September 1984.
14) T.R. Myers, "A Portable Digital Speech Processor for an Auditory Prosthesis," *Wescon/84 Conference Record*, USA, 1984.

Development Support

1) M. Karjalainen, "A LISP-Based High-Level Programming Environment for the TMS320C30," *Proceedings of ICASSP 89*, USA, pages 1150–1153, May 1989.
2) B.C. Mather, "Digital Filter Design Package (DFP2), Version 2.12," *IEEE Spectrum*, USA, Volume 25, Number 7, page 16, July 1988.
3) R. Weiss, "PC Package Ends DSP Drugery. (Monarch Software Package,Digital Signal Processing)," *Electronic Engineering Times*, USA, Number 494, p. 57, July 1988.
4) A. Kohl, "PC Development Environment For Signal Processors," *Elektron. Prax.*, West Germany, Volume 23, Number 4, April 1988.

5) R. Simar, Jr., A. Davis, "The Application of High-Level Languages to Single-Chip digital Signal Processors," *Proceedings of ICASSP 88*, USA, Volume 3, pages 1678–1681, April 1988.
6) A. Bindra, "New chips, Tools For Signal Processing on Tap: DSP Seminar Attracts Third-Party Developers," *Electronic Engineering Times*, USA, Number 470, page 6, January 1988.
7) R.J. Chance, "Simulation of Multiple Digital Signal Processor Systems," *Journal of Microcomputer Applications*, Great Britain, Volume 11, Number 1, pages 1–19, January 1988.
8) A. Kohl, "Pascal and C-Compilers for the Type TMS320C25 Signal Processor," *Elektron. Ind.*, West Germany, Volume 19, Number 4, pages 58,60,62, 1988.
9) R.J. Chance, B.S. Jones, "A Combined Software/Hardware Development Tool For The TMS32020 Digital Signal Processor," *Journal of Microcomputer Applications*, Great Britain, Volume 10, Number 3, pages 179–197, July 1987.
10) M.A. Zissman, G.C. O'Leary, D.H. Johnson, "A Block Diagram For a Digital Signal Processing MIMD Computer," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 4, pages 1867–1870, April 1987.
11) M.J. Tracy, "Forth as a Language For Digital Signal Processing," *1987 Rochester Forth Conference on Comparative Computer Architectures*, USA, Volume 5, Number 1, pages 221–224, 1987.
12) A. Gharahgozlou, M. Banaouas, E. Babani, "Software Development For a Microprocessor on a 'Host' Computer," *Electronic Industry*, France, Number 115, pages 61–64, November 1986.
13) D.R. Campbell, C. Canning, K. Miller, "Crossassembler For The TMS32010 Digital Signal Processor," *Microprocessors and Microsystems*, Great Britain, Volume 10, Number 8, pages 434–441, October 1986.
14) J. Chance, "Simulation Experiences in the Development of Software For Digital Signal Processors," *Microprocessors and Microsystems*, Great Britain, Volume 10, Number 8, pages 419–426, October 1986.
15) A.C.P. van Meer, "TMS32010 Evaluation Module Controller," *Einhoven University of Technology, Report Number EUT-86,E–162,* 42 pages, October 1986.
16) S.E. Reyer, "A Demonstration Unit For Digital Signal Processing Development and Experimentation," *Proceedings of IECON '86*, Catalog Number 86CH2334-1, Volume 2, pages 641–646, September/October 1986
17) J.R. Parker, "A Subset FORTRAN Compiler For a Modified Harvard Architecture," *SIGPLAN Not*, USA, Volume 21, Number 9, pages 57–62, September 1986.
18) H. Harrison, "A High-Level Language Programming Environment for Speech and Signal Processing," *Proceedings of Speech Technology 86*, April 1986.
19) S. Suehiro, K. Sugimoto, "Forth Machine With Hardware Interpreter Designed to Increase Execution Speed," *Nekkei Electron.*, Japan, Number 396, pages 213–245, 1986.
20) G. Frantz and K. Lin, "The TMS320 Family Design Tools," *Proceedings of SPEECH TECH 85*, pages 238–240, April 1985.
21) R. Schafer, R. Merseraeau, and T. Barnwell, "Software Package Brings Filter Design to PCs," *Computer Design*, USA, Volume 23, Number 13, pages 119–125, November 1984.

22) G. Pawle and T. Faherty, "DSP/Development Board Offers Host Independence," *Computer Design*, USA, Volume 23, Number 12, pages 109–116, October 15, 1984.
23) R. Mersereau, R. Schafer, T. Barnwell, and D. Smith, "A Digital Filter Design Package for PCs and TMS320," *MIDCON/84 Electronic Show and Convention*, USA, 1984.
24) R. Cushman, "Sophisticated Development Tool Simplifies DSP-Chip Programming," *Electronic Design*, USA, Volume 28, Number 20, pages 165–178, September 1983.
25) W. Gass and M. McMahan, "Software Development Techniques for the TMS320," *SOUTHCON/83 Electronics Show and Convention*, USA, 1983.
26) R. Wyckoff, "A Forth Simulator for the TMS320 IC," *Rochester Forth Applications Conference*, USA, pages 141–150, June 1983.

Index

S

T

V

W

X

TEXAS INSTRUMENTS

Printed in U.S.A., March 1990

SPRA017