TEXAS INSTRUMENTS

TMS320C4x General-Purpose Applications

User's Guide

1999

Digital Signal Processing Solutions

Texas Instruments

Printed in U.S.A., May 1999

SPRU159A

User's Guide

TMS320C4x General-Purpose Applications

1999

TMS320C4x General-Purpose Applications User's Guide

SPRU159A
May 1999

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

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

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

Preface

Read This First

About This Manual

This user's guide serves as an applications reference book for the TMS320C40 and TMS320C44 digital signal processors (DSP). Throughout the book, all references to the TMS320C4x apply to both devices (exceptions are noted).

Specifically, this book complements the *TMS320C4x User's Guide* by providing information to assist managers and hardware/software engineers in application development. It includes example code and hardware connections for various applications.

The guide shows how to use the instruction set, the architecture, and the 'C4x interface. It presents examples for frequently used applications and discusses more involved examples and applications. It also defines the principles involved in many applications and gives the corresponding assembly language code for instructional purposes and for immediate use. Whenever the detailed explanation of the underlying theory is too extensive to be included in this manual, appropriate references are given for further information.

How to Use This Manual

The following table summarizes the information contained in this user's guide:

If you are looking for information about:	Turn to these chapters:
Arithmetic	Chapter 3, *Logical and Arithmetic Operations*
Communication Ports	Chapter 8, *Using the Communication Ports*
Companding	Chapter 6, *Applications-Oriented Operations*
Development Support	Chapter 10, *Development Support and Part Order Information*

If you are looking for information about:	Turn to these chapters:
DMA Coprocessor	Chapter 7, *Programming the DMA Coprocessor*
FTTs	Chapter 6, *Applications-Oriented Operations*
Filters	Chapter 6, *Applications-Oriented Operations*
Ordering Parts	Chapter 10, *Development Support and Part Order Information*
Repeat Modes	Chapter 2, *Program Control*
Reset	Chapter 1, *Processor Initialization*
Stacks	Chapter 2, *Program Control*
Tips	Chapter 5, *Programming Tips*
Wait States	Chapter 4, *Memory Interfacing*
XDS510 Emulator	Chapter 11, *XDS510 Emulator Design Considerations*

Style and Symbol Conventions

This document uses the following conventions:

- Program listings, program examples, file names, and symbol names are shown in a special font. Examples use a bold version of the special font for emphasis. Here is a sample program listing segment:

```
*
LOOP1  RPTB   MAX
       CMPF   *AR0,R0      ;Compare number to the maximum
MAX    LDFLT  *AR0,R0      ;If greater, this is a new max
       B      NEXT
LOOP2  RPTB   MIN
       CMPF   *AR0++(1),R0 ;Compare number to the minimum
MIN    LDFLT  *–AR0(1),R0  ;If smaller, this is new minimum
NEXT   .
       .
```

- Throughout this book MSB indicates the most significant bit and LSB indicates the least significant bit. MS indicates the most significant byte and LS indicates the least significant byte.

Information About Cautions and Warnings

This book may contain cautions and warnings.

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to you.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320 floating-point devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C4x User's Guide (literature number SPRU063) describes the 'C4x 32-bit floating-point processor, developed for digital signal processing as well as parallel processing applications. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, and operation of its six DMA channels and six communication ports.

TMS320C4x Parallel Processing Development System Technical Reference (literature number SPRU075) describes the TMS320C4x parallel processing system, a system with four C4xs with shared and distributed memory.

Parallel Processing with the TMS320C4x (literature number SPRA031) describes parallel processing and how the 'C4x can be used in parallel processing. Also provides sample parallel processing applications.

TMS320C3x/C4x Assembly Language Tools User's Guide (literature number SPRU035) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C3x and 'C4x generations of devices.

TMS320 Floating-Point DSP Optimizing C Compiler User's Guide (literature number SPRU034) describes the TMS320 floating-point C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C3x and 'C4x generations of devices.

TMS320C4x C Source Debugger User's Guide (literature number SPRU054) tells you how to invoke the 'C4x emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C4x Technical Brief (literature number SPRU076) gives a condensed overview of the 'C4x DSP and its development tools. It also lists TMS320C4x third parties.

TMS320 Family Development Support Reference Guide (literature number SPRU011) describes the ’320 family of digital signal processors and the various products that support it. This includes code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). This book also lists related documentation, outlines seminars and the university program, and gives factory repair and exchange information.

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that supply various products that serve the family of ’320 digital signal processors—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

TMS320 DSP Designer’s Notebook: Volume 1 (literature number SPRT125) presents solutions to common design problems using ’C2x, ’C3x, ’C4x, ’C5x, and other TI DSPs.

Related Articles and Books

A wide variety of related documentation is available on digital signal processing. These references fall into one of the following application categories:

- ❑ General-Purpose DSP
- ❑ Graphics/Imagery
- ❑ Speech/Voice
- ❑ Control
- ❑ Multimedia
- ❑ Military
- ❑ Telecommunications
- ❑ Automotive
- ❑ Consumer
- ❑ Medical
- ❑ Development Support

In the following list, references appear in alphabetical order according to author. The documents contain beneficial information regarding designs, operations, and applications for signal-processing systems; all of the documents provide additional references. Texas Instruments strongly suggests that you refer to these publications.

General-Purpose DSP*:*

1) Antoniou, A., *Digital Filters: Analysis and Design*, New York, NY: McGraw-Hill Company, Inc., 1979.

2) Brigham, E.O., *The Fast Fourier Transform*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974.

3) Burrus, C.S., and T.W. Parks, *DFT/FFT and Convolution Algorithms*, New York, NY: John Wiley and Sons, Inc., 1984.

4) Chassaing, R., Horning, D.W., "Digital Signal Processing with Fixed and Floating-Point Processors." *CoED*, USA, Volume 1, Number 1, pages 1–4, March 1991.

5) Defatta, David J., Joseph G. Lucas, and William S. Hodgkiss, *Digital Signal Processing: A System Design Approach*, New York: John Wiley, 1988.

6) Erskine, C., 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.

7) Essig, D., 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.

8) Frantz, G., K. Lin, J. Reimer, and J. Bradley, "The Texas Instruments TMS320C25 Digital Signal Microcomputer." *IEEE Microelectronics*, USA, Volume 6, Number 6, pages 10–28, December 1986.

9) Gass, W., R. Tarrant, T. Richard, B. Pawate, M. Gammel, P. Rajasekaran, R. Wiggins, and C. Covington, "Multiple Digital Signal Processor Environment for Intelligent Signal Processing." *Proceedings of the IEEE, USA*, Volume 75, Number 9, pages 1246–1259, September 1987.

10) Gold, Bernard, and C.M. Rader, *Digital Processing of Signals*, New York, NY: McGraw-Hill Company, Inc., 1969.

11) Hamming, R.W., *Digital Filters*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

12) IEEE ASSP DSP Committee (Editor), *Programs for Digital Signal Processing*, New York, NY: IEEE Press, 1979.

13) Jackson, Leland B., *Digital Filters and Signal Processing*, Hingham, MA: Kluwer Academic Publishers, 1986.

14) Jones, D.L., and T.W. Parks, *A Digital Signal Processing Laboratory Using the TMS32010*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

15) Lim, Jae, and Alan V. Oppenheim, *Advanced Topics in Signal Processing*, Englewood Cliffs, NJ: Prentice- Hall, Inc., 1988.

16) Lin, K., G. Frantz, and R. Simar, Jr., "The TMS320 Family of Digital Signal Processors." *Proceedings of the IEEE*, USA, Volume 75, Number 9, pages 1143–1159, September 1987.

17) Lovrich, A., Reimer, J., *"*An Advanced Audio Signal Processor.*" Digest of Technical Papers for 1991 International Conference on Consumer Electronics*, June 1991.

18) Magar, S., 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.

19) Morris, Robert L., *Digital Signal Processing Software*, Ottawa, Canada: Carleton University, 1983.

20) Oppenheim, Alan V. (Editor), *Applications of Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

21) Oppenheim, Alan V., and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975 and 1988.

22) Oppenheim, A.V., A.N. Willsky, and I.T. Young, *Signals and Systems*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

23) Papamichalis, P.E., and C.S. Burrus, "Conversion of Digit-Reversed to Bit-Reversed Order in FFT Algorithms." *Proceedings of ICASSP 89*, USA, pages 984–987, May 1989.

24) Papamichalis, P., and R. Simar, Jr., "The TMS320C30 Floating-Point Digital Signal Processor." *IEEE Micro Magazine*, USA, pages 13–29, December 1988.

25) Parks, T.W., and C.S. Burrus, *Digital Filter Design*, New York, NY: John Wiley and Sons, Inc., 1987.

26) Peterson, C., Zervakis, M., Shehadeh, N., *"*Adaptive Filter Design and Implementation Using the TMS320C25 Microprocessor.*" Computers in Education Journal*, USA, Volume 3, Number 3, pages 12–16, July–September 1993.

27) Prado, J., and 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.

28) Rabiner, L.R. and B. Gold, *Theory and Applications of Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

29) Simar, Jr., R., and 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.

30) Simar, Jr., R., T. Leigh, P. Koeppen, J. Leach, J. Potts, and 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.

31) Simar, Jr., R., and J. Reimer, “The TMS320C25: a 100 ns CMOS VLSI Digital Signal Processor.” *1986 Workshop on Applications of Signal Processing to Audio and Acoustics*, September 1986.

32) Texas Instruments, *Digital Signal Processing Applications with the TMS320 Family*, 1986; Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

33) Treichler, J.R., C.R. Johnson, Jr., and M.G. Larimore, *A Practical Guide to Adaptive Filter Design*, New York, NY: John Wiley and Sons, Inc., 1987.

***Graphics/Imagery**:*

1) Andrews, H.C., and B.R. Hunt, *Digital Image Restoration*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

2) Gonzales, Rafael C., and Paul Wintz, *Digital Image Processing*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1977.

3) Papamichalis, P.E., “FFT Implementation on the TMS320C30.” *Proceedings of ICASSP 88*, USA, Volume D, page 1399, April 1988.

4) Pratt, William K., *Digital Image Processing*, New York, NY: John Wiley and Sons, 1978.

5) Reimer, J., and A. Lovrich, “Graphics with the TMS32020.” *WESCON/85 Conference Record*, USA, 1985.

***Speech/Voice**:*

1) DellaMorte, J., and P. 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.

2) Frantz, G.A., and K.S. Lin, “A Low-Cost Speech System Using the TMS320C17.” *Proceedings of SPEECH TECH ’87*, pages 25–29, April 1987.

3) Gray, A.H., and J.D. Markel, *Linear Prediction of Speech*, New York, NY: Springer-Verlag, 1976.

4) Jayant, N.S., and Peter Noll, *Digital Coding of Waveforms*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

5) Papamichalis, Panos, *Practical Approaches to Speech Coding*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

6) Papamichalis, P., and D. Lively, “Implementation of the DOD Standard LPC–10/52E on the TMS320C25.” *Proceedings of SPEECH TECH ’87*, pages 201–204, April 1987.

7) Pawate, B.I., and G.R. Doddington, “Implementation of a Hidden Markov Model-Based Layered Grammar Recognizer.” *Proceedings of ICASSP 89*, USA, pages 801–804, May 1989.

8) Rabiner, L.R., and R.W. Schafer, *Digital Processing of Speech Signals*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

9) Reimer, J.B. and K.S. Lin, “TMS320 Digital Signal Processors in Speech Applications.” *Proceedings of SPEECH TECH ’88*, April 1988.

10) Reimer, J.B., M.L. McMahan, and 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.

Control*:*

1) Ahmed, I., “16-Bit DSP Microcontroller Fits Motion Control System Application.” *PCIM*, October 1988.

2) Ahmed, I., “Implementation of Self Tuning Regulators with TMS320 Family of Digital Signal Processors.” *MOTORCON ’88*, pages 248–262, September 1988.

3) Ahmed, I., and S. Lindquist, “Digital Signal Processors: Simplifying High-Performance Control.” *Machine Design*, September 1987.

4) Ahmed, I., and S. Meshkat, “Using DSPs in Control.” *Control Engineering*, February 1988.

5) Allen, C. and P. Pillay, *“*TMS320 Design for Vector and Current Control of AC Motor Drives.*” Electronics Letters*, UK, Volume 28, Number 23, pages 2188–2190, November 1992.

6) Bose, B.K., and 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.

7) Hanselman, H., “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.

8) Jacquot, R., *Modern Digital Control Systems*, New York, NY: Marcel Dekker, Inc., 1981.

9) Katz, P., *Digital Control Using Microprocessors*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

10) Kuo, B.C., *Digital Control Systems*, New York, NY: Holt, Reinholt, and Winston, Inc., 1980.

11) Lovrich, A., G. Troullinos, and R. Chirayil, “An All-Digital Automatic Gain Control.” *Proceedings of ICASSP 88*, USA, Volume D, page 1734, April 1988.

12) Matsui, N. and M. Shigyo, *“*Brushless DC Motor Control Without Position and Speed Sensors.*” IEEE Transactions on Industry Applications*, USA, Volume 28, Number 1, Part 1, pages 120–127, January–February 1992.

13) Meshkat, S., and I. Ahmed, “Using DSPs in AC Induction Motor Drives.” *Control Engineering*, February 1988.

14) Panahi, I. and R. Restle, *“*DSPs Redefine Motion Control.*” Motion Control Magazine*, December 1993.

15) Phillips, C., and H. Nagle, *Digital Control System Analysis and Design*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Multimedia*:*

1) Reimer, J., *“*DSP-Based Multimedia Solutions Lead Way Enhancing Audio Compression Performance.*” Dr. Dobbs Journal*, December 1993.

2) Reimer, J., G. Benbassat, and W. Bonneau Jr., *“*Application Processors: Making PC Multimedia Happen.*” Silicon Valley PC Design Conference*, July 1991.

Military*:*

1) Papamichalis, P., and J. Reimer, “Implementation of the Data Encryption Standard Using the TMS32010.” *Digital Signal Processing Applications*, 1986.

Telecommunications*:*

1) Ahmed, I., and A. Lovrich, “Adaptive Line Enhancer Using the TMS320C25.” *Conference Records of Northcon/86*, USA, 14/3/1–10, September/October 1986.

2) Casale, S., R. Russo, and G. Bellina, “Optimal Architectural Solution Using DSP Processors for the Implementation of an ADPCM Transcoder.” *Proceedings of GLOBECOM ’89*, pages 1267–1273, November 1989.

3) Cole, C., A. Haoui, and P. 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.

4) Cole, C., A. Haoui, and P. 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.

5) Lovrich, A., and J. Reimer, “A Multi-Rate Transcoder.” *Transactions on Consumer Electronics*, USA, November 1989.

6) Lovrich, A. and J. Reimer, *“*A Multi-Rate Transcoder.*” Digest of Technical Papers for 1989 International Conference on Consumer Electronics*, June 7–9, 1989.

7) Lu, H., D. Hedberg, and 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.

8) Mock, P., “Add DTMF Generation and Decoding to DSP– µP Designs.” *Electronic Design*, USA, Volume 30, Number 6, pages 205–213, March 1985.

9) Reimer, J., M. McMahan, and M. Arjmand, “ADPCM on a TMS320 DSP Chip.” *Proceedings of SPEECH TECH 85*, pages 246–249, April 1985.

10) Troullinos, G., and 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.

***Automotive**:*

1) Lin, K., “Trends of Digital Signal Processing in Automotive.” *International Congress on Transportation Electronic (CONVERGENCE ’88*), October 1988.

***Consumer**:*

1) Frantz, G.A., J.B. Reimer, and R.A. Wotiz, “Julie, The Application of DSP to a Product.” *Speech Tech Magazine*, USA, September 1988.

2) Reimer, J.B., and G.A. Frantz, “Customization of a DSP Integrated Circuit for a Customer Product.” *Transactions on Consumer Electronics*, USA, August 1988.

3) Reimer, J.B., 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.

***Medical**:*

1) Knapp and Townshend, “A Real-Time Digital Signal Processing System for an Auditory Prosthesis.” *Proceedings of ICASSP 88*, USA, Volume A, page 2493, April 1988.

2) Morris, L.R., and 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.

***Development Support**:*

1) Mersereau, R., R. Schafer, T. Barnwell, and D. Smith, “A Digital Filter Design Package for PCs and TMS320.” *MIDCON/84 Electronic Show and Convention*, USA, 1984.

2) Simar, Jr., R., and 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.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated Market Communications Manager MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Contact the DSP hotline: Phone: **(713) 274–2320** FAX: **(713) 274–2324** Electronic Mail: **4389750@mcimail.com.**
Obtain the source code in this user’s guide.	Call the TI BBS: **(713) 274–2323** Ftp from: **ftp.ti.com** log in as user **ftp** cd to **/mirrors/tms320bbs**
Visit TI online, including TI&ME™, your own customized web page.	Point your browser at: **http://www.ti.com**
Report mistakes or make comments about this or any other TI documentation.	Send electronic mail to: **comments@books.sc.ti.com** Send printed comments to: Texas Instruments Incorporated Technical Publications Mgr., MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Trademarks

MS is a registered trademark of Microsoft Corp.

MS-Windows is a registered trademark of Microsoft Corp.

MS-DOS is a registered trademark of Microsoft Corp.

OS/2 is a trademark of International Business Machines Corp.

Sun and SPARC are trademarks of Sun Microsystems, Inc.

VAX and VMS are trademarks of Digital Equipment Corp.

Contents

Figures

Tables

Examples

Chapter 1

Processor Initialization

Before you execute a DSP algorithm, it is necessary to initialize the processor. Initialization brings the processor to a known state. Generally, initialization takes place any time after the processor is reset. This chapter reviews the concepts explained in the user's guide and provides examples.

Topic **Page**

1.1 Reset Process

After $\overline{\text{RESET}}$ is applied, the ’C4x jumps to the address stored in the reset vector location and starts execution from that point.

In order to reset the ’C4x correctly, you need to comply with several hardware and software requirements:

- Select the reset vector location:
  - The RESET vector of the ’C4x can be mapped to one of four different locations that are controlled by the value of the RESETLOC(1,0) pins at RESET. Table 1–1 shows possible reset vectors for the ’C40 and ’C44.
  - If the DSP is in microcomputer mode (ROMEN pin =1), RESETLOC(1,0) must be equal to 0,0 for the boot loader to operate correctly.
- If the DSP is in microcomputer mode, set the IIOFx pins as discussed in the bootloader chapter *TMS320C4x User’s Guide* so that the bootloader works properly.
- Provide the correct reset vector value:
  - The RESET vector normally contains the address of the system initialization routine.
  - In microcomputer mode the reset vector is initialized automatically by the processor to point to the beginning of the on–chip boot loader code. No user action is required.
  - In microprocessor mode, the reset vector is typically stored in an EPROM. Example 1–1 shows how you can initialize that vector.
- Apply a low level to the RESET input. (See section 1.2).

Table 1–1. RESET Vector Locations in the ’C40 and ’C44

Value at RESETLOCx Pin		Get Reset Vector From Hex Memory Address	Bus
RESETLOC1	RESETLOC0		
0	0	00000 0000	Local
0	1	07FFF FFFF†	Local
1	0	08000 0000†	Global
1	1	0FFFF FFFF†	Global

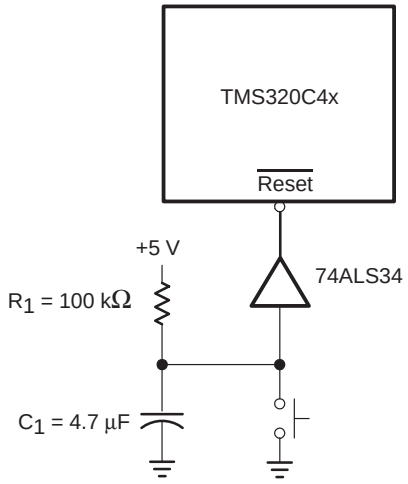
† This corresponds to the 32-bit address that the processor accesses. However, in the ’C44 only the 24-LSBs of the reset address are driven on pins A0–A23 and pins LA0–LA23. The corresponding LSTRBx pins are also activated.

1.2 Reset Signal Generation

Several aspects of ’C4x system hardware design are critical to overall system operation. One such aspect is reset signal generation.

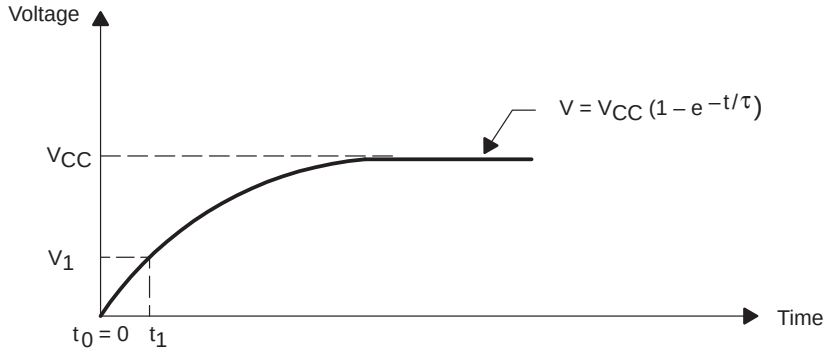
The reset input controls initialization of internal ’C4x logic and execution of the system initialization software. For proper system initialization, *the $\overline{RESET}$ signal must be applied for at least ten H1 cycles*, that is, 400 ns for a ’C4x operating at 50 MHz. Upon power up, however, it can take 20 ms or more before the system oscillator reaches a stable operating state. Therefore, the power-up reset circuit should generate a low pulse on the $\overline{RESET}$ pin 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 1–1 shows a circuit that will generate an appropriate power-up or push-button reset signal.

Figure 1–1. Reset Circuit



The voltage on the $\overline{RESET}$ pin 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 1–2. In Figure 1–1, the 74ALS34 provides a clean $\overline{RESET}$ signal to the ’C4x.

Figure 1–2. Voltage on the $\overline{RESET}$ Pin



The duration of the low pulse on the $\overline{RESET}$ 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 expressed as

$$V = V_{CC}\left[1 - e^{-\frac{t}{\tau}}\right] \qquad (5)$$

where $\tau = R_1C_1$ is the reset circuit time constant. Solving (5) for t results in

$$t = -R_1C_1\ln\left[1 - \frac{V}{V_{CC}}\right] \qquad (6)$$

Setting the following:

$R_1 = 100\ k\Omega$

$C_1 = 4.7\ \mu F$

$V_{CC} = 5\ V$

$V = V_1 = 1.5\ V$

results in t = 167 ms. Therefore, the reset circuit of Figure 1–1 provides a low pulse for a long enough time to ensure the stabilization of the system oscillator upon powerup.

Note:

Reset does not have internal Schmidt hysteresis. To ensure proper reset operation, avoid low rise and fall times. Rise/fall time should not exceed one CLKIN cycle.

1.3 Multiprocessing System Reset Considerations

If synchronization of multiple ’C4x DSPs is required, all processors should be provided with the same input clock and the same reset signal. After powerup, when the clock has stabilized, set $\overline{\text{RESET}}$ high for a few H1/H3 cycles and then set it low to synchronize their H1/H3 clock phases. Following the falling edge, $\overline{\text{RESET}}$ should remain low for at least ten H1 cycles and then be driven high. The circuit in Figure 1–1 can be used for $\overline{\text{RESET}}$ generation.

Pullup resistors are recommended at each end of the connection to avoid unintended triggering after reset when $\overline{\text{RESET}}$ going low is not received on all ’C4x devices at the same time.

CAUTION

It is recommended that you power up the system with $\overline{\text{RESET}}$ low. This prevents ’C4x asynchronous signals from driving unknown values before $\overline{\text{RESET}}$ goes low, which could create bus contention in communication-port pins, resulting in damage to the device.

1.4 How to Initialize the Processor

After reset, the C4x jumps to the address stored in the reset vector location and starts execution from that point. The RESET vector normally contains the address of the system initialization routine.

The initialization routine should typically perform several tasks:

- Set the DP register.
- Set the stack pointer.
- Set the interrupt vector table.
- Set the trap vector table.
- Set the memory control register.
- Clear/enable cache.

Note:

When running under microcomputer mode (ROMEN = 1). The address stored in the reset vector location points to the beginning of the bootloader code. The on-chip bootloader automatically initializes the memory-control register values from the bootloader table

The following examples illustrate how to initialize the ’C4x when using assembly language and when using C.

Processor initialization under assembly language

If you are running under an assembly-only environment, Example 1–1 provides a basic initialization routine. This example shows code for initializing the ’C4x to the following machine state:

- Timer 0 interrupt is enabled.
- Trap 0 is initialized.
- The program cache is enabled.
- The DP is initialized to point to the .text section.
- The stack pointer is initialized to the beginning of the *mystack* section.
- The memory control registers are initialized.
- The ’C4x is initialized to run in microcontroller mode with the reset vector located at address 08000 0000h (RESETLOC(1,0)=1,0).
- The program has already been loaded into memory location at address = 0x4000 0000.

You need to allocate the section addresses using a linker command file (see the *TMS320 Floating-Point DSP Assembly Language Tools User’s Guide* book for more information about linker command files) as shown in Example 1–2.

Example 1–1.Processor Initialization Example

```
;
;  Create Reset Vector
;
        .sect ”rst_sect” ;Named section for RESET vector
reset   .word init       ;RESET vector
;
;  Create Interrupt Vector Table
;
_myvect  .sect ”myvect”   ;Named section for int. vectors
        .space 2         ;Reserved space
        .word tint0      ;Timer 0 ISR address
;
;  Create Trap Vector Table
;
_mytrap  .sect ”mytrap”   ; named section for trap vectors
        .word trap0      ;Trap 0 subroutine address
;
;  Create Stack
;
_mystack .usect ”mystack”,500 ; reserve 500 locations for
                              ; stack
        .text
stacka   .word _mystack   ; address of mystack section
ivta     .word _myvect    ; address of myvect section
tvta     .word _mytrap    ; address of mytrap section
ieval    .word 1          ; IE register value
gctrl    .word ????????   ; target board specific
lctrl    .word ????????   ; target board specific
mctrla   .word 100000h    ; address of the global memory
                          ; control register
init:
;
; Initialize the DP Register
;

        ldp stacka
;
;  Set Expansion Register IVTP
;
        LDI   @ivta,AR0
        LDPE  AR0,IVTP
;
;  Set Expansion Register TVTP
;
        LDI   @tvta,AR0
        LDPE  AR0,TVTP
```

Example 1–1. Processor Initialization Example (Continued)

```
;
;  Initialize global memory interface control
;
        ldi   @mctrla,ar0
        LDI   @gctrl,R0
        STI   R0,*AR0
;
; Initialize local memory interface control
;
        LDI   @lctrl,R0
        STI   R0,*+AR0(4)
;
;  Initialize the Stack Pointer
;
        LDI   @stacka,SP
;
;  Enable timer interrupt
;    This is equivalent to ldi 1,iie
;
        LDI   @ieval,IIE
;
;  Clear/Enable Cache and Enable Global Interrupts
;
        OR    3800H,ST   ;
;
; Global interrupt enable
;
        BR    BEGIN      ; Branch to the beginning of
                         ; the application
     ..................
begin
              < this is your application code>
trap0
           .. < this is your trap0 trap code>
        reti
tint0
           .. < this is your tint0 interrupt
                 service routine>
        reti
        .end
```

Example 1–2.Linker Command File for Linking the Previous Example

```
MEMORY
{
    EPROM:  org = 0x80000000 len = 0x10     /* EPROM reset vector location */
    RAM:   org = 0x40000000   len = 0x100    /* extend RAM */
}

                    /* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

                    SECTIONS
                    {

                       rst_sect: > EPROM
                       myvect: > RAM
                       mystack: > RAM
                       .text:   > RAM
                       mytrap: > RAM

                    }
```

Processor initialization under C language

If you are running under a C environment, your initialization routine is typically boot.asm (from the RTS40.LIB library that comes with the floating-point compiler). In addition to initializing global variables, boot.asm initializes the DP register (pointing to the .bss section) and the SP register (pointing to the .stack section). You need to enable the cache, as shown in Example 1–3, and setup your interrupts inside your main routine before you enable interrupts. See the Application Report, *Setting Up TMS320 DSP Interrupts in C* (SPRA036), for more information.

Example 1–3.Enabling the Cache

```
main() {
asm(" or 1800,st")       ; enable cache
/* asm(" or 3800,st") */ ; enable cache and interrupts
}
```

Chapter 2

Program Control

Several ’C4x instructions provide program control and facilitate high-speed processing. These instructions directly handle:

- Regular and zero-overhead subroutine calls
- Software stack
- Interrupts
- Delayed branches
- Single- and multiple-instruction loops without overhead

Topic **Page**

2.1 Subroutines

The ’C4x provides two ways to invoke subroutine calls: regular calls and zero-overhead calls. The regular and zero-overhead subroutine calls use the software stack and extended-precision register R11, respectively, to save the return address. The following subsections use example programs to explain how this works.

2.1.1 Regular Subroutine Calls

The ’C4x has a 32-bit program counter (PC) and a virtually unlimited software stack. The CALL and CALL*cond* subroutine calls increment the stack pointer and store the contents of the next value of the PC counter on the stack. At the end of the subroutine, RETS*cond* performs a conditional return.

Example 2–1 illustrates the use of a subroutine to determine the dot product of two vectors. Given two vectors of length N, represented by the arrays a[0], a[1], ..., a[N–1] and b[0], b[1],..., b[N–1], the dot product is computed from the expression

d = a[0] b[0] + a[1] b[1] + ... + a[N–1] b[N–1]

Processing proceeds in the main routine to the point where the dot product is to be computed. It is assumed that the arguments of the subroutine have been appropriately initialized. At this point, a CALL is made to the subroutine, transferring control to that section of the program memory for execution, then returning to the calling routine via the RETS instruction when execution has completed. Note that for this particular example, it would suffice to save the register R2. However, a larger number of registers are saved for demonstration purposes. The saved registers are stored on the system stack, which should be large enough to accommodate the maximum anticipated storage requirements. Other methods of saving registers could be used equally well.

Example 2–1.Regular Subroutine Call (Dot Product)

```
*
*   TITLE REGULAR SUBROUTINE CALL (DOT PRODUCT)
*
*
*   MAIN ROUTINE THAT CALLS THE SUBROUTINE 'DOT' TO COMPUTE THE
*   DOT PRODUCT OF TWO VECTORS.
          .
          .
          LDI     @blk0,AR0   ;AR0 points to vector a
          LDI     @blk1,AR1   ;AR1 points to vector b
          LDI     N,RC        ;RC contains the number of elements
          CALL    DOT
          .
          .
*
*
*SUBROUTINE DOT
*
*
*EQUATION: d = a(0) * b(0) + a(1) * b(1) + ... + a(N–1) * b(N–1)
*
*
*THE DOT PRODUCT OF a AND b IS PLACED IN REGISTER R0. N MUST
*BE GREATER THAN OR EQUAL TO 2.
*
*
*   ARGUMENT ASSIGNMENTS:
*    ARGUMENT     |   FUNCTION
*   ------------- +-------------------------
*         AR0     |   ADDRESS OF a(0)
*         AR1     |   ADDRESS OF b(0)
*         RC      |   LENGTH OF VECTORS (N)
*
*
*   REGISTERS USED AS INPUT: AR0, AR1, RC
*   REGISTER MODIFIED: R0
*   REGISTER CONTAINING RESULT: R0
*
*
          .global DOT
*
DOT PUSH  ST      ;Save status register
          PUSH    R2          ;Use the stack to save R2's
          PUSHF   R2          ;bottom 32 and top 32 bits
          PUSH    AR0         ;Save AR0
          PUSH    AR1         ;Save AR1
          PUSH    RC          ;Save RC
          PUSH    RS
          PUSH    RE
*
*
*   Initialize R0:
          MPYF3   *AR0,*AR1,R0;a(0) * b(0) –> R0
```

Example 2–1.Regular Subroutine Call (Dot Product) (Continued)

```
||       SUBF    R2,R2,R2                ;Initialize R2.
         SUBI    2,RC                    ;Set RC = N–2
*
*
*   DOT PRODUCT (1 <= i < N)*
         RPTS    RC                      ; Setup the repeat single.
         MPYF3   *++AR0(1),*++AR1(1),R0  ; a(i) * b(i) –> R0
||       ADDF3   R0,R2,R2                ; a(i–1)*b(i–1) + R2 –> R2
*
         ADDF3   R0,R2,R0                ; a(N–1)*b(N–1) + R2 –> R0
*
*
*   RETURN  SEQUENCE
*
         POP     RE
         POP     RS
         POP     RC                      ;Restore RC
         POP     AR1                     ;Restore AR1
         POP     AR0                     ;Restore AR0
         POPF    R2                      ;Restore top 32 bits of R2
         POP     R2                      ;Restore bottom 32 bits of R2
         POP     ST                      ;Restore ST
         RETS                            ;Return
*
*  end
*
         .end
```

2.1.2 Zero-Overhead Subroutine Calls

Two instructions, link and jump (LAJ) and link and jump conditional (LAJ*cond*), implement zero-overhead subroutine calls to be implemented on the ’C4x. Unlike CALL and CALL*cond*, which put the value of PC + 1 into the software stack, LAJ and LAJ*cond* put the value of PC + 4 into extended-precision register R11. Three instructions following LAJ or LAJ*cond* are executed before going to the subroutine. The restriction that applies to these three instructions is the same as that of the three instructions following a delayed branch. At the end of the subroutine, you can use a delayed branch conditional, B*cond*D, in the register addressing mode with R11 as source, to perform a zero-overhead subroutine return.

For comparison, the same dot product example with a zero-overhead subroutine call is given in the following example program.

Example 2–2.Zero-Overhead Subroutine Call (Dot Product)

```
*
*   TITLE ZERO-OVERHEAD SUBROUTINE CALL (DOT PRODUCT)
*
*
*   MAIN ROUTINE THAT CALLS THE SUBROUTINE ‘DOT’ TO COMPUTE THE
*   DOT PRODUCT OF TWO VECTORS.

          .
          .
          .
          LAJ     DOT
          LDI     @blk0,AR0       ; AR0 points to vector a
          LDI     @blk1,AR1       ; AR1 points to vector b
          LDI     N,RC            ; RC contains the number of elements
          .
          .
          .

*
*SUBROUTINE       DOT
*
*EQUATION:    d = a(0) * b(0) + a(1) * b(1) + ... + a(N–1) * b(N–1)
*
*   THE DOT PRODUCT OF a AND b IS PLACED IN REGISTER R0. N MUST
*   BE GREATER THAN OR EQUAL TO 2.
*
*   ARGUMENT ASSIGNMENTS:
*   ARGUMENT       |        FUNCTION
*   -------------- +------------------------
*          AR0     |        ADDRESS OF a(0)
*          AR1     |        ADDRESS OF b(0)
*          RC      |        LENGTH OF VECTORS (N)


*
*   REGISTERS USED AS INPUT: AR0, AR1, RC
*   REGISTER MODIFIED: R0
*   REGISTER CONTAINING RESULT: R0
*
*


*
          .global DOT
*
DOT       PUSH    ST              ;Save status register
          PUSH    R2              ;Use the stack to save R2’s
          PUSHF   R2              ;bottom 32 and top 32 bits
          PUSH    AR0             ;Save AR0
          PUSH    AR1             ;Save AR1
          PUSH    RC              ;Save RC
          PUSH    RS
          PUSH    RE
```

Example 2–2.Zero-Overhead Subroutine Call (Dot Product) (Continued)

```
*   Initialize R0:
          MPYF3   *AR0,*AR1,R0          ;a(0) * b(0) –> R0
||        SUBF    R2,R2,R2              ;Initialize R2.
          SUBI    2,RC                  ;Set RC = N–2
*
*   DOT PRODUCT (1 <= i < N)
*
          RPTS    RC                    ; Setup the repeat single
          MPYF3   *++AR0(1),*++AR1(1),R0; a(i) * b(i) –> R0
||        ADDF3   R0,R2,R2              ; a(i–1)*b(i–1) + R2 –> R2
*
          ADDF3   R0,R2,R0              ; a(N–1)*b(N–1) + R2 –> R0
*
*   RETURN SEQUENCE
*
          POP     RE
          POP     RS
          POP     RC                    ;Restore RC
          POP     AR1                   ;Restore AR1
          POP     AR0                   ;Restore AR0
          BUD     R11                   ;Return
          POPF    R2                    ;Restore top 32 bits of R2
          POP     R2                    ;Restore bottom 32 bits of R2
          POP     ST                    ;Restore ST
*
*   end
*
          .end
```

2.2 Stacks and Queues

The ’C4x provides a dedicated stack pointer (SP) for building stacks in memory. Also, the auxiliary registers can be used to build user stacks and a variety of more general linear lists. This section discusses the implementation of the following types of linear lists:

Stack A linear list for which all insertions and deletions are made at one end of the list.

Queue A linear list for which all insertions are made at one end of the list, and all deletions are made at the other end.

Dequeue A double-ended queue linear list for which insertions and deletions are made at either end of the list.

2.2.1 System Stacks

A stack in the ’C4x fills from a low-memory address to a high-memory address, as is shown in Figure 2–1. A system stack stores addresses and data during subroutine calls, traps, and interrupts.

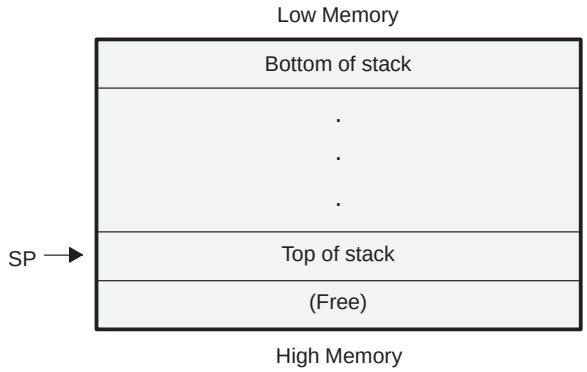
The stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. A push performs a preincrement, and a pop performs a postdecrement of the SP. Provisions should be made to accommodate your software’s anticipated storage requirements.

The stack pointer (SP) can be read from as well as written to; multiple stacks can be created by updating the SP. The SP is not initialized by the hardware during reset; it is important to remember to initialize its value so that the it points to a predetermined memory location. Example 1–1 on page 1-7, shows how to initialize the SP. You must initialize the stack to a valid free memory space. Otherwise, use of the stack could corrupt data or program memory.

The program counter is pushed onto the system stack on subroutine calls, traps, and interrupts. It is popped from the system stack on returns. The PUSH, POP, PUSHF, and POPF instructions push and pop the system stack. The stack can be used inside of subroutines as a place of temporary storage of registers, as is the case shown in Example 2–1, on page 2-3.

Two instructions, PUSHF and POPF, are for floating-point numbers. These instructions can pop and push floating-point numbers to registers R0 — R11. This feature is very useful for saving the extended-precision registers (see Example 2–1 and Example 2–2). PUSH saves the lower 32 bits of an extended-precision register, and PUSHF saves the upper 32 bits. To recover this extended-precision number, execute a POPF followed by POP. It is important to perform the integer and floating-point PUSH and POP in the above order, since POPF forces the last eight bits of the extended-precision registers to zero.

Figure 2–1. System Stack Configuration



2.2.2 User Stacks

User stacks can be built to store data from low-to-high memory or from high-to-low memory. Two cases for each type of stack are shown. You can build stacks by using the preincrement/decrement and postincrement/decrement modes of modifying the auxiliary registers (AR).

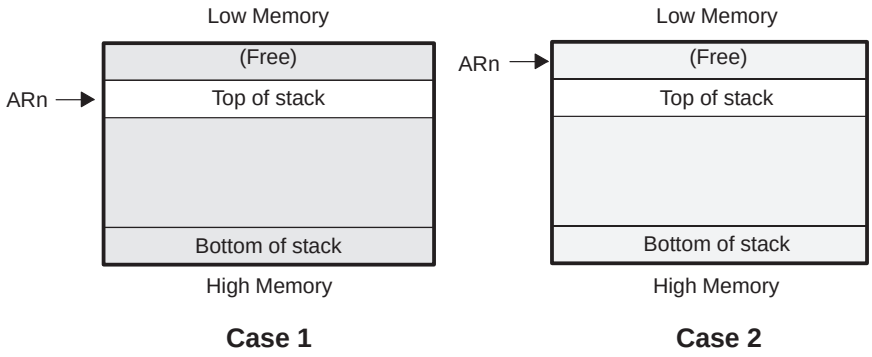
You can implement stack growth from high to low memory in two ways:

Case 1: Store to memory using *– –AR*n* to push data onto the stack, and read from memory using *AR*n*++ to pop data off the stack.

Case 2: Store to memory using *AR*n*– – to push data onto the stack, and read from memory using * ++AR*n* to pop data off the stack.

Figure 2–2 illustrates these two cases. The only difference is that in case 1, the AR always points to the top of the stack, and in case 2, the AR always points to the next free location on the stack.

Figure 2–2. Implementations of High-to-Low Memory Stacks



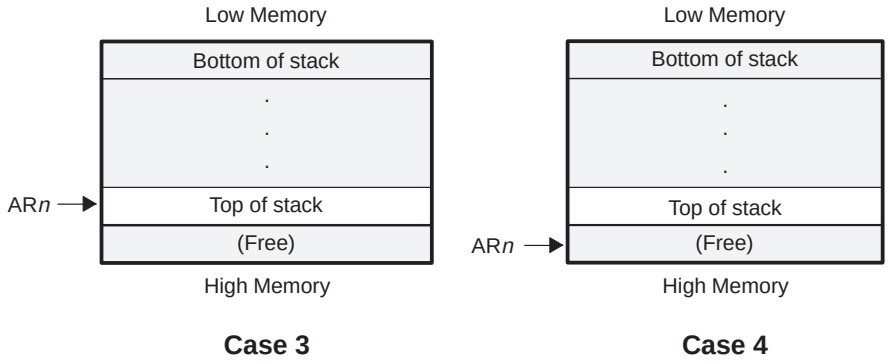
You can implement stack growth from low to high memory in two ways:

Case 3: Store to memory using *++ AR*n* to push data onto the stack, and read from memory using *AR*n*– – to pop data off the stack.

Case 4: Store to memory using *AR*n*++ to push data onto the stack, and read from memory using *– –AR*n* to pop data off the stack.

Figure 2–3 shows these two cases. In case 3, the AR always points to the top of the stack. In case 4, the AR always points to the next free location on the stack.

Figure 2–3. Implementations of Low-to-High Memory Stacks



2.2.3 Queues and Double-Ended Queues

The implementations of queues and double-ended queues is based upon the manipulation of the auxiliary registers for user stacks.

For queues, two auxiliary registers are used: one to mark the front of the queue from which data is popped and the other to mark the rear of the queue to where data is pushed.

For double-ended queues, two auxiliary registers are also necessary. One register marks one end of the double-ended queue, and the other register marks the other end. Data can be popped from or pushed onto either end.

2.3 Interrupt Examples

When using interrupts, you must consider several issues. This section offers examples of several interrupt-related topics:

- Interrupt Service Routines
- Context Switching
- Interrupt-Vector Table (IVTP)
- Interrupt Priorities

2.3.1 Correct Interrupt Programming

For interrupts to work properly you need to execute the following sequence of steps, as is shown in Example 1–1:

1) Set the interrupt-vector table in a 512-word boundary.
2) Initialize the IVTP register.
3) Create a software stack.
4) Enable the specific interrupt.
5) Enable global interrupts.
6) Generate the interrupt signal.

2.3.2 Software Polling of Interrupts

The interrupt flag register can be polled, and action can be taken, depending on whether an interrupt has occurred. This is true even when maskable interrupts are disabled.This can be useful when an interrupt-driven interface is not implemented. Example 2–3 shows the case in which a subroutine is called when external interrupt 1 has not occurred.

Example 2–3.Use of Interrupts for Software Polling

```
* TITLE INTERRUPT POLLING
    .
    .
    .
    TSTB    40H,IIF         ;Test if interrupt 1 has occurred
    CALLZ   SUBROUTINE      ;If not, call subroutine
    .
    .
    .
```

When interrupt processing begins, the program counter is pushed onto the stack, and the interrupt vector is loaded in the program counter. Interrupts are disabled when GIE is cleared to 0 and the program continues from the address loaded in the program counter. Because all maskable interrupts are disabled, interrupt processing can proceed without further interruption unless the interrupt service routine re-enables interrupts, or the NMI occurs.

2.3.3 Using One Interrupt for Two Services

The IVTP can be changed to point to alternate interrupt-vector tables. This relocatable feature of the table allows you to use a single interrupt signal for more than one service.

In Example 2–4, the IVTP is reset in the external INT0 interrupt service routines EINT0A and EINT0B. After the value of the IVTP is changed, the CPU goes to a different interrupt service routine when the same interrupt signal reoccurs.

Example 2–4. Use of One Interrupt Signal for Two Different Services

```
*   TITLE USE OF ONE INTERRUPT SIGNAL FOR TWO DIFFERENT SERVICES
*
*   IN THIS EXAMPLE, THE ADDRESS OF EINT0A AND EINT0B ARE IN
*   MEMORY LOCATION 03H AND 1003H, RESPECTIVELY. ASSUME THE IVTP
*   HAS NOT BEEN CHANGED AFTER DEVICE RESET AND THE EXTERNAL
*   INTERRUPT IIOF0 IS ENABLED. WHEN THE FIRST IIOF0 INTERRUPT
*   SIGNAL COMES IN, THE EINT0A ROUTINE WILL BE EXECUTED AND THEN
*   IF THE NEXT IIOF0 INTERRUPT SIGNAL OCCURS, THE EINT0B ROUTINE
*   WILL BE EXECUTED, AND SO ON. THE EINT0A AND EINT0B ROUTINES
*   WILL TAKE TURNS TO BE EXECUTED WHEN THE IIOF0 INTERRUPT
*   SIGNAL OCCURS.
*
*   External IIOF0 interrupt service routine A
*
          .global EINT0A
EINT0A:   .
          .
          .
          .
          LDI     1000H,R0    ;Change IVTP to point to 1000H
          LDPE    R0,IVTP
          .
          .
*
          RETI                ;Return and enable interrupts
*


*   External IIOF0 interrupt service routine A
*
         .global EINT0B
EINT0B:  .
         .
         .
         .
         LDI     0,R0         ;Change IVTP to point to 0
         LDPE    R0,IVTP
         .
         .
*
         RETI                 ;Return and enable interrupts
```

2.3.4 Nesting Interrupts

In Example 2–5, the interrupt service routine for INT2 temporarily modifies the interrupt enable register (IIE) and interrupt flag register (IIF) to permit interrupt processing when an interrupt to INT0 or NMI (but no other interrupt) occurs. When the routine finishes processing, the IIE register is restored to its original state. Notice that the RETI*cond* instruction not only pops the next program counter address from the stack, but also restores GIE and CF bits from the PGIE and PCF bits. This re-enables all interrupts that were enabled before the INT2 interrupt was serviced.

Example 2–5. Interrupt Service Routine

```
*   TITLE INTERRUPT SERVICE ROUTINE
          .global   ISR2
*
ENABLE    .set   2000h
MASK      .set   9h
*
*   INTERRUPT PROCESSING FOR EXTERNAL INTERRUPT INT2–
*
ISR2:
          PUSH   ST          ;Save status register
          PUSH   DP          ;Save data page pointer
          PUSH   IIE         ;Save interrupt enable register
          PUSH   IIF
          PUSH   R0          ;Save lower 32 bits and
          PUSHF  R0          ;upper 32 bits of R0
          PUSH   R1          ;Save lower 32 bits and
          PUSHF  R1          ;upper 32 bits of R1
          LDI    0,IIE       ;Unmask all internal interrupts
          LDI    MASK, R0
          MH0    R0, IIF     ;Enable INT2
          OR     ENABLE,ST   ;Enable all interrupts
*
*   MAIN PROCESSING SECTION FOR ISR2
          .
          .
          .


          XOR    ENABLE,ST   ;Disable all interrupts
          POPF   R1          ;Restore upper 32 bits and
          POP    R1          ;lower 32 bits of R1
          POPF   R0          ;Restore upper 32 bits and
          POP    R0          ;lower 32 bits of R0
          POP    IIF
          POP    IIE         ;Restore interrupt enable register
          POP    DP          ;Restore data page register
          POP    ST          ;Restore status register
*
          RETI               ;Return and enable interrupts
```

2.4 Context Switching in Interrupts and Subroutines

Context switching is commonly required when a subroutine call or interrupt is processed. It can be extensive or simple, depending on system requirements. For the ’C4x, the program counter is automatically pushed onto the stack. Important information in other ’C4x registers, such as the status, auxiliary, or extended-precision registers, must be saved in the stack with PUSH/PUSHF and recovered later with POP/POPF instructions.

You need to preserve only the registers that are modified inside of your subroutine or interrupt/trap service routine and that could potentially affect the previous context environment.

Note:

The status register should be saved first and restored last to preserve the processor status without any further change caused by other context-switching instructions.

If the previous context environment was in C, then your program must perform one of two tasks:

- If the program is in a subroutine, it must preserve the dedicated C registers:

Save as integers		Save as floating-point	
R4	RS	R6	R7
AR4	AR5		
AR6	AR7		
FP	DP (small model only)		
SP	R8 (‘C4x only)		

- If the program is in an interrupt service routine, it must preserve all of the ’C4x registers, as Example 2–6 shows.

If the previous context environment was in assembly language, you need to determine which registers you must save based on the operations of your assembly-language code.

Example 2–6. Context Save and Context Restore

```
*         .global ISR1
*
*   TOTAL CONTEXT SAVE ON INTERRUPT.
*
ISR1:     PUSH    ST       ;Save status register
*
*   SAVE THE EXTENDED PRECISION REGISTERS
*
          PUSH    R0       ;Save the lower 32 bits of R0
          PUSHF   R0       ;and the upper 32 bits
          PUSH    R1       ;Save the lower 32 bits of R1
          PUSHF   R1       ;and the upper 32 bits
          PUSH    R2       ;Save the lower 32 bits of R2
          PUSHF   R2       ;and the upper 32 bits
          PUSH    R3       ;Save the lower 32 bits of R3
          PUSHF   R3       ;and the upper 32 bits
          PUSH    R4       ;Save the lower 32 bits of R4
          PUSHF   R4       ;and the upper 32 bits
          PUSH    R5       ;Save the lower 32 bits of R5
          PUSHF   R5       ;and the upper 32 bits
          PUSH    R6       ;Save the lower 32 bits of R6
          PUSHF   R6       ;and the upper 32 bits
          PUSH    R7       ;Save the lower 32 bits of R7
          PUSHF   R7       ;and the upper 32 bits
          PUSH    R8       ;Save the lower 32 bits of R8
          PUSHF   R8       ;and the upper 32 bits
          PUSH    R9       ;Save the lower 32 bits of R9
          PUSHF   R9       ;and the upper 32 bits
          PUSH    R10      ;Save the lower 32 bits of R10
          PUSHF   R10      ;and the upper 32 bits
          PUSH    R11      ;Save the lower 32 bits of R11
          PUSHF   R11      ;and the upper 32 bits
*
*   SAVE THE AUXILIARY REGISTERS
*
          PUSH    AR0      ;Save AR0
          PUSH    AR1      ;Save AR1
          PUSH    AR2      ;Save AR2
          PUSH    AR3      ;Save AR3
          PUSH    AR4      ;Save AR4
          PUSH    AR5      ;Save AR5
          PUSH    AR6      ;Save AR6
          PUSH    AR7      ;Save AR7
*
```

Example 2–6.Context Save and Context Restore (Continued)

```
*   SAVE THE REST OF THE REGISTERS FROM THE REGISTER FILE
*
          PUSH   DP     ;Save data page pointer
          PUSH   IR0    ;Save index register IR0
          PUSH   IR1    ;Save index register IR1
          PUSH   BK     ;Save block-size register
          PUSH   IIE    ;Save interrupt enable register
          PUSH   IIF    ;Save interrupt flag register
          PUSH   DIE    ;Save DMA interrupt enable register
          PUSH   RS     ;Save repeat start address

          PUSH   RE     ;Save repeat end address
          PUSH   RC     ;Save repeat counter
*
*   SAVE IS COMPLETE
*
*
*   YOUR INTERRUPT SERVICE ROUTINE CODE GOES HERE*

          .global      RESTR
*
*   CONTEXT RESTORE AT THE END OF A SUBROUTINE CALL OR
    INTERRUPT.
RESTR:
*
*   RESTORE THE REST REGISTERS FROM THE REGISTER FILE
*
          POP    RC     ;Restore repeat counter
          POP    RE     ;Restore repeat end address
          POP    RS     ;Restore repeat start address
          POP    DIE    ;Restore DMA interrupt enable register
          POP    IIF    ;Restore interrupt flag register
          POP    IIE    ;Restore interrupt enable register
          POP    BK     ;Restore block-size register
          POP    IR1    ;Restore index register IR1
          POP    IR0    ;Restore index register IR0
          POP    DP     ;Restore data page pointer
*
*   RESTORE THE AUXILIARY REGISTERS
*
          POP    AR7    ;Restore AR7
          POP    AR6    ;Restore AR6
          POP    AR5    ;Restore AR5
          POP    AR4    ;Restore AR4
          POP    AR3    ;Restore AR3
          POP    AR2    ;Restore AR2
          POP    AR1    ;Restore AR1
          POP    AR0    ;Restore AR0
*
```

Example 2–6. Context Save and Context Restore (Continued)

```
*   RESTORE THE EXTENDED PRECISION REGISTERS
*
          POPF    R11      ;Restore the upper 32 bits and
          POP     R11      ;the lower 32 bits of R11
          POPF    R10      ;Restore the upper 32 bits and
          POP     R10      ;the lower 32 bits of R10
          POPF    R9       ;Restore the upper 32 bits and
          POP     R9       ;the lower 32 bits of R9
          POPF    R8       ;Restore the upper 32 bits and
          POP     R8       ;the lower 32 bits of R8
          POPF    R7       ;Restore the upper 32 bits and
          POP     R7       ;the lower 32 bits of R7
          POPF    R6       ;Restore the upper 32 bits and
          POP     R6       ;the lower 32 bits of R6
          POPF    R5       ;Restore the upper 32 bits and
          POP     R5       ;the lower 32 bits of R5
          POPF    R4       ;Restore the upper 32 bits and
          POP     R4       ;the lower 32 bits of R4
          POPF    R3       ;Restore the upper 32 bits and
          POP     R3       ;the lower 32 bits of R3
          POPF    R2       ;Restore the upper 32 bits and
          POP     R2       ;the lower 32 bits of R2
          POPF    R1       ;Restore the upper 32 bits and
          POP     R1       ;the lower 32 bits of R1
          POPF    R0       ;Restore the upper 32 bits and
          POP     R0       ;the lower 32 bits of R0
          POP     ST       ;Restore status register
*
*   RESTORE IS COMPLETE
*
          RETI
```

2.5 Repeat Modes

The RPTB, RPTBD, and RPTS instructions support looping without overhead. Loop execution parameters are specified by three registers, as can be seen in the following examples:

- RS (Repeat start address)
- RE (Repeat end address)
- RC (Repeat counter)

In principle, it is possible to nest repeat blocks. However, there is only one set of control registers: RS, RE, and RC. It is, therefore, necessary to save these registers before entering an inside loop and to restore these registers after completing the inside loop. It takes four cycles of overhead to save and restore these registers. Hence, sometimes it may be more economical to implement a nested loop by the more traditional method of using a register as a counter and then using a delayed branch, rather than by using the nested repeat block approach. Often, implementing the outer loop as a counter and the inner loop as a RPTB/RPTBD instruction produces the fastest execution.

2.5.1 Block Repeat

Example 2–7 shows the use of the block repeat to find the maximum or the minimum value of 147 numbers. The elements of the array are either all positive or all negative numbers. Because the loop cannot be predetermined, the RPTBD instruction is not suitable here.

Example 2–7. Use of Block Repeat to Find a Maximum or a Minimum

```
*
*   TITLE USE OF BLOCK REPEAT TO FIND A MAXIMUM OR A MINIMUM
*
*   THIS ROUTINE FINDS MAXIMUM OR MINIMUM OF N=147 NUMBERS

        .
        .
        .

        LDI     146,RC              ;Initialize repeat counter to 147–1
        LDI     @ADDR,AR0           ;AR0 points to beginning of array
        LDF     *AR0++(1),R0        ;Initialize MAX or MIN to first value
        BLT     LOOP2               ;If negative array, find minimum
*
LOOP1   RPTB    MAX
        CMPF    *AR0,R0             ;Compare number to the maximum
MAX     LDFLT   *AR0,R0             ;If greater, this is a new maximum
        B       NEXT
LOOP2   RPTB    MIN
        CMPF    *AR0++(1),R0        ;Compare number to the minimum
MIN     LDFLT   *–AR0(1),R0         ;If smaller, this is new minimum
NEXT    .
        .
        .
```

2.5.2 Delayed Block Repeat

Example 2–8 shows an application of the delayed block-repeat construct. In this example, an array of 64 elements is flipped over by exchanging the elements that are equidistant from the end of the array. In other words, if the original array is:

a(1), a(2),..., a(31), a(32),..., a(63), a(64);

then the final array after the rearrangement is:

a(64), a(63),..., a(32), a(31),..., a(2), a(1).

Because the exchange operation is performed on two elements at the same time, it requires 32 operations. The repeat counter (RC) is initialized to 31. In general, if RC contains the number N, the loop is executed N + 1 times. In the example, the loop begins at the fourth instruction following the RPTBD instruction (at the EXCH label). *RC should not be initiated in the next three instructions following the RPTBD*.

Example 2–8. Loop Using Delayed Block Repeat

```
*  TITLE LOOP USING DELAYED BLOCK REPEAT
*
*  THIS CODE SEGMENT EXCHANGES THE VALUES OF ARRAY
*  ELEMENTS THAT ARE SYMMETRIC AROUND THE MIDDLE OF THE
*  ARRAY.
*
       .
       .
       .
       LDI   31,RC        ;Initialize repeat counter
*
       RPTBD EXCH         ;Repeat RC + 1 times between
                          ;START and EXCH
       LDI   @ADDR,AR0    ;AR0 points to
                           beginning of array
       LDI   AR0,AR1
       ADDI  63,AR1       ;AR1 points to the end of the
                           array
*
*  The loop starts here
START  LDI   *AR0,R0      ;Load one memory element in R0,
||     LDI   *AR1,R1      ;and the other in R1
EXCH   STI   R1,*AR0++(1) ;Then, exchange their locations
||     STI   R0,*AR1--(1)
       .
       .
       .
```

2.5.3 Single-Instruction Repeat

Example 2–9 shows an application of the repeat-single construct. In this example, the sum of the products of two arrays is computed. The arrays are not necessarily different. If the arrays are a(i) and b(i), and if each is of length N = 512, register R2 contains the following quantity:

a(1) b(1) + a(2) b(2) +...+ a(N) b(N).

The value of the repeat counter (RC) is specified to be 511 in the instruction.

Example 2–9. Loop Using Single Repeat

```
*   TITLE LOOP USING SINGLE REPEAT
*
          .
          .
          .
          LDI     @ADDR1,AR0              ;AR0 points to array a(i)
          LDI     @ADDR2,AR1              ;AR1 points to array b(i)
*
          LDF     0.0,R2                  ;Initialize R0
*
          MPYF3   *AR0++(1),*AR1++(1),R1  ;Compute first product
*
          RPTS    511                     ;Repeat 512 times
*
          MPYF3   *AR0++(1),*AR1++(1),R1  ;Compute next product and
||        ADDF3   R1,R2,R2                ;accumulate the previous
*
          ADDF    R1,R2                   ;One final addition
          .
          .
          .
```

2.6 Computed GOTOs to Select Subroutines at Runtime

Occasionally, it is convenient to select during runtime, not during assembly, the subroutine to be executed. The ’C4x’s computed GOTO supports this selection. You can implement the computed GOTO by using the CALL*cond* instruction in the register addressing mode. This instruction uses the contents of the register as the address of the call. Example 2–10 shows the case of a task controller.

Example 2–10. Computed GOTO

```
*   TITLE COMPUTED GOTO
*
*   TASK CONTROLLER
*
*   THIS MAIN ROUTINE CONTROLS THE ORDER OF TASK EXECUTION
*   (6 TASKS IN THE PRESENT EXAMPLE). TASK0 THROUGH TASK5 ARE
*   THE NAMES OF SUBROUTINES TO BE CALLED. THEY ARE EXECUTED
*   IN ORDER, TASK0, TASK1, ... TASK5. WHEN AN INTERRUPT
*   OCCURS, THE INTERRUPT SERVICE ROUTINE IS EXECUTED, AND THE
*   PROCESSOR CONTINUES WITH THE INSTRUCTION FOLLOWING THE
*   IDLE INSTRUCTION. THIS ROUTINE SELECTS THE APPROPRIATE
*   TASK FOR THE CURRENT CYCLE, CALLS THE TASK AS A SUBROUTINE,
*   AND BRANCHES BACK TO THE IDLE INSTRUCTION TO WAIT FOR THE
*   NEXT SAMPLE INTERRUPT WHEN THE SCHEDULED TASK HAS COMPLETED
*   EXECUTION. R0 HOLDS THE OFFSET FROM THE BASE ADDRESS OF THE
*   TASK TO BE EXECUTED. BIT 15 (SET COND BIT) OF STATUS REGISTER
*   (ST) SHOULD BE SET TO 1.
*
          LDI     5,IR0             ;Initialize IR0
          LDI     @ADDR,AR1         ;AR1 holds the base address
                                    ;of the table
WAIT      IDLE                      ;Wait for the next interrupt
          ADDI    *+AR1(IR0),R1     ;Add base address to the
                                    ;table entry number
          SUBI    1,IR0             ;Decrement IR0
          LDILT   5,IR0             ;If IR0<0, reinitialize it to 5
          CALLU   R1                ;Execute appropriate task
          BR      WAIT
*
TSKSEQ    .word   TASK5             ;Address of TASK5
          .word   TASK4             ;Address of TASK4
          .word   TASK3             ;Address of TASK3
          .word   TASK2             ;Address of TASK2
          .word   TASK1             ;Address of TASK1
          .word   TASK0             ;Address of TASK0
ADDR      .word   TSKSEQ
```

Chapter 3

Logical and Arithmetic Operations

The ’C4x instruction set supports both integer and floating-point arithmetic and logical operations. The basic functions of such instructions can be combined to form more complex operations. This chapter contains the following operations examples:

- Bit manipulation
- Block moves
- Byte and half-word manipulation
- Bit-reversed addressing
- Integer and floating-point division
- Square root
- Extended-precision arithmetic
- Floating-point format conversion between IEEE and ’C4x formats

Topic **Page**

3.1 Bit Manipulation

Instructions for logical operations, such as AND, OR, NOT, ANDN, and XOR, can be used together with shift instructions for bit manipulation. A special instruction, TSTB, tests bits. TSTB does the same operation as AND, but the result of the TSTB is used only to set the condition flags and is not written anywhere. Example 3–1 and Example 3–2 demonstrate the use of several instructions for bit manipulation and testing.

Example 3–1. Use of TSTB for Software-Controlled Interrupt

```
*   TITLE USE OF TSTB FOR SOFTWARE-CONTROLLED INTERRUPT
*
*   IN THIS EXAMPLE, ALL INTERRUPTS HAVE BEEN DISABLED BY
*   RESETTING THE GIE BIT OF THE STATUS REGISTER. WHEN AN
*   INTERRUPT ARRIVES, IT IS STORED IN THE IF REGISTER. THE
*   PRESENT EXAMPLE ACTIVATES THE INTERRUPT SERVICE ROUTINE INTR
*   WHEN IT DETECTS THAT INT2- HAS OCCURRED.
          .
          .
          .
          TSTB    4,IIF           ; Check if bit 2 of IF is set,
          CALLNZ  INTR            ; and, if so, call subroutine INTR
          .
          .
          .
```

Example 3–2. Copy a Bit from One Location to Another

```
*   TITLE COPY A BIT FROM ONE LOCATION TO ANOTHER
*
*   BIT I OF R1 NEEDS TO BE COPIED TO BIT J OF R2. AR0 POINTS TO A LOCATION
*   HOLDING I, AND IT IS ASSUMED THAT THE NEXT MEMORY LOCATION HOLDS THE VALUE J.
*
          .
          .
          .
          LDI     1,R0
          LSH     *AR0,R0         ;Shift 1 to align it with bit I
          TSTB    R1,R0           ;Test the I–th bit of R1
          BZD     CONT            ;If bit = 0, branch delayed
          LDI     1,R0
          LSH     *+AR0(1),R0     ;Align 1 with J–th location
          ANDN    R0,R2           ;If bit = 0, reset J–th bit of R2
          OR      R0,R2           ;If bit = 1, set J–th bit of R2
          CONT
          .
          .
          .
```

3.2 Block Moves

Because the ’C4x directly addresses a large amount of memory, blocks of data or program code can be stored off-chip in slow memories and then loaded on-chip for faster execution. Data can also be moved from on-chip memory to off-chip memory for storage or for multiprocessor data transfers.

The DMA can transfer data efficiently in parallel with CPU operations. Alternatively, you can use the load and store instructions in a repeat mode to perform data transfers under program control. Example 3–3 shows how to transfer a block of 512 floating-point numbers from external memory to block 1 of on-chip RAM.

Example 3–3. Block Move Under Program Control

```
*   TITLE BLOCK MOVE UNDER PROGRAM CONTROL
*
extern    .word   01000H
block1    .word   02FFC00H
          .
          .
          .
          LDI     @extern,AR0     ;Source address
          LDI     @block1,AR1     ;Destination address
          LDF     *AR0++,R0       ;Load the first number
          RPTS    510             ;Repeat following instruction 511 times
          LDF     *AR0++,R0       ;Load the next number, and...
||        STF     R0,*AR1++       ;store the previous one
          STF     R0,*AR1         ;Store the last number
          .
          .
          .
```

3.3 Byte and Half-Word Manipulation

A set of instructions for byte and half-word accessibility, such as LB(3,2,1,0), LBU(3,2,1,0), LH(1,0), LHU(1,0), LWL(0,1,2,3), LWR(0,1,2,3), MB(3,2,1,0), and MH(1,0), is available on the 'C4x. In an application such as image processing, it is often important to be able to manipulate packed data. For example, the pixels in color images are often represented by four 8-bit unsigned quantities — red, green, blue and alpha — which are packed into a single 32-bit word. The byte and half-word instruction makes it very easy to manipulate this packed data.

Example 3–4 shows the packing of data from a half-word FIFO to 32-bit data memory, and Example 3–5 shows the unpacking of a 32-bit data array into a 4-byte-wide data array (assuming the 32-bit data array contains four 8-bit unsigned numbers).

Example 3–4. Use of Packing Data From Half-Word FIFO to 32-Bit Data Memory

```
*   TITLE USE OF PACKING DATA FROM HALF-WORD FIFO
*   TO 32-BIT DATA MEMORY
*
*   IN THIS EXAMPLE, EVERY TWO INPUT 16 BITS DATA HAS BEEN
*   PACKED INTO ONE 32-BIT DATA MEMORY. THE LOOP SIZE
*   USED HERE IS ARRAY SIZE, NOT THE INPUT DATA LENGTH.
          .
          .
          .
          LDI     size-1,RC       ;Load array size
          RPTBD   PACK
          LDI     @fifo_adr,AR1   ;Load fifo address
          LDI     @array,AR2      ;Load data array address
          NOP
*   >>>>>>>>>>>>>>>>              ;Loop starts here
          LWL0    *AR1,R9         ;Pack 16 LSBs
          LWL1    *AR1,R9         ;Pack 16 MSBs
PACK      STI     R9,*AR2++(1)    ;Store the data
          .
          .
          .
```

Example 3–5. Use of Unpacking 32-Bit Data Into Four-Byte-Wide Data Array

```
*   TITLE  USE OF UNPACKING 32-BIT DATA INTO FOUR BYTE-WIDE
*   DATA ARRAY
*
*   THIS EXAMPLE ASSUMED THAT THE 32-BIT DATA CONTAINS FOUR 8-BIT
*   UNSIGNED DATA.
          .
          .
          .       LDI size–1,RC     ;Load array size
          LDI     @input_adr,AR0    ;Load RPTBD UNPACK input address
          LDI     @array1,AR1       ;Load output data array 1 address
          RPTBD   UNPACK
          LDI     @array2,AR2       ;Load output data array 2 address
          LDI     @array3,AR3       ;Load output data array 3 address
          LDI     @array4,AR4       ;Load output data array 4 address
*   >>>>>>>>>>>>>>>>                ;Loop starts here
          LBU0    *AR0,R8           ;Unpack first byte
          STI     R8,*AR1++(1)
          LBU1    *AR0,R8           ;Unpack second byte
          STI     R8,*AR2++(1)
          LBU2    *AR0,R8           ;Unpack third byte
          STI     R8,*AR3++(1)
          LBU3    *AR0++(1),R8      ;Unpack fourth byte
UNPACK    STI     R8,*AR4++(1)
          .
          .
          .
```

3.4 Bit-Reversed Addressing

The ’C4x can implement fast Fourier transforms (FFT) with bit-reversed addressing. If the data to be transformed is in the correct order, the final result of the FFT is scrambled in bit-reversed order. To recover the frequency-domain data in the correct order, certain memory locations must be swapped. The bit-reversed addressing mode makes swapping unnecessary. The next time data is accessed, the access is bit-reversed rather than sequential. In ’C4x, this bit-reversed addressing can be implemented through both the CPU and DMA.

For correct CPU or DMA bit-reversed operation, the base address of bit-reversed addressing must be located on a boundary of the size of the table. To clarify this point, assume an FFT of size $N = 2^n$. When real and imaginary data are stored in separate arrays, the *n* LSBs of the base address must be zero, (0) and IR0 must be initialized to 2^{n-1} (half of the FFT size). When real and imaginary data are stored in consecutive memory locations (*Re–Im–Re–Im*) the *n+1* LSBs of the base address must be zero (0), and IR0 must be equal to $IR0 = 2^n = N$ (FFT size).

3.4.1 CPU Bit-Reversed Addressing

One auxiliary register (AR0, in this case) points to the physical location of a data value. When you add IR0 to the auxiliary register by using bit-reversed addressing, addresses are generated in a bit-reversed fashion (reverse carry propagation). The largest index (IR0, in this case) for bit reversing is 00FF FFFFh.

Example 3–6 illustrates how to move a 512-point complex FFT from the place of computation (pointed at by AR0) to a location pointed at by AR1. Reads are executed in a bit-reversed fashion and writes in a linear fashion. In this example, real and imaginary parts XR(i) and XI(i) of the data are not stored in separate arrays, but they are interleaved with XR(0), XI(0), XR(1), XI(1), ..., XR(N1), XI(N1). Because of this arrangement, the length of the array is 2N instead of N, and IR0 is set to 512 instead of 256.

Example 3–6.CPU Bit-Reversed Addressing

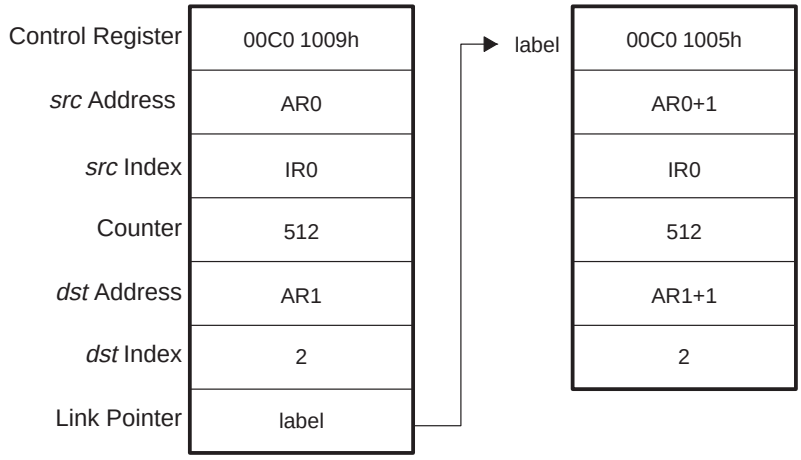
```
*
*  TITLE BIT-REVERSED ADDRESSING
*
*  THIS EXAMPLE MOVES THE RESULT OF THE 512-POINT FFT COMPUTATION, POINTED AT BY
*  AR0, TO A LOCATION POINTED AT BY AR1. REAL AND IMAGINARY POINTS ARE ALTERNATING.
*
           .
           .
           .
           LDI     511,RC             ;Repeat 511+1 times
           RPTBD   LOOP
           LDI     512,IR0            ;Load FFT size
           LDI     2,IR1
           LDF     *+AR0(1),R1        ;Load first imaginary point
*
           LDF     *AR0++(IR0)B,R0    ;Load real value (and point to next
||         STF     R1,*+AR1(1)        ;location) and store the imaginary value
LOOP       LDF     *+AR0(1),R1        ;Load next imaginary point and store
||         STF     R0,*AR1++(IR1)     ;previous real value
           .
           .
           .
```

3.4.2 DMA Bit-Reversed Addressing

In DMA bit-reversed addressing, two bits in the DMA control register enable bit-reversed addressing on DMA reads (READ BIT REV) and DMA writes (WRITE BIT REV). The source address index register and destination address index register define the size of the bit-reversed addressing. Their function is similar to the CPU index register IR0 described in the previous subsection. Two DMA block transfers are required when the DMA is used for bit-reversed transfer of complex numbers: one to transfer the real ports and one to transfer the imaginary ports.

Figure 3–1 illustrates the DMA settings required for a DMA operation equivalent to Example 3–6. Unified-autoinitialization mode and bit-reversed read are used. For more detailed information about DMA operation, refer to *The DMA Coprocessor* in the *TMS320C4x User's Guide*.

Figure 3–1. DMA Bit-Reversed Addressing



3.5 Integer and Floating-Point Division

You can use the single-cycle instruction, RCPF, to generate an estimate of the reciprocal of a floating-point number. This estimate has the correct exponent, and the mantissa is accurate to the eighth binary place (the error of the mantissa is $< 2^{-8}$). Often, this is a satisfactory estimate of the reciprocal of a floating-point number. In other cases, this estimate can be used as a seed for an algorithm that computes the reciprocal to even greater accuracy. The Newton-Raphson algorithm described later is one such case.

Although it provides no special instruction for integer division, the instruction set can perform an efficient division routine. Additionally, the FLOAT, RCPF, and FIX instructions can produce a rough estimate.

3.5.1 Integer Division

You can implement division on the ’C4x by repeating SUBC, a special conditional subtract instruction. Consider the case of a 32-bit positive dividend with *i* significant bits (and *32–i* sign bits), and a 32-bit positive divisor with *j* significant bits (and *32–j* sign bits). The repetition of the SUBC command *i–j + 1* times produces a 32-bit result in which the lower *i–j + 1* bits are the quotient, and the upper *31–i + j* bits are the remainder of the division.

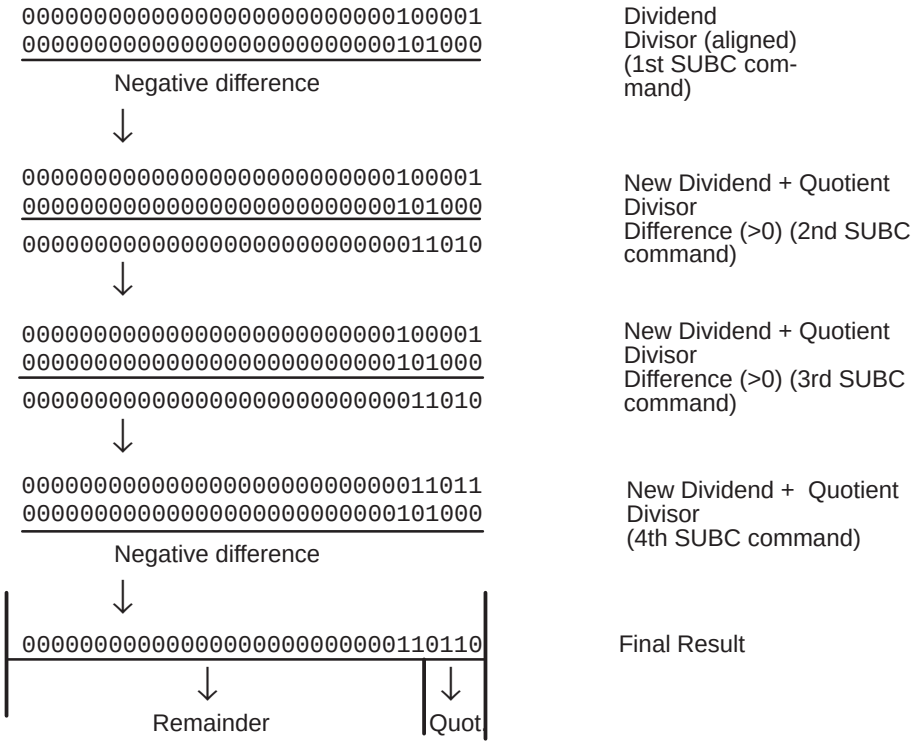
SUBC implements binary division in the same manner as long division. The divisor (assumed to be smaller than the dividend) is shifted left *i–j* times to align with the dividend. Then, using SUBC, the shifted divisor is subtracted from the dividend. For each subtract that does not produce a negative answer, the dividend is replaced by the difference. It is then shifted to the left, and the LSB is set to 1. If the difference is negative, the dividend is simply shifted left by one. This operation is repeated *i–j + 1* times.

As an example, consider the division of 33 by 5 using both long division and the SUBC method. In this case, i = 6, j = 3, and the SUBC operation is repeated 6–3 + 1 = 4 times.

LONG DIVISION:

```
                                        Quotient
                                  00000000000000000000000000000110
                                 ----------------------------------
00000000000000000000000000000101 | 00000000000000000000000000100001
                                                          -101
                                                          ----
                                                            1101
                                                           -101
                                                           ----
                                              Remainder      11
```

SUBC METHOD:



When the SUBC command is used, both the dividend and the divisor must be positive. Example 3–7 shows a realization of the integer division in which the sign of the quotient is properly handled. The last instruction before returning modifies the condition flag, in case subsequent operations depend on the sign of the result.

Example 3–7. Integer Division

```
*
*   TITLE INTEGER DIVISION
*
*         SUBROUTINE DIVI
*
*
*   INPUTS:          SIGNED INTEGER DIVIDEND IN R0,
*                    SIGNED INTEGER DIVISOR IN R1.
*
*   OUTPUT: R0/R1 into R0.
*
*   REGISTERS USED: R0–R3, IR0, IR1
*
*   OPERATION:    1. NORMALIZE DIVISOR WITH DIVIDEND
*                 2. REPEAT SUBC
*                 3. QUOTIENT IS IN LSBs OF RESULT
*
*   CYCLES: 31–62 (DEPENDS ON AMOUNT OF NORMALIZATION)
*         .globl  DIVI
SIGN      .set    R2
TEMPF     .set    R3
TEMP      .set    IR0
COUNT     .set    IR1
*   DIVI – SIGNED DIVISION
DIVI:
*
*   DETERMINE SIGN OF RESULT. GET ABSOLUTE VALUE OF OPERANDS.
*
          XOR     R0,R1,SIGN       ;Get the sign
          ABSI    R0
          ABSI    R1
          CMPI    R0,R1            ;Divisor > dividend ?
          BGTD    ZERO             ;If so, return 0
*
*   NORMALIZE OPERANDS. USE DIFFERENCE IN EXPONENTS AS SHIFT COUNT
*   FOR DIVISOR, AND AS REPEAT COUNT FOR 'SUBC'.
*
          FLOAT   R0,TEMPF         ;Normalize dividend
          PUSHF   TEMPF            ;PUSH as float
          POP     COUNT            ;POP as int
          LSH     –24,COUNT        ;Get dividend exponent
          FLOAT   R1,TEMPF         ;Normalize divisor
          PUSHF   TEMPF            ;PUSH as float
          POP     TEMP             ;POP as int
          LSH     –24,TEMP         ;Get divisor exponent
          SUBI    TEMP,COUNT       ;Get difference in exponents
          LSH     COUNT,R1         ;Align divisor with dividend
*
*   DO COUNT+1 SUBTRACT & SHIFTS.
*
          RPTS    COUNT
          SUBC    R1,R0
*
```

Example 3–7. Integer Division (Continued)

```
*   MASK OFF THE LOWER COUNT+1 BITS OF R0
*
          SUBRI   31,COUNT          ;Shift count is (32 – (COUNT+1))
          LSH     COUNT,R0          ;Shift left
          NEGI    COUNT
          LSH     COUNT,R0          ;Shift right to get result
*
*   CHECK SIGN AND NEGATE RESULT IF NECESSARY.
*
          NEGI    R0,R1             ;Negate result
          ASH     –31,SIGN          ;Check sign
          LDINZ   R1,R0             ;If set, use negative result
          CMPI    0,R0              ;Set status from result RETS
*
*   RETURN ZERO.
*
ZERO:
          LDI     0,R0
          RETS
          .end
```

If the dividend is less than the divisor and you want fractional division, you can perform a division after you determine the desired accuracy of the quotient in bits. If the desired accuracy is *k* bits, start by shifting the dividend left by *k* positions. Then apply the algorithm described above, and replace with *i + k*. It is assumed that *i + k* is less than 32.

3.5.2 Computation of Floating-Point Inverse and Division

When you use the RCPF (reciprocal of a floating-point number) instruction to generate an estimate of the reciprocal of a floating-point number, you can also use Newton-Raphson algorithm to extend the precision of the mantissa of the reciprocal of a floating-point number that the instruction generates. The floating-point division can be obtained by multiplying the dividend and the reciprocal of the divisor.

The input to RCPF is assumed to be $v = v(man) \times 2^{v(exp)}$. The output is $x = x(man) \times 2^{x(exp)}$. The value v(man) (or x(man)) is composed of three fields: the sign bit v(sign), an implied nonsign bit, and the fraction field v(frac).

Four rules apply to generating the reciprocal of a floating-point number:

1) If v > 0, then x(exp) = –v(exp) – 1, and x(man) = 2/v(man).
   For the special case in which the 10 MSBs of v(man) = 01.00000000b, then $x(man) = 2 – 2^{-8}$ = 01.11111111b. In both cases, the 23 LSBs of x(frac) = 0.

2) If v < 0, then x(exp) = –v(exp) – 1, and x(man) = 2/v(man).
   For the special case in which the 10 MSBs of v(man) = 10.00000000b,

then x(man) = $-1 - 2^{-8}$ = 10.11111111b. In both cases, the 23 LSBs of x(frac) = 0.

3) If v = 0 (v(exp) = –128), then x(exp) = 127, and
x(man) = 01.11111111111111111111111111111111b.
In other words, if v = 0, then *x* becomes the largest positive number representable in the extended-precision floating-point format. The overflow flag (V) is set to 1.

4) If v(exp) = 127, then x(exp) = –128, and x(man) = 0.
The zero flag (Z) is set to 1.

The Newton-Raphson algorithm is:

x[n+1] = x[n](2.0 – vx[n])

In this algorithm, *v* is the number for which the reciprocal is desired. *x*[0] is the seed for the algorithm and is given by RCPF. At every iteration of the algorithm, the number of bits of accuracy in the mantissa doubles. Using RCPF, accuracy starts at eight bits. With one iteration, accuracy increases to16 bits in the mantissa, and with the second iteration, accuracy increases to 32 bits in the mantissa. Example 3–8 shows the program for implementing this algorithm on the ’C4x.

Example 3–8. Inverse of a Floating-Point Number With 32-Bit Mantissa Accuracy

```
*
*   TITLE INVERSE OF A FLOATING-POINT NUMBER WITH 32-BIT
*   MANTISSA ACCURACY
*
*   SUBROUTINE INVF
*
*   THE FLOATING-POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, 1/v IS STORED IN R1.
*
*   TYPICAL CALLING SEQUENCE:
*   LAJU  INVF
*   LDF   v, R0
*   NOP   <---- can be other non-pipeline-break
*   NOP   <---- instructions
*
*   ARGUMENT ASSIGNMENTS:
*
*   ARGUMENT     |FUNCTION
*   ------------ +-----------------------------------------
*        R0      |   v = NUMBER TO FIND THE RECIPROCAL OF
*                |           (UPON THE CALL)
*        R1      |   1/v (UPON THE RETURN)
*
*   REGISTER USED AS INPUT:          R0
*   REGISTERS MODIFIED:              R1, R2
*   REGISTER CONTAINING RESULT:      R1
*   REGISTER FOR SUBROUTINE CALL:    R11
*
*   CYCLES: 7 (not including subroutine overhead)
*   WORDS: 8 (not including subroutine overhead)
*
*
         .global INVF
*
INVF:    RCPF    R0,R1       ;Get x[0] = the
                             ;estimate of 1/v, R0 = v
*
         MPYF3   R1,R0,R2
         SUBRF   2.0,R2
         MPYF    R2,R1       ;End of first iteration
                             ;(16 bits accuracy)
*
         BUD     R11         ;Delayed return to caller
*
         MPYF3   R1,R0,R2
         SUBRF   2.0,R2
         MPYF    R2,R1       ;End of second iteration
                             ;(32 bits accuracy)
*
*        R1 = 1/v, Return to caller
*
         .end
```

3.6 Calculating a Square Root

In many applications, normalization of data values is necessary. Often, the normalizing factor is the square root of another quantity. For example, given a vector, the unit vector in the same direction as the original vector can be found by normalizing the original vector by its length. This involves a division by a square root. The ’C4x single-cycle instruction RSQRF generates an estimate of the reciprocal of the square root of a positive floating-point number. This estimate has the correct exponent, and the mantissa is accurate to the eighth binary place (the error of the mantissa is $< 2^{-8}$). Three rules apply to this algorithm:

1) If v(exp) is even, then x(exp) = –(v(exp)/2) – 1, and
   x(man) = 2/sqrt(v(man)).

   For the special case where the 10 MSBs of y(man) = 01.00000000b, then x(man) = $2 – 2^{-8}$ = 01.11111111b. In both cases, the 23 LSBs of x(frac) = 0.

2) If v(exp) is odd, then x(exp) = –((v(exp) – 1)/2) – 1 and
   x(man) = sqrt(2/v(man)). The 23 LSBs of x(frac) = 0.

3) If v = 0 (v(exp) = –128), then x(exp) = 127, and
   x(man) = 01.11111111111111111111111111111111b.
   In other words, if v = 0, then *x* becomes the largest positive number representable in the extended-precision floating-point format. The overflow flag (V) is set to 1.

If you need larger precision than the RSQRF instruction gives for the estimate of the reciprocal of the square root, you can use the Newton-Raphson algorithm to further extend the precision of the mantissa. The algorithm is:

x[n+1] = x[n](1.5 – (v/2) x [n] x [n])

In this equation, *v* is the number for which the reciprocal is desired. *x*[0] is the seed for the algorithm and is given by RSQRF. At every iteration of the algorithm, the number of bits of accuracy in the mantissa doubles. Using RSQRF, accuracy starts at eight bits. With one iteration, accuracy increases to16 bits, and with the second iteration, accuracy increases to 32 bits in the mantissa. Example 3–9 shows the program for implementing this algorithm on the ’C4x.

Example 3–9.Reciprocal of the Square Root of a Positive Floating Point

```
*   TITLE RECIPROCAL OF THE SQUARE ROOT OF A POSITIVE
*                 FLOATING-POINT
*
*   SUBROUTINE RCPSQRF
*
*   THE FLOATING-POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, 1/SQRT(v) IS STORED IN R1.
*
*   TYPICAL CALLING SEQUENCE:
*   LDF   v, R0
*   LAJU  RCPSQRF
*
*   ARGUMENT ASSIGNMENTS:
*
*   ARGUMENT      |  FUNCTION
*    ------------ +-------------------------------------
*   R0            |  v = NUMBER TO FIND THE RECIPROCAL OF
*                 |           (UPON THE CALL)
*   R1            |  1/sqrt(v) (UPON THE RETURN)
*
*   REGISTER USED AS INPUT:       R0
*   REGISTERS MODIFIED:           R1, R2
*   REGISTER CONTAINING RESULT:   R1
*   REGISTER FOR SUBROUTINE CALL: R11
*
*   CYCLES: 10 (not including subroutine overhead)
*   WORDS: 10 (not including subroutine overhead)
*
          .global RCPSQRF
*
RCPSQRF: RSQRF   R0,R1       ;Get x[0] = the estimate of 1/sqrt(v), R0 = v
          MPYF    0.5,R0     ;R0 = v/2
*
          MPYF3   R1,R1,R2   ;First iteration
          MPYF    R0,R2
          SUBRF   1.5,R2
          MPYF    R2,R1      ;End of first iteration (16 bits accuracy)
*
          MPYF3   R1,R1,R2   ;Second iteration
*
          BRD     R11        ;Delayed return to caller
*
          MPYF    R0,R2
          SUBRF   1.5,R2
          MPYF    R2,R1      ;End of second iteration (32 bits accuracy)
*
*   R1 = 1/SQRT(v), Return to caller
*
          .end
```

You can find the square root by a simple multiplication: sqrt(v) = vx[n] in which x[n] is the estimate of 1/sqrt(v) as determined by the Newton-Raphson algorithm or another algorithm.

3.7 Extended-Precision Arithmetic

The ’C4x offers 32 bits of precision in the mantissa for integer arithmetic, and 24 bits of precision in the mantissa for floating-point arithmetic. For higher precision in floating-point operations, the twelve extended-precision registers, R0 to R11, contain eight more bits of accuracy. Because no comparable extension is available for fixed-point arithmetic, this section discusses how to achieve fixed-point double precision. The technique consists of performing the arithmetic by parts and is similar to the way in which longhand arithmetic is done.

The instructions, ADDC (add with carry) and SUBB (subtract with borrow) use the status carry bit for extended-precision arithmetic. The carry bit is affected by the arithmetic operations of the ALU and by the rotate and shift instructions. You can also manipulate it directly by setting the status register to certain values. For proper operation, the overflow mode bit should be reset (OVM = 0) so that the accumulator results are not loaded with the saturation values. Example 3–10 and Example 3–11 show 64-bit addition and 64-bit subtraction, respectively. The first operand is stored in the registers R0 (low word) and R1 (high word). The second operand is stored in registers R2 and R3, respectively. The result is stored in R0 and R1.

Example 3–10. 64-Bit Addition

```
*
*   TITLE 64-BIT ADDITION
*
*   TWO 64-BIT NUMBERS ARE ADDED TO EACH OTHER PRODUCING
*
*   A 64-BIT RESULT. THE NUMBERS X (R1,R0) AND Y (R3,R2)
*
*   ARE ADDED, RESULTING IN W (R1,R0).
*
*      R1   R0
*    + R3   R2
*      -------
*      R1   R0
*
       ADDI   R2,R0
       ADDC   R3,R1
```

Example 3–11. 64-Bit Subtraction

```
*
*   TITLE 64-BIT SUBTRACTION
*
*   TWO 64-BIT NUMBERS ARE SUBTRACTED FROM EACH OTHER
*   PRODUCING A 64-BIT RESULT. THE NUMBERS X (R1,R0) AND
*   Y (R3,R2) ARE SUBTRACTED, RESULTING IN W (R1,R0).
*
*      R1   R0
*    – R3   R2
*      -------
*      R1   R0
*
       SUBI   R2,R0
       SUBB   R3,R1
```

When two 32-bit numbers are multiplied, a 64-bit product results. To do this, ’C4x provides a 32 bit x 32-bit multiplier and two special instructions, MPYSHI (multiply signed integer and produce 32 MSBs) and MPYUHI (multiply unsigned integer and produce 32 MSBs). Example 3–12 shows the implementation of a 32-bit x 32-bit multiplication.

Example 3–12. 32-Bit by 32-Bit Multiplication

```
*
*   TITLE 32 BIT × 32-BIT MULTIPLICATION
*
*   MULTIPLIES 2 32-BIT NUMBERS, PRODUCING A 64-BIT RESULT.
*   THE TWO NUMBERS R0 AND R1 ARE MULTIPLIED, RESULTING
*   IN W (R3,R2).
*
*        R0
*     ×  R1
*        ----
*        R3  R2
*
       MPYI3  R0,R1,R2
       MPYSHI3   R0,R1,R3
```

3.8 Floating-Point Format Conversion: IEEE to/From ’C4x

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. Therefore, if the binary point of a 32-bit number is fixed after the most significant bit (which is also the sign bit), only a fractional number (a number with an absolute value less than 1), can be represented. In other words, there is a number with 31 fractional bits. All operations assume that the binary point is fixed at this location. The fixed-point system, although simple to implement in hardware, imposes limitations in the dynamic range of the represented number. This causes scaling problems in many applications. You can avoid this difficulty by using floating-point numbers.

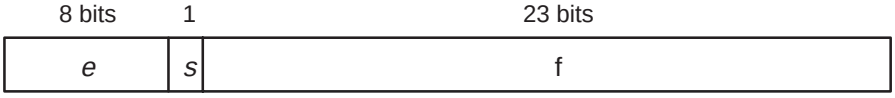
A floating-point number consists of a mantissa *m* multiplied by base *b* raised to an exponent *e*:

$m \times b^e$

In current hardware implementations, the mantissa is typically a normalized number with an absolute value between 1 and 2, and the base is b = 2. Although the mantissa is represented as a fixed-point number, the actual value of the overall number floats the binary point because of the multiplication by b^e. The exponent *e* is an integer whose value determines the position of the binary point in the number. IEEE has established a standard format for the representation of floating-point numbers.

To achieve higher efficiency in the hardware implementation, the ’C4x uses a floating-point format that differs from the IEEE standard. However, ’C4x has two single-cycle instructions, TOIEEE and FRIEEE, for the format conversion. These two instructions can also be used with the STF instruction, which allows the data format to be converted within memory-to-memory transfer. Here are descriptions of both formats and an example program to convert between them.

’C4x floating-point format:

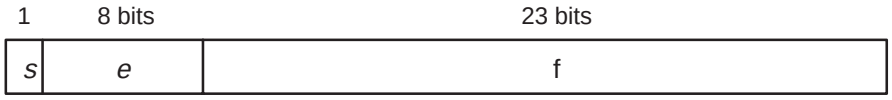


In a 32-bit word representing a floating-point number, the first 8 bits correspond to the exponent expressed in twos-complement format. One bit is for sign, and 23 bits are for the mantissa. The mantissa is expressed in twos-complement form with the binary point after the most significant nonsign bit. Because this bit is the complement of the sign bit *s*, it is suppressed. In other words, the mantissa actually has 24 bits. One special case occurs when

e = –128. In this case, the number is interpreted as zero, independently of the values of *s* and *f* (which are, by default, set to zero). To summarize, the values of the represented numbers in the ’C4x floating-point format are as follows:

2^e * (01.f)	if *s* = 0
2^e * (10.f)	if *s* = 1
0	if *e* = –128

IEEE floating-point format:



The IEEE floating-point format uses sign-magnitude notation for the mantissa. In a 32-bit word representing a floating-point number, the first bit is the sign bit. The next 8 bits correspond to the exponent, expressed in an offset-by-127 format (the actual exponent is *e*–127). The following 23 bits represent the absolute value of the mantissa with the most significant 1 implied. The binary point is fixed after this most significant 1. In other words, the mantissa actually has 24 bits. Several special cases are summarized below.

These are values of the represented numbers in the IEEE floating-point format:

$(-1)^s$ * 2^{e-127} * (01.f)	if 0 < *e* < 255

Special cases:

$(-1)^s$ * 0.0	if *e* = 0 and f = 0 (zero)
$(-1)^s$ * 2^{-126} * (0.f)	if *e* = 0 and f <> 0 (denormalized)
$(-1)^s$ * infinity	if *e* = 255 and f = 0 (infinity)
NaN (not a number)	if *e* = 255 and f <> 0

The ’C4x performs the conversion according to these definitions of the formats. It assumes that the source data for the IEEE format is in memory only and that the source data for the ’C4x floating-point format is in either memory or an extended-precision register. The destination for both conversions must be in an extended-precision register. In the case of block memory transfer, the no-penalty data-format conversion can be executed by parallel instruction with STF. Example 3–13 and Example 3–14 show the data-format conversion within the data transformation between communication port and internal RAM.

Example 3–13. IEEE to 'C4x Conversion Within Block Memory Transfer

```
*   TITLE IEEE TO 'C4x CONVERSION WITHIN BLOCK MEMORY
*   TRANSFER
*
*   PROGRAM ASSUMES THAT INPUT FIFO OF COMMUNICATION PORT 0
*   IS FULL OF IEEE FORMAT DATA. EIGHT DATA WORDS ARE
*   TRANSFERRED FROM COMMUNICATION PORT 0 TO INTERNAL RAM
*   BLOCK 0 AND THE DATA FORMAT IS CONVERTED FROM IEEE FORMAT
*   TO 'C4x FLOATING-POINT FORMAT.
*
       .
       .
       .
       LDI    @CP0_IN,AR0   ;Load comm port0 input FIFO address
       LDI    @RAM0,AR1     ;Load internal RAM block 0 address
       FRIEEE *AR0,R0       ;Convert first data
       RPTS   6
       FRIEEE *AR0,R0       ;Convert next data
||     STF    R0,*AR1++(1)  ;Store previous data
       STF    R0,*AR1++(1)  ;Store last data
       .
       .
       .
```

Example 3–14. 'C4x to IEEE Conversion Within Block Memory Transfer

```
*   TITLE  'C4x TO IEEE CONVERSION WITHIN BLOCK MEMORY
*   TRANSFER
*
*   PROGRAM ASSUMES THAT OUTPUT FIFO OF COMMUNICATION PORT 0
*   IS EMPTY. EIGHT DATA WORDS ARE TRANSFERRED FROM INTERNAL
*   RAM BLOCK 0 TO COMMUNICATION PORT 0 AND THE DATA FORMAT
*   IS CONVERTED FROM 'C4x FLOATING-POINT FORMAT TO
*   IEEE FORMAT.
*
       .
       .
       .
       LDI    @CP0_OUT,AR0  ;Load comm port0 output FIFO address
       LDI    @RAM0,AR1     ;Load internal RAM block 0 address
       TOIEEE *AR1++(1),R0  ;Convert first data
       RPTS   6
       TOIEEE *AR1++(1),R0  ;Convert next data
||     STF    R0,*AR0       ;Store previous data
       STF    R0,*AR0       ;Store last data
       .
       .
       .
```

Chapter 4

Memory Interfacing

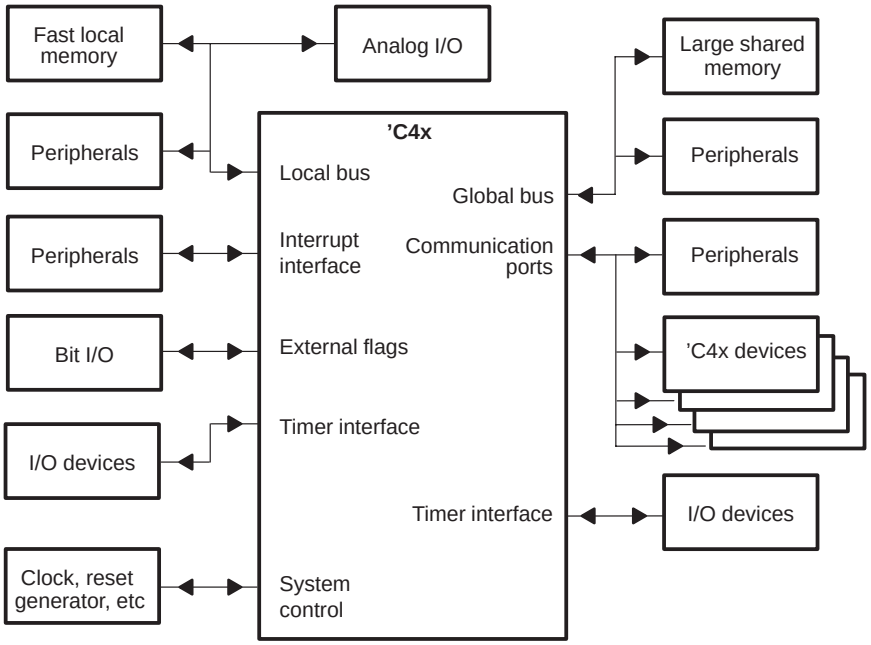
The ’C4x’s advanced interface design can be used to implement a wide variety of system configurations. Its two external buses and DMA capability provide a flexible parallel 32-bit interface to byte- or word-wide devices.

This chapter describes how to use the ’C4x’s memory interfaces to connect to various external devices. Specific discussions include implementation of a parallel interface to devices with and without wait states and implementing system control functions.

4.1 System Configuration

Figure 4–1 illustrates an expanded configuration of a ’C4x system with different types of external devices and the interfaces to which they are connected.

Figure 4–1. Possible System Configurations

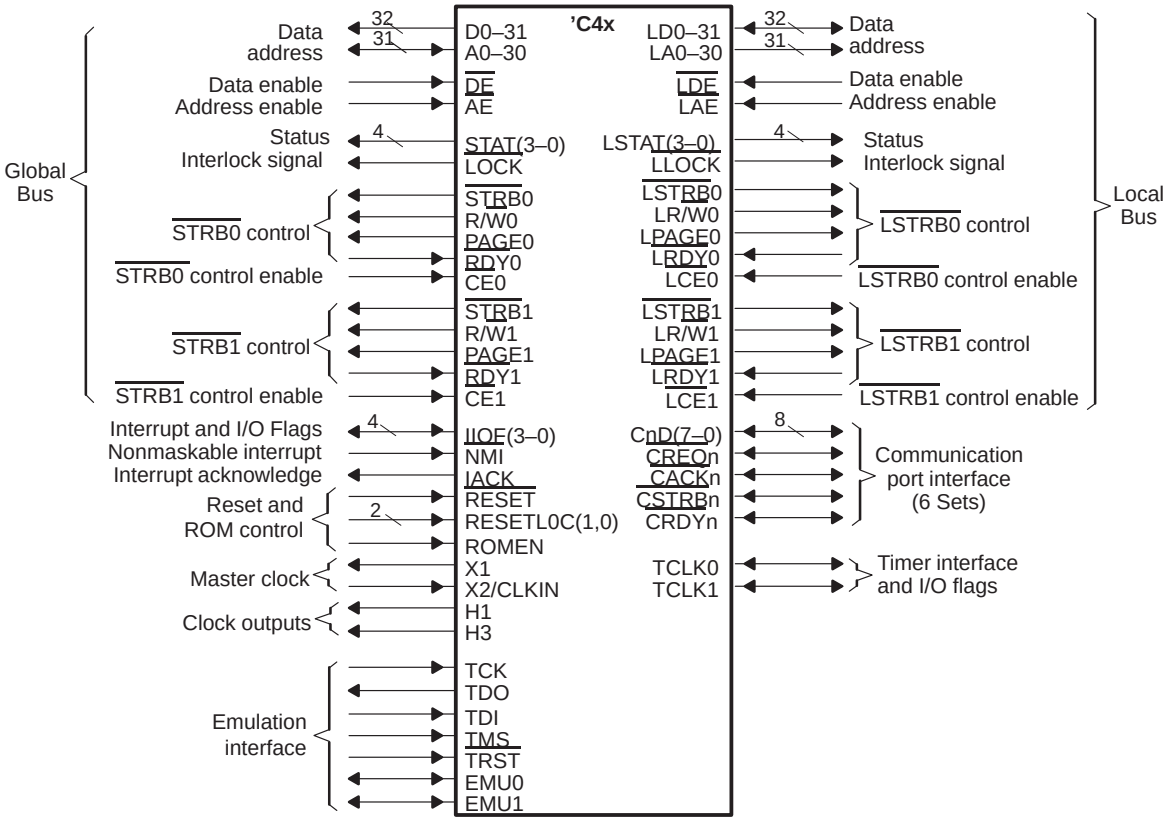


In your design, you can use any subset or superset of the illustrated components.

4.2 External Interfacing

The ’C4x interfaces connect to a wide variety of device types. Each of these interfaces is tailored to a particular type of device such as memory, DMA, parallel and serial peripherals, and I/O. In addition, ’C4x devices can interface directly with each other, without external logic, through their communication ports or their external flag pins $\overline{\text{IIOF}}$(0–3). Each interface comprises one or more signal lines, which transfer information and control its operation. Figure 4–2 shows the signal groups for these interfaces.

Figure 4–2. External Interfaces



Note: n = 0 for communication port 0, n = 1 for communication port 1, etc.

The global and local buses implement the primary memory-mapped interfaces to the device. These interfaces allow external devices such as DMA controllers and other microprocessors to share resources with one or more ’C4x devices through a common bus.

4.3 Global and Local Bus Interfaces

The ’C4x uses the global and local buses to access the majority of its memory-mapped locations. Since these two memory interfaces are identical in every way, except for their positions in the memory map, each example in this memory interface section focuses on only one of the two interfaces. However, all of the examples are applicable to either the local or global bus. The buses have identical but mutually exclusive sets of control signals:

Table 4–1. Local/Global Bus Control Signals

Global Bus	Local Bus
$\overline{\text{STRB0}}$	$\overline{\text{LSTRB0}}$
$\overline{\text{STRB1}}$	$\overline{\text{LSTRB1}}$
$\overline{\text{CE0}}$	$\overline{\text{LCE0}}$
$\overline{\text{CE1}}$	$\overline{\text{LCE1}}$
$\overline{\text{RDY0}}$	$\overline{\text{LRDY0}}$
$\overline{\text{RDY1}}$	$\overline{\text{LRDY1}}$
$\overline{\text{AE}}$	$\overline{\text{LAE}}$
$\overline{\text{DE}}$	$\overline{\text{LDE}}$
$\overline{\text{PAGE0}}$	$\overline{\text{LPAGE0}}$
$\overline{\text{PAGE1}}$	$\overline{\text{LPAGE1}}$
R/$\overline{\text{W}}$0	LR/$\overline{\text{W}}$0
R/$\overline{\text{W}}$1	LR/$\overline{\text{W}}$1

While both the global bus and the local bus can interface to a wide variety of devices, they most commonly interface to memories.

4.4 Zero Wait-State Interfacing to RAMs

A memory-read access time is normally defined as the time between address valid and data valid. This time can be determined by:

Read access time $= t_{c(H)} - (t_{d(H1L-A)} + t_{su(D)R})$

where:

$t_{c(H)}$ = H1/H3 cycle time

$t_{d(H1L-A)}$ = H1 low to address valid

$t_{su(D)R}$ = Data valid before next H1 low (read)

For a full-speed, zero wait-state interface to any device, a 50-MHz ’C4x (40-ns instruction cycle time) requires a read access time of 21 ns from address stable to data valid. For most memories, the access time from chip enable is the same as access time from address; thus, it is possible to use 20-ns memories at full speed with a 50-MHz ’C4x. However, to use 20-ns memories properly, you must avoid long delays between the processor and the memories.

Avoiding these delays is not always possible, because interconnections and gating for chip-enable generation can cause them. In addition, if you choose a memory device with an output enable, the output enable must become active quickly enough to ensure that the memory can meet the data valid timing requirements of the ’C4x. For memories with 20-ns access times, the output enable active to data valid timing parameter is typically less than 10 ns.

Currently available RAMs without output-enable (OE) control lines include the 1-bit wide organized RAMs and most of the 4-bit wide RAMs. Those with OE controls include the byte-wide and a few of the 4-bit wide RAMs. Many of the fastest RAMs do not provide OE control; they use chip-enable (CE) controlled write cycles to ensure that data outputs do not turn on for write operations. In CE-controlled write cycles, the write control line ($\overline{WE}$) goes low before CE goes low, and internal logic holds the outputs disabled until the cycle is completed. Using CE-controlled write cycles is an efficient way to interface fast RAMs without OE controls to the ’C4x at full speed.

Note:

You can find timing parameters for CLKIN, H1, H3, and memory in the TMS320C40 and TMS320C44 data sheets.

4.4.1 Consecutive Reads Followed by a Write Interface Timing

Figure 4–3 shows the timing of consecutive reads followed by a write. For consecutive reads, $\overline{\text{LSTRB}}0$ stays active (low), and LR/$\overline{W}$ stays high as long as read cycles continue. For back-to-back reads, the ’C4x requires zero-wait-state memories to have an address-valid to data-valid time of less than 21 ns.

For most memory devices, this time is the same as the memory access time, which is t_1 = 20 ns. Thus, memories with access times of 25 ns or more cannot meet this timing.

Memory device timing is not as critical for zero-wait-state as for nonzero-wait-state write cycles, because of the two H1 cycle writes of the ’C4x. The extra cycle gives $\overline{\text{LSTRB}}0$ enough time to frame LR/$\overline{W}$, preventing memories that go into high impedance slowly at the end of a read cycle from driving the bus during the subsequent write cycle. For the memory device used in this design (Figure 4–3), the data lines are guaranteed to into high impedance (t_2 = 10 ns) after $\overline{\text{CS}}$ goes inactive, which gives more than 23 ns of margin before the ’C4x starts driving the bus with write data. Also, the extra cycle with $\overline{\text{LSTRB}}0$ inactive prevents writes to random locations in memory while the address is changing between consecutive writes.

For the write cycles shown in Figure 4–3 and Figure 4–4, the RAM requires 15 ns of write data setup before $\overline{\text{CS}}$ goes high, and this design provides at least 24 ns (t_3). A data hold time of 0 ns (t_4) is required by the RAM, and this design provides greater than 13 ns. Finally, the RAM’s 20-ns setup and 0-ns hold times for address (with respect to $\overline{\text{CS}}$ high) ensure a clear margin.

Figure 4–3. Consecutive Reads Followed by a Write

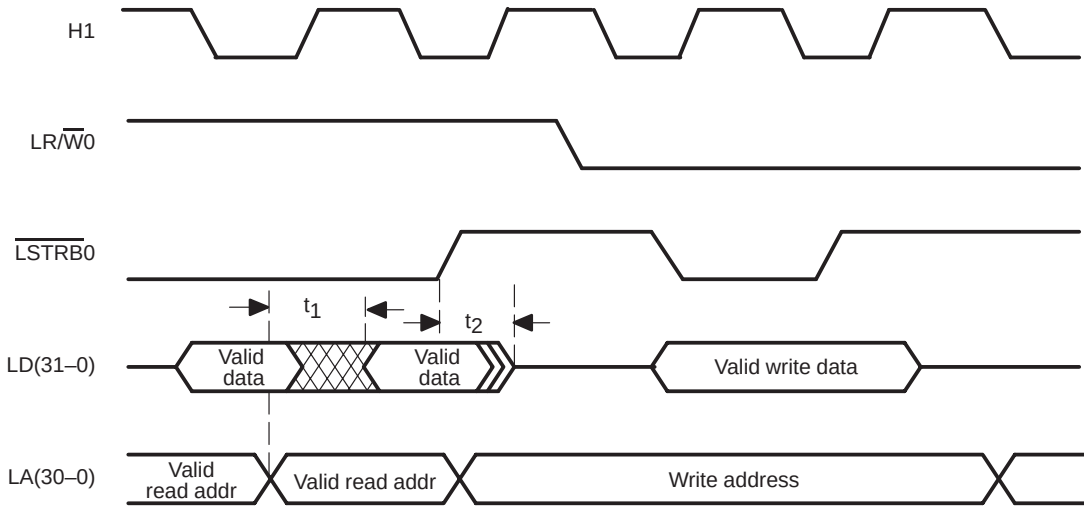
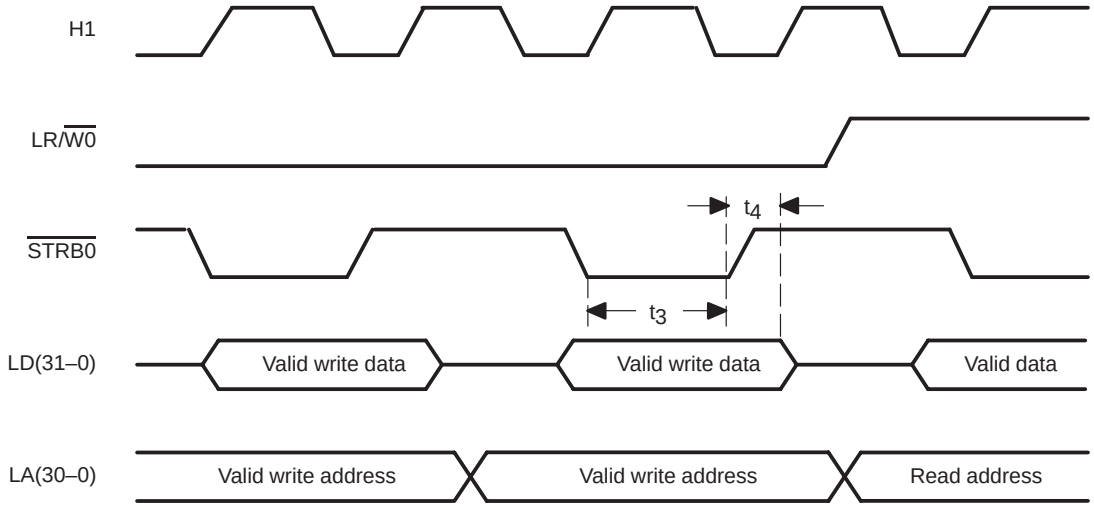


Figure 4–4. Consecutive Writes Followed by a Read



4.4.2 Consecutive Writes Followed by a Read Interface Timing

Figure 4–4 shows the timing of consecutive writes followed by a read. Notice that between consecutive writes, LR/$\overline{W}$ stays low, but $\overline{STRB}$0 goes inactive to frame the write cycles. Although ’C4x zero-wait-state writes take two H1 cycles, writes appear to take one cycle internally (from the perspective of the CPU and DMA) if no access to the interface is already in progress.

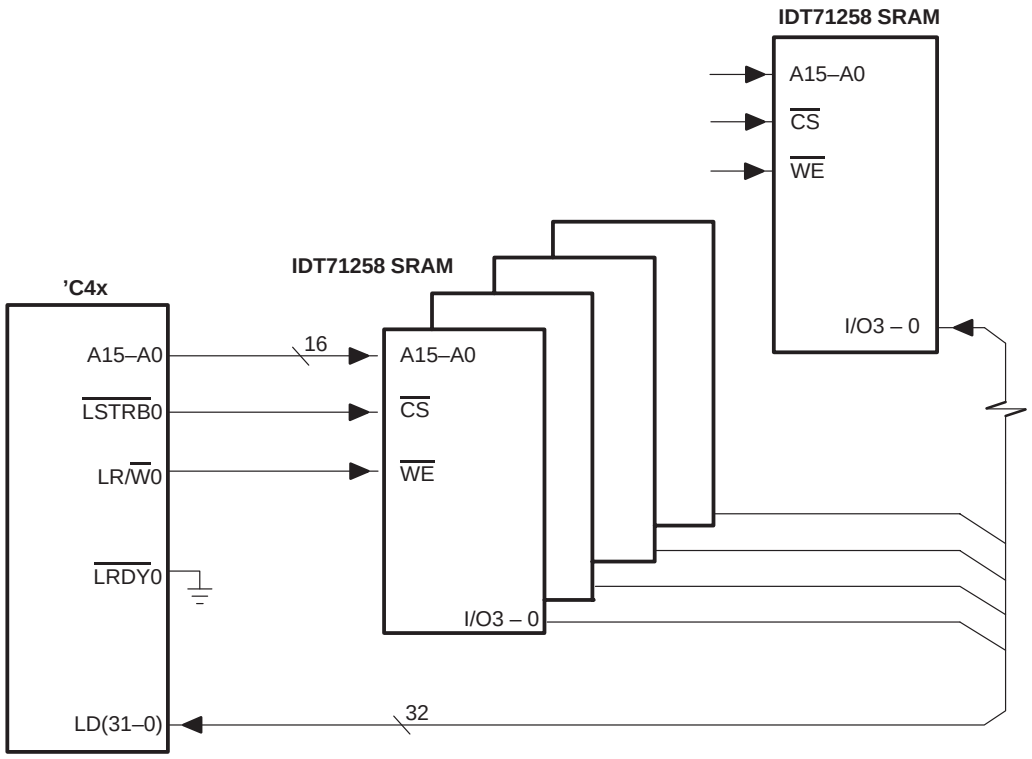
In the read cycle following the writes in Figure 4–4, the ’C4x requires zero-wait-state memories to have a $\overline{LSTRB}$-active to data-valid time of less than 21 ns (one H1 cycle minus (H1 low to $\overline{LSTRB}$ active plus data setup before H1 low)). For most memory devices, this time is the same as the memory access time, which is t_1 = 20 ns in this design. Thus, a margin of only 1 ns exists, leaving little allowance for $\overline{STRB}$ gating if desired.

4.4.3 RAM Interface Using One Local Strobe

Figure 4–5 shows the ’C4x’s local bus interfaced to eight Integrated Device Technology IDT71258 20-ns 64K $\times$ 4-bit CMOS static RAMs with zero wait states using chip-enable controlled write cycles. The SRAMs are arranged to implement the first 64K, 32-bit words in external memory, located at addresses 00000h thru 0FFFFh (internal ROM is assumed to be disabled). If these 64K words of SRAM are the only memory controlled by $\overline{LSTRB}$0, the LSTRB ACTIVE field of the local memory interface control register (LMICR) should be set to its minimum value of 01111_2, allowing $\overline{LSTRB}$0 to be active for only the first

64K words of the ’C4x’s memory space. In addition, if this memory is the only memory interfaced to $\overline{\text{LSTRB}}$0, $\overline{\text{LSTRB}}$0 requires only one page, and the PAGESIZE field of the LMICR should be set to 01111_2. Also note that in Figure 4–5, the $\overline{\text{LRDY}}$0 input is tied low, selecting zero wait states for all $\overline{\text{LSTRB}}$0 accesses on the local bus. With all of the zero-wait-state memory controlled by $\overline{\text{LSTRB}}$0, $\overline{\text{LSTRB}}$1 can be used to control accesses to slower read-only memory devices or other types of memory.

Figure 4–5. ’C4x Interface to Eight Zero-Wait-State SRAM



In this circuit implementation, no external logic is necessary to interface the ’C4x to the memory device. Typically, memory devices must be held inactive ($\overline{\text{CS}}$ inactive) during changes in $\overline{\text{WE}}$; this avoids undesired memory accesses while the address changes. The ’C4x ensures this glueless interface because $\overline{\text{LSTRB}}$ always frames changes in LR/$\overline{\text{W}}$.

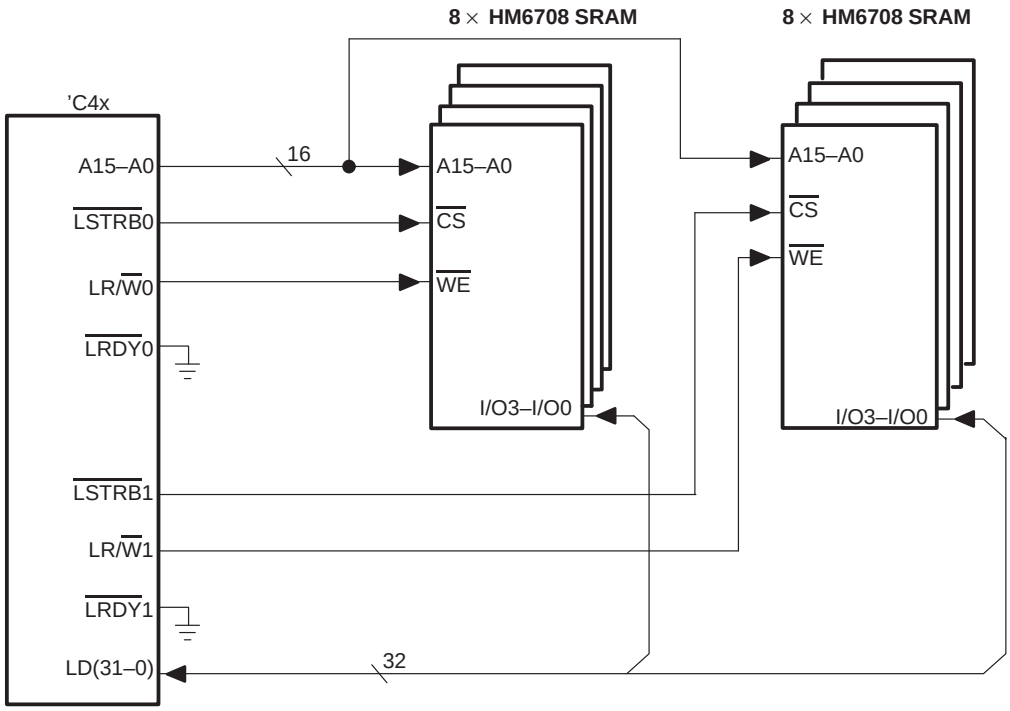
4.4.4 RAM Interface Using Both Local Strobes

Figure 4–6 shows the ’C4x’s local bus interfaced to HM6708 — 20-ns 64K × 4-bit CMOS static RAMs with zero wait states using $\overline{\text{CS}}$ controlled write cycles.

These RAMs are arranged to allow 128K 32-bit words of local memory, which are implemented as two 64K × 32-bit banks. One bank is controlled by each of the two sets of control signals on the local bus. To map these memory devices properly in the ’C4x’s memory space, you must use the local-memory-interface control register (LMICR) to define which part of the local bus’s memory space is mapped to each of the two strobes. In this implementation with internal ROM disabled, $\overline{\text{LSTRB}}$0 is mapped to the first 64K words of the local space (addresses 0h through 0FFFFh), and $\overline{\text{LSTRB}}$1 is mapped to the rest of the local space (addresses 10000h through 7FFF FFFFh). For this memory configuration, the LSTRB ACTIVE field of the local-memory-interface control register (LMICR) should be set to 01111_2. Also, each $\overline{\text{LSTRB}}$ requires only one page. The PAGESIZE field of the LMICR should be set to 01111_2. Note that in Figure 4–6, the $\overline{\text{LRDY}}$ inputs are tied low, selecting zero wait states for all accesses on the local bus.

Hence, through the use of the ’C4x’s four strobes (two each on the local and global buses), four different banks of memory can be decoded. In addition, through program control, you can change the address decoding under program control by changing the LSTRB active field (bits 24–28) of the LMICR or the global-memory-interface control register (GMICR). If you must decode more than four banks of memory or if the chosen memory device cannot meet the read cycle timing requirements for the ’C4x at zero wait states, you should use page switching (discussed in subsection 4.5.6 on page 4-18) to add an extra cycle to read accesses outside the current bank boundary.

Figure 4–6. ’C4x Interface to Zero-Wait-State SRAMs, Two Strobes



4.5 Wait States and Ready Generation

Using wait states can greatly increase a system's flexibility and reduce its hardware requirement. The 'C4x is capable of generating wait states on either the global bus or the local bus, and both buses have independent sets of ready control logic. The buses' wait-state configuration is determined by the SWW and WTCNT fields of the local and global-bus-interface control registers.

This section discusses ready generation from the perspective of the *global-bus* interface; however, wait-state operation on the *local bus* is the same as on the global bus, so this discussion pertains equally well to both (local and global). Also, the local and global buses each have two sets of control signals — R/$\overline{W}$0, $\overline{STRB}$0, $\overline{RDY}$0, $\overline{PAGE}$0, $\overline{CE}$0 and R/$\overline{W}$1, $\overline{STRB}$1, $\overline{RDY}$1, $\overline{PAGE}$1, $\overline{CE}$1— with each set of control signals having its own ready signal, providing for more flexibility in support of external devices with different speeds. Since both strobes' ready signals share the same electrical characteristics, the following discussion focuses on one of the global bus's set of control signals.

Wait states are generated by:

- The internal wait-state generator
- The external ready inputs ($\overline{RDY}$0 or $\overline{RDY}$1)
- The logical AND or OR of the two ready signals

When enabled, internally generated wait states affect 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 can be used to customize wait-state generation to specific system requirements.

If either 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 the two signals will generate a ready condition and allow the cycle to be completed. It is not required that both signals be present.

4.5.1 ORing of the Ready Signals ($\overline{\text{STRB}}$x SWW = 10)

You can use the OR of the two ready signals to implement wait states for devices that require more wait states than internal logic can implement (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 ready externally with a minimum of logic. When fast devices are accessed, the external hardware responds promptly with ready, which terminates the cycle.
- **Slow devices** can use the internal wait counter for larger numbers of wait states. 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 can also terminate the bus cycle before the number of wait states implemented with external logic allows termination. 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. Also, this feature can be used as a safeguard against inadvertent accesses to nonexistent memory that would never respond with ready and would, therefore, lock up the ’C4x.

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 be able to reset its sequencing to allow a new cycle to begin immediately following the end of the internal wait count. Also, the consecutive cycles must be from independently decoded areas of memory (or from different pages in memory). Otherwise, the external ready generation logic may lose synchronization with bus cycles and generate improperly timed wait states.

4.5.2 ANDing of the Ready Signals ($\overline{\text{STRB}}$x SWW = 11)

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, but both signals must be asserted. Accordingly, external ready control must be implemented for each wait-state device, and the wait count ready signal must be enabled.

This feature is useful if devices in a system are equipped to provide a ready signal but cannot respond quickly enough to meet the ’C4x’s timing requirements. 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 provide wait states initially, and then the external ready can provide wait states 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 can 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.

4.5.3 External Ready Generation

The optimum technique for implementing external ready generation hardware depends on the specific characteristics of the system, including the relative number of wait-state and nonwait-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 to distinguish fast and slow devices

2) Generation of properly timed ready indications

3) Logical ORing of all the separate ready timing signals together to connect to the physical ready input

Segmentation of the address space is required to obtain a unique indication of each particular area within the address space that requires wait states. This segmentation is commonly implemented in the form of chip-select generation. Chip-select signals can initiate wait states in many cases; however, occasionally, chip-select decoding considerations may provide signals that do not allow ready input timing requirements to be met. In this case, you can segment *coarse* address space 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 that indicates that a particular area of memory is being addressed also normally initiates the ready or wait-state signal.

When address space to be accessed has been established, a timing circuit 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, you should logically OR the signals by using a single gate to drive the $\overline{\text{RDY}}$ input.

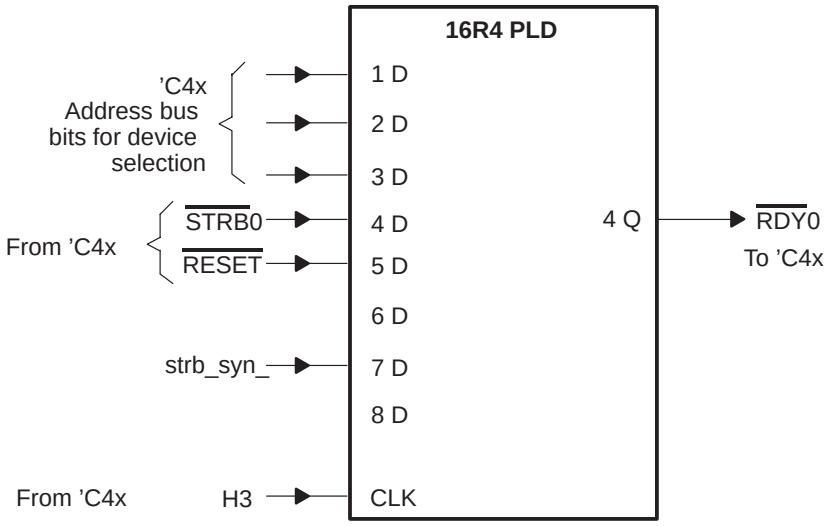
4.5.4 Ready Control Logic

You can take one of two basic approaches to implement ready control logic, depending on the state of the ready input 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 $\overline{\text{RDY}}$ i**s low between accesses**, control of devices that are zero-wait-state at full speed is straightforward; no action is necessary, because 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 difficult in many circumstances because 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 can simply delay their select signals appropriately to generate a ready. Typically, this approach results in the most efficient implementation of ready control logic. Figure 4–7 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 4–7. Logic for Generation of 0, 1, or 2 Wait States for Multiple Devices



4.5.5 Example Circuit

Figure 4–7 shows how a single, 7-ns 16R4 programmable logic device (PLD) can be used to generate 0, 1, and 2 wait states for multiple devices that are interfaced to a ’C4x. In this example, distinct address bits are used to select the different wait-state devices. Here, each of the three address lines input to the 16R4 corresponds to a different speed device. For a single 16R4 implementation, up to nine different address bits can be used to select different speed devices.

The single output, 4Q, of the PLD is connected directly to the $\overline{\text{RDY}}$0 input of the ’C4x to signal the completion of a bus access for external wait-state generation. Because $\overline{\text{RDY}}$0 is sampled on the falling of H1, the H3 output clock is used as the PLD clock input.

Example 4–1 shows the ready logic equations for programming the 16R4 PLD. The PLD language used is ABEL. $\overline{\text{STRB}}$0 is an input into the PLD that indicates that a valid ’C4x bus cycle is occurring. Also, a delayed version of $\overline{\text{STRB}}$0 (synchronized with H1 going high) is provided as the *strb_syn_* input signal. This delayed signal is needed to avoid problems with a race condition that may exist between $\overline{\text{STRB}}$0 going low and H3 rising. $\overline{\text{RESET}}$ can be used to bring the state machine back to the idle state.

Notice that the $\overline{\text{RDY}}$0 output of the PLD is not registered. An asynchronous $\overline{\text{RDY}}$0 signal is necessary to generate a ready signal for zero-wait-state devices. When a zero-wait-state device is selected (ahi1 high in Example 4–1) and $\overline{\text{STRB}}$0 is low, the PLD asserts $\overline{\text{RDY}}$0 low within 7 ns. Hence, $\overline{\text{RDY}}$0 goes active fast enough to satisfy the 20-ns setup time of $\overline{\text{RDY}}$0 low before H1 low.

For generation of $\overline{\text{RDY}}$0 for one and two wait states, the device select address bits and *strb_syn_* are delayed one and two cycles, respectively, by the PLD before a $\overline{\text{RDY}}$0 is brought active low. The one H3-cycle delay, required for one-wait-state device ready generation, corresponds to state wait_one in Example 4–1 and the two H3-cycle delay required for two-wait-state devices corresponds to state *wait_twoa* and *wait_twob*.

This 16R4 PLD-based design can be used to implement different numbers of wait states for multiple devices. More devices can be selected with ’C4x address lines, and a higher number of wait states can be produced with a PLD logic. Furthermore, this approach can be used in conjunction with the ’C4x’s internal wait-state generator.

Example 4–1. PLD Equations for Ready Generation

```
0001    |   module     ready_generation
0002    |   title'     ready generation logic for 0, 1 and 2 wait state devices interfaced
0003    |              to TMS320C4x'
0004    |
0005    |       C40u5 device 'P16R4';
0006    |
0007    |       "inputs
0008    |       h3          Pin 1;
0009    |
0010    |
0011    |       "The following are TMS320C40 address bits used to
0012    |       "select the different speed devices. More can be used if
0013    |       "necessary. In this example, a zero wait state, a one wait
0014    |       "state, and a two wait state device are decoded with these
        |       "three address bits
0015    |
0016    |       ahi1        Pin 2;  "when high selects zero wait state device
0017    |       ahi2        Pin 3;  "when high selects one wait state device
0018    |       ahi3        Pin 4;  "when high selects two wait state device
0019    |       strb0_      Pin 5;  "indicates valid TMS320C40 bus cycle
0020    |       reset_      Pin 6;  "reset signal from TMS320C40
0021    |       strb_syn_   Pin 7;  "reset strb0_ synchronized with H1 rising edge.
0022    |       "output
0023    |       rdy0_       Pin 12; "ready signal to TMS320C40
0024    |
0025    |       one_wait    Pin 14; "internal flip-flop signal for 1 wait state
0026    |                           "device ready signal generation
0027    |       two_waita   Pin 15; "internal flip-flop signal for first of the two
0028    |       "wait states for 2 wait state devices
0029    |       two_waitb   Pin 16; "internal flip-flop signal for second
0030    |                           "of the two wait states for 2 wait
0031    |                           "state devices
0032    |
0033    |       "name substitutions for test vectors
0034    |       c,H,L,X  = .C.,1,0,.X.;
0035    |
0036    |
0037    |       "state bits
0038    |       outstate = [one_wait, two_waita, two_waitb];
0039    |
0040    |       idle      = ^b111;
0041    |       wait_one  = ^b011;
0042    |       wait_twoa = ^b101;
0043    |       wait_twob = ^b110;
0044    |
0045    |
0046    |state_diagram  outstate
0047    |
0048    |state idle:
0049    |           if  (reset_ & ahi2 & !strb_syn_) then wait_one
0050    |           else if (reset_ & ahi3 & !strb_syn_) then wait_twoa
```

Example 4–1.PLD Equations for Ready Generation (Continued)

```
0051   |          else   idle;
0052   |
0053   |
0054   |state wait_one:
0055   |          GOTO   idle;
0056   |
0057   |state wait_twoa:
0058   |          if (reset_)  then wait_twob
0059   |          else   idle;
0060   |
0061   |state wait_twob:
0062   |          GOTO   idle;
0063   |
0064   |equations
0065   |          !rdy0_   =  reset_ & ((ahi1 & !strb0_) # !one_wait #
       |          !two_waitb) ;
0066   |
0067   |@page
0068   |“Test 1st level global arbitration logic
0069   |test_vectors
0070   |([h3,ahi1,ahi2,ahi3,strb0_, _strb_syn_ reset_] -> [outstate,  rdy0_])
0071   |[ c,  X,   X,    X,   X,        X,       L   ] -> [idle,       H   ];
0072   |[ c,  L,   H,    L,   L,        L,       H   ] -> [wait_one,   L   ];
0073   |[ c,  X,   X,    X,   X,        X,       L   ] -> [idle,       H   ];
0074   |[ c,  L,   L,    H,   L,        L,       H   ] -> [wait_twoa,  H   ];
0075   |[ c,  X,   X,    X,   X,        X,       L   ] -> [idle,       H   ];
0076   |[ c,  L,   L,    H,   L,        L,       H   ] -> [wait_twoa,  H   ];
0077   |[ c,  L,   L,    H,   L,        L,       H   ] -> [wait_twob,  L   ];
0078   |[ c,  X,   X,    X,   X,        X,       L   ] -> [idle,       H   ];
0079   |[ L,  H,   L,    L,   L,        L,       H   ] -> [idle,       L   ];
0080   |[ c,  H,   L,    L,   L,        L,       H   ] -> [idle,       L   ];
0081   |[ L,  L,   L,    L,   L,        L,       H   ] -> [idle,       H   ];
0082   |[ c,  L,   H,    L,   L,        L,       H   ] -> [wait_one,   L   ];
0083   |[ c,  X,   X,    X,   X,        X,       H   ] -> [idle,       H   ];
0084   |[ c,  L,   L,    H,   L,        L,       H   ] -> [wait_twoa,  H   ];
0085   |[ c,  L,   L,    H,   L,        L,       H   ] -> [wait_twob,  L   ];
0086   |[ c,  H,   L,    L,   L,        L,       H   ] -> [idle,       L   ];
0087   |[ c,  X,   X,    X,   H,        H,       H   ] -> [idle,       H   ];
0088   |[ c,  X,   X,    X,   H,        H,       H   ] -> [idle,       H   ];
0089   |end    ready_generation
```

4.5.6 Page Switching Techniques

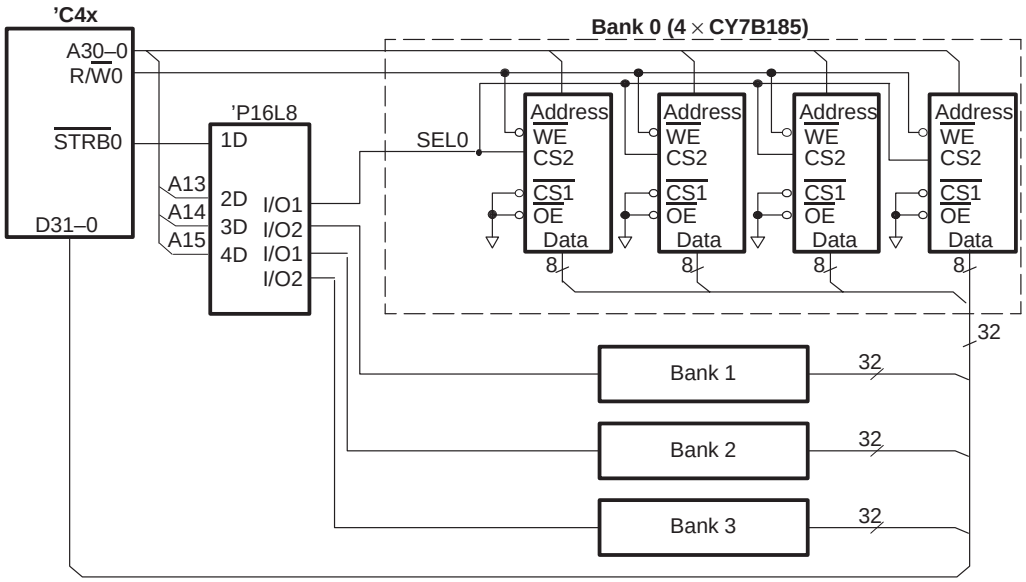
The ’C4x’s programmable page-switching feature can greatly ease system design when large amounts of memory or slow external peripheral devices are required. This feature provides a time period for disabling all device selects. During the interval, slow devices are allowed time to turn off before other devices have the opportunity to drive the data bus, thus avoiding bus contention.

When page switching is enabled, any time a portion of the high-order address lines changes, as defined by the contents of the $\overline{\text{STRB}}$0 and $\overline{\text{STRB}}$1 PAGE-SIZE fields (in the global and local memory interface control registers), the corresponding $\overline{\text{STRB}}$ and PAGE go high for one full H1 cycle. Provided that $\overline{\text{STRB}}$ is included in chip-select decodes, this causes all devices selected by that $\overline{\text{STRB}}$ to be disabled during this period. The next page of devices is not enabled until $\overline{\text{STRB}}$ and PAGE go low again.

If the high-order address lines remain constant during a read cycle, the memory access time with page switching is the same as memory access time without page switching. In addition, page switching is not required during writes, because these write cycles exhibit an inherent one-half H1 cycle setup of address information before $\overline{\text{STRB}}$ goes low. Thus, when you use page switching for read/write devices, a minimum of half of one H1 cycle of address setup is provided for all accesses outside a page boundary. Therefore, large amounts of memory can be implemented without wait states or extra hardware required for isolation between pages. Also, note that access time for cycles during page switching is the same as that of cycles without page switching, and, accordingly, full-speed accesses may still be accomplished within each page.

The circuit shown in Figure 4–8 illustrates page switching with the CY7B185 15-ns 8K $\times$ 8 BiCMOS static RAM. This circuit implements 32K 32-bit words of memory with full-speed zero wait-state accesses within each page.

Figure 4–8. Page Switching for the CY7B185



A 5-ns, 16L8 PLD decodes lines A15 – A13. These lines along with $\overline{\text{STRB}}$0 select each of the four pages in this circuit. With the PAGESIZE field of $\overline{\text{STRB}}$0 of the global memory interface control register set to 0Ch, the pages are selected on even 8K-word boundaries, starting at location zero in external memory space.

This circuit cannot be implemented without page switching, because the data output's turn-on and turn-off delays cause bus conflicts, and full-speed accesses do not allow enough time for chip-select decoding for the four pages. Here, the propagation delay of the 16L8 is involved only during page switches, where there is sufficient time between cycles to allow new chip-selects to be decoded.

The timing of this circuit for read operations with page switching is shown in Figure 4–9. When a page switch occurs, the page address on address lines A30 – A13 is updated during the extra H1 cycle while $\overline{\text{STRB}}$0 is high. Then, after chip-select decodes have stabilized and the previously selected page has disabled its outputs, $\overline{\text{STRB}}$ goes low for the next read cycle. Further accesses occur at full speed with the normal bus timings, as long as another page switch is not necessary. Write cycles do not require page switching, because of the inherent address setup provided in their timings.

This timing is summarized in Table 4–2.

Figure 4–9. Timing for Read Operations Using Bank Switching

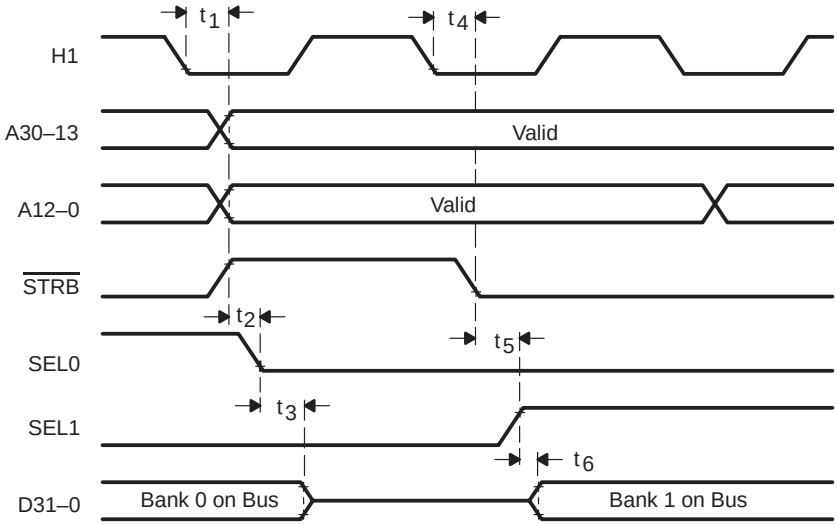


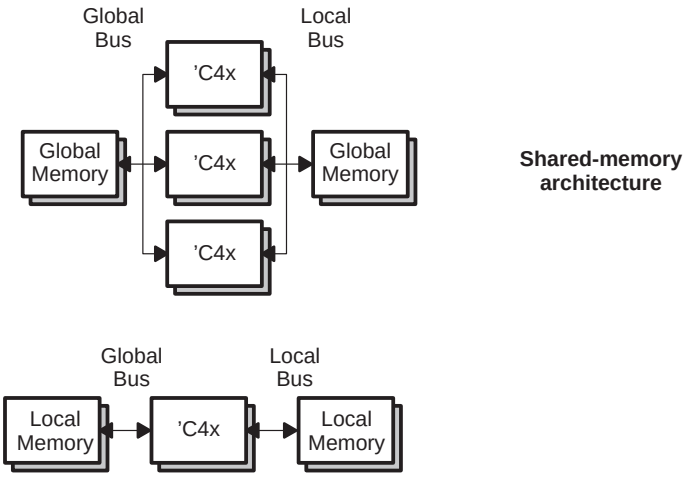
Table 4–2. Page Switching Interface Timing

Time Interval	Event	Time Period
t_1	H1 falling to address/$\overline{\text{STRB}}$ valid	7 ns
t_2	$\overline{\text{STRB}}$ to select delay	5 ns
t_3	Memory disable from select	8 ns
t_4	H1 falling to $\overline{\text{STRB}}$	7 ns
t_5	$\overline{\text{STRB}}$ to select delay	5 ns
t_6	Memory output enable delay	3 ns

4.6 Parallel Processing Through Shared Memory

The ’C4x’s two memory interfaces allow flexibility to design shared-memory interfaces for parallel processing. Many processors can be linked together in a wide variety of network configurations through these ports. In this section, Figure 4–10 illustrates ’C4x shared-memory networks that you can use to fulfill many signal processing system needs.

Figure 4–10. ’C4x Shared/Distributed-Memory Networks



4.6.1 Shared Global-Memory Interface

One of the most common multiprocessor configurations is the sharing of memory by all processors in a system. Shared memory is typically implemented by tying the processors’ data and address lines together. However, the shared memory interface must guarantee that no more than one processor is driving the shared bus at any one time; it must also allow all processors sharing the bus to have a chance to access shared resources.

The ’C4x supports shared memory multiprocessing with its identical global- and local-port interfaces. Both interfaces have four status output signals, (L)STAT3–0, which identify what type of access is beginning on the bus. These signals identify whether the ’C4x port is idle, a DMA read is occurring, a $\overline{\text{STRB1}}$ write is occurring, a $\overline{\text{LOCK}}$ed access to memory is pending, etc. The signals can be interpreted by the interface to issue single access or locked access bus requests to a shared bus arbiter.

The $\overline{\text{(L)CE}}$, $\overline{\text{(L)AE}}$, and $\overline{\text{(L)DE}}$ input signals support shared address control and data lines. When the signals are disabled (high), they put the port’s control

signals, address lines, and data lines, respectively, in the high-impedance state. These bus enable lines are asynchronous inputs to the ’C4x, which can quickly turn off bus drivers when another processor is accessing a shared resource. However, these signals asynchronously turn off the ’C4x’s local and global buses, without memory accesses being suspended. To ensure that data written is seen externally and data read is valid, you should use the external $\overline{\text{(L)RDY}}$ should be used for wait-state generation in shared memory designs. An $\overline{\text{(L)RDY}}$ signal should not be sent to the ’C4x until the processor has regained access to the bus ($\overline{\text{CE}}$, $\overline{\text{AE}}$, $\overline{\text{DE}}$ enabled) and has had enough time to complete its access. Hence, with bus enable and status signals, the ’C4x flexible bus interfaces easily implement high-speed shared bus configurations.

4.6.2 Shared-Memory Interface Design Example

For an example of a ’C4x shared-memory interface, see the *TMS320C4x Parallel Processing Development System Technical Reference* (SPRU075). In the example in that text, four ’C4x devices share SRAM with their global buses tied together. A bus arbitrator implemented as a programmable logic device provides a fair scheme for processor access to the shared bus. The design uses high-speed parts but employs a fully asynchronous handshake protocol that allows ’C4x devices of various speeds and also processors other than ’C4x devices to be added to this bus configuration.

The shared-memory interface in the PPDS works for ’C4x devices running at a speed of up to 32 MHz. For higher speeds, the arbitrator incorrectly takes away bus master privileges from a ’C4x between back-to-back reads to the same page (the page size is determined by the page size field in the global bus control register. The default page size for the PPDS global memory is 64K). If this occurs while two or more ’C4x devices are requesting the bus to perform write cycles, random shared memory locations can be corrupted.

To fix this problem for higher speeds, the *busenable_* signal of each ’C4x local interface can be used to generate *gmce0_* and *gmce1_* to prevent these signals from going low (active) if all the processors *busenable_* signals are high (inactive). The *busenable_* signal is shown in the PLD equations in the *Global Bus Interface Logic* section the of the *TMS320C4x Parallel Processing Development System Technical Reference*). The *gmce0* and *gmce1* signals are shown in the *Global Memory Control* section of the same book.

Chapter 5

Programming Tips

Programming style is highly personal and reflects each individual's preferences and experiences. The purpose of this chapter is not to impose any particular style. Instead, it emphasizes some of the features of the 'C4x that can help in producing faster and/or shorter programs. The tips in this chapter cover both C and assembly language programming.

Topic **Page**

5.1 Hints for Optimizing C Code

The ’C4x’s large register file, software stack, and large memory space easily support the ’C4x C Compiler. The C compiler translates standard ANSI C programs into assembly language source. It also increases the portability and decreases the porting time of applications.

The suggested methodology for developing your application follows five steps:

1) Write the application in C.
2) Debug the program.
3) Estimate if the program runs in real-time.
4) If the program does not run in real time:
   - Use the *–o2* or *–o3* option when compiling
   - Use registers to pass parameters (*–mr* compiling option)
   - Use inlining (*–x* compiling option)
   - Remove the *–g* option when compiling
   - Follow some of the efficient code generation tips listed below.
5) Identify places where most of the execution time is spent and optimize these areas by writing assembly language routines that implement the functions.

The efficiency of the code generated by the floating point compiler depends to a large extent on how well you take advantage of the compiler strengths described above when writing your C code. There are specific constructs that can vastly improve the compiler’s effectiveness:

- **Use register variables for often–used variables.** This is particularly true for pointer variables. Example 5–1 shows a code fragment that exchanges one object in memory with another.

Example 5–1. Exchanging Objects in Memory

```
do
  {
      temp  = *++src;
      *src  = *++dest;
      *dest = temp;
  }
  while (--n);
```

- **Pre-compute subexpressions**, especially array references in loops. Assign commonly used expressions to register variables where possible.

- **Use *++ to step through arrays**, rather than using an index to recalculate the address each time through a loop.

As an example of the previous 2 points, consider the loops in Example 5–2:

Example 5–2. Optimizing a Loop

```
/* loop 1 */
    main()
    {
       float a[10], b[10];
       int i;
       for (i = 0; i < 10; ++i)
          a[i] = (a[i] * 20) + b[i];
    }

/* loop 2 */
   main()
   {
     float a[10], b[10];
     int i;
     register float *p = a, *q = b;
     for (i = 0; i < 10; ++i)
     *p++ = (*p * 20) + *q++;
   }
```

Loop 1 executes in 19 cycles. Loop 2, which is the equivalent of loop 1, executes in 12 cycles.

- **Use structure assignments to copy blocks of data.** The compiler generates very efficient code for structure assignments, so nest objects within structures and use simple assignments to copy them.

- **Avoid large local frames and declare the most often used local variables first.** The compiler uses indirect addressing with an 8-bit offset to access local data. To access objects on the local frame with offsets greater than 255, the compiler must first load the offset into an index register. This causes 1 extra instruction and incurs 2 cycles of pipeline delay.

- **Avoid the large model.** The large model is inefficient because the compiler reloads the data-page pointer (DP) before each access to a global or static variable. If you have large array objects, use ”malloc()” to dynamically allocate them and access them via pointers rather than declaring them globally. Example 5–3 illustrates two methods for allocating large array objects:

Example 5–3.Allocating Large Array Objects

```
/* Bad Method */
   int a[100000]; /* BAD */
   ...
   a[i] = 10;


/* Good Method */

   int *a = (int *)malloc(100000); /* GOOD */
   ...
   a[i] = 10;
```

5.2 Hints for Optimizing Assembly-Language Code

Each program has particular requirements. Not all possible optimizations make sense in every case. The suggestions presented in this section can be used as a checklist of available software tools.

- **Use delayed branches**. Delayed branches execute in a single cycle; regular branches execute in four. The three instructions that follow the delayed branch are executed whether the branch is taken or not. If fewer than three instructions are used, use the delayed branch and append NOPs. Machine cycles (time) are still being saved.
- **Use delayed subroutine call and return**. Regular subroutine CALL and RETS execute in four cycles. You can implement a delayed subroutine call by using link and jump (LAJ) and delayed branches with R11 register mode (BUD R11) instructions. Both LAJ and BUD instructions execute in a single cycle. Guidelines for using the LAJ instruction are the same as for delayed branches.
- **Use the repeat single/block construct**. This method produces loops with no overhead. Nesting such constructs will not normally increase efficiency, so try to use the feature on the most often performed loop. The RPTBD is a single-cycle instruction, and the RPTS and RPTB are four-cycle instructions. RPTBD and delayed branches are used in similar ways. Note that RPTS is not interruptible, and the executed instruction is not refetched for execution. This frees the buses for operands.
- **Use parallel instructions**. You can have a multiply in parallel with an add (or subtract) and stores in parallel with any multiply or ALU operation. This increases the number of operations executed in a single cycle. For maximum efficiency, observe the addressing modes used in parallel instructions and arrange the data appropriately. You can have loads in parallel with any multiply or add (or subtract). The result of a multiply by one or an add of zero is the same as a load. Therefore, to implement parallel instructions with a data load, you can substitute a multiply or an add instruction, with one extra register containing a one or zero, in place of the load instruction.
- **Maximize the use of registers**. The registers are an efficient way to access scratch-pad memory. Extensive use of the register file facilitates the use of parallel instructions and helps avoid pipeline conflicts when you use register addressing.
- **Use the cache**. The cache speeds instruction fetches and enables simple-cycle access, even with slow external memory. The cache is transparent to the user, so make sure that it is enabled.

- **Use internal memory instead of external memory**. The internal memory (2K × 32 bits RAM and 4K × 32 bits ROM) is considerably faster to access than external memory. In a single cycle, two operands can be brought from internal memory. You can maximize performance if you use the DMA coprocessor in parallel with the CPU to transfer data you want to operate on to internal memory.

- **Avoid pipeline conflicts**. For time-critical operations, make sure that cycles are not missed because of pipeline conflicts. If there is no problem with program speed, ignore this suggestion.

- **Plan your linker command file in advance**. Memory allocation for code and data sections can have a big impact on your algorithm performance. One of the 'C4x's strengths is its sustained bandwidth achieved by having two external busses. By carefully dividing data and program between the two busses, you can minimize pipeline conflicts. You need to apply the same concept to minimize DMA/CPU access conflicts.

The above checklist is not exhaustive, and it does not address some features in detail. To learn how to exploit the full power of the 'C4x, carefully study its architecture, hardware configuration, and instruction set, which are all described in the *TMS320C4x User's Guide* (SPRU063).

Chapter 6

Applications-Oriented Operations

The ’C4x architecture and instruction set features facilitate the solution of numerically intensive problems. This chapter presents examples of applications that use these features, such as companding, filtering, matrix arithmetic, and fast Fourier transforms (FFT).

Topic **Page**

6.1 Companding

In telecommunications, one of the primary concerns is to conserve the channel bandwidth and, at the same time, to preserve high speech quality. This is achieved by quantizing the speech samples logarithmically. It has been demonstrated that an 8-bit logarithmic quantizer produces speech quality equivalent to that of a 13-bit uniform quantizer. The logarithmic quantization is achieved by companding (COMpress/exPANDing). Two international standards have been established for companding: the μ-law (used in the United States and Japan), and the A-law (used in Europe). Detailed descriptions of μ-law and A-law companding are presented in an application report on companding routines included in the book *Digital Signal Processing Applications with the TMS320 Family* (literature number SPRA012A).

During transmission, logarithmically compressed data in sign-magnitude form are transmitted along the communications channel. If any processing is necessary, these data should be expanded to a 14-bit (for μ-law) or 13-bit (for A-law) linear format. This operation occurs when data is received at the digital signal processor. After processing, and in order to continue transmission, the result is compressed back to 8-bit format and transmitted through the channel.

Example 6–1 and Example 6–2 show μ-law compression and expansion (such as linear to μ-law and μ-law to linear conversion), while Example 6–3 and Example 6–4 show A-law compression and expansion. For expansion, using a look-up table is an alternative approach. It trades memory space for speed of execution. Because the compressed data is 8 bits long, a table with 256 entries can be constructed to contain the expanded data. If the compressed data is stored in the register AR0, the following two instructions put the expanded data in register R0:

```
ADDI   @TABL,AR0 ; @TABL = BASE ADDRESS OF TABLE
LDI    *AR0,R0   ; PUT EXPANDED NUMBER IN R0
```

The same look-up table approach could be used for compression, but the required table length would then be 16,384 words for μ-law or 8,192 words for A-law. If this memory size is not acceptable, you should use the subroutines presented in Example 6–1 or Example 6–3.

Example 6–1.μ-Law Compression

```
*
*   TITLE μ-LAW COMPRESSION
*
*   SUBROUTINE MUCMPR
*
*   TYPICAL CALLING SEQUENCE:
*   LAJU   MUCMPR
*   LDI       v, R0
*   NOP    <----    can be other non-pipeline break
*   NOP    <----    instructions
*
*   ARGUMENT ASSIGNMENTS:
*
*   ARGUMENT  |     FUNCTION
*   --------- +----------------------------
*   R0        |     v = NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1
*   REGISTER  CONTAINING RESULT: R0
*
*
*   BENCHMARKS:     CYCLES: 14 (not including the BUD instruction)
*                   WORDS: 15 (not including the BUD instruction)
*
*
          .global   MUCMPR
*
MUCMPR    LSH3      –6,R0,R1        ;Save sign of number
          ABSI      R0,R0
          CMPI      1FDEH,R0        ;If R0>0x1FDE,
          LDIGT     1FDEH,R0        ;saturate the result
          ADDI      33,R0           ;Add bias
          FLOAT     R0              ;Normalize: (seg+5)0WXYZx...x
          MPYF      0.03125,R0      ;Adjust segment number by 2**(–5)
          LSH       1,R0            ;(seg)WXYZx...x
          PUSHF     R0
          POP       R0              ;Treat number as integer
          LSH       –20,R0          ;Right–justify
          BUD       R11             ;Delayed return
          AND       080H,R1         ;Set sign bit
          ADDI      R1,R0           ;R0 = compressed number
          NOT       R0              ;Reverse all bits for transmission
```

Example 6–2.μ-Law Expansion

```
*
*TITLE 'μ-LAW EXPANSION'
*
*   SUBROUTINE MUXPND
*
*   TYPICAL CALLING SEQUENCE:
*   LAJU   MUXPND
*   LDI    v, R0
*   NOP    <---- can be other non-pipeline-break
*   NOP    <---- instructions
*
*   ARGUMENT ASSIGNMENTS:
*
*   ARGUMENT  |     FUNCTION
*   --------- +------------------------------
*   R0        |     v = NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2
*   REGISTER  CONTAINING RESULT: R0
*
*   BENCHMARKS:     CYCLES: 11/10 (worst/best, not including subroutine overhead)
*                   WORDS: 11 (not including subroutine overhead)
*
*
          .global  MUXPND
*
MUXPND    NOT      R0,R0          ;Complement bits
          AND3     0FH,R0,R1      ;Isolate quantization bin
          LSH      1,R1
          ADDI     33,R1          ;Add bias to introduce 1xxxx1
          LSH3     -4,R0          ;Isolate segment code
          TSTB     08H,R0         ;Test sign
          BZD      R11            ;If positive, delayed return
          AND      7,R0
          LSH3     R0,R1,R0       ;Shift and put result in R0
          SUBI     33,R0          ;Subtract bias
          BUD      R11            ;Delayed return
          NEGI     R0             ;Negate if a negative number
          NOP
          NOP
```

Example 6–3.A-Law Compression

```
*
*   TITLE A-LAW COMPRESSION
*
*   SUBROUTINE ACMPR
*
*   TYPICAL CALLING SEQUENCE:
*   LAJ    ACMPR
*   LDI    v, R0
*   NOP    <---- can be other non-pipeline-break
*   NOP    <---- instructions
*
*   ARGUMENT ASSIGNMENTS:
*   ARGUMENT  |     FUNCTION
*   --------- +--------------------------
*   R0        |     v = NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1
*   REGISTER CONTAINING RESULT: R0
*
*
*   BENCHMARKS:     CYCLES: 16/10 (worst/best, not including subroutine overhead)
*                   WORDS: 16 (not including subroutine overhead)
*
          .global  ACMPR
*
ACMPR     LSH3     -5,R0,R1  ;Save sign of number
          ABSI     R0,R0
          CMPI     1FH,R0    ;If R0<0x20,
          BLED     END       ;do linear coding
          CMPI     0FFFH,R0  ;If R0>0xFFF,
          LDIGT    0FFFH,R0  ;saturate the result
          LSH      -1,R0     ;Eliminate rightmost bit
          FLOAT    R0        ;Normalize: (seg+3)0WXYZx...x
          MPYF     0.125,R0  ;Adjust segment number by 2**(-3)
          LSH      1,R0      ;(seg)WXYZx...x
          PUSHF    R0
          POP      R0        ;Treat number as integer
          LSH      -20,R0    ;Right-justify
END       BUD      R11       ;Delayed return
          AND      080H,R1   ;Set sign bit
          ADDI     R1,R0     ;R0 = compressed number
          XOR      0D5H,R0   ;Invert even bits for transmission
*
```

Example 6–4.A-Law Expansion

```
*
*   TITLE A-LAW EXPANSION
*
*   SUBROUTINE AXPND
*
*   TYPICAL CALLING SEQUENCE:
*   LAJU   AXPND
*   LDI    v, R0
*   NOP    <----    can be other non-pipeline-break
*   NOP    <----    instructions
*
*   ARGUMENT ASSIGNMENTS:
*
*          ARGUMENT |  FUNCTION
*          ---------+------------------------------
*          R0       |  v = NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2
*   REGISTER CONTAINING RESULT: R0
*
*   BENCHMARKS:  CYCLES: 15/13 (worst/best – not including subroutine overhead)
*                WORDS: 15 (not including subroutine overhead)
*
*
          .global  AXPND
*
AXPND     XOR      0D5H,R0,R2     ;Invert even bits
          ASH3     -4,R2,R0       ;Store for bit sign
          AND      7,R0           ;Isolate segment code
          BZD      SKIP1
          AND3     0FH,R2,R1      ;Isolate quantization bin
          LSH      1,R1
          ADDI     1,R1           ;Create 0xxxx1
          ADDI     32,R1          ;Or 1xxxx1
          SUBI     1,R0
SKIP1     LSH3     R0,R1,R0       ;Shift and put result in R0
          TSTB     80H,R2         ;Test sign bit
          BZAT     R11            ;If positive, delayed return and
                                  ;annul next three instructions
          NEGI     R0             ;Negate if a negative number
          NOP
          NOP
          BU       R11            ;Return
```

6.2 FIR, IIR, and Adaptive Filters

Digital filters are a common requirement for digital signal processing systems. There are two types of digital filters: finite impulse response (FIR) and infinite impulse response (IIR). Each of these types can have either fixed or adaptable coefficients. In this section, the fixed-coefficient filters are presented first, and then the adaptive filters are discussed.

6.2.1 FIR Filters

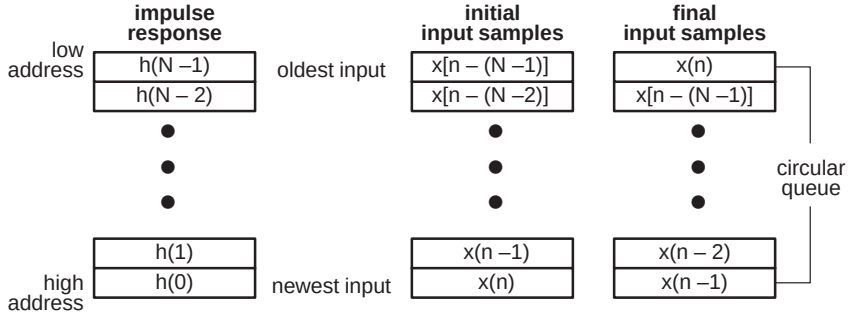
If the FIR filter has an impulse response h[0], h[1],..., h[N–1], and x[n] represents the input of the filter at time n, the output y[n] at time n is given by this equation:

$$y[n] = h[0]\,x[n] + h[1]\,x[n-1] + \ldots + h[N-1]\,x[n-(N-1)]$$

Two features of the ’C4x that facilitate the implementation of the FIR filters are parallel multiply/add operations and circular addressing. The first permits the performance of a multiplication and an addition in a single machine cycle, while the second makes a finite buffer of length N sufficient for the data x.

Figure 6–1 shows the arrangement of the memory locations to implement circular addressing, while Example 6–5 presents the ’C4x assembly code for an FIR filter.

Figure 6–1. Data Memory Organization for an FIR Filter



To set up circular addressing, initialize the block-size register BK to block length N. Also, the locations for signal x should start from a memory location whose address is a multiple of the smallest power of 2 that is greater than N. For instance, if N = 24, the first address for x should be a multiple of 32 (the lower 5 bits of the beginning address should be zero). To understand see *Circular Addressing* in the TMS320C4x User’s Guide.

In Example 6–5, 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 point to the position for the next input sample.

Example 6–5.FIR Filter

```
*
*   TITLE FIR FILTER
*
*
*   SUBROUTINE FIR
*
*   EQUATION: y(n) =   h(0) * x(n) + h(1) * x(n-1) +
*                      ... + h(N-1) * x(n-(N-1))
*
*   TYPICAL CALLING SEQUENCE:
*
*   LOAD   AR0
*   LAJU   FIR
*   LOAD   AR1
*   LOAD   RC
*   LOAD   BK
*
*
*   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
*
*
*   BENCHMARKS:     CYCLES: 3 + N (not including subroutine overhead)
*                   WORDS: 6 (not including subroutine overhead)
*
*
FIR       .global  FIR
*
          RPTBD    CONV                    ;Set up the repeat cycle
*   Initialize R0:
          MPYF3    *AR0++(1),*AR1++(1)%,R0 ;h(N–1) *x(n–(N–1)) –>R0
          LDF      0.0,R2                  ;Initialize R2
          NOP
*
*   FILTER (1 <= i < N)
*
CONV      MPYF3    *AR0++(1),*AR1++(1)%,R0 ;h(N–1–i)*x(n–(N–1–i))–>R0
||        ADDF3    R0,R2,R2                ;Multiply and add operation
*
          BUD      R11                     ;Delayed return
          ADDF     R0,R2,R0                ;Add last product
          NOP
          NOP
*
*   end
*
          .end
```

6.2.2 IIR Filters

The transfer function of the IIR filters has both poles and zeros. Its output depends on both the input and the past output. As a rule, the filters need less computation than an FIR with similar frequency response, but the filters 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. Example 6–6 and Example 6–7 show the implementation for one biquad and for any number of biquads, respectively.

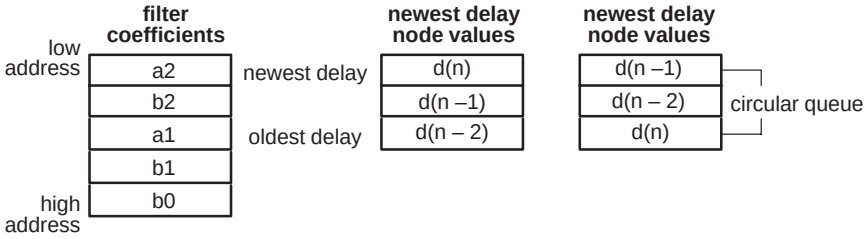
$$y[n] = a1\ y[n–1] + a2\ y[n–2] + b0\ x[n] + b1\ x[n–1] + b2\ x[n–2]$$

However, the following two equations are more convenient and have smaller storage requirements:

$$d[n] = a2\ d[n–2] + a1\ d[n–1] + x[n]$$
$$y[n] = b2\ d[n–2] + b1\ d[n–1] + b0\ d[n]$$

Figure 6–2 shows the memory organization for this two-equation approach to the implementation of a single biquad on the ’C4x.

Figure 6–2. Data Memory Organization for a Single Biquad



As in the case of FIR filters, the address for the start of the values d must be a multiple of 4; that is, the last two bits of the beginning address must be zero. The block-size register BK must be initialized to 3.

Example 6–6. IIR Filter (One Biquad)

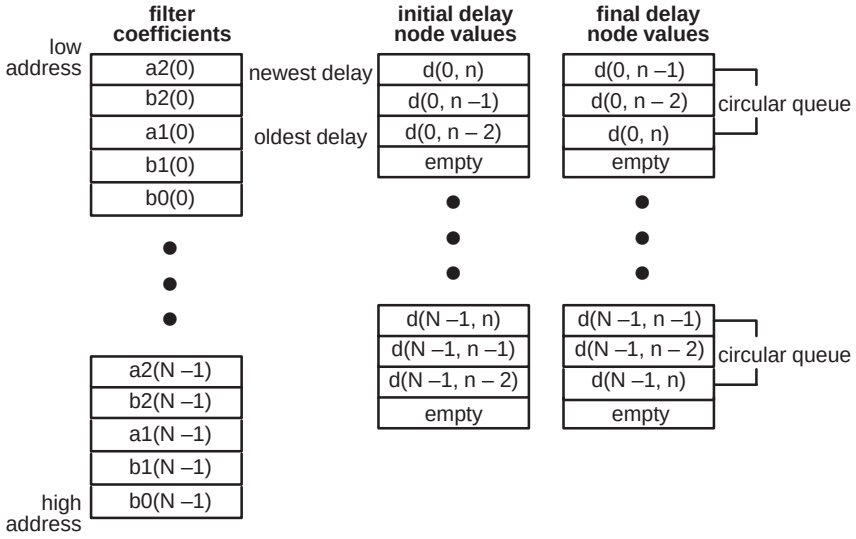
```
*   TITLE IIR FILTER
*
*   SUBROUTINE IIR1
*
*   IIR1 == IIR FILTER (ONE BIQUAD)
*
*   EQUATIONS: d(n) = a2 * d(n–2) + a1 * d(n–1) + x(n)
*                     y(n) = b2 * d(n–2) + b1 * d(n–1) + b0 * d(n)
*
*   OR                y(n) = a1*y(n–1) + a2*y(n–2) + b0*x(n) + b1*x(n–1)
*                     + b2*x(n–2)
*
*
*   TYPICAL CALLING SEQUENCE:
*
*   load   R2
*   LAJU   IIR1
*   load   AR0
*   load   AR1
*   load   BK
*
*
*   ARGUMENT ASSIGNMENTS:
*      ARGUMENT       |  FUNCTION
*     -------------- +-----------------------------------------
*       R2            |  INPUT SAMPLE X(N)
*       AR0           |  ADDRESS OF FILTER COEFFICIENTS (A2)
*       AR1           |  ADDRESS OF DELAY MODE VALUES (D(N–2))
*       BK            |  BK = 3
*
*   REGISTERS USED AS INPUT:         R2, AR0, AR1, BK
*   REGISTERS MODIFIED:              R0, R1, R2, AR0, AR1
*   REGISTER CONTAINING RESULT:      R0
*
*   BENCHMARKS:       CYCLES: 7 (not including subroutine overhead)
*                     WORDS: 7 (not including subroutine overhead)
*
*
           .global   IIR1
*
IIR1       MPYF3     *AR0,*AR1,R0                ;a2 * d(n–2) –> R0
           MPYF3     *++AR0(1),*AR1– –(1)%,R1    ;b2 * d(n–2) –> R1
*
           MPYF3     *++AR0(1),*AR1,R0           ;a1 * d(n–1) –> R0
||         ADDF3     R0,R2,R2                    ;a2*d(n–2)+x(n) –> R2
*
           MPYF3     *++AR0(1),*AR1– –(1)%,R0    ;b1 * d(n–1) –> R0
||         ADDF3     R0,R2,R2                    ;a1*d(n–1)+a2*d(n–2)
                                                 ;+x(n) –> R2
*
           BUD       R11                         ;Delayed return
*
           MPYF3     *++AR0(1),R2,R2             ;b0 * d(n) –> R2
||         STF       R2,*AR1++(1)%               ;Store d(n) and point to d(n–1)
*
           ADDF      R0,R2                       ;b1*d(n–1)+b0*d(n) –> R2
           ADDF      R1,R2,R0                    ;b2*d(n–2)+b1*d(n–1)
                                                 ;+b0*d(n) –> R0
*
*   end
*
           .end
```

Generally, the IIR filter contains N>1 biquads. The equations for its implementation are given by the following pseudo-C language code:

```
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–2] + b1[i] d[i,n–1] + b0[i] d[i,n]
}
y[n] = y[N–1,n]
```

Figure 6–3 shows the memory organization, and Example 6–7 shows the corresponding ’C4x assembly-language code.

Figure 6–3. Data Memory Organization for N Biquads



The block size register BK should be initialized to 3, and each set of d values (i.e., d[i,n], i = 0...N–1) should begin at an address that is a multiple of 4 (the last two bits zero), as stated in the case of a single biquad.

Example 6–7. IIR Filter (N > 1 Biquads)

```
*
*   TITLE IIR FILTER (N > BIQUADS)
*
*   SUBROUTINE IIR2
*
*   EQUATIONS: 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)
*
*   TYPICAL CALLING SEQUENCE:
*
*   load   R2
*   load   AR0
*   load   AR1
*   load   IR0
*   LAJU   IIR2
*   load   IR1
*   load   BK
*   load   RC
*
*   ARGUMENT ASSIGNMENT:
*    ARGUMENT |     FUNCTION*
*   ---------  +-----------------------------------------
*      R2     |     INPUT SAMPLE x(n)
*      ARO    |     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
*   REGISTERS CONTAINING RESULT: R0
*
*   BENCHMARKS:     CYCLES: 2 + 6N (not including subroutine overhead)
*                   WORDS: 15 (not including subroutine overhead)
*
*
           .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
*
           RPTBD    LOOP                   ;Set loop for 1 <= i < n
*
           MPYF3    *++AR0(1),*AR1,R0      ;a1(0) * D(0,n–1) –> R0
||         ADDF     R0,R2,R2               ;First sum term of d(0,n).
*
```

Example 6–7.IIR Filter (N > 1 Biquads) (Continued)

```
          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)
**  LOOP STARTS HERE
*
          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)
*                                           ;Pipeline hit on previous
                                            ;instruction
*
          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).
*
LOOP      MPYF3     *++AR0(1),R2,R2         ;b0(i) * d(i,n) –> R2
||        STF       R2, *AR1––(1)%          ;Store d(i,n) point to d(i,n–2)
*
*   FINAL SUMMATION
*
          ADDF3     R1,R2,R0                ;Second sum term of y(n–1,n
          BRD       R11                     ;Delayed return

*


          ADDF      R0,R2                   ;First sum term of y(n–1,n)
          NOP       *AR1––(IR1)             ;Return to first biquad
          NOP       *AR1––(1)%              ;Point to d(0,n–1)
*
*   end
*
          .end
```

6.2.3 Adaptive Filters (LMS Algorithm)

In some applications in digital signal processing, a filter must be adapted over time to keep track of changing conditions. The book *Theory and Design of Adaptive Filters* by Treichler, Johnson, and Larimore (Wiley-Interscience, 1987) presents the theory of adaptive filters. Although in theory, both FIR and IIR structures can be used as adaptive filters, the stability problems and the local optimum points that the IIR filters exhibit make them less attractive for such an application. Hence, until further research makes IIR filters a better choice, only the FIR filters are used in adaptive algorithms of practical applications.

In an adaptive FIR filter, the filtering equation takes this form:

$$y[n] = h[n,0]\,x[n] + h[n,1]x[n–1] +...+ h[n,N–1]x[n–(N–1)]$$

The filter coefficients are time-dependent. In a least-mean-squares (LMS) algorithm, the coefficients are updated by an equation in this form:

$$h[n+1,i] = h[n,1] + b\ x[n-i],\ i = 0, 1, ..., N-1$$

b is a constant for the computation. The updating of the filter coefficients can be interleaved with the computation of the filter output so that it takes 3 cycles per filter tap to do both. The updated coefficients are written over the old filter coefficients. Example 6–8 shows the implementation of an adaptive FIR filter on the ’C4x. The memory organization and the positioning of the data in memory should follow the same rules as the above FIR filter with fixed coefficients.

Example 6–8.Adaptive FIR Filter (LMS Algorithm)

```
*   TITLE ADAPTIVE FIR FILTER (LMS ALGORITHM)
*
*   SUBROUTINE LMS
*
*   LMS == LMS ADAPTIVE FILTER
*
*   EQUATIONS:      y(n) = h(n,0)*x(n) + h(n,1)*x(n–1) + ...
*                   + h(n,N–1)*x(n–(N–1))
*   FOR             (i = 0; i < N; i++) h(n+1,i) = h(n,i)
*                   + tmuerr * x(n–i)
*
*   TYPICAL CALLING SEQUENCE:
*
*   load   R4
*   load   AR0
*   LAJU   LMS
*   load   AR1
*   load   RC
*   load   BK
*
*
*   ARGUMENT ASSIGNMENTS:
*     ARGUMENT      |  FUNCTION
*   --------------- +---------------------------------
*      R4           |  SCALE FACTOR (2 * mu * err)
*      AR0          |  ADDRESS OF h(n,N–1)
*      AR1          |  ADDRESS OF x(n–(N–1))
*      RC           |  LENGTH OF FILTER – 2 (N–2)
*      BK           |  LENGTH OF FILTER (N)*
*   REGISTERS USED AS INPUT: R4, AR0, AR1, RC, BK
*   REGISTERS MODIFIED: R0, R1, R2, AR0, AR1, RC
*   REGISTER CONTAINING RESULT: R0
*
*   BENCHMARKS:     CYCLES:     4 + 3N (not including subroutine overhead)
*                   PROGRAM SIZE: 9 words (not including subroutine overhead)
*
*   SETUP (i = 0)
*
          .global  LMS
LMS       RPTBD    LOOP                  ;Setup the delayed repeat block
*   Initialize R0:
          MPYF3    *AR0,*AR1,R0          ;h(n,N–1) * x(n–(N–1)) –> R0
||        SUBF3    R2, R2, R2            ;Initialize R2
*
*   Initialize R1:
          MPYF3    *AR1++(1)%,R4,R1      ;x(n–(N–1)) * tmuerr –> R1
          ADDF3    *AR0++(1),R1,R1       ;h(n,N–1) + x(n–(N–1)) *
                                         ;tmuerr –> R1
*
*   FILTER AND UPDATE (1 <= I < N)
*   Filter:
          MPYF3    *AR0– –(1),*AR1,R0    ;h(n,N–1–i) * x(n–(N–1–i)) –> R0
||        ADDF3    R0,R2,R2              ;Multiply and add operation.
*
*   UPDATE:
          MPYF3    *AR1++(1)%,R4,R1      ;x(n,N–(N–1–i)) * tmuerr –> R1
||        STF      R1,*AR0++(1)          ;R1 –> h(n+1,N–1–(i–1))
*
```

Example 6–8.Adaptive FIR Filter (LMS Algorithm) (Continued)

```
LOOP      ADDF3    *AR0++(1),R1,R1        ;h(n,N–1–i) + x(n–(N–1–i))
                                          ;*tmuerr –> R1
*
          BUD      R11                    ;Delayed return
*
          ADDF3    R0,R2,R0               ;Add last product.
          STF      R1,*–AR0(1)            ;h(n,0) + x(n)* tmuerr –>
                                          ;h(n+1 , 0)
          NOP
*
*   end
*
          .end
```

6.3 Lattice Filters

The lattice form is an alternative way of implementing digital filters; it has applications in speech processing, spectral estimation, and other areas. In this discussion, the notation and terminology from speech processing applications are used.

If H(z) is the transfer function of a digital filter that has only poles, A(z) = 1/H(z) will be a filter having only zeros, and it will be called the inverse filter. The inverse lattice filter is shown in Figure 6–4. These equations describe the filter in mathematical terms:

$f(i,n) = f(i–1,n) + k(i)\ b(i–1,n–1)$
$b(i,n) = b(i–1,n–1) + k(i)\ f(i–1,n)$

Initial conditions:

$f(0,n) = b(0,n) = x(n)$

Final conditions:

$y(n) = f(p,n)$

In the above equation, f(i,n) is the forward error, b(i,n) is the backward error, k(i) is the i-h reflection coefficient, x(n) is the input, and y(n) is the output signal. The order of the filter (that is, the number of stages) is p. In the linear predictive coding (LPC) method of speech processing, the inverse lattice filter is used during analysis, and the (forward) lattice filter is used during speech synthesis.

Figure 6–4. Structure of the Inverse Lattice Filter

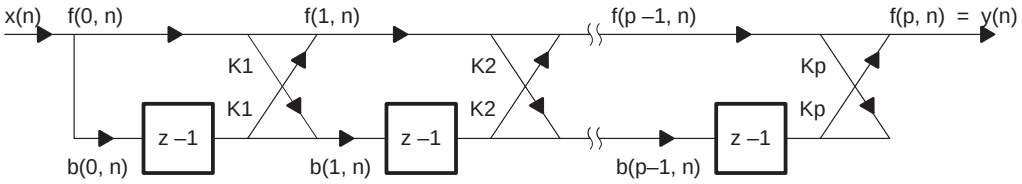
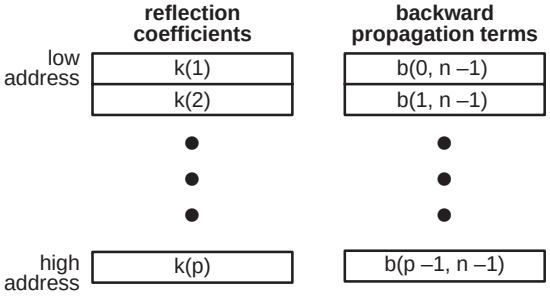


Figure 6–5 shows the data memory organization of the inverse lattice filter on the ’C40.

Figure 6–5. Data Memory Organization for Inverse Lattice Filters



Example 6–9. Inverse Lattice Filter

```
*   TITLE INVERSE LATTICE FILTER
*
*   SUBROUTINE LATINV
*
*   LATINV == LATTICE FILTER (LPC INVERSE FILTER – ANALYSIS)
*
*   TYPICAL CALLING SEQUENCE:
*
*   load   R2
*   LAJU   LATINV
*   load   AR0
*   load   AR1
*   load   RC
*
*
*   ARGUMENT ASSIGNMENTS:
*    ARGUMENT |     FUNCTION
*   --------- +-------------------------------------------------
*      R2     |     f(0,n) = x(n)
*      AR0    |     ADDRESS OF FILTER COEFFICIENTS (k(1))
*      AR1    |     ADDRESS OF BACKWARD PROPAGATION VALUES (b(0,n–1))
*      RC     |     RC = p – 2
*
*   REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*   REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*   REGISTER CONTAINING RESULT: R2 (f(p,n))
*
    BENCHMARKS:     CYCLES: 3 + 3p (not including subroutine overhead)
                    PROGRAM SIZE: 9 WORDS (not including subroutine overhead)
*
*
*
*
           .global LATINV
*
*   i = 1
*
LATINV     RPTBD     LOOP                    ;Setup the delayed repeat block loop
           MPYF3     *AR0,*AR1,R0            ;k(1) * b(0,n–1) –> R0
                                             ;Assume f(0,n) –> R2.
           LDF       R2,R3                   ;Put b(0,n) = f(0,n) –> R3.
           MPYF3     *AR0++(1),R2,R1         ;k(1) * f(0,n) –> R1
```

Example 6–9.Inverse Lattice Filter (Continued)

```
*
*   2 <= i <= p  (Repeat block loop start here)
*
          MPYF3     *AR0,*++AR1(1),R0       ;k(i) * b(i–1,n–1) –> R0
||        ADDF3     R2,R0,R2                ;f(i–1–1,n) + k(i–1) *b(i–1–1,n–1)
                                            ;= f(i–1,n) –> R2
*
                                            ;b(i–1–1,n–1) + k(i–1)*f(i–1–1,n)
          ADDF3     *–AR1(1),R1,R3          ;= b(i–1,n) –> R3
||        STF       R3,*–AR1(1)             ;b(i–1–1,n) –> b(i–1–1,n–1)
*
LOOP      MPYF3     *AR0++(1),R2,R1         ;k(i) * f(i–1,n) –> R1
*
*   I = P + 1 (CLEANUP)
*
          BUD       R11                     ;Delayed return
          ADDF3     R2,R0,R2                ;f(p–1,n) + k(p)*b(p–1,n–1)
                                            ;= f(p,n) –> R2
*
          ADDF3     *AR1,R1,R3              ;b(p–1,n–1) + k(p)*f(p–1,n)
                                            ;= b(p,n) –> R3
||        STF       R3,*AR1                 ;b(p–1,n) –> b(p–1,n–1)
          NOP
*
*   end
*
          .end
```

The structure of the forward lattice filter, shown in Figure 6–6, is similar to that of the inverse filter (also shown in the figure). These corresponding equations describe the lattice filter:

f(i–1,n) = f(i,n) – k(i) b(i–1,n–1)
b(i,n) = b(i–1,n–1) + k(i) f(i–1,n)

Initial conditions:

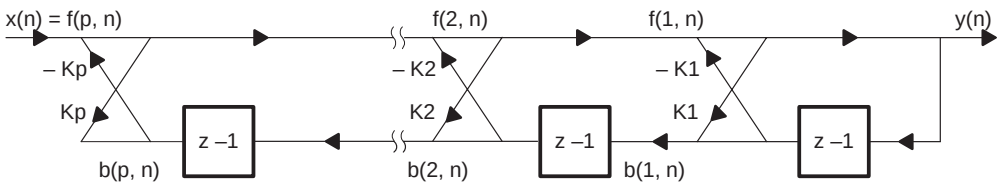
f(p,n) = x(n), b(i,n–1) = 0 for i = 1, ..., p

Final conditions:

y(n) = f(0,n).

The data memory organization is identical to that of the inverse filter shown in Figure 6–5. Example 6–10 shows the implementation of the lattice filter on the ’C4x.

Figure 6–6. Structure of the Forward Lattice Filter



Example 6–10. Lattice Filter

```
*   TITLE LATTICE FILTER
*
*   SUBROUTINE LATTICE
*
*         LAJU     LATTICE
*         LOAD     AR0
*         LOAD     AR1
*         LOA      RC
*
*   ARGUMENT ASSIGNMENTS:
*    ARGUMENT |     FUNCTION
*   --------- +-------------------------------------
*      R2     |     F(P,N) = E(N) = EXCITATION
*      AR0    |     ADDRESS OF FILTER COEFFICIENTS (K(P))
*      AR1    |     ADDRESS OF BACKWARD PROPAGATION
*             |     VALUES (B(P–1,N–1))
*      RC     |     RC = P – 2
*
*   REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*   REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*   REGISTER CONTAINING RESULT: R2 (f(0,n))
*
*   BENCHMARKS:     CYCLES:         1 + 5P (not including subroutine overhead)
*                   PROGRAM SIZE: 11 words (not including subroutine overhead)
*
          .global  LATTICE
*
LATTICE   RPTBD    LOOP                  ;Setup the delayed repeat block loop
          MPYF3    *AR0,*AR1,R0          ;K(P) * B(P–1,N–1) –> R0
          SUBF3    R0,R2,R2              ;Assume F(P,N) –> R2
          NOP                            ;F(P,N)–K(P)*B(P–1,N–1)
                                         ;= F(P–1,N) –> R2
*
*   2 <= I <= P  (Repeat block loop start here)
*
          MPYF3    *AR0,R2,R1            ;K(I) * F(I–1,N) –> R1
          MPYF3    *––AR0(1),*–AR1(1),R0 ;K(I–1) *
                                         ;B(I–1–1,N–1) –> R0
          ADDF3    *AR1––(1),R1,R3       ;B(I–1,N–1) + K(I)*F(I–1,N)
*                                        ;= B(I,N) –> R3
          STF      R3,*+AR1(2)           ;B(I,N) –> B(I,N–1)
LOOP      SUBF3    R0,R2,R2              ;F(I–1,N)–K(I–1)
                                         ;*B(I–1–1,N–1)
*                                        ;= F(I–1–1,N) –> R2
*
*   I = 1 (CLEANUP)
*
          BUD R11                        ;Delayed return
          MPYF     *AR0,R2,R1            ;K(1) * F(0,N) –> R1
          ADDF3    *AR1,R1,R3            ;B(0,N–1) + K(1)*F(0,N)
*                                        ;= B(1,N) –> R3
          STF      R3,*+AR1(1)           ;B(1,N) –> B(1,N–1)
||        STF      R2,*AR1               ;F(0,N) –> B(0,N–1)
*
*   end
*
          .end
```

6.4 Matrix-Vector Multiplication

In matrix-vector multiplication, a K × N matrix of elements m(i,j), having K rows and N columns, is multiplied by an N × 1 vector to produce a K × 1 result. The multiplier vector has elements v(j), and the product vector has elements p(i). Each one of the product-vector elements is computed by the following expression:

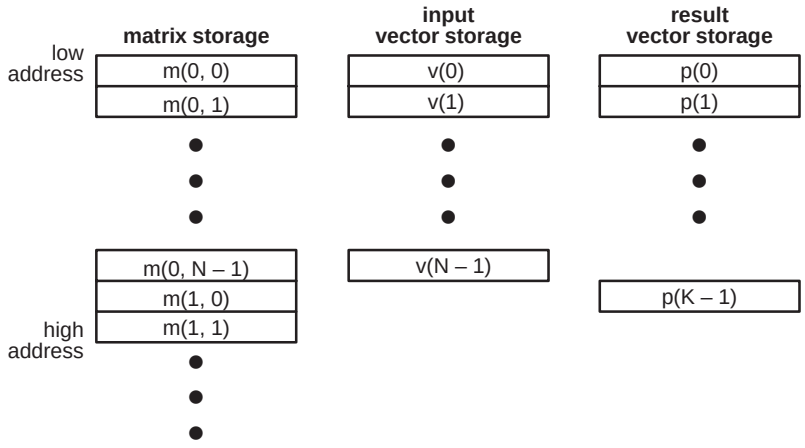
$$p(i) = m(i,0)\,v(0) + m(i,1)\,v(1) + \ldots + m(i,N-1)\,v(N-1) \quad i = 0,1,\ldots,K-1$$

This is essentially a dot product, and the matrix-vector multiplication contains, as a special case, the dot product presented in Example 2–1 on page 2-3 and Example 2–2 on page 2-5. In pseudo-C format, the computation of the matrix multiplication is expressed by

```
for (i = 0; i < K; i++) {
        p(i) = 0
        for (j = 0; j < N; j++)
                p(i) = p(i) + m(i,j) * v(j)
}
```

Figure 6–7 shows the data memory organization for matrix-vector multiplication, and Example 6–11 shows the 'C4x assembly code that implements it. Note that in Example 6–11, K (number of rows) should be greater than 0, and N (number of columns) should be greater than 1.

Figure 6–7. Data Memory Organization for Matrix-Vector Multiplication



Example 6–11. Matrix Times a Vector Multiplication

```
*
*       TITLE MATRIX TIMES A VECTOR MULTIPLICATION
*
*       SUBROUTINE MAT
*
*       MAT == MATRIX TIMES A VECTOR OPERATION
*
*       TYPICAL CALLING SEQUENCE:
*
*       load    AR0
*       load    AR1
*       load    AR2
*       load    AR3
*       load    R1
*       CALL    MAT
*
*       ARGUMENT ASSIGNMENTS:
*
*       ARGUMENT|        FUNCTION
*       --------------   +-------------------------------
*               AR0      |      ADDRESS OF M(0,0)
*               AR1      |      ADDRESS OF V(0)
*               AR2      |      ADDRESS OF P(0)
*               AR3      |      NUMBER OF ROWS – 1 (K–1)
*               RC       |      NUMBER OF COLUMNS – 2 (N–2)
*
*       REGISTERS USED AS INPUT: AR0, AR1, AR2, AR3, RC
*       REGISTERS MODIFIED: R0, R2, AR0, AR1, AR2, AR3, IR0, RC
*
*
*       MATRIX  -VECTOR BENCHMARKS:      CYCLES: 1 + 7K + KN = 1 + K (N + 7)
*                                        (not including subroutine overhead)
*                                        PROGRAM SIZE: 10 words (not including subroutine
                                         overhead)
*
*
        .global  MAT
*
*       SETUP
*
MAT     ADDI3   RC,2,IR0                 ;IR0 = N
*
*       FOR (i = 0; i < K; i++) LOOP OVER THE ROWS.
*
ROWS    RPTBD   DOT                      ;Setup multiply a row by a column
                                         ;Set loop counter
        LDF     0.0,R2                   ;Initialize R2
        MPYF3   *AR0++(1),*AR1++(1),R0   ;m(i,0) * v(0) –> R0
        NOP
*       FOR (j = 1; j < N; j++) DO DOT PRODUCT OVER COLUMNS
*
DOT     MPYF3   *AR0++(1),*AR1++(1),R0   ;m(i,j) * v(j) –> R0
||      ADDF3   R0,R2,R2                 ;m(i,j–1) * v(j–1) +
                                         ;R2 –> R2
*
        DBD     AR3,ROWS                 ;counts the number of rows left
*
```

Example 6–11. Matrix Times a Vector Multiplication (Continued)

```
*
        ADDF    R0,R2                   ;last accumulate
        STF     R2,*AR2++(1)            ;result –> p(i)
        NOP     *– –AR1(IR0)            ;set AR1 to point to v(0)
*       !!! DELAYED BRANCH HAPPENS HERE !!!
*
*       RETURN SEQUENCE
*
        RETS                            ;return
*
*       end
*
        .end
```

6.5 Fast Fourier Transforms (FFTs)

Fourier transforms are an important tool often used in digital signal processing systems. The transform converts information from the time domain to the frequency domain. The inverse Fourier transform converts information back to the time domain from the frequency domain. Implementation of Fourier transforms that are computationally efficient are known as fast Fourier transforms (FFTs). The theory of FFTs can be found in books such as *DFT/FFT and Convolution Algorithms* by C.S. Burrus and T.W. Parks (John Wiley, 1985) and *Digital Signal Processing Applications With the TMS320 Family*.

'C4x features that increase efficient implementation of numerically intensive algorithms are particularly well-suited for FFTs. The high speed of the 'C4x (40-ns cycle time) makes the implementation of real-time algorithms easier, while the floating-point capability eliminates the problems associated with dynamic range. The powerful indexing scheme in indirect addressing facilitates the access of FFT butterfly legs that have different spans. The repeat block implemented by the RPTB or RPTBD instruction reduces the looping overhead in algorithms heavily dependent on loops (such as the FFTs). This gives the efficiency of in-line coding with the form of a loop. Since the output of the FFT is in scrambled (bit-reversed) order when the input is in regular order, it must be restored to the proper order. This rearrangement does not require extra cycles. The device has a special form of indirect addressing (bit-reversed addressing mode) that can be used when the FFT output is needed.

The 'C4x can implement the bit-reversed addressing mode on either the CPU or DMA. This mode makes it possible to access the FFT output in the proper order. If the DMA transfer with bit-reversed addressing mode is used, there is no overhead for data input and output.

There are several types of FFT examples in this section:

- Radix-2 and radix-4 algorithms, depending on the size of the FFT butterfly
- Decimation in time or frequency (DIT or DIF)
- Complex or real FFTs
- FFTs of different lengths, etc.

The following C-callable FFT code examples are provided in this section:

- Complex radix-2 DIF FFT: subsection 6.5.1
- Complex radix-4 DIF FFT: subsection 6.5.2
- Faster Complex radix-2 DIT FFT: subsection 6.5.3
- Real radix-2 DIF FFT: subsection 6.5.4

Code for these different FFTs can be found in the DSP Bulletin Board Service (under the filename: C40FFT.EXE). This file includes code, input data and sine table examples, and batch files for compiling and linking. For instructions on how to access the BBS, see subsection 10.1.3, *The Bulletin Board Service (BBS)*. To use these FFT codes, you need to perform two steps:

- Provide a sine table in the format required by the program. This sine table is FFT size specific, with the exception of the sine table required for Complex radix-2 DIT and the real radix-2 DIF FFT programs (as noted in Example 6–18)
- Align the input data buffer on a n+1 memory boundary, i.e the n+1 LSBs of the input buffer base address must be zero. (n = log FFT_SIZE).

For most applications, the ’C4x quickly executes FFT lengths of up to 1024 points (complex) or 2048 points (real) because it can do so almost entirely in on-chip memory.

For FFTs larger than 1024 (complex), see the application report, *Parallel 1-D FFT Implementation with the TMS320C4x DSPs,* in the book *Parallel Processing Applications with the TMS320C4x DSP* (literature number SPRA031). This application note covers unprocessed partitioned FFT implementation for large FFTs. The source code is also available on the TI DSP Bulletin Board (under the filename: C40PFFT.EXE).

6.5.1 Complex Radix-2 DIF FFT

Example 6–12 shows a simple implementation of a complex radix-2, DIF FFT on the ’C4x. The code is generic and can be used with any length number. However, for the complete implementation of an FFT, a table of twiddle factors (sines/cosines) is needed, and this table depends on the size of the transform. To retain the generic form of Example 6–12, the table with the twiddle factors (containing 1-1/4 complete cycles of a sine) is presented separately in Example 6–13 for the case of a 64-point FFT. A full cycle of a sine should have a number of points equal to the FFT size. If the table with the twiddle factors and the FFT code are kept in separate files, they should be connected at link time.

Example 6–12. Complex Radix-2 DIF FFT

```
*******************************************************************************
*
*  FILENAME     : CR2DIF.ASM
*  DESCRIPTION  : COMPLEX, RADIX–2 DIF FFT FOR TMS320C40 (C callable)
*  DATE         : 6/29/93
*  VERSION      : 4.0
*
*******************************************************************************
*
*  VERSION        DATE         COMMENTS
*  -------        ----         --------
*   1.0           10/87        PANNOS PAPAMICHALIS (TI Houston) Original Release
*   2.0           1/91         DANIEL CHEN (TI Houston): C40 porting
*   3.0           7/1/92       ROSEMARIE PIEDRA (TI Houston): made it C–callable
*   4.0           6/29/93      ROSEMARIE PIEDRA (TI Houston): added support for
*                              in-place bit reversing
*
*******************************************************************************
*
*  SYNOPSIS: int  cr2dif(SOURCE_ADDR,FFT_SIZE,LOGFFT,DST_ADDR)
*                             ar2        r2       r3    rc
*
*            float   *SOURCE_ADDR    ; input address
*            int     FFT_SIZE        ;64, 128, 256, 512, 1024, ...
*            int     LOGFFT          ;log (base 2) of FFT_SIZE
*            float   *DST_ADDR       ;destination address
*
*   – The computation is done in–place.
*   – Sections to be allocated in linker command file: .ffttxt : FFT code
*                                                      .fftdat : FFT data
*  If SOURCE_ADDR=DST_ADDR, then in-place bit reversing is performed
*
*******************************************************************************
*
*  DESCRIPTION:
*
*  Generic program for a radix–2 DIF FFT computation using the TMS320C4x family.
*  The computation is done in–place and the result is bit–reversed. The program
*   is from the Burrus and Parks book, p. 111. The input data array is 2*FFT_SIZE–
*  long with real and imaginary data in consecutive memory locations: Re–Im–Re–Im
*
*  The twiddle factors are supplied in a table put in a section with a global
*  label _SINE pointing to the beginning of the table. This data is included in a
*  separate file to preserve the generic nature of the program. The sine table
*  size  is (5*FFT_SIZE)/4.
*
*  Note: Sections needed in the linker command file: .ffttxt : FFT code
*                                                    .fftdat : FFT data
*
```

Example 6–12. Complex Radix-2 DIF FFT (Continued)

```
*******************************************************************************
*                              +
*   AR + j AI ------------------------------------- AR' + j AI'
*                        \         / +
*                          \     /
*                            \ /
*                            / \
*                          /     \
*                        /         \ +
*   BR + j BI ----------------------- COS – j SIN ---- BR' + j BI'
*                              –
*
*   AR'= AR + BR
*   AI'= AI + BI
*   BR'= (AR–BR)*COS + (AI–BI)*SIN
*   BI'= (AI–BI)*COS – (AR–BR)*SIN
*
*******************************************************************************
*
*
        .globl   _SINE              ;Address of sine/cosine table
        .globl   _cr2dif            ;Entry point for execution
        .globl   STARTB,ENDB        ;starting/ending point for benchmarks
        .sect    ".fftdat"
SINTAB  .word    _SINE
OUTPUTP .space   1
FFTSIZE .space   1
        .sect   ".ffttxt"

_cr2dif:
        LDI     SP,AR0
        PUSH    DP
        PUSH    R4                  ;Save dedicated registers
        PUSH    R5
        PUSH    R6                  ;lower 32 bits
        PUSHF   R6                  ;upper 32 bits
        PUSH    AR4
        PUSH    AR5
        PUSH    AR6
        PUSH    R8
        LDP     SINTAB
       .if      .REGPARM == 0       ;stack is used for parameter passing
        LDI     *–AR0(1),AR2        ;points input data
        LDI     *–AR0(2),R10        ;R10=N
        LDI     *–AR0(3),R9         ;R9 holds the remain stage number
        LDI     *–AR0(4),RC         ;points where FFT result should move to
        .else                       ;registers are used for parameter passing
        LDI     R2,R10
        LDI     R3,R9
       .endif
        STI     RC, @OUTPUTP
        STI     R10,@FFTSIZE
```

Example 6–12. Complex Radix-2 DIF FFT (Continued)

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

Example 6–12. Complex Radix-2 DIF FFT (Continued)

```
||      STF     R4,*+AR6            ;Y(L)=R1*COS-R2*SIN
        CMPI    R10,AR1
        BNEAF   INLOP               ;Loop back to the inner loop
        ADDI    AR5,AR4             ;IA=IA+IE;AR4 points to cosine
        ADDI    AR2,AR1,AR0         ;(X(I),Y(I)) pointer
        SUBI    1,R8,RC
        LSH     1,R8                ;Increment loop counter for next time
        BRD     LOOP                ;Next FFT stage (delayed)
        LSH     1,AR5               ;IE=2*IE
        LDI     R10,IR0             ;N1=N2
        SUBI3   1,R8,RC
ENDB:
*
*
*************************************************************************
*------------- BITREVERSAL ---------------------------------------------*
* This bit-reversal section assume input and output in Re-Im-Re-Im format *
*************************************************************************
        cmpi    @OUTPUTP,ar2
        beqd    INPLACE
        nop
        ldi     @FFTSIZE,ir0        ;ir0 = FFT_SIZE
        subi    2,ir0,rc            ;rc  = FFT_SIZE-2
                                    ;SRC different from DST
                                    ;ar2 = SRC_ADDR
        rptbd   BITRV
        ldi     2,ir1               ;ir1 = 2
        ldi     @OUTPUTP,ar1        ;ar1 = DST_ADDR
        ldf     *+ar2(1),r0         ;read first Im value
        ldf     *ar2++(ir0)b,r1
||      stf     r0,*+ar1(1)
BITRV   ldf     *+ar2(1),r0
||      stf     r1,*ar1++(ir1)
        bud     END
        ldf     *ar2++(ir0)b,r1
||      stf     r0,*+ar1(1)
        nop
        stf     r1,*ar1
INPLACE
        rptbd   BITRV2              ;in place bit reversing
        ldi     ar2,ar1
        nop     *++ar1(2)
        nop     *ar2++(ir0)b
        cmpi    ar1,ar2
        bgeat   CONT
        ldf     *ar1,r0
||      ldf     *ar2,r1
        stf     r0,*ar2
||      stf     r1,*ar1
        ldf     *+ar1(1),r0
||      ldf     *+ar2(1),r1
```

Example 6–12. Complex Radix-2 DIF FFT (Continued)

```
       stf      r0,*+ar2(1)
||     stf      r1,*+ar1(1)
CONT   nop      *++ar1(2)
BITRV2 nop      *ar2++(ir0)b
;
;Return to C environment.
;
END:   POP      R8
       POP      AR6                    ;Restore the register values and return
       POP      AR5
       POP      AR4
       POPF     R6
       POP      R6
       POP      R5
       POP      R4
       POP      DP
       RETS
       .end
```

Example 6–13. Table With Twiddle Factors for a 64-Point FFT

```
*************************************************************
*
*  TITLE  TABLE WITH TWIDDLE FACTORS FOR A 64–POINT FFT
*
*  FILE TO BE LINKED WITH THE SOURCE CODE FOR A 64–POINT,
*  RADIX–2 DIF COMPLEX FFT OR A RADIX–4 DIF COMPLEX FFT.
*
* SINE TABLE LENGTH = 5*FFTSIZE/4
*
*************************************************************
   .globl    _SINE
   .sect     ".sintab"
_SINE
   .float    0.000000
   .float    0.098017
   .float    0.195090
   .float    0.290285
   .float    0.382683
   .float    0.471397
   .float    0.555570
   .float    0.634393
   .float    0.707107
   .float    0.773010
   .float    0.831470
   .float    0.881921
   .float    0.923880
   .float    0.956940
   .float    0.980785
   .float    0.995185
_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
```

Example 6–13. Table With Twiddle Factors for a 64-Point FFT (Continued)

```
    .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
```

6.5.2 Complex Radix-4 DIF FFT

The radix-2 algorithm has tutorial value because it is relatively easy to understand how the FFT algorithm functions. However, radix-4 implementations can increase the speed of the execution by reducing the overall arithmetic required. Example 6–14 shows the generic implementation of a complex, DIF FFT in radix-4. A companion table like the one Example 6–13 should be used to provide the twiddle factor.

Example 6–14. Complex Radix-4 DIF FFT

```
*******************************************************************************
*
*  FILENAME     : CR4DIF.ASM
*  DESCRIPTION  : COMPLEX, RADIX–4 DIF FFT FOR TMS320C40 (C callable)
*  DATE         : 6/29/93
*  VERSION      : 4.0
*
*******************************************************************************
*
*  VERSION        DATE         COMMENTS
*  -------        ----         --------
*   1.0           10/87        PANNOS PAPAMICHALIS (TI Houston)
*                              Original Release
*   2.0           1/91         DANIEL CHEN (TI Houston): C40 porting
*   3.0           7/1/91       ROSEMARIE PIEDRA (TI Houston): made it C–callable
*   4.0           6/29/93      ROSEMARIE PIEDRA (TI Houston):added support for
*                              in–place bit reversing.
*
*******************************************************************************
*
*  SYNOPSIS: int  cr4dif(SOURCE_ADDR,FFT_SIZE,LOGFFT,DST_ADDR)
*                             ar2        r2       r3    rc
*
*            float   *SOURCE_ADDR    ;input address
*            int     FFT_SIZE        ;64, 256, 1024, ...
*            int     LOGFFT          ;log (base 4) of FFT_SIZE
*            float   *DST_ADDR       ;destination address
*
*   – The computation is done in–place.
*   – Sections to be allocated in linker command file: .ffttxt : FFT code
*                                                      .fftdat : FFT data
*  If SOURCE_ADDR=DST_ADDR, then in-place bit reversing is performed
*
*******************************************************************************
*
*  DESCRIPTION:
*
*  Generic program for a radix–4 DIF FFT computation using the TMS320C4x
*  family. The computation is done in–place and the result is bit–reversed.
*  The program is taken from the Burrus and Parks book, p. 117.
*  The input data array is 2*FFT_SIZE–long with real and imaginary data
*  in consecutive memory locations: Re–Im–Re–Im
*
*  The twiddle factors are supplied in a table put in a section
*  with a global label _SINE pointing to the beginning of the table
*  This data is included in a separate file to preserve the generic
*  nature of the program. The sine table size is (5*FFT_SIZE)/4.
*
*  In order to have the final results in bit–reversed order, the two
*  middle branches of the radix–4 butterfly are interchanged during
*  storage. Note the difference when comparing with the program in p.117
*  of the Burrus and Parks  book.
*
```

Example 6–14. Complex Radix-4 DIF FFT (Continued)

```
*  Note: Sections needed in the linker command file: .ffttxt : FFT code
*                                                    .fftdat : FFT data
*
*******************************************************************************
*
*  WARNING:
*
*  For optimization purposes, LDF *+AR1,R0 (see **1**) will fetch memory outside
*  the input buffer range during the "first loop" execution (RC=0). Even though
*  the read value (R0) is not used in the code, this could cause a halt situa
*  tion if AR1 points to a no-ready external memory
*
*******************************************************************************
       .globl   _SINE               ;Address of sine/cosine table
       .globl   _cr4dif             ;Entry point for execution
       .globl   STARTB,ENDB         ;starting/ending point for benchmarks
       .sect    ".fftdat"
FFTSIZ .space   1
SINTAB .word    _SINE
SINTAB1         .word      _SINE-1
INPUTP .space   1
OUTPUTP         .space     1
       .sect    ".ffttxt"
_cr4dif:
       LDI      SP,AR0
       PUSH     DP
       PUSH     R4                  ;Save dedicated registers

       PUSH     R5
       PUSH     R6                  ;lower 32 bits
       PUSHF    R6                  ;upper 32 bits
       PUSH     R7                  ;lower 32 bits
       PUSHF    R7                  ;upper 32 bits
       PUSH     AR3
       PUSH     AR4
       PUSH     AR5
       PUSH     AR6
       PUSH     AR7
       PUSH     R8
       .if      .REGPARM == 0
       LDI      *-AR0(1),AR2        ;points to input data
       LDI      *-AR0(2),R10        ;R10=N
       LDI      *-AR0(3),R9         ;R9 holds the remain stage number
       LDI      *-AR0(4),RC         ;points to where FFT result should move to
       .else
       LDI      R2,R10
       LDI      R3,R9
       .endif
       LDP      FFTSIZ              ;Command to load data page pointer
       STI      AR2, @INPUTP
       STI      RC, @OUTPUTP
       STI      R10,@FFTSIZ
```

Example 6–14. Complex Radix-4 DIF FFT (Continued)

```
STARTB:
       LDI      @FFTSIZ,BK
       LSH3     1,BK,IR0            ;IR0=2*N1 (because of real/imag)
       LSH3     -2,BK,IR1           ;IR1=N/4, pointer for SIN/COS table
       LDI      1,AR7               ;Initialize IE index
       LDI      1,R8                ;Initialize repeat counter of first loop
       ADDI     2,IR1,R9            ;R9=JT
       LSH      -1,BK               ;BK=N2
*  OUTER LOOP
LOOP:  LDI          @INPUTP,AR0     ;AR0 points to X(I)
       SUBI3    1,R8,RC             ;RC should be one less than desired #
       ADDI     BK,AR0,AR1          ;AR1 points to X(I1)
       RPTBD    BLK1                ;Setup loop BLK1
       ADDI     BK,AR1,AR2          ;AR2 points to X(I2)
       ADDI     BK,AR2,AR3          ;AR3 points to X(I3)
       LDF           *+AR1,R0       ;R0=Y(I1)

*  FIRST LOOP: BLK1
       ADDF     R0,*+AR3,R3;R3=Y(I1)+Y(I3)
       ADDF     *+AR0,*+AR2,R1      ;R1=Y(I)+Y(I2)
       ADDF     R3,R1,R6            ;R6=R1+R3
       SUBF     *+AR2,*+AR0,R4      ;R4=Y(I)-Y(I2)
       LDF      *AR2,R5             ;R5=X(I2)
||     STF      R6,*+AR0            ;Y(I)=R1+R3
       SUBF     R3,R1               ;R1=R1-R3
       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     R0,*+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
       STF      R2,*AR3++(IR0)      ;X(I3)=R2+R3
BLK1   STF      R5,*AR2++(IR0)      ;X(I2)=R2-R3
||     LDF      *+AR1,R0            ;R0=Y(I1)              ; **1**

*  IF THIS IS THE LAST STAGE, YOU ARE DONE
       CMPI     IR1,R8
       BZD           ENDB
```

Example 6–14. Complex Radix-4 DIF FFT (Continued)

```
*
*   MAIN INNER LOOP
*
          LDI      1,R10                ;Init IA1 index
          LDI      2,R11                ;Init loop counter for inner loop
          LDI      R11,AR0
          ADDI     @INPUTP,AR0          ;(X(I),Y(I)) pointer
          ADDI     2,R11                ;Increment inner loop counter
INLOP:    ADDI     AR7,R10              ;IA1=IA1+IE
          ADDI     BK,AR0,AR1           ;(X(I1),Y(I1)) pointer
          CMPI     R9,R11               ;If LPCNT=JT, go to
          BZD      SPCL                 ;special butterfly
          ADDI     BK,AR1,AR2           ;(X(I2),Y(I2)) pointer
          ADDI     BK,AR2,AR3           ;(X(I3),Y(I3)) pointer
          SUBI3    1,R8,RC              ;RC should be one less than desired #
          LDI      R10,AR4
          ADDI     @SINTAB1,AR4         ;Create cosine index AR4
          ADDI     AR4,R10,AR5
          SUBI     1,AR5                ;IA2=IA1+IA1–1
          RPTBD    BLK2                 ;Setup loop BLK2
          ADDI     R10,AR5,AR6
          SUBI     1,AR6                ;IA3=IA2+IA1–1
          LDF      *+AR2,R7             ;R7=Y(I2)
*
*   SECOND LOOP: BLK2
*
          ADDF     R7,*+AR0,R3          ;R3=Y(I)+Y(I2)
          ADDF     *+AR3,*+AR1,R5       ;R5=Y(I1)+Y(I3)
          ADDF     R5,R3,R6             ;R6=R3+R5
          SUBF     R7,*+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,R0             ;R0=R1+R5
          SUBF     *AR2,*AR0,R2         ;R2=X(I)–X(I2)
          SUBF     R5,R1                ;R1=R1–R5
          MPYF     R1,*AR5,R0           ;R0=R1*SI2
||        STF      R0,*AR0++(IR0)       ;X(I)=R1+R5
          SUBF     R0,R6                ;R6=R3*CO2–R1*SI2
          SUBF     *+AR3,*+AR1,R5       ;R5=Y(I1)–Y(I3)
          MPYF     R1,*+AR5(IR1),R0     ;R0=R1*C02
||        STF      R6,*+AR1             ;Y(I1)=R3*CO2–R1*SI2
          MPYF     R3,*AR5,R6           ;R6=R3*SI2
          ADDF     R0,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,R0           ;R0=R1*SI1
          SUBF     R0,R6                ;R6=R3*CO1+R1*SI1
          MPYF     R1,*+AR4(IR1),R6     ;R6=R1*CO1
||        STF      R6,*+AR2             ;Y(I2)=R3*CO1–R1*SI1
```

Example 6–14. Complex Radix-4 DIF FFT (Continued)

```
          MPYF    R3,*AR4,R0          ;R0=R3*SI1
          ADDF    R0,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,R0          ;R0=R2*SI3
          SUBF    R0,R6               ;R6=R1*CO3–R2*SI3
          MPYF    R2,*+AR6(IR1),R6    ;R6=R2*CO3
||        STF     R6,*+AR3            ;Y(I3)=R4*CO3–R2*SI3
          MPYF    R4,*AR6,R0          ;R0=R4*SI3
          ADDF    R0,R6               ;R6=R2*CO3+R4*SI3
BLK2      STF     R6,*AR3++(IR0)      ;x(i3)=R2*CO3+R4*SI3
||        LDF     *+AR2,R7            ;Load next Y(I2)
          CMPI    R11,BK
          BPD     INLOP               ;LOOP BACK TO THE INNER LOOP
          LDI     R11,AR0
          ADDI    @INPUTP,AR0         ;(X(I),Y(I)) pointer
          ADDI    2,R11               ;Increment inner loop counter
          BRD     CONT
          LSH     2,R8                ;Increment repeat counter for next time
          LSH     2,AR7               ;IE=4*IE
          LDI     BK,IR0              ;N1=N2

*  SPECIAL BUTTERFLY FOR W=J
SPCL      RPTBD   BLK3                ;Setup loop BLK3
          LSH     –1,IR1,AR4          ;Point to SIN(45)
          ADDI    @SINTAB,AR4         ;Create cosine index AR4=CO21
          LDF     *AR2,R7             ;R7=X(I2)
*  SPCL LOOP: BLK3
          ADDF    R7,*AR0,R1          ;R1=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,R0            ;R0=R3–R5
          ADDF    R5,R3               ;R3=R3+R5
          SUBF    R7,*AR0,R2          ;R2=X(I)–X(I2)
||        STF     R3,*+AR0            ;Y(I)=R3+R5
          LDF     *AR3,R7             ;R7=X(I3)
||        STF     R1,*AR0++(IR0)      ;X(I)=R1+R5
          SUBF    *+AR3,*+AR1,R3      ;R3=Y(I1)–Y(I3)
          SUBF    R7,*AR1,R1          ;R1=X(I1)–X(I3)
||        STF     R6,*+AR1            ;Y(I1)=R5–R1
          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    R1,*AR4,R1          ;R1=R1*CO21
||        STF     R0,*AR1++(IR0)      ;X(I1)=R3–R5
          ADDF    R5,R3               ;R3=R3+R5
          MPYF    R3,*AR4,R3          ;R3=R3*CO21
||        STF     R1,*+AR2            ;Y(I2)=(R3–R5)*CO21
          SUBF    R4,R2,R1            ;R1=R2–R4
          MPYF    R1,*AR4,R1          ;R1=R1*CO21
```

Example 6–14. Complex Radix-4 DIF FFT (Continued)

```
||        STF     R3,*AR2++(IR0)  ;X(I2)=(R3+R5)*CO21
          ADDF    R4,R2           ;R2=R2+R4
          MPYF3   R2,*AR4,R2      ;R2=R2*CO21
||        STF     R1,*+AR3        ;Y(I3)=-(R4-R2)*CO21
BLK3      LDF     *AR2,R7         ;Load next X(I2)
||        STF     R2,*AR3++(IR0)  ;X(I3)=(R4+R2)*CO21
          CMPI    R11,BK
          BPD     INLOP           ;Loop back to the inner loop
          LDI     R11,AR0
          ADDI    @INPUTP,AR0     ;(X(I),Y(I)) pointer
          ADDI    2,R11           ;Increment inner loop counter
          LSH     2,R8            ;Increment repeat counter for next time
          LSH     2,AR7           ;IE=4*IE
          LDI     BK,IR0          ;N1=N2
CONT      BRD     LOOP            ;Next FFT stage (delayed)
          LSH     -2,BK           ;N2=N2/4
          LSH3    -1,BK,R9
          ADDI    2,R9            ;JT=N2/2+2
ENDB:
*******************************************************************************
*-------------- BIT REVERSAL ------------------------------------------------*
* This bit-reversal section assumes input and output in Re-Im-Re-Im format *
*******************************************************************************
          LDI     @INPUTP,ar0
          CMPI    @OUTPUTP,ar0
          BEQD    INPLACE
          LDI     @OUTPUTP,ar1    ;ar1=DST_ADDR
          LDI     @FFTSIZ,ir0     ;ir0=FFT_SIZE
          SUBI    2,ir0,rc        ;rc=FFT_SIZE-2

          RPTBD   bitrv1
          LDI     2,ir1           ;ir1=2
          LDF     *+ar0(1),r0     ;read first Im value
          NOP     LDF             *ar0++(ir0)b,r1
||        STF     r0,*+ar1(1)
bitrv1    LDF     *+ar0(1),r0
||        STF     r1,*ar1++(ir1) BUD     END
          LDF     *ar0++(ir0)b,r1
||        STF     r0,*+ar1(1)
          NOP
          STF     r1,*ar1INPLACE
          RPTBD   BITRV2
          NOP     *++ar1(2)
          NOP     *ar0++(ir0)b
          NOP     CMPI            ar1,ar0
          BGEAT   CONT2
```

Example 6–14. Complex Radix-4 DIF FFT (Continued)

```
           LDF     *ar1,r0
||         LDF     *ar0,r1
           STF     r0,*ar0
||         STF     r1,*ar1
           LDF     *+ar1(1),r0
||         LDF     *+ar0(1),r1
           STF     r0,*+ar0(1)
||         STF     r1,*+ar1(1)
CONT2      NOP     *++ar1(2)
BITRV2     NOP     *ar0++(ir0)b
END:       POP     R8              ;Restore the register values and return
           POP     AR7
           POP     AR6
           POP     AR5
           POP     AR4
           POP     AR3
           POPF    R7
           POP     R7
           POPF    R6
           POP     R6
           POP     R5
           POP     R4
           POP     DP
           RETS
           .end
```

6.5.3 Faster Complex Radix-2 DIT FFT

Example 6–12 and Example 6–14 provide an easy understanding of the FFT algorithm functions. However, those examples are not optimized for fast execution of the FFT. Example 6–15 shows a faster version of a radix-2 DIT FFT algorithm. This program uses a different twiddle factor table than the previous examples. The twiddle factors are stored in bit-reversed order and with a table length of N/2 (N = FFT length) as shown in Example 6–16. For instance, if the FFT length is 32, the twiddle factor table should be:

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

Example 6–15. Faster Version Complex Radix-2 DIT FFT

```
*******************************************************************************
*
*  FILENAME    : CR2DIT.ASM
*
*  DESCRIPTION : COMPLEX, RADIX–2 DIT FFT FOR TMS320C40
*
*  DATE        : 6/29/93
*
*  VERSION     : 4.0
*
*******************************************************************************
*    VERSION        DATE         COMMENTS
*    -------        ----         --------
*     1.0           7/89         Original version
*                                RAIMUND MEYER, KARL SCHWARZ
*                                LEHRSTUHL FUER NACHRICHTENTECHNIK
*                                UNIVERSITAET ERLANGEN–NUERNBERG
*                                CAUERSTRASSE 7, D–8520 ERLANGEN, FRG
*
*     2.0           1/91         DANIEL CHEN (TI HOUSTON): C40 porting
*     3.0           7/1/92       ROSEMARIE PIEDRA (TI HOUSTON): made it
*                                C–callable and implemented changes in the order
*                                of the operands for some mpyf instructions for
*                                faster execution when sine table is off–chip
*     4.0           6/29/93      ROSEMARIE PIEDRA (TI Houston): Added support
*                                for in–place bit reversing.
*******************************************************************************
*
*   SYNOPSIS: int  cr2dit(SOURCE_ADDR,FFT_SIZE, DST_ADDR)
*                              ar2        r2       r3
*
*             float   *SOURCE_ADDR    ; Points to where data is originated
*                                     ; and operated on.
*             int     FFT_SIZE        ; 64, 128, 256, 512, 1024, ...
*
*             float   *DST_ADDR       ; Points to where FFT results should be
*                                     ; moved
*
*******************************************************************************
*
*   THE COMPUTATION IS DONE IN–PLACE.
*   FOR THIS PROGRAM THE MINIMUM FFT LENGTH IS 32 POINTS BECAUSE OF THE
*   SEPARATE STAGES (THIS IS NOT CHECKED INSIDE THE
*   FIRST TWO PASSES ARE REALIZED AS A FOUR BUTTERFLY LOOP SINCE THE
*   MULTIPLIES ARE TRIVIAL. THE MULTIPLIER IS ONLY USED FOR A LOAD IN
*   PARALLEL WITH AN ADDF OR SUBF.
*******************************************************************************
*   SECTIONS NEEDED IN LINKER COMMAND FILE:  .ffttxt : fft code
*                                            .fftdat : fft data
*******************************************************************************
*
*   THE TWIDDLE FACTORS ARE STORED IN BIT-REVERSED ORDER AND WITH A TABLE LENGTH
*   OF N/2 (N = FFTLENGTH). THE SINE TABLE IS PROVIDED IN A SEPARATE FILE
*   WITH GLOBAL LABEL _SINE POINTING TO THE BEGINNING OF THE TABLE.
*
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
*   EXAMPLE: SHOWN FOR N=32, WN(n) = COS(2*PI*n/N) – j*SIN(2*PI*n/N)
*
*               ADDRESS                  COEFFICIENT
*                  0          R{WN(0)} = COS(2*PI*0/32) = 1
*                  1         –I{WN(0)} = SIN(2*PI*0/32) = 0
*                  2          R{WN(4)} = COS(2*PI*4/32) = 0.707
*                  3         –I{WN(4)} = SIN(2*PI*4/32) = 0.707
*
*                  :                          :
*
*                  12         R{WN(3)} = COS(2*PI*3/32) = 0.831
*                  13        –I{WN(3)} = SIN(2*PI*3/32) = 0.556
*                  14         R{WN(7)} = COS(2*PI*7/32) = 0.195
*                  15        –I{WN(7)} = SIN(2*PI*7/32) = 0.981
*
*   WHEN GENERATED FOR A FFT LENGTH OF 1024, THE TABLE IS FOR ALL FFT
*   LENGTH LESS OR EQUAL AVAILABLE.
*   THE MISSING TWIDDLE FACTORS (WN(),WN(),....) ARE GENERATED BY USING
*   THE SYMMETRY WN(N/4+n) = –j*WN(n). THIS CAN BE REALIZED VERY EASY, BY
*   CHANGING REAL– AND IMAGINARY PART OF THE TWIDDLE FACTORS AND BY
*   NEGATING THE NEW REAL PART.
*******************************************************************************
**
*
*                                                +
*    AR + j AI ------------------------------------------------ AR' + j AI'
*                                       \         / +
*                                         \     /
*                                           \ /
*                                           / \
*                                         /     \
*                                       /         \ +
*    BR + j BI ---- ( COS – j SIN ) ---------------------------- BR' + j BI'
*                                                –*
*    TR = BR * COS + BI * SIN
*    TI = BI * COS – BR * SIN
*    AR'= AR + TR
*    AI'= AI + TI
*    BR'= AR – TR
*    BI'= AI – TI
*
*******************************************************************************
**
          .global  _cr2dit        ; Entry execution point.
          .global  _SINE          ; sine table pointer
          .global  STARTB,ENDB    ; starting/ending point for given
                                  ; benchmarks
          .sect    ".fftdat"
fg        .space   1              ; is FFT_SIZE
fg2       .space   1              ; is FFT_SIZE/2
fg4m2     .space   1              ; is FFT_SIZE/4 – 2
fg8m2     .space   1              ; is FFT_SIZE/8 – 2
sintab    .word    _SINE          ; pointer to sine table
sintp2    .word    _SINE+2        ; pointer to sine table +2
inputp2   .space   1              ; pointer to input +2
inputp    .space   1              ; pointer to source address
outputp   .space   1              ; pointer to dst address
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
;
; Initialize C Function.
;
          .sect     ".ffttxt"
_cr2dit:  LDI       SP,AR0
          PUSH      R4
          PUSH      R5
          PUSH      R6
          PUSHF     R6
          PUSH      R7
          PUSHF     R7
          PUSH      AR3
          PUSH      AR4
          PUSH      AR5
          PUSH      AR6
          PUSH      AR7
          PUSH      DP
          .if       .REGPARM == 0   ; arguments passed in stack
          LDI       *-AR0(1),AR2    ; src address
          LDI       *-AR0(2),R2     ; FFT size
          LDI       *-AR0(3),R3     ; dst address
          .endif
          LDP       fg              ; Initialize DP pointer.
          STI       R2,@fg          ; fg = FFT_SIZE
          LSH       -1,R2           ; R2 = FFT_SIZE/2
          STI       AR2,@inputp     ; inputp = SOURCE_ADDR
          ADDI      2,AR2,R0
          STI       R0,@inputp2     ; inputp2= SOURCE_ADDR + 2
          STI       R3,@outputp     ; output = DST_ADDR
          STI       R2,@fg2         ; fg2 = nhalb = (FFT_size/2)
          LSH       -1,R2
          SUBI      2,R2,R0
          STI       R0,@fg4m2       ; fg4m2 = NVIERT-2 : (FFT_SIZE/4)-2
          LSH       -1,R2
          SUBI      2,R2,R0
          STI       R0,@fg8m2
*       ar0 : AR + AI
*       ar1 : BR + BI
*       ar2 : CR + CI + CR' + CI'
*       ar3 : DR + DI
*       ar4 : AR' + AI'
*       ar5 : BR' + BI'
*       ar6 : DR' + DI'
*       ar7 : first twiddle factor = 1
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
STARTB:
          ldi      @fg2,ir0           ; ir0 = n/2 = offset between SOURCE_ADDRs
          ldi      @sintab,ar7        ; ar7 points to twiddle factor 1
          ldi      ar2,ar0            ; ar0 points to AR
          addi     ir0,ar0,ar1        ; ar1 points to BR
          addi     ir0,ar1,ar2        ; ar2 points to CR
          addi     ir0,ar2,ar3        ; ar3 points to DR
          ldi      ar0,ar4            ; ar4 points to AR'
          ldi      ar1,ar5            ; ar5 points to BR'
          ldi      ar3,ar6            ; ar6 points to DR'
          ldi      2,ir1              ; addressoffset
          lsh      -1,ir0             ; ir0 = n/4 = number of R4-butterflies
          subi     2,ir0,rc
*************************************************************************
* ------------ FIRST 2 STAGES AS RADIX-4 BUTTERFLY ------------------------ *
*************************************************************************
fill pipeline
          addf     *ar2,*ar0,r4       ; r4 = AR + CR
          subf     *ar2,*ar0++,r5     ; r5 = AR - CR
          addf     *ar1,*ar3,r6       ; r6 = DR + BR
          subf     *ar1++,*ar3++,r7   ; r7 = DR - BR
          addf     r6,r4,r0           ; AR' = r0 = r4 + r6
          mpyf     *ar7,*ar3++,r1     ; r1 = DI , BR' = r3 = r4 - r6
||        subf     r6,r4,r3
          addf     r1,*ar1,r0         ; r0 = BI + DI , AR' = r0
||        stf      r0,*ar4++
          subf     r1,*ar1++,r1       ; r1 = BI - DI , BR' = r3
||        stf      r3,*ar5++
          addf     r1,r5,r2           ; CR' = r2 = r5 + r1
          mpyf     *ar7,*+ar2,r1      ; r1 = CI , DR' = r3 = r5 - r1
||        subf     r1,r5,r3
          rptbd    blk1               ; Setup for radix-4 butterfly loop
          addf     r1,*ar0,r2         ; r2 = AI + CI , CR' = r2
||        stf      r2,*ar2++(ir1)
          subf     r1,*ar0++,r6       ; r6 = AI - CI , DR' = r3
||        stf      r3,*ar6++
          addf     r0,r2,r4           ; AI' = r4 = r2 + r0
```

Example 6–15. Faster Version Complex, Radix-2 DIT FFT (Continued)

```
* radix-4 butterfly loop
*
          mpyf     *ar7,*ar2--,r0 ; r0 = CR , (BI' = r2 = r2 - r0)
||        subf     r0,r2,r2
          mpyf     *ar7,*ar1++,r1 ; r1 = BR , (CI' = r3 = r6 + r7)
||        addf     r7,r6,r3
          addf     r0,*ar0,r4     ; r4 = AR + CR , (AI' = r4)
||        stf      r4,*ar4++
          subf     r0,*ar0++,r5   ; r5 = AR - CR , (BI' = r2)
||        stf      r2,*ar5++
          subf     r7,r6,r7       ; (DI' = r7 = r6 - r7)
          addf     r1,*ar3,r6     ; r6 = DR + BR , (DI' = r7)
||        stf      r7,*ar6++
          subf     r1,*ar3++,r7   ; r7 = DR - BR , (CI' = r3)
||        stf      r3,*ar2++
          addf     r6,r4,r0       ; AR' = r0 = r4 + r6
          mpyf     *ar7,*ar3++,r1 ; r1 = DI , BR' = r3 = r4 - r6
||        subf     r6,r4,r3
          addf     r1,*ar1,r0     ; r0 = BI + DI , AR' = r0
||        stf      r0,*ar4++
          subf     r1,*ar1++,r1   ; r1 = BI - DI , BR' = r3
||        stf      r3,*ar5++
          addf     r1,r5,r2       ; CR' = r2 = r5 + r1
          mpyf     *+ar2,*ar7,r1   ; r1 = CI , DR' = r3 = r5 - r1
||        subf     r1,r5,r3
          addf     r1,*ar0,r2     ; r2 = AI + CI , CR' = r2
||        stf      r2,*ar2++(ir1)
          subf     r1,*ar0++,r6   ; r6 = AI - CI , DR' = r3
||        stf      r3,*ar6++
blk1      addf     r0,r2,r4       ; AI' = r4 = r2 + r0
* clear pipeline
*
          subf     r0,r2,r2       ; BI' = r2 = r2 - r0
          addf     r7,r6,r3       ; CI' = r3 = r6 + r7
          stf      r4,*ar4        ; AI' = r4 , BI' = r2
||        stf      r2,*ar5
          subf     r7,r6,r7       ; DI' = r7 = r6 - r7
          stf      r7,*ar6        ; DI' = r7 , CI' = r3
||        stf      r3,*--ar2
*******************************************************************************
* ------------ THIRD TO LAST-2 STAGE ------------------------------------- *
*******************************************************************************
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
          ldi      @fg2,ir1
          subi     1,ir0,ar5
          ldi      1,ar6
          ldi      @sintab,ar7       ; pointer to twiddle factor
          ldi      0,ar4             ; group counter
          ldi      @inputp,ar0
stufe     ldi      ar0,ar2           ; upper real butterfly output
          addi     ir0,ar0,ar3       ; lower real butterfly output
          ldi      ar3,ar1           ; lower real butterfly input
          lsh      1,ar6             ; double group count
          lsh      -2,ar5            ; half butterfly count
          lsh      1,ar5             ; clear LSB
          lsh      -1,ir0            ; half step from upper to lower real part
          lsh      -1,ir1
          addi     1,ir1             ; step from old imaginary to new
                                     ; real value
          ldf      *ar1++,r6         ; dummy load, only for address update
||        ldf      *ar7,r7           ; r7 = COS
gruppe
* fill pipeline
*
*   ar0 = upper real butterfly input
*   ar1 = lower real butterfly input
*   ar2 = upper real butterfly output
*   ar3 = lower real butterfly output
*   the imaginary part has to follow
          ldf      *++ar7,r6         ; r6 = SIN
          mpyf     *ar1--,r6,r1      ; r1 = BI * SIN
||        addf     *++ar4,r0,r3      ; dummy addf for counter update
          mpyf     *ar1,r7,r0        ; r0 = BR * COS
          ldi      ar5,rc
          rptbd    bfly1             ; Setup for loop bfly1
          mpyf     *ar7--,*ar1++,r0  ; r3 = TR = r0 + r1 , r0 = BR * SIN
||        addf     r0,r1,r3
          mpyf     *ar1++,r7,r1      ; r1 = BI * COS , r2 = AR - TR
||        subf     r3,*ar0,r2
          addf     *ar0++,r3,r5      ; r5 = AR + TR , BR' = r2
||        stf      r2,*ar3++
*    FIRST BUTTERFLY-TYPE:
*
*    TR = BR * COS + BI * SIN
*    TI = BR * SIN - BI * COS
*    AR'= AR + TR
*    AI'= AI - TI
*    BR'= AR - TR
*    BI'= AI + TI
*           loop bfly1
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
          mpyf     *+ar1,r6,r5          ; r5 = BI * SIN , (AR' = r5)
||        stf      r5,*ar2++
          subf     r1,r0,r2             ; (r2 = TI = r0 – r1)
          mpyf     *ar1,r7,r0           ; r0 = BR * COS , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++,r4         ; (r4 = AI – TI , BI' = r3)
||        stf      r3,*ar3++
          addf     r0,r5,r3             ; r3 = TR = r0 + r5
          mpyf     *ar1++,r6,r0         ; r0 = BR * SIN , r2 = AR – TR
||        subf     r3,*ar0,r2
          mpyf     *ar1++,r7,r1         ; r1 = BI * COS , (AI' = r4)
||        stf      r4,*ar2++
bfly1     addf     *ar0++,r3,r5         ; r5 = AR + TR , BR' = r2
||        stf      r2,*ar3++
* switch over to next group
          subf     r1,r0,r2             ; r2 = TI = r0 – r1
          addf     r2,*ar0,r3           ; r3 = AI + TI , AR' = r5
||        stf      r5,*ar2++
          subf     r2,*ar0++(ir1),r4    ; r4 = AI – TI , BI' = r3
||        stf      r3,*ar3++(ir1)
          nop      *ar1++(ir1)          ; address update
          mpyf     *ar1--,r7,r1         ; r1 = BI * COS , AI' = r4
||        stf      r4,*ar2++(ir1)
          mpyf     *ar1,r6,r0           ; r0 = BR * SIN
          ldi      ar5,rc
          rptbd    bfly2                ; Setup for loop bfly2
          mpyf     *ar7++,*ar1++,r0     ; r3 = TR = r1 – r0 , r0 = BR * COS
||        subf     r0,r1,r3
          mpyf     *ar1++,r6,r1         ; r1 = BI * SIN , r2 = AR – TR
||        subf     r3,*ar0,r2
          addf     *ar0++,r3,r5         ; r5 = AR + TR , BR' = r2
||        stf      r2,*ar3++
*   SECOND BUTTERFLY-TYPE:
*
*   TR = BI * COS – BR * SIN
*   TI = BI * SIN + BR * COS
*   AR'= AR + TR
*   AI'= AI – TI
*   BR'= AR – TR
*   BI'= AI + TI
*         loop bfly2
          mpyf     *+ar1,r7,r5          ; r5 = BI * COS , (AR' = r5)
||        stf      r5,*ar2++
          addf     r1,r0,r2             ; (r2 = TI = r0 + r1)
          mpyf     *ar1,r6,r0           ; r0 = BR * SIN , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++,r4         ; (r4 = AI – TI , BI' = r3)
||        stf      r3,*ar3++
          subf     r0,r5,r3             ; TR = r3 = r5 – r0
          mpyf     *ar1++,r7,r0         ; r0 = BR * COS , r2 = AR – TR
||        subf     r3,*ar0,r2
          mpyf     *ar1++,r6,r1         ; r1 = BI * SIN , (AI' = r4)
||        stf      r4,*ar2++
bfly2     addf     *ar0++,r3,r5         ; r5 = AR + TR , BR' = r2
||        stf      r2,*ar3++
* clear pipeline
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

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

*   1. butterfly: w^0
          addf     *ar0,*ar1,r2                ; AR' = r2 = AR + BR
          subf     *ar1++,*ar0++,r3            ; BR' = r3 = AR - BR
          addf     *ar0,*ar1,r0                ; AI' = r0 = AI + BI
          subf     *ar1++,*ar0++,r1            ; BI' = r1 = AI - BI
*   2. butterfly: w^0

          addf     *ar0,*ar1,r6                ; AR' = r6 = AR + BR
          subf     *ar1++,*ar0++,r7            ; BR' = r7 = AR - BR
          addf     *ar0,*ar1,r4                ; AI' = r4 = AI + BI
          subf     *ar1++(ir0),*ar0++(ir0),r5  ; BI' = r5 = AI - BI
          stf      r2,*ar2++                   ; (AR' = r2)
||        stf      r3,*ar3++                   ; (BR' = r3)
          stf      r0,*ar2++                   ; (AI' = r0)
||        stf      r1,*ar3++                   ; (BI' = r1)
          stf      r6,*ar2++                   ; AR' = r6
||        stf      r7,*ar3++                   ; BR' = r7
          stf      r4,*ar2++(ir0)              ; AI' = r4
||        stf      r5,*ar3++(ir0)              ; BI' = r5
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
*   3. butterfly: w^M/4
          addf     *ar0++,*+ar1,r5        ; AR' = r5 = AR + BI
          subf     *ar1,*ar0,r4           ; AI' = r4 = AI - BR
          addf     *ar1++,*ar0--,r6       ; BI' = r6 = AI + BR
          subf     *ar1++,*ar0++,r7       ; BR' = r7 = AR - BI
*   4. butterfly: w^M/4
          addf     *+ar1,*++ar0,r3        ; AR' = r3 = AR + BI
          ldf      *-ar7,r1               ; r1 = 0 (for inner loop)
||        ldf      *ar1++,r0              ; r0 = BR (for inner loop)
          rptbd    bf2end                 ; Setup for loop bf2end
          subf     *ar1++(ir0),*ar0++,r2  ; BR' = r2 = AR - BI
          stf      r5,*ar2++              ; (AR' = r5)
||        stf      r7,*ar3++              ; (BR' = r7)
          stf      r6,*ar3++              ; (BI' = r6)
*   5. to M. butterfly:
*         loop bf2end
          ldf      *ar7++,r7              ; r7 = COS , ((AI' = r4))
||        stf      r4,*ar2++
          ldf      *ar7++,r6              ; r6 = SIN , (BR' = r2)
||        stf      r2,*ar3++
          mpyf     *+ar1,r6,r5            ; r5 = BI * SIN , (AR' = r3)
||        stf      r3,*ar2++
          addf     r1,r0,r2               ; (r2 = TI = r0 + r1)
          mpyf     *ar1,r7,r0             ; r0 = BR * COS , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++(ir0),r4      ; (r4 = AI - TI , BI' = r3)
||        stf      r3,*ar3++(ir0)
          addf     r0,r5,r3               ; r3 = TR = r0 + r5
          mpyf     *ar1++,r6,r0           ; r0 = BR * SIN , r2 = AR - TR
||        subf     r3,*ar0,r2
          mpyf     *ar1++,r7,r1           ; r1 = BI * COS , (AI' = r4)
||        stf      r4,*ar2++(ir0)
          addf     *ar0++,r3,r5           ; r5 = AR + TR , BR' = r2
||        stf      r2,*ar3++
          mpyf     *+ar1,r6,r5            ; r5 = BI * SIN , (AR' = r5)
||        stf      r5,*ar2++
          subf     r1,r0,r2               ; (r2 = TI = r0 - r1)
          mpyf     *ar1,r7,r0             ; r0 = BR * COS , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++,r4           ; (r4 = AI - TI , BI' = r3)
||        stf      r3,*ar3++
          addf     r0,r5,r3               ; r3 = TR = r0 + r5
          mpyf     *ar1++,r6,r0           ; r0 = BR * SIN , r2 = AR - TR
||        subf     r3,*ar0,r2
          mpyf     *ar1++(ir0),r7,r1      ; r1 = BI * COS , (AI' = r4)
||        stf      r4,*ar2++
          addf     *ar0++,r3,r3           ; r3 = AR + TR , BR' = r2
||        stf      r2,*ar3++
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
          mpyf     *+ar1,r7,r5                ; r5 = BI * COS , (AR' = r3)
||        stf      r3,*ar2++
          subf     r1,r0,r2                   ; (r2 = TI = r0 - r1)
          mpyf     *ar1,r6,r0                 ; r0 = BR * SIN , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++(ir0),r4          ; (r4 = AI - TI , BI' = r3)
||        stf      r3,*ar3++(ir0)
          subf     r0,r5,r3                   ; r3 = TR = r5 - r0
          mpyf     *ar1++,r7,r0               ; r0 = BR * COS , r2 = AR - TR
||        subf     r3,*ar0,r2
          mpyf     *ar1++,r6,r1               ; r1 = BI * SIN , (AI' = r4)
||        stf      r4,*ar2++(ir0)
          addf     *ar0++,r3,r5               ; r5 = AR + TR , BR' = r2
||        stf      r2,*ar3++
          mpyf     *+ar1,r7,r5                ; r5 = BI * COS , (AR' = r5)
||        stf      r5,*ar2++
          addf     r1,r0,r2                   ; (r2 = TI = r0 + r1)
          mpyf     *ar1,r6,r0                 ; r0 = BR * SIN , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++,r4               ; (r4 = AI - TI , y(L) = BI' =
r3)
||        stf      r3,*ar3++
          subf     r0,r5,r3                   ; r3 = TR = r5 - r0
          mpyf     *ar1++,r7,r0               ; r0 = BR * COS , r2 = AR - TR
||        subf     r3,*ar0,r2
bf2end    mpyf     *ar1++(ir0),r6,r1          ; r1 = BI * SIN , r3 = AR + TR
||        addf     *ar0++,r3,r3
*  clear pipeline
          stf         r2,*ar3++                     ; BR' = r2 , AI' = r4
||        stf         r4,*ar2++
          addf     r1,r0,r2                   ; r2 = TI = r0 + r1
          addf     r2,*ar0,r3                 ; r3 = AI + TI , AR' = r3
||        stf         r3,*ar2++
          subf     r2,*ar0,r4                 ; r4 = AI - TI , BI' = r3
||        stf         r3,*ar3
          stf         r4,*ar2                           ; AI' = r4
*******************************************************************************
*------------- LAST STAGE ----------------------------------------------------*
*******************************************************************************
          ldi      @inputp,ar0
          ldi      ar0,ar2                    ; upper output
          ldi      @inputp2,ar1
          ldi      ar1,ar3                    ; lower output
          ldi      @sintp2,ar7                ; pointer to twiddle factors
          ldi      3,ir0                      ; group offset
          ldi      @fg4m2,rc
*  fill pipeline
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
*  1. butterfly: w^0
          addf     *ar0,*ar1,r6              ; AR' = r6 = AR + BR
          subf     *ar1++,*ar0++,r7          ; BR' = r7 = AR – BR
          addf     *ar0,*ar1,r4              ; AI' = r4 = AI + BI
          subf     *ar1++(ir0),*ar0++(ir0),r5; BI' = r5 = AI – BI
* 2. butterfly: w^M/4
          addf     *+ar1,*ar0,r3             ; AR' = r3 = AR + BI
          ldf      *–ar7,r1                  ; r1 = 0 (for inner loop)
||        ldf      *ar1++,r0                 ; r0 = BR (for inner loop)
          rptbd    bflend                    ; Setup for loop bflend
          subf     *ar1++(ir0),*ar0++,r2             ; BR' = r2 = AR – BI
          stf      r6,*ar2++                 ; (AR' = r6)
||        stf      r7,*ar3++                 ; (BR' = r7)
          stf      r5,*ar3++(ir0)            ; (BI' = r5)
* 3. to M. butterfly:
*  loop bflend
          ldf      *ar7++,r7                 ; r7 = COS , ((AI' = r4))
||        stf      r4,*ar2++(ir0)
          ldf      *ar7++,r6                 ; r6 = SIN , (BR' = r2)
||        stf      r2,*ar3++
          mpyf     *+ar1,r6,r5               ; r5 = BI * SIN, (AR' = r3)
||        stf      r3,*ar2++
          addf     r1,r0,r2                  ; (r2 = TI = r0 + r1)
          mpyf     *ar1,r7,r0                ; r0 = BR * COS , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++(ir0),r4         ; (r4 = AI – TI , BI' = r3)
||        stf      r3,*ar3++(ir0)
          addf     r0,r5,r3                  ; r3 = TR = r0 + r5
          mpyf     *ar1++,r6,r0              ; r0 = BR * SIN , r2 = AR – TR
||        subf     r3,*ar0,r2
          mpyf     *ar1++(ir0),r7,r1         ; r1 = BI * COS , (AI' = r4)
||        stf      r4,*ar2++(ir0)
          addf     *ar0++,r3,r3              ; r3 = AR + TR , BR' = r2
||        stf      r2,*ar3++
          mpyf     *+ar1,r7,r5               ;r5 = BI * COS , (AR' = r3)
||        stf      r3,*ar2++
          subf     r1,r0,r2                  ;(r2 = TI = r0 – r1)
          mpyf     *ar1,r6,r0                ;r0 = BR * SIN , (r3 = AI + TI)
||        addf     r2,*ar0,r3
          subf     r2,*ar0++(ir0),r4         ;(r4 = AI – TI , BI' = r3)
||        stf      r3,*ar3++(ir0)
          subf     r0,r5,r3                  ;r3 = TR = r0 – r5
          mpyf     *ar1++,r7,r0              ;r0 = BR * COS , r2 = AR – TR
||        subf     r3,*ar0,r2
bflend    mpyf     *ar1++(ir0),r6,r1         ;r1 = BI * SIN , r3 = AR + TR
||        addf     *ar0++,r3,r3
* clear pipeline
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
          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 -------------------------------------------------*
*****************************************************************************
ENDB:
*****************************************************************************
*-------------- BITREVERSAL ------------------------------------------------*
* This bit-reversal section assume input and output in Re-Im-Re-Im format *
*****************************************************************************
          ldi       @inputp,ar0
          cmpi      @outputp,ar0
          beqd      INPLACE
          ldi       @outputp,ar1            ;ar1=DSR_ADDR
          ldi       @fg,ir0                 ;ir0=FFT_SIZE
          subi      2,ir0,rc                ;rc=FFT_SIZE-2
          rptbd     bitrv1
          ldi       2,ir1                   ;ir1=2
          ldf       *+ar0(1),r0             ;read first Im value
          nop
          ldf       *ar0++(ir0)b,r1
||        stf       r0,*+ar1(1)
bitrv1    ldf       *+ar0(1),r0
||        stf       r1,*ar1++(ir1)
          bud       end
          ldf       *ar0++(ir0)b,r1
||        stf       r0,*+ar1(1)
          nop
          stf       r1,*ar1
;
; Return to C environment.
;
INPLACE
          rptbd     BITRV2
          nop       *++ar1(2)
          nop       *ar0++(ir0)b
          nop
```

Example 6–15. Faster Version Complex Radix-2 DIT FFT (Continued)

```
          cmpi     ar1,ar0
          bgeat    CONT
          ldf      *ar1,r0
||        ldf      *ar0,r1
          stf      r0,*ar0
||        stf      r1,*ar1
          ldf      *+ar1(1),r0
||        ldf      *+ar0(1),r1
          stf      r0,*+ar0(1)
||        stf      r1,*+ar1(1)
CONT      nop      *++ar1(2)
BITRV2    nop      *ar0++(ir0)b
; Return to C environment
end:      POP      DP
          POP      AR7
          POP      AR6
          POP      AR5
          POP      AR4
          POP      AR3
          POPF     R7
          POP      R7
          POPF     R6
          POP      R6
          POP      R5
          POP      R4
          RETS
          .end
```

Example 6–16. Bit-Reversed Sine Table

```
*****************************************************************
*
*   SINTAB.ASM : Bit–reversed sine table for a 64–point
*                File to be linked with the source code for a
*                64–point radix–2 DIT FFT
*                Sine table length = FFT size / 2
*
*****************************************************************
    .global    _SINE
    .sect ".sintab"
_SINE
    .float     1.000000
    .float     0.000000
    .float     0.707107
    .float     0.707107
    .float     0.923880
    .float     0.382683
    .float     0.382683
    .float     0.923880
    .float     0.980785
    .float     0.195090
    .float     0.555570
    .float     0.831470
    .float     0.831470
    .float     0.555570
    .float     0.195090
    .float     0.980785
    .float     0.995185
    .float     0.098017
    .float     0.634393
    .float     0.773010
    .float     0.881921
    .float     0.471397
    .float     0.290285
    .float     0.956940
    .float     0.956940
    .float     0.290285
    .float     0.471397
    .float     0.881921
    .float     0.773010
    .float     0.634393
    .float     0.098017
    .float     0.995185
    .end
```

6.5.4 Real Radix-2 FFT

Most often, the data to be transformed is a sequence of real numbers. In this case, the FFT demonstrates certain symmetries that permit the reduction of the computational load even further. Example 6–17 and Example 6–18 show the generic implementation of a real-valued radix-2 FFT (forward and inverse). For such an FFT, the total storage required for a length-N transform is only N locations; in a complex FFT, 2N are necessary. Recovery of the rest of the points is based on the symmetry conditions. A companion table (Example 6–13) should be used to provide the twiddle factors.

Example 6–17. Real Forward Radix-2 FFT

```
*******************************************************************************
*
*  FILENAME     : FFFT_RL.ASM
*  DESCRIPTION  : REAL, RADIX-2 DIF FFT FOR TMS320C40
*  DATE         : 1/19/93
*  VERSION      : 3.0
*
*******************************************************************************
*
*  VERSION     DATE      COMMENTS
*  -------     ----      --------
*  1.0        7/18/91     ALEX TESSAROLO(TI Australia):
*                         Original Release (C30 version)
*  2.0        7/23/92     ALEX TESSAROLO(TI Australia):
*                         Most Stages Modified (C30 version).
*                         Minimum FFT Size increased from 32 to 64.
*                         Faster in place bit reversing algorithm.
*                         Program size increased by about 100 words.
*                         One extra data word required.
*  3.0        1/19/93     ROSEMARIE PIEDRA(TI Houston):
*                         C40 porting started from  C30 forward real FFT
*                         version 2.0. Expanded calling conventions to the use
*                         of registers for parameter passing.
*
*******************************************************************************
*
*   SYNOPSIS:
*
*   int ffft_rl (FFT_SIZE,LOG_SIZE,SOURCE_ADDR,DEST_ADDR,SINE_TABLE,BIT_REVERSE)
*                  ar2       r2        r3         rc        rs         re
*
*     int     FFT_SIZE        ; 64, 128, 256, 512, 1024, ...
*     int     LOG_SIZE        ;  6,   7,   8,   9,   10, ...
*     float   *SOURCE_ADDR    ; Points to location of source data.
*     float   *DEST_ADDR      ; Points to where data will be
*                             ; operated on and stored.
*     float   *SINE_TABLE     ; Points to the SIN/COS table.
*     int     BIT_REVERSE     ; =  0, Bit Reversing is disabled.
*                             ; <> 0, Bit Reversing is enabled.
*
*     NOTE:   1) If SOURCE_ADDR = DEST_ADDR, then in place bit reversing
*                is performed, if enabled (more processor intensive).
*             2) FFT_SIZE must be >= 64   (this is not checked).
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
*
*****************************************************************************
*
*   DESCRIPTION:
*
*  Generic function to do a radix–2 FFT computation on the C40.
*  The input data array is FFT_SIZE–long with only real data. The output is
*  stored in the same locations (in–place) with real and imaginary
*  points R and I as follows:
*
*   DEST_ADDR[0]    –>             R(0)
*                                  R(1)
*                                  R(2)
*                                  R(3)
*                                  .
*                                  .
*                                  R(FFT_SIZE/2)
*                                  I(FFT_SIZE/2 – 1)
*                                  .
*                                        .
*                                  I(2)
*   DEST_ADDR[FFT_SIZE – 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.
*
*  Bit reversal is optionally implemented at the beginning of the function.
*
*  The sine/cosine table for the twiddle factors is expected to be supplied in
*  the following format:
*
*   SINE_TABLE[0]             –>   sin(0*2*pi/FFT_SIZE)
*                                  sin(1*2*pi/FFT_SIZE)
*                                  .
*                                  .
*                                  sin((FFT_SIZE/2–2)*2*pi/FFT_SIZE)
*   SINE_TABLE[FFT_SIZE/2–1]  –>   sin((FFT_SIZE/2–1)*2*pi/FFT_SIZE)
*
*  NOTE: The table is the first half period of a sine wave.
*
*****************************************************************************
*
*  NOTES:   1. Calling C program can be compiled with large or small model. Both
*              calling conventions methods: stack or register for parameter
*              passing are supported.
*
*           2. Sections needed in linker command file: .ffttxt  : fft code
*                                                      .fftdat  : fft data
*
*           3. The DEST_ADDR must be aligned such that the first LOG_SIZE bits
*               are zero (this is not checked by the program)
*
*  Caution: DP initialized only once in the program. Be wary with interrupt
*           service routines. Make sure interrupt service routines save the DP
*           pointer.
*
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
*******************************************************************************
*
*  REGISTERS USED: R0, R1, R2, R3, R4, R5, R6, R7
*                  AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7
*                  IR0, IR1
*                  RC, RS, RE
*                  DP
*
*
*  MEMORY REQUIREMENTS:  Program = 405 Words (approximately)
*                        Data    =   7 Words
*                        Stack   =  12 Words
*
*******************************************************************************
*
*  BENCHMARKS:   Assumptions     – Program in RAM0
*                                – Reserved data in RAM0
*                                – Stack on Local/Global Bus RAM
*                                – Sine/Cosine tables in RAM0
*                                – Processing and data destination in RAM1.
*                                – Local/Global Bus RAM, 0 wait state.
*
*  FFT Size        Bit Reversing   Data Source     Cycles(C40)
*  --------        -------------   -----------     -----------
*   1024            OFF             RAM1            19404 approx.
*
*  Note: This number does not include the C callable overheads.
*        This benchmark is the number of cycles between labels STARTB and ENDB.
*
*  NOTE:
*  – If .ffttxt is located off–chip, enable cache for faster performance
*
*******************************************************************************
*
FP           .set  AR3
             .global  _ffft_rl    ; Entry execution point.
             .global  STARTB,ENDB
FFT_SIZE:    .usect   ".fftdat",1 ; Reserve memory for arguments.
LOG_SIZE:    .usect   ".fftdat",1
SOURCE_ADDR: .usect   ".fftdat",1
DEST_ADDR:   .usect   ".fftdat",1
SINE_TABLE:  .usect   ".fftdat",1
BIT_REVERSE: .usect   ".fftdat",1
SEPARATION:  .usect   ".fftdat",1
*
*  Initialize C Function
*
          .sect    ".ffttxt"
_ffft_rl: PUSH     FP             ; Preserve C environment.
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
          LDI       SP,FP
          PUSH      R4
          PUSH      R5
          PUSH      R6
          PUSHF     R6
          PUSH      R7
          PUSHF     R7
          PUSH      AR4
          PUSH      AR5
          PUSH      AR6
          PUSH      AR7
          PUSH      DP
          LDP       FFT_SIZE        ; Initialize DP pointer.
          .if       .REGPARM==0     ; arguments passed in stack
          LDA       *-FP(2),AR2
          LDI       *-FP(3),R2
          LDI       *-FP(4),R3
          LDI       *-FP(5),RC
          LDI       *-FP(6),RS
          LDI       *-FP(7),RE
          .endif
          STI       AR2,@FFT_SIZE
          STI       R2,@LOG_SIZE
          STI       R3,@SOURCE_ADDR
          STI       RC,@DEST_ADDR
          STI       RS,@SINE_TABLE
          STI       RE,@BIT_REVERSE
;
; Check Bit Reversing Mode (on or off).
;
; BIT_REVERSING =  0, then OFF (no bit reversing).
; BIT_REVERSING <> 0, Then ON.
;
          LDI       @BIT_REVERSE,R0
          BZ        MOVE_DATA
;
; Check Bit Reversing Type.
;
; If SourceAddr =  DestAddr, Then In Place Bit Reversing.
; If SourceAddr <> DestAddr, Then Standard Bit Reversing.
;
          LDI       @SOURCE_ADDR,R0
          CMPI      @DEST_ADDR,R0
          BEQ       IN_PLACE
;
; Bit reversing Type 1 (From Source to Destination).
;
; NOTE: abs(SOURCE_ADDR – DEST_ADDR) must be > FFT_SIZE, this is not checked.
;
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
          LDI     @FFT_SIZE,R0
          SUBI    2,R0
          LDA     @FFT_SIZE,IR0
          LSH     -1,IR0          ;IRO = Half FFT size.
          LDA     @SOURCE_ADDR,AR0
          LDA     @DEST_ADDR,AR1
          LDF     *AR0++,R1
          RPTS    R0
          LDF     *AR0++,R1
||        STF     R1,*AR1++(IR0)B
          STF     R1,*AR1++(IR0)B
          BR      STARTB
;
;  In Place Bit Reversing.
;  Bit Reversing On Even Locations, 1st Half Only.
IN_PLACE: LDA     @FFT_SIZE,IR0
          LSH     -2,IR0          ;IRO = Quarter FFT size.
          LDA     2,IR1
          LDI     @FFT_SIZE,RC
          LSH     -2,RC
          SUBI    3,RC
          LDA     @DEST_ADDR,AR0
          LDA     AR0,AR1
          LDA     AR0,AR2
          NOP     *AR1++(IR0)B
          NOP     *AR2++(IR0)B
          LDF     *++AR0(IR1),R0
          LDF     *AR1,R1
          RPTBD   BITRV1
          CMPI    AR1,AR0         ;Xchange Locations only if AR0<AR1.
          LDFGT   R0,R1
          LDFGT   *AR1++(IR0)B,R1
          LDF     *++AR0(IR1),R0
||        STF     R0,*AR0
          LDF     *AR1,R1
||        STF     R1,*AR2++(IR0)B
          CMPI    AR1,AR0
          LDFGT   R0,R1

BITRV1:   LDFGT   *AR1++(IR0)B,R0
          STF     R0,*AR0
          STF     R1,*AR2
;
;Perform Bit Reversing, Odd Locations, 2nd Half Only
;
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
              LDI     @FFT_SIZE,RC
              LSH     -1,RC
              LDA     @DEST_ADDR,AR0
              ADDI    RC,AR0
              ADDI    1,AR0
              LDA     AR0,AR1
              LDA     AR0,AR2
              LSH     -1,RC
              SUBI    3,RC
              NOP     *AR1++(IR0)B
              NOP     *AR2++(IR0)B
              LDF     *++AR0(IR1),R0
              LDF     *AR1,R1
              RPTBD   BITRV2
              CMPI    AR1,AR0             ;Xchange Locations only if AR0<AR1
              LDFGT   R0,R1
              LDFGT   *AR1++(IR0)B,R1
              LDF     *++AR0(IR1),R0
||            STF     R0,*AR0
              LDF     *AR1,R1
||            STF     R1,*AR2++(IR0)B
              CMPI    AR1,AR0
              LDFGT   R0,R1
BITRV2:       LDFGT   *AR1++(IR0)B,R0
              STF     R0,*AR0
              STF     R1,*AR2
;Perform Bit Reversing, Odd Locations, 1st Half Only
              LDI     @FFT_SIZE,RC
              LSH     -1,RC
              LDA     RC,IR0
              LDA     @DEST_ADDR,AR0
              LDA     AR0,AR1
              ADDI    1,AR0
              ADDI    IR0,AR1
              LSH     -1,RC
              LDA     RC,IR0
              SUBI    2,RC
              RPTBD   BITRV3
              NOP                         ;Note: could be instruction
              LDF     *AR0,R0
              LDF     *AR1,R1
              LDF     *++AR0(IR1),R0
||            STF     R0,*AR1++(IR0)B
BITRV3:       LDF     *AR1,R1
||            STF     R1,*-AR0(IR1)
              STF     R0,*AR1
||            STF     R1,*AR0
              BR      STARTB
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
;
; Check Data Source Locations.
;
; If SourceAddr =  DestAddr, Then do nothing.
; If SourceAddr <> DestAddr, Then move data.
;
MOVE_DATA: LDI      @SOURCE_ADDR,R0
           CMPI     @DEST_ADDR,R0
           BEQ      STARTB
           LDI      @FFT_SIZE,R0
           SUBI     2,R0
           LDA      @SOURCE_ADDR,AR0
           LDA      @DEST_ADDR,AR1
           LDF      *AR0++,R1
           RPTS     R0
           LDF      *AR0++,R1
||         STF      R1,*AR1++
           STF      R1,*AR1
;
; Perform first and second FFT loops.
;
;   |  AR1 -> |__I1__| 0  <-  [X(I1) + X(I2)] + [X(I3) + X(I4)]
;   |  AR2 -> |__I2__| 1  <-  [X(I1) - X(I2)]
;   |  AR3 -> |__I3__| 2  <-  [X(I1) + X(I2)] - [X(I3) + X(I4)]
;   |_ AR4 -> |__I4__| 3  <- -[X(I3) - X(I4)]
;      AR1 -> |______| 4
;              |   .  |
;                  .
;                  .
;                 \|/
;
STARTB:    LDA      @DEST_ADDR,AR1
           LDA      AR1,AR2
           LDA      AR1,AR3
           LDA      AR1,AR4
           ADDI     1,AR2
           ADDI     2,AR3
           ADDI     3,AR4
           LDA      4,IR0
           LDI      @FFT_SIZE,RC
           LSH      -2,RC
           SUBI     2,RC
           LDF      *AR2,R0               ; R0 = X(I2)
||         LDF      *AR3,R1               ; R1 = X(I3)
           ADDF3    R1,*AR4,R4            ; R4 = X(I3) + X(I4)
           SUBF3    R1,*AR4++(IR0),R5     ; R5 = -[X(I3) - X(I4)] --+
           SUBF3    R0,*AR1,R6            ; R6 = X(I1) - X(I2) --+  |
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
                                                ;                          |   |
          RPTBD     LOOP1_2                     ;                          |   |
          ADDF3     R0,*AR1++(IR0),R7           ; R7 = X(I1) + X(I2)       |   |
          ADDF3     R7,R4,R2                    ; R2 = R7 + R4 -----+      |   |
          SUBF3     R4,R7,R3                    ; R3 = R7 - R4 --+  |      |   |
                                                ;                |  |      |   |
          LDF       *+AR2(IR0),R0               ;                |  |      |   |
||        LDF       *+AR3(IR0),R1               ;                |  |      |   |
          ADDF3     R1,*AR4,R4                  ;                |  |      |   |
||        STF       R3,*AR3++(IR0)              ; X(I3) <--------+  |      |   |
          SUBF3     R1,*AR4++(IR0),R5           ;                   |      |   |
||        STF       R5,*-AR4(IR0)               ; X(I4) <-----------|--|--|--+
          SUBF3     R0,*AR1,R6                  ;                   |  |
||        STF       R6,*AR2++(IR0)              ; X(I2) <-----------|--+
          ADDF3     R0,*AR1++(IR0),R7           ;                   |
||        STF       R2,*-AR1(IR0)               ; X(I1) <-----------+
          ADDF3     R7,R4,R2
LOOP1_2:  SUBF3     R4,R7,R3
          STF       R3,*AR3
||        STF       R5,*-AR4(IR0)
          STF       R6,*AR2
||        STF       R2,*-AR1(IR0)
;
; Perform Third FFT Loop.
;
; Part A:
;      _
;   |      AR1 ->    ______
;   |               |__I1__| 0  <-  X(I1) + X(I3)
;   |               |______| 1
;   |               |__I2__| 2
;   |               |______| 3
;   |      AR2 ->   |__I3__| 4  <-  X(I1) - X(I3)
;   |               |______| 5
;   |      AR3 ->   |__I4__| 6  <- -X(I4)
;   |_              |______| 7
;          AR1 ->   |______| 8
;                   |______| 9
;                   |   .  |
;                      .
;                      .
;                     \|/
          LDA       @DEST_ADDR,AR1
          LDA       AR1,AR2
          LDA       AR1,AR3
          ADDI      4,AR2
          ADDI      6,AR3
          LDA       8,IR0
          LDI       @FFT_SIZE,RC
          LSH       -3,RC
          SUBI      2,RC
          RPTBD     LOOP3_A
          SUBF3     *AR2,*AR1,R1
          ADDF3     *AR2,*AR1,R2
          NEGF      *AR3,R3
          LDF       *+AR2(IR0),R0               ; R0 =  X(I3)
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
||        STF     R2,*AR1++(IR0)
          SUBF3   R0,*AR1,R1              ; R1 =  X(I1) – X(I3) -----+
||        STF     R1,*AR2++(IR0)          ;                          |
          ADDF3   R0,*AR1,R2              ; R2 =  X(I1) + X(I3) --+  |
||        STF     R3,*AR3++(IR0)          ;                       |  |
LOOP3_A:  NEGF    *AR3,R3                 ; R3 = –X(I4) --+       |  |
                                          ;               |       |  |
          STF     R2,*AR1                 ; X(I1) <-------|-------+  |
||        STF     R1,*AR2                 ; X(I3) <-------|----------+
          STF     R3,*AR3                 ; X(I4) <-------+
;
; Part B:
;
;          _         ______
;         |          |______| 0
;         | AR0 –>   |__I1__| 1 <-  X(I1) + [X(I3)*COS + X(I4)*COS]
;         |          |______| 2
;         | AR1 –>   |__I2__| 3 <-  X(I1) – [X(I3)*COS + X(I4)*COS]
;         |          |______| 4
;         | AR2 –>   |__I3__| 5 <- –X(I2) – [X(I3)*COS – X(I4)*COS]
;         |          |______| 6
;         |_AR3 –>   |__I4__| 7 <-  X(I2) – [X(I3)*COS – X(I4)*COS]
;                    |______| 8
;          AR0 –>    |______| 9     NOTE: COS(2*pi/8) = SIN(2*pi/8)
;                    |   .  |
;                      .
;                      .
;                     \|/
;
            LDI     @FFT_SIZE,RC
            LSH     –3,RC
            LDA     RC,IR1
            SUBI    3,RC
            LDA     8,IR0
            LDA     @DEST_ADDR,AR0
            LDA     AR0,AR1
            LDA     AR0,AR2
            LDA     AR0,AR3
            ADDI    1,AR0
            ADDI    3,AR1
            ADDI    5,AR2
            ADDI    7,AR3
            LDA     @SINE_TABLE,AR7     ; Initialize table pointers.
            LDF     *++AR7(IR1),R7      ; R7   = COS(2*pi/8)
                                        ; *AR7 = COS(2*pi/8)
            MPYF3   *AR7,*AR2,R0        ; R0 =  X(I3)*COS
            MPYF3   *AR3,R7,R1          ; R5 =  X(I4)*COS
            ADDF3   R0,R1,R2            ; R2 =  [X(I3)*COS + X(I4)*COS]
            MPYF3   *AR7,*+AR2(IR0),R0
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
||            SUBF3    R0,R1,R3            ; R3 = -[X(I3)*COS - X(I4)*COS]
              SUBF3    *AR1,R3,R4          ; R4 = -X(I2) + R3 --+
                                           ;                    |
              RPTBD    LOOP3_B             ;                    |
              ADDF3    *AR1,R3,R4          ; R4 =  X(I2) + R3 --|--+
||            STF      R4,*AR2++(IR0)      ; X(I3) <------------+  |
              SUBF3    R2,*AR0,R4          ; R4 =  X(I1) - R2 --+  |
||            STF      R4,*AR3++(IR0)      ; X(I4) <------------|--+
              ADDF3    *AR0,R2,R4          ; R4 =  X(I1) + R2 --|--+
||            STF      R4,*AR1++(IR0)      ; X(I2) <------------+  |
              MPYF3    *AR3,R7,R1          ;                       |
||            STF      R4,*AR0++(IR0)      ; X(I1) <---------------+
              ADDF3    R0,R1,R2
              MPYF3    *AR7,*+AR2(IR0),R0
||            SUBF3    R0,R1,R3
              SUBF3    *AR1,R3,R4
              ADDF3    *AR1,R3,R4
||            STF      R4,*AR2++(IR0)
              SUBF3    R2,*AR0,R4
||            STF      R4,*AR3++(IR0)
LOOP3_B:      ADDF3    *AR0,R2,R4
||            STF      R4,*AR1++(IR0)
              MPYF3    *AR3,R7,R1
||            STF      R4,*AR0++(IR0)
              ADDF3    R0,R1,R2
              SUBF3    R0,R1,R3
              SUBF3    *AR1,R3,R4
              ADDF3    *AR1,R3,R4
||            STF      R4,*AR2
              SUBF3    R2,*AR0,R4
||            STF      R4,*AR3
              ADDF3    *AR0,R2,R4
||            STF      R4,*AR1
              STF      R4,*AR0
;
; Perform Fourth FFT Loop.
;
; Part A:
;   _          _____
;   |     AR1->    |__I1__| 0  <-  X(I1) + X(I3)
;   |              |______| 1
;   |              |______| 2
;   |              |______| 3
;   |              |__I2__| 4
;   |              |______| 5
;   |              |______| 6
;   |              |______| 7
;   |     AR2->    |__I3__| 8  <-  X(I1) - X(I3)
;   |              |______| 9
;   |              |______| 10
;   |              |______| 11
;   |     AR3->    |__I4__| 12 <- -X(I4)
;   |              |______| 13
;   |              |______| 14
;   |_             |______| 15
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
;          AR1->    |__I5__| 16
;             |_____| 17
;             |   .  |
;                 .
;                    .
;                   \|/
           LDA      @DEST_ADDR,AR1
           LDA      AR1,AR2
           LDA      AR1,AR3
           ADDI     8,AR2
           ADDI     12,AR3
           LDA      16,IR0
           LDI      @FFT_SIZE,RC
           LSH      -4,RC
           SUBI     2,RC
           RPTBD    LOOP4_A
           SUBF3    *AR2,*AR1,R1
           ADDF3    *AR2,*AR1,R2
           NEGF     *AR3,R3
           LDF      *+AR2(IR0),R0  ;R0 =  X(I3)
||         STF      R2,*AR1++(IR0)
           SUBF3    R0,*AR1,R1              ;R1 =  X(I1) - X(I3) -----+
||         STF      R1,*AR2++(IR0)          ;                         |
           ADDF3    R0,*AR1,R2              ;R2 =  X(I1) + X(I3) --+  |
||         STF      R3,*AR3++(IR0)          ;                      |  |
LOOP4_A:   NEGF     *AR3,R3                 ;R3 = -X(I4) --+       |  |
                                            ;              |       |  |
           STF      R2,*AR1                 ;X(I1) <-------|-------+  |
||         STF      R1,*AR2                 ;X(I3) <-------|----------+
           STF      R3,*AR3                 ;X(I4) <-------+
;
; Part B:
;    _                 ___________
;   |                 |___________| 0
;   |      AR0  ->    |__I1_(3rd)_| 1  <-  X(I1) + [X(I3)*COS + X(I4)*SIN]
;   |                 |__I1_(2nd)_| 2       .
;   |                 |__I1_(1st)_| 3       .
;   |                 |___________| 4
;   |                 |__I2_(1st)_| 5       .
;   |                 |__I2_(2nd)_| 6       .
;   |      AR1  ->    |__I2_(3rd)_| 7  <-  X(I1) - [X(I3)*COS + X(I4)*SIN]
;   |                 |___________| 8
;   |      AR2  ->    |__I3_(3rd)_| 9  <- -X(I2) - [X(I3)*SIN - X(I4)*COS]
;   |                 |__I3_(2nd)_| 10      .
;   |      AR4  ->    |__I3_(1st)_| 11      .
;   |                 |___________| 12
;   |                 |__I4_(1st)_| 13      .
;   |                 |__I4_(2nd)_| 14      .
;   |_     AR3  ->    |__I4_(3rd)_| 15 <-  X(I2) - [X(I3)*SIN - X(I4)*COS]
;             |_____________| 16
;          AR0  ->    |___________| 17
;             |       .     |
;                       .
;                       .
;                      \|/
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
          LDI     @FFT_SIZE,RC
          LSH     -4,RC
          LDA     RC,IR1
          LDA     2,IR0
          SUBI    3,RC
          LDA     @DEST_ADDR,AR0
          LDA     AR0,AR1
          LDA     AR0,AR2
          LDA     AR0,AR3
          LDA     AR0,AR4
          ADDI    1,AR0
          ADDI    7,AR1
          ADDI    9,AR2
          ADDI    15,AR3
          ADDI    11,AR4
          LDA     @SINE_TABLE,AR7
          LDF     *++AR7(IR1),R7          ;R7 = SIN(1*[2*pi/16])
                                          ;*AR7 = COS(3*[2*pi/16])
          LDA     AR7,AR6
          LDF     *++AR6(IR1),R6          ;R6 = SIN(2*[2*pi/16])
                                          ;*AR6 = COS(2*[2*pi/16])
          LDA     AR6,AR5
          LDF     *++AR5(IR1),R5          ;R5 = SIN(3*[2*pi/16])
                                          ;*AR5 = COS(1*[2*pi/16])
          LDA     16,IR1
          MPYF3   *AR7,*AR4,R0            ;R0 =  X(I3)*COS(3)
          MPYF3   *++AR2(IR0),R5,R4       ;R4 =  X(I3)*SIN(3)
          MPYF3   *--AR3(IR0),R5,R1       ;R1 =  X(I4)*SIN(3)
          MPYF3   *AR7,*AR3,R0            ;R0 =  X(I4)*COS(3)
||        ADDF3   R0,R1,R2                ;R2 =  [X(I3)*COS + X(I4)*SIN]
          MPYF3   *AR6,*-AR4,R0
||        SUBF3   R4,R0,R3                ;R3 = -[X(I3)*SIN - X(I4)*COS]
          SUBF3   *--AR1(IR0),R3,R4       ;R4 = -X(I2) + R3 --+
          ADDF3   *AR1,R3,R4              ;R4 =  X(I2) + R3 --|--+
||        STF     R4,*AR2--               ;X(I3) <------------+  |
          SUBF3   R2,*++AR0(IR0),R4       ;R4 =  X(I1) - R2 --+  |
||        STF     R4,*AR3                 ;X(I4) <---------------|--+
          ADDF3   *AR0,R2,R4              ;R4 =  X(I1) + R2 --|---+
||        STF     R4,*AR1                 ;X(I2) <------------+   |
                                          ;                       |
          MPYF3   *++AR3,R6,R1            ;                       |
||        STF     R4,*AR0                 ;X(I1) <----------------+
          ADDF3   R0,R1,R2
          MPYF3   *AR5,*-AR4(IR0),R0
||        SUBF3   R0,R1,R3
          SUBF3   *++AR1,R3,R4
          ADDF3   *AR1,R3,R4
||        STF     R4,*AR2
          SUBF3   R2,*--AR0,R4
||        STF     R4,*AR3
          ADDF3   *AR0,R2,R4
||        STF     R4,*AR1
          MPYF3   *--AR2,R7,R4
||        STF     R4,*AR0
          MPYF3   *++AR3,R7,R1
          MPYF3   *AR5,*AR3,R0
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
||          ADDF3     R0,R1,R2
            MPYF3     *AR7,*++AR4(IR1),R0
||          SUBF3     R4,R0,R3
            SUBF3     *++AR1,R3,R4
            RPTBD     LOOP4_B
            ADDF3     *AR1,R3,R4
||          STF       R4,*AR2++(IR1)
            SUBF3     R2,*--AR0,R4
||          STF       R4,*AR3++(IR1)
            ADDF3     *AR0,R2,R4
||          STF       R4,*AR1++(IR1)
            MPYF3     *++AR2(IR0),R5,R4
||          STF       R4,*AR0++(IR1)
            MPYF3     *--AR3(IR0),R5,R1
            MPYF3     *AR7,*AR3,R0
||          ADDF3     R0,R1,R2
            MPYF3     *AR6,*-AR4,R0
||          SUBF3     R4,R0,R3
            SUBF3     *--AR1(IR0),R3,R4
            ADDF3     *AR1,R3,R4
||          STF       R4,*AR2--
            SUBF3     R2,*++AR0(IR0),R4
||          STF       R4,*AR3
            ADDF3     *AR0,R2,R4
||          STF       R4,*AR1
            MPYF3     *++AR3,R6,R1
||          STF       R4,*AR0
            ADDF3     R0,R1,R2
            MPYF3     *AR5,*-AR4(IR0),R0
||          SUBF3     R0,R1,R3
            SUBF3     *++AR1,R3,R4
            ADDF3     *AR1,R3,R4
||          STF       R4,*AR2
            SUBF3     R2,*--AR0,R4
||          STF       R4,*AR3
            ADDF3     *AR0,R2,R4
||          STF       R4,*AR1
            MPYF3     *--AR2,R7,R4
||          STF       R4,*AR0
            MPYF3     *++AR3,R7,R1
            MPYF3     *AR5,*AR3,R0
||          ADDF3     R0,R1,R2
            MPYF3     *AR7,*++AR4(IR1),R0
||          SUBF3     R4,R0,R3
            SUBF3     *++AR1,R3,R4
            ADDF3     *AR1,R3,R4
||          STF       R4,*AR2++(IR1)
            SUBF3     R2,*--AR0,R4
||          STF       R4,*AR3++(IR1)
LOOP4_B:    ADDF3     *AR0,R2,R4
||          STF       R4,*AR1++(IR1)
            MPYF3     *++AR2(IR0),R5,R4
||          STF       R4,*AR0++(IR1)
            MPYF3     *--AR3(IR0),R5,R1
            MPYF3     *AR7,*AR3,R0
||          ADDF3     R0,R1,R2
            MPYF3     *AR6,*-AR4,R0
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
||        SUBF3    R4,R0,R3
          SUBF3    *--AR1(IR0),R3,R4
          ADDF3    *AR1,R3,R4
||        STF      R4,*AR2--
          SUBF3    R2,*++AR0(IR0),R4
||        STF      R4,*AR3
          ADDF3    *AR0,R2,R4
||        STF      R4,*AR1
          MPYF3    *++AR3,R6,R1
||        STF      R4,*AR0
          ADDF3    R0,R1,R2
          MPYF3    *AR5,*-AR4(IR0),R0
||        SUBF3    R0,R1,R3
          SUBF3    *++AR1,R3,R4
          ADDF3    *AR1,R3,R4
||        STF      R4,*AR2
          SUBF3    R2,*--AR0,R4
||        STF      R4,*AR3
          ADDF3    *AR0,R2,R4
||        STF      R4,*AR1
          MPYF3    *--AR2,R7,R4
||        STF      R4,*AR0
          MPYF3    *++AR3,R7,R1
          MPYF3    *AR5,*AR3,R0
||        ADDF3    R0,R1,R2
          SUBF3    R4,R0,R3
          SUBF3    *++AR1,R3,R4
          ADDF3    *AR1,R3,R4
||        STF      R4,*AR2
          SUBF3    R2,*--AR0,R4
||        STF      R4,*AR3
          ADDF3    *AR0,R2,R4
||        STF      R4,*AR1
          STF      R4,*AR0
;
; Perform Remaining FFT loops (loop 4 onwards).
;
;                                  LOOP
;                                  1st  2nd
;                                   \/   \/
;        _          _____________   \/   \/
;         |        |____X'(I1)____| 0    0   <-  X'(I1) + X'(I3)
;         |  AR1-> |__X(I1)_(1st)_| 1    1   <-  X(I1) + [X(I3)*COS + X(I4)*SIN]
;         |        |__X(I1)_(2nd)_| 2    2        .
;         |        |__x(I1)_(3rd)_| 3    3        .
;         |        |       .      |
;         |        |       .
;         |  A ->  |______________|
;         |        |____X'(I2)____| 8    16
;         |  B ->  |       .      |
;         |        |       .
;         |        |______________|
;         |        |__X(I2)_(3rd)_| 13   29       .
;         |        |__X(I2)_(2nd)_| 14   30       .
;         |  AR2-> |__X(I2)_(1st)_| 15   31  <-  X(I1) - [X(I3)*COS + X(I4)*SIN]
;         |        |____X'(I3)____| 16   32  <-  X'(I1) - X'(I3)
;         |  AR3-> |__X(I3)_(1st)_| 17   33  <- -X(I2) - [X(I3)*SIN - X(I4)*COS]
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
;          |          |__X(I3)_(2nd)_| 18  34       .
;          |          |__X(I3)_(3rd)_| 19  35       .
;          |          |       .      |
;          |          |          .
;          |  C ->    |______________|
;          |          |____X'(I4)____| 24  48 <- -X'(I4)
;          |  D ->    |       .      |
;          |          |          .
;          |          |______________|
;          |          |__X(I4)_(3rd)_| 29  61       .
;          |          |__X(I4)_(2nd)_| 30  62       .
;          |_ AR4->   |__X(I4)_(1st)_| 31  63 <-  X(I2) - [X(I3)*SIN - X(I4)*COS]
;                     |______________| 32  64
;             AR1->   |______________| 33  65
;                     |       .      |
;                              .
;                              .
;                            \|/
;
           LDA       @FFT_SIZE,IR0
           LSH       -2,IR0
           STI       IR0,@SEPARATION
           LSH       -2,IR0
           LDI       5,R5
           LDI       3,R7
           LDI       16,R6
           LDA       @DEST_ADDR,AR5
           LDA       @DEST_ADDR,AR1
           LSH       -1,IR0
           LSH       1,R7
LOOP:      ADDI      1,R7
           LSH       1,R6
           LDA       AR1,AR4
           ADDI      R7,AR1                    ;AR1 points at A.
           LDA       AR1,AR2
           ADDI      2,AR2                     ;AR2 points at B.
           ADDI      R6,AR4
           SUBI      R7,AR4                    ;AR4 points at D.
           LDA       AR4,AR3
           SUBI      2,AR3                     ;AR3 points at C.
           LDA       @SINE_TABLE,AR0           ;AR0 points at SIN/COS table.
           LDA       R7,IR1
           LDI       R7,RC

INLOP:     ADDF3     *--AR1(IR1),*++AR2(IR1),R0   ;R0 = X'(I1) + X'(I3) --+
           SUBF3     *--AR3(IR1),*AR1++,R1        ;R1 = X'(I1) - X'(I3) -+|
           NEGF      *--AR4,R2                    ;R2 = -X'(I4) --+      ||
||         STF       R0,*-AR1                     ;X'(I1) <-------|------|+
           STF       R1,*AR2--                    ;X'(I3) <-------|------+
||         STF       R2,*AR4++(IR1)               ;X'(I4) <-------+
           LDA       @SEPARATION,IR1              ;IR1=SEPARATION BETWEEN SIN/COS
                                                   TABLES
           SUBI      3,RC
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
          MPYF3     *++AR0(IR0),*AR4,R4    ;R4 = X(I4)*SIN
          MPYF3     *AR0,*++AR3,R1         ;R1 = X(I3)*SIN
          MPYF3     *++AR0(IR1),*AR4,R0    ;R0 = X(I4)*COS
          MPYF3     *AR0,*AR3,R0           ;R0 = X(I3)*COS
||        SUBF3     R1,R0,R3               ;R3 = -[X(I3)*SIN - X(I4)*COS]
          MPYF3     *++AR0(IR0),*-AR4,R0
||        ADDF3     R0,R4,R2               ;R2 = X(I3)*COS + X(I4)*SIN
          SUBF3     *AR2,R3,R4             ;R4 = R3 - X(I2) --*
                                           ;                  |
          RPTBD     IN_BLK                 ;                  |
          ADDF3     *AR2,R3,R4             ;R4 = R3 + X(I2) --|--*
||        STF       R4,*AR3++              ;X(I3) <-----------*  |
          SUBF3     R2,*AR1,R4             ;R4 = X(I1) - R2 --*  |
||        STF       R4,*AR4- -             ;X(I4) <-----------|--*
          ADDF3     *AR1,R2,R4             ;R4 = X(I1) + R2 --|--*
||        STF       R4,*AR2--              ;X(I2) <-----------*  |
          LDF       *-AR0(IR1),R3          ;                     |
          MPYF3     *AR4,R3,R4             ;                     |
||        STF       R4,*AR1++              ;X(I1) <--------------*
          MPYF3     *AR3,R3,R1
          MPYF3     *AR0,*AR3,R0
||        SUBF3     R1,R0,R3
          MPYF3     *++AR0(IR0),*-AR4,R0
||        ADDF3     R0,R4,R2
          SUBF3     *AR2,R3,R4
          ADDF3     *AR2,R3,R4
||        STF       R4,*AR3++
          SUBF3     R2,*AR1,R4
||        STF       R4,*AR4- -
IN_BLK:   ADDF3     *AR1,R2,R4
||        STF       R4,*AR2- -
          LDF       *-AR0(IR1),R3
          MPYF3     *AR4,R3,R4
||        STF       R4,*AR1++
          MPYF3     *AR3,R3,R1
          MPYF3     *AR0,*AR3,R0
||        SUBF3     R1,R0,R3
          LDA       R6,IR1
          ADDF3     R0,R4,R2
          SUBF3     *AR2,R3,R4
          ADDF3     *AR2,R3,R4
||        STF       R4,*AR3++(IR1)
          SUBF3     R2,*AR1,R4
||        STF       R4,*AR4++(IR1)
          ADDF3     *AR1,R2,R4
||        STF       R4,*AR2++(IR1)
          STF       R4,*AR1++(IR1)
          SUBI3     AR5,AR1,R0
          CMPI      @FFT_SIZE,R0
          BLTD      INLOP                  ;LOOP BACK TO THE INNER LOOP
          LDA       @SINE_TABLE,AR0        ;AR0 POINTS TO SIN/COS TABLE
```

Example 6–17. Real Forward Radix-2 FFT (Continued)

```
          LDA       R7,IR1
          LDI       R7,RC
          ADDI      1,R5
          CMPI      @LOG_SIZE,R5
          BLED      LOOP
          LDA       @DEST_ADDR,AR1
          LSH       -1,IR0
          LSH       1,R7
;
; Return to C environment.
;
ENDB:     POP       DP                    ;Restore C environment variables.
          POP       AR7
          POP       AR6
          POP       AR5
          POP       AR4
          POPF      R7
          POP       R7
          POPF      R6
          POP       R6
          POP       R5
          POP       R4
          POP       FP
          RETS
          .end
*
* No more.
*
*******************************************************************************
```

Example 6–18. Real Inverse Radix-2 FFT

```
*******************************************************************************
*
*  FILENAME     : IFFT_RL.ASM
*  DESCRIPTION  : INVERSE FFT FOR TMS320C40
*  DATE         : 1/19/93
*  VERSION      : 2.0
*
*******************************************************************************
*
*  VERSION   DATE        COMMENTS
*  -------   ----        --------
*   1.0      2/18/92    DANIEL MAZZOCCO(TI Houston):
*                       Original Release (C30 version)
*                       Started from forward real FFT routine written by Alex
*                       Tessarolo, rev 2.0 .
*   2.0      1/19/93    ROSEMARIE PIEDRA(TI Houston): C40 porting started from
*                       C30 inverse real FFT version 1.0 (C30). Expanded calling
*                       conventions to registers for parameter passing.
*
*******************************************************************************
*
*  SYNOPSIS:
*
*  int ifft_rl(FFT_SIZE,LOG_SIZE,SOURCE_ADDR,DEST_ADDR,SINE_TABLE,BIT_REVERSE);
*                 ar2      r2         r3        rc        rs          re
*
*         int      FFT_SIZE       ; 64, 128, 256, 512, 1024, ...
*         int      LOG_SIZE       ;  6,   7,   8,   9,   10, ...
*         float    *SOURCE_ADDR   ; Points to where data is originated
*                                 ; and operated on.
*         float    *DEST_ADDR     ; Points to where data will be stored.
*         float    *SINE_TABLE    ; Points to the SIN/COS table.
*         int      BIT_REVERSE    ; =  0, Bit Reversing is disabled.
*                                 ; <> 0, Bit Reversing is enabled.
*
*  NOTE:   1) If SOURCE_ADDR = DEST_ADDR, then in place bit reversing is
*             performed, if enabled (more processor intensive).
*          2) FFT_SIZE must be >= 64 (this is not checked).
*
*******************************************************************************
*
*  DESCRIPTION:
*
*  Generic function to do an inverse radix–2 FFT computation on the C40.
*  The input data array is FFT_SIZE–long with real and imaginary  points R and
*  I as follows:
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
*
*         SOURCE_ADDR[0]     –> R(0)
*                            R(1)
*                            R(2)
*                            R(3)
*                            .
*                            .
*                            R(FFT_SIZE/2)
*                            I(FFT_SIZE/2 – 1)
*                            .
*                            .
*                            I(2)
*                            SOURCE_ADDR[FFT_SIZE – 1] –> I(1)
*  The output data array will contain only real values. Bit reversal is
*  optionally implemented at the end of the function.
*
*  The sine/cosine table for the twiddle factors is expected to be supplied in
*  the following format:
*
*   SINE_TABLE[0]                  –>      sin(0*2*pi/FFT_SIZE)
*                                          sin(1*2*pi/FFT_SIZE)
*                                               .
*                                               .
*                                          sin((FFT_SIZE/2–2)*2*pi/FFT_SIZE)
*   SINE_TABLE[FFT_SIZE/2–1]       –>      sin((FFT_SIZE/2–1)*2*pi/FFT_SIZE)
*
*  NOTE: The table is the first half period of a sine wave.
*
*************************************************************************
*
*  NOTE:  1.Calling C program can be compiled using either large or small model.
*           Both calling conventions methods: stack or register for parameter
*           passing are supported.
*
*         2. Sections needed in linker command file: .ffttxt  : fft code
*                                                    .fftdat  : fft data
*
*         3.The SOURCE_ADDR must be aligned such that the first LOG_SIZE bits
*           are zero (this is not checked by the program).
*
*  CAUTION: DP initialized only once in the program. Be wary with interrupt
*           service routines.Ensure interrupt service routines save DP pointer.
*
*************************************************************************
*
*  REGISTERS USED: R0, R1, R2, R3, R4, R5, R6, R7
*                  AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7
*                  IR0, IR1
*                  RC, RS, RE
*                  DP
*
*  MEMORY REQUIREMENTS:  Program = 322 Words (approximately)
*                        Data    =   7 Words
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
*                        Stack   =  12 Words
*
*************************************************************************
*
*  BENCHMARKS:   Assumptions     – Program in RAM0
*                                – Reserved data in RAM0
*                                – Stack on Local/Global Bus RAM
*                                – Sine/Cosine tables in RAM0
*                                – Processing and data destination in RAM1.
*                                – Local/Global Bus RAM, 0 wait state.
*
*  FFT Size        Bit Reversing   Data Source     Cycles(C30)
*  --------        -------------   -----------     -----------
*   1024            OFF             RAM1            25120 approx.
*
*  Note: This number does not include the C callable overheads.
*        This benchmark is the number of cycles between labels STARTB and ENDB
*
*  NOTE: If .ffttxt is located in external SRAM, enable cache for faster
*  performance
*
*************************************************************************

FP        .set        AR3
          .global     ifft_rl             ;Entry execution point.
          .global     STARTB,ENDB
FFT_SIZE: .usect      ".ifftdat",1        ;Reserve memory for arguments.
LOG_SIZE: .usect      ".ifftdat",1
SOURCE_ADDR:  .usect  ".ifftdat",1
DEST_ADDR:    .usect  ".ifftdat",1
SINE_TABLE:   .usect  ".ifftdat",1
BIT_REVERSE:  .usect  ".ifftdat",1
SEPARATION:   .usect  ".ifftdat",1

;
; Initialize C Function.
;
          .sect    ".ffttxt"
_ifft_rl: PUSH     FP                     ;Preserve C environment.
          LDI      SP,FP
          PUSH     R4
          PUSH     R5
          PUSH     R6
          PUSHF    R6
          PUSH     R7
          PUSHF    R7
          PUSH     AR4
          PUSH     AR5
          PUSH     AR6
          PUSH     AR7
          PUSH     DP
          LDP FFT_SIZE                    ;Initialize DP pointer.
          .if .REGPARM == 0               ;arguments passed in stack
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
          LDA       *-FP(2),AR2
          LDI       *-FP(3),R2
          LDI       *-FP(4),R3
          LDI       *-FP(5),RC
          LDI       *-FP(6),RS
          LDI       *-FP(7),RE
          .endif
          STI       AR2,@FFT_SIZE
          STI       R2,@LOG_SIZE
          STI       R3,@SOURCE_ADDR
          STI       RC,@DEST_ADDR
          STI       RS,@SINE_TABLE
          STI       RE,@BIT_REVERSE
;
; Perform Last FFT loops first (loop 2 onwards).
;
;                                                LOOP
;                                              1st 2nd
;          _          _______________ \/  \/
;         |          |____X'(I1)____| 0   0  <-  X'(I1) + X'(I3)
;         |  AR1-> |__X(I1)_(1st)_| 1   1  <-  X(I1) + X(I2)
;         |          |__X(I1)_(2nd)_| 2   2       .
;         |          |__x(I1)_(3rd)_| 3   3       .
;         |          |       .      |
;         |                  .
;         |  A ->  |_______________|
;         |          |____X'(I2)____| 8   16  <- X'(I2) * 2
;         |  B ->  |       .      |
;         |                  .
;         |          |_______________|
;         |          |__X(I2)_(3rd)_| 13  29      .
;         |          |__X(I2)_(2nd)_| 14  30      .
;         |  AR2-> |__X(I2)_(1st)_| 15  31 <-  X(I4) - X(I3)
;         |          |____X'(I3)____| 16  32 <-  X'(I1) - X'(I3)
;         |  AR3-> |__X(I3)_(1st)_| 17  33 <-
[X(I1)-X(I2)]*COS-[X(I3)+X(I4)]*SIN
;         |          |__X(I3)_(2nd)_| 18  34      .
;         |          |__X(I3)_(3rd)_| 19  35      .
;         |          |       .      |
;         |                   .
;         |  C ->  |_______________|
;         |          |____X'(I4)____| 24  48 <- -X'(I4) * 2
;         |  D ->  |       .      |
;         |                  .
;         |          |_______________|
;         |          |__X(I4)_(3rd)_| 29  61      .
;         |          |__X(I4)_(2nd)_| 30  62      .
;         |_ AR4-> |__X(I4)_(1st)_| 31  63 <-
[X(I1)-X(I2)]*SIN+[X(I3)+X(I4)]*COS
;                    |_______________| 32  64
;             AR1-> |_______________| 33  65
;                    |       .      |
;                           .
;                           .
;                          \|/
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
STARTB:   LDA      1,IR0                      ;step between two consecutive sines
          LDI      4,R5                       ;stage number from 4 to M.
          LDI      @FFT_SIZE,R7
          LSH      -2,R7                      ;R7 is FFT_SIZE/4-1 (ie 15 for 64
                                              ;pts)
          SUBI     1,R7                       ;and will be used to point at A & D.
          LDI      @FFT_SIZE,R6               ;R6 will be used to point at D.
          LSH      1,R6
          LDA      @SOURCE_ADDR,AR5
          LDA      @SOURCE_ADDR,AR1
LOOP:     LSH      -1,R6                      ;R6 is FFT_SIZE at the 1st loop
          LDA      AR1,AR4
          ADDI     R7,AR1                     ;AR1 points at A.
          LDA      AR1,AR2
          ADDI     2,AR2                      ;AR2 points at B.
          ADDI     R6,AR4
          SUBI     R7,AR4                     ;AR4 points at D.
          LDA      AR4,AR3
          SUBI     2,AR3                      ;AR3 points at C.
          LDA      R7,IR1
          LDI      R7,RC
INLOP:    ADDF3    *--AR1(IR1),*--AR3(IR1),R0 ; R0 = X'(I1) + X'(I3) ---+
          SUBF3    *AR3,*AR1,R1               ; R1 = X'(I1) - X'(I3) -+ |
          LDF      *--AR4,R2                  ;                       | |
||        STF      R0,*AR1++                  ; X'(I1) <--------------|-+
          MPYF     -2.0,R2                    ; R2 = -2*X'(I4) --+    |
          LDF      *--AR2,R3                  ;                  |    |
||        STF      R1,*AR3++                  ; X'(I3) <---------|----+
          MPYF     2.0,R3                     ; R3 = 2*X'(I2) -, |
          STF      R3,*AR2++(IR1)             ; X'(I2) <-------' |
||        STF      R2,*AR4++(IR1)             ; X'(I4) <---------+
          LDA      @FFT_SIZE,IR1              ; IR1=SEPARATION BETWEEN SIN/COS TBLS
          LDA      @SINE_TABLE,AR0            ; AR0 points at SIN/COS table
          LSH      -2,IR1
          SUBI     3,RC
          SUBF3    *AR2,*AR1,R3               ; R3 = X(I1)-X(I2)
          ADDF3    *AR1,*AR2,R2               ; R2 = X(I1)+X(I2) ---+
          MPYF3    R3,*++AR0(IR0),R1          ; R1 = R3*SIN         |
          LDF      *AR4,R4                    ; R4 = X(I4)          |
          MPYF3    R3,*++AR0(IR1),R0          ; R0 = R3*COS         |
||        SUBF3    *AR3,R4,R3                 ; R3 = X(I4)-X(I3)  --|--+
          ADDF3    R4,*AR3,R2                 ; R2 = X(I3)+X(I4)    |  |
||        STF      R2,*AR1++                  ; X(I1) <-------------+  |
          MPYF3    R2,*AR0--(IR1),R4          ; R4 = R2*COS            |
||        STF      R3,*AR2--                  ; X(I2) <----------------+
          RPTBD    IN_BLK
          ADDF3    R4,R1,R3                   ; R3 = R3*SIN + R2*COS -----+
          MPYF3    R2,*AR0,R1                 ; R1 = R2*SIN               |
||        STF      R3,*AR4--                  ; X(I4) <-------------------+
          SUBF3    R1,R0,R4                   ; R4 = R3*COS - R2*SIN
          SUBF3    *AR2,*AR1,R3               ; R3 = X(I1)-X(I2)
          ADDF3    *AR1,*AR2,R2               ; R2 = X(I1)+X(I2) ---+
          MPYF3    R3,*++AR0(IR0),R1          ; R1 = R3*SIN         |
||        STF      R4,*AR3++                  ; X(I3)               |
          LDF      *AR4,R4                    ; R4 = X(I4)          |
          MPYF3    R3,*++AR0(IR1),R0          ; R0 = R3*COS         |
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
||        SUBF3    *AR3,R4,R3                ; R3 = X(I4)–X(I3)  --|--+
          ADDF3    R4,*AR3,R2                ; R2 = X(I3)+X(I4)    |  |
||        STF      R2,*AR1++                 ; X(I1) <-------------+  |
          MPYF3    R2,*AR0– –(IR1),R4        ; R4 = R2*COS            |
||        STF      R3,*AR2– –                ; X(I2) <----------------+
          ADDF3    R4,R1,R3                  ; R3 = R3*SIN + R2*COS -----+
          MPYF3    R2,*AR0,R1                ; R1 = R2*SIN               |
||        STF      R3,*AR4– –                ; X(I4) <-------------------+
IN_BLK:   SUBF3    R1,R0,R4                  ; R4 = R3*COS – R2*SIN
          SUBF3    *AR2,*AR1,R3              ; R3 = X(I1)–X(I2)
          ADDF3    *AR1,*AR2,R2              ; R2 = X(I1)+X(I2) ---+
          MPYF3    R3,*++AR0(IR0),R1         ; R1 = R3*SIN         |
||        STF      R4,*AR3++                 ; X(I3)               |
          LDF      *AR4,R4                   ; R4 = X(I4)          |
          MPYF3    R3,*++AR0(IR1),R0         ; R0 = R3*COS         |
||        SUBF3    *AR3,R4,R3                ; R3 = X(I4)–X(I3)  --|--+
          ADDF3    R4,*AR3,R2                ; R2 = X(I3)+X(I4)    |  |
||        STF      R2,*AR1                   ; X(I1) <-------------+  |
          MPYF3    R2,*AR0– –(IR1),R4        ; R4 = R2*COS            |
||        STF      R3,*AR2                   ; X(I2) <----------------+
          LDA      R6,IR1                    ; Get prepared for the next
          ADDF3    R4,R1,R3                  ; R3 = R3*SIN + R2*COS -----+
          MPYF3    R2,*AR0,R1                ; R1 = R2*SIN               |
||        STF      R3,*AR4++(IR1)            ; X(I4) <-------------------+
          SUBF3    R1,R0,R4                  ; R4 = R3*COS – R2*SIN
          NEGF     *AR1++(IR1),R2            ; DUMMY
||        STF      R4,*AR3++(IR1)            ; X(I3)               |
          SUBI3    AR5,AR1,R0
          CMPI     @FFT_SIZE,R0
          BLTD     INLOP                     ; LOOP BACK TO THE INNER LOOP
          NOP      *AR2++(IR1)               ; DUMMY
          LDA      R7,IR1
          LDI      R7,RC
          ADDI     1,R5
          CMPI     @LOG_SIZE,R5              ; next stage if any left
          BLED     LOOP
          LDA      @SOURCE_ADDR,AR1
          LSH      1,IR0                     ; double step in sine table
          LSH      –1,R7
;
; Perform Third FFT loop .
;
;
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
; Part A:
;
;          _  _______
;         |  AR1-> |__I1__| 0  <-  X(I1) + X(I3)
;         |        |______| 1
;         |  AR2   |__I2__| 2  <-  2 * X(I2)
;         |        |______| 3
;         |  AR3-> |__I3__| 4  <-  X(I1) - X(I3)
;         |        |______| 5
;         |  AR4-> |__I4__| 6  <- -2 * X(I4)
;         |_       |______| 7
;            AR1-> |______| 8
;                  |______| 9
;                  |   .  |
;                     .
;                     .
;                    \|/
          LDA       @SOURCE_ADDR,AR1
          LDA       AR1,AR2
          LDA       AR1,AR3
          LDA       AR1,AR4
          ADDI      2,AR2
          ADDI      4,AR3
          LDI       @FFT_SIZE,RC
          LSH       -3,RC
          SUBI      1,RC
          RPTBD     LOOP3_A
          ADDI      6,AR4
          LDA       8,IR0
          LDA       @SINE_TABLE,AR0        ; AR0 points at SIN/COS table
          LDF       *AR3,R3
          ADDF3     R3,*AR1,R0             ; R0 = X'(I1) + X'(I3) ---+
          SUBF3     R3,*AR1,R1             ; R1 = X'(I1) - X'(I3) -+ |
          LDF       *AR4,R2                ;                       | |
||        STF       R0,*AR1++(IR0)         ; X'(I1) <--------------|-+
          MPYF      -2.0,R2                ; R2 = -2*X'(I4) --+    |
          LDF       *AR2,R3                ;                  |    |
||        STF       R1,*AR3++(IR0)         ; X'(I3) <---------|----+
          MPYF      2.0,R3                 ; R3 = 2*X'(I2) -, |
LOOP3_A:  STF       R3,*AR2++(IR0)         ; X'(I2) <-------' |
||        STF       R2,*AR4++(IR0)         ; X'(I4) <---------+
;
; Part B:
;
;          _        _______
;         |         |______| 0
;         |  AR1-> |__I1__| 1 <-  X(I1) + X(I2)
;         |        |______| 2
;         |  AR2-> |__I2__| 3 <-  X(I4) - X(I3)
;         |        |______| 4
;         |  AR3-> |__I3__| 5 <- [X(I1)-X(I2)]*COS-[X(I3)+X(I4)]*SIN
;         |        |______| 6
;         |_AR4 ->  |__I4__| 7 <- [X(I1)-X(I2)]*SIN+[X(I3)+X(I4)]*COS
;                  |______| 8
;            AR1-> |______| 9       NOTE: COS(2*pi/8) = SIN(2*pi/8)
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
;                   |   .  |
;                       .
;                       .
;                      \|/
;          LDA      @SOURCE_ADDR,AR1
           LDA      AR1,AR2
           LDA      AR1,AR3
           LDA      AR1,AR4
           ADDI     1,AR1
           ADDI     3,AR2
           ADDI     5,AR3
           ADDI     7,AR4
           LDA      @SINE_TABLE,AR7        ; AR7 points at SIN/COS table
           LDI      @FFT_SIZE,RC
           LSH      -3,RC
           LDA      RC,IR1
           SUBI     2,RC
           LDF      *AR2,R6                ; R6 = X(I2)
           LDF      *AR3,R0                ; R0 = X(I3)
           ADDF3    R6,*AR1,R5             ; R5 = X(I1)+X(I2) ---------+
           SUBF3    R6,*AR1,R4             ; R4 = X(I1)-X(I2)          |
           SUBF3    R0,R4,R3               ; R3 = X(I1)-X(I2)-X(I3)    |
           ADDF3    R0,R4,R2               ; R2 = X(I1)-X(I2)+X(I3)    |
           SUBF3    R0,*AR4,R1             ; R1 = X(I4)-X(I3) ---------|-+
||         STF      R5,*AR1++(IR0)         ; X(I1) <-------------------+ |
                                            ;                             |
           RPTBD    LOOP3_B                ;                             |
           ADDF3    R2,*AR4,R5             ; R5 = X(I1)-X(I2)+X(I3)+X(I4)|
||         STF      R1,*AR2++(IR0)         ; X(I2) <---------------------+
           MPYF3    R5,*++AR7(IR1),R1      ; R1 = R5*SIN    --------------+
||         SUBF3    *AR4,R3,R2             ; R2 = X(I1)-X(I2)-X(I3)-X(I4) |
           MPYF3    R2,*AR7,R0             ; R0 = R2*SIN    ---+          |
||         STF      R1,*AR4++(IR0)         ; X(I4) <-----------|----------+
                                            ;                   |
           LDF      *AR2,R6                ; R6 = X(I2)        |
||         STF      R0,*AR3++(IR0)         ; X(I3) <-----------+
           ADDF3    R6,*AR1,R5             ; R5 = X(I1)+X(I2) ---------+
           LDF      *AR3,R0                ; R0 = X(I3)                |
           SUBF3    R6,*AR1,R4             ; R4 = X(I1)-X(I2)          |
           SUBF3    R0,R4,R3               ; R3 = X(I1)-X(I2)-X(I3)    |
           ADDF3    R0,R4,R2               ; R2 = X(I1)-X(I2)+X(I3)    |
           SUBF3    R0,*AR4,R1             ; R1 = X(I4)-X(I3) ---------|-+
||         STF      R5,*AR1++(IR0)         ; X(I1) <-------------------+ |
           ADDF3    R2,*AR4,R5             ; R5 = X(I1)-X(I2)+X(I3)+X(I4)|
||         STF      R1,*AR2++(IR0)         ; X(I2) <---------------------+
           MPYF3    R5,*AR7,R1             ; R1 = R5*SIN  <--------------+
||         SUBF3    *AR4,R3,R2             ; R2 = X(I1)-X(I2)-X(I3)-X(I4) |
LOOP3_B:   MPYF3    R2,*AR7,R0             ; R0 = R2*SIN                  |
||         STF      R1,*AR4++(IR0)         ; X(I4) <----------------------+
           STF      R0,*AR3                ;X(I3)
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
;
; Perform first and second FFT loops.
;
;    _          _____
;    |  AR1 -> |__I1__| 0  <-  X(I1) + X(I3) + 2*X(I2)
;    |  AR2 -> |__I2__| 1  <-  X(I1) + X(I3) - 2*X(I2)
;    |  AR3 -> |__I3__| 2  <-  X(I1) - X(I3) - 2*X(I4)
;    |_ AR4 -> |__I4__| 3  <-  X(I1) - X(I3) + 2*X(I4)
;       AR1 -> |______| 4
;              |  .   |
;                .
;                .
;               \|/
;         LDA      @SOURCE_ADDR,AR1
          LDA      AR1,AR2
          LDA      AR1,AR3
          LDA      AR1,AR4
          ADDI     1,AR2
          ADDI     2,AR3
          ADDI     3,AR4
          LDA      4,IR0
          LDI      @FFT_SIZE,RC
          LSH      -2,RC
          SUBI     2,RC
          LDF      *AR4,R6         ; R6 = X(I4)
          LDF      *AR2,R7         ; R7 = X(I2)
||        LDF      *AR1,R1         ; R1 = X(I1)
          MPYF     2.0,R6          ; R6 = 2 * X(I4)
          MPYF     2.0,R7          ; R7 = 2 * X(I2)
          SUBF3    R6,*AR3,R5      ; R5 = X(I3) - 2*X(I4)
          SUBF3    R5,R1,R4        ; R4 = X(I1)-X(I3)+2X(I4) --+
          SUBF3    R7,*AR3,R5      ; R5 = X(I3) - 2*X(I2)      |
||        STF      R4,*AR4++(IR0)  ; X(I4)  <------------------+
          ADDF3    R5,R1,R3        ; R3 = X(I1)+X(I3)-2X(I2) --+
          ADDF3    R6,*AR3,R4      ; R4 = X(I3) + 2*X(I4)      |
||        STF      R3,*AR2++(IR0)  ; X(I2)  <------------------+
                                   ;                           |
          RPTBD    LOOP1_2         ;                           |
          SUBF3    R4,R1,R4        ; R4 = X(I1)-X(I3)-2X(I4) --+
          ADDF3    R7,*AR3,R0      ; R0 = X(I3) + 2*X(I2)      |
||        STF      R4,*AR3++(IR0)  ; X(I3)  <------------------+
          ADDF3    R0,R1,R0        ; R0 = X(I1)+X(I3)+2X(I2) --+
                                   ;                           |
          LDF      *AR4,R6         ; R6 = X(I4)                |
||        STF      R0,*AR1++(IR0)  ; X(I1)  <------------------+
          MPYF     2.0,R6          ; R6 = 2 * X(I4)
          LDF      *AR2,R7         ; R7 = X(I2)
||        LDF      *AR1,R1         ; R1 = X(I1)
          MPYF     2.0,R7          ; R7 = 2 * X(I2)
          SUBF3    R6,*AR3,R5      ; R5 = X(I3) - 2*X(I4)
          SUBF3    R5,R1,R4        ; R4 = X(I1)-X(I3)+2X(I4) --+
          SUBF3    R7,*AR3,R5      ; R5 = X(I3) - 2*X(I2)      |
||        STF      R4,*AR4++(IR0)  ; X(I4)  <------------------+
          ADDF3    R5,R1,R3        ; R3 = X(I1)+X(I3)-2X(I2) --+
          ADDF3    R6,*AR3,R4      ; R4 = X(I3) + 2*X(I4)      |
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
||        STF     R3,*AR2++(IR0)          ; X(I2)  <------------------+
          SUBF3   R4,R1,R4                ; R4 = X(I1)-X(I3)-2X(I4) --+
          ADDF3   R7,*AR3,R0              ; R0 = X(I3) + 2*X(I2)      |
||        STF     R4,*AR3++(IR0)          ; X(I3)  <------------------+
LOOP1_2:  ADDF3   R0,R1,R0                ; R0 = X(I1)+X(I3)+2X(I2) --+
                                          ;                           |
          STF     R0,*AR1                 ; LAST X(I1) <--------------+
;
; Check Bit Reversing Mode (on or off)
;
; BIT_REVERSING =  0, then OFF (no bit reversing)
; BIT_REVERSING <> 0, Then ON
;
ENDB:     LDI     @BIT_REVERSE,R0
          BZ      MOVE_DATA
;
; Check Bit Reversing Type.
;
; If SourceAddr =  DestAddr, Then In Place Bit Reversing
; If SourceAddr <> DestAddr, Then Standard Bit Reversing
;
          LDI     @SOURCE_ADDR,R0
          CMPI    @DEST_ADDR,R0
          BEQ     IN_PLACE
;
; Bit reversing Type 1 (From Source to Destination).
;
; NOTE: abs(SOURCE_ADDR – DEST_ADDR) must be > FFT_SIZE, this is not checked.
;
          LDI     @FFT_SIZE,R0
          SUBI    2,R0
          LDA     @FFT_SIZE,IR0
          LSH -1,IR0                      ; IRO = Half FFT size.
          LDA     @SOURCE_ADDR,AR0
          LDA     @DEST_ADDR,AR1
          LDF     *AR0++,R1
          RPTS    R0
          LDF     *AR0++,R1
||        STF     R1,*AR1++(IR0)B
          STF     R1,*AR1++(IR0)B
          BR      DIVISION;
; In Place Bit Reversing.
;
; Bit Reversing On Even Locations, 1st Half Only.
IN_PLACE: LDA     @FFT_SIZE,IR0
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
          LSH     -2,IR0                  ; IRO = Quarter FFT size.
          LDA     2,IR1
          LDI     @FFT_SIZE,RC
          LSH     -2,RC
          SUBI    3,RC
          LDA     @DEST_ADDR,AR0
          LDA     AR0,AR1
          LDA     AR0,AR2
          NOP     *AR1++(IR0)B
          NOP     *AR2++(IR0)B
          LDF     *++AR0(IR1),R0
          LDF     *AR1,R1
          RPTBD   BITRV1
          CMPI    AR1,AR0                 ; Xchange Locations only if AR0<AR1.
          LDFGT   R0,R1
          LDFGT   *AR1++(IR0)B,R1
          LDF     *++AR0(IR1),R0
||        STF     R0,*AR0
          LDF     *AR1,R1
||        STF     R1,*AR2++(IR0)B
          CMPI    AR1,AR0
          LDFGT   R0,R1
BITRV1:   LDFGT   *AR1++(IR0)B,R0
          STF     R0,*AR0
          STF     R1,*AR2
;Perform Bit Reversing Odd Locations, 2nd Half Only
          LDI     @FFT_SIZE,RC
          LSH     -1,RC
          LDA     @DEST_ADDR,AR0
          ADDI    RC,AR0
          ADDI    1,AR0
          LDA     AR0,AR1
          LDA     AR0,AR2
          LSH     -1,RC
          SUBI    3,RC
          NOP     *AR1++(IR0)B
          NOP     *AR2++(IR0)B
          LDF     *++AR0(IR1),R0
          LDF     *AR1,R1
          RPTBD   BITRV2
          CMPI    AR1,AR0                 ; Xchange Locations only if AR0<AR1.
          LDFGT   R0,R1
          LDFGT   *AR1++(IR0)B,R1
          LDF     *++AR0(IR1),R0
||        STF     R0,*AR0
          LDF     *AR1,R1
||        STF     R1,*AR2++(IR0)B
          CMPI    AR1,AR0
          LDFGT   R0,R1

BITRV2:   LDFGT   *AR1++(IR0)B,R0
          STF     R0,*AR0                 ; STF     R1,*AR2  later
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
; Perform Bit Reversing On Odd Locations, 1st Half Only.
          LDI      @FFT_SIZE,RC
          LSH      -1,RC
          LDA      RC,IR0
          LDA      @DEST_ADDR,AR0
          LDA      AR0,AR1
          ADDI     1,AR0
          ADDI     IR0,AR1
          LSH      -1,RC
          LDA      RC,IR0
          SUBI     2,RC
          RPTBD    BITRV3
          STF      R1,*AR2
          LDF      *AR0,R0
          LDF      *AR1,R1
          LDF      *++AR0(IR1),R0
||        STF      R0,*AR1++(IR0)B
BITRV3:   LDF      *AR1,R1
||        STF      R1,*-AR0(IR1)
          STF      R0,*AR1
          STF      R1,*AR0
          BR       DIVISION
;
; Check Data Source Locations.
;
; If SourceAddr =  DestAddr, Then do nothing.
; If SourceAddr <> DestAddr, Then move data.
;
MOVE_DATA: LDI     @SOURCE_ADDR,R0
          CMPI     @DEST_ADDR,R0
          BEQ      DIVISION
          LDI      @FFT_SIZE,R0
          SUBI     2,R0
          LDA      @SOURCE_ADDR,AR0
          LDA      @DEST_ADDR,AR1
          LDF      *AR0++,R1
          RPTS     R0
          LDF      *AR0++,R1
||        STF      R1,*AR1++
          STF      R1,*AR1
DIVISION: LDA      2,IR0
          LDI      @FFT_SIZE,R0
          FLOAT    R0              ; exp = LOG_SIZE
          PUSHF    R0              ; 32 MSB'S saved
          POP      R0
          NEGI     R0              ; Neg exponent
          PUSH     R0
          POPF     R0              ; R0 = 1/FFT_SIZE
          LDA      @DEST_ADDR,AR1
          LDI      @FFT_SIZE,RC
          LSH      -1,RC
          SUBI     2,RC
          RPTBD    LAST_LOOP
          LDA      @DEST_ADDR,AR2
          NOP      *AR2++
```

Example 6–18. Real Inverse Radix-2 FFT (Continued)

```
          MPYF3     R0,*AR1,R1      ; 1st location
          MPYF3     R0,*AR2,R2      ; 2nd,4th,6th,... location
||        STF       R1,*AR1++(IR0)
LAST_LOOP:MPYF3     R0,*AR1,R1      ; 3rd,5th,7th,... location
||        STF       R2,*AR2++(IR0)
          MPYF3     R0,*AR2,R2      ; last location
||        STF       R1,*AR1
          STF       R2,*AR2
; Return to C environment
          POP       DP              ; Restore C environment variables.
          POP       AR7
          POP       AR6
          POP       AR5
          POP       AR4
          POPF      R7
          POP       R7
          POPF      R6
          POP       R6
          POP       R5
          POP       R4
          POP       FP
          RETS
          .end
*
* No more.
*
*******************************************************************************
```

6.6 'C4x Benchmarks

Table 6–1 provides benchmarks for common DSP operations. Table 6–2 summarizes the FFT execution time required for FFT lengths between 64 and 1024 points for the four algorithms in Example 6–12, Example 6–14, Example 6–17, Example 6–18, and Example 6–15.

The benchmarks are given in cycles (the H1 internal processor cycle). To get the benchmark (time), multiply the number of cycles by the processor's internal clock period. For example, for a 50 MHz 'C4x, multiply by 40 ns.

Table 6–1. 'C4x Application Benchmarks

Application	Words	Cycles
Inverse of a float (32-bit mantissa accuracy)	7	7
Double-precision integer multiply	2	2
Square root (32-bit mantissa accuracy)	11	11
Vector dot product†	6	N + 4
Matrix Times a Vector	10	1 + R (C + 7)
FIR Filter	6	3+N
IIR Filter (One Biquad)	7	7
IIR Filter (N>1 Biquads)	15	2 + 6N
LMS Lattice Filter	11	1 + 5P
Inverse LPC Lattice Filter	9	3 + 3P
Mu–law (A–law) Compression	15 (16)	14 (16 / 10)
Mu–law (A–law) Expansion	11 (15)	11/10 (15/13)

† Based on a modification of the matrix times a vector benchmark

Table 6–2. FFT Timing Benchmarks (Cycles)

	Complex			Real	
Points	Radix-2 Example 6–12	Radix-4 Example 6–14	Radix-2 Example 6–15	Forward Example 6–17	Inverse Example 6–18
64	2290†	1745†	1425†	752†	1012†
128	5179†	——	3336†	1683†	2269†
256	11588†	9216†	7655†	3814†	5086†
512	25677†	——	17302†	8633†	11343†
1024	56411‡	47237‡	38945‡	19404†	25120†

Assumptions:

† The data is in on-chip RAM1. Program (.fftxt) and reserved data (.fftdat) are in on-chip RAM0. The sine/Cosine table is in on-chip RAM0. Bit-reversing is not considered. The cache is enabled

‡ The data is in on-chip RAM. Program (.ffttxt) and reserved data (.fftdat) are a in local(global) bus RAM with 0-wait states. Bit reversing is not considered. The sine/cosine table is on the global(local) bus. The cache is enabled

Chapter 7

Programming the DMA Coprocessor

The ’C4x DMA (Direct Memory Access) coprocessor is a ’C4x peripheral module. With its six channels, the DMA maximizes sustained CPU performance by alleviating the CPU of burdensome I/O. Any of the six DMA channels can transfer data to and from anywhere in the ’C4x’s memory map for maximum flexibility.

Topic **Page**

7.1 Hints for DMA Programming

The following hints will help you improve your DMA programming and also help you avoid unexpected results:

- Reset the DMA register before starting it. This clears any previously latched interrupt that may no longer exist. Also, set the DIE register (enabling interrupts for sync transfer) after starting the DMA channel.

- Take care in selecting the priority used to arbitrate between the CPU and DMA and also between DMA channels. If a DMA channel fails to finish a block transfer, it may have lower priority in a conflicting environment and and not be granted access to the resource. CPU/DMA rotating priority is considered a safe first choice. Depending on CPU/DMA execution load, selection of other priority schemes could result in faster code. Fine tuning may be needed.

- Ensure that each interrupt is received when you use interrupt synchronization; otherwise, the DMA will never complete the block transfer.

- For faster execution, avoid memory/resource access conflicts between the CPU and DMA. Carefully allocate the different sections of the program in memory. Use the same care with DMA autoinitialization values in memory.

- Try to use read/write synchronization when reading from or writing to communication ports. This avoids a peripheral-bus halt during a read from an empty-input FIFO or a write to a full-output FIFO.

  Choose between DMA read and write synchronization when using a DMA channel to transfer from one communication port to another. The ’C4x does not allow synchronization of DMA channel reads/writes with ICRDY*i*/OCRDY*j* signals coming from two different communication ports ($i \neq j$)

- When your application requires initializing the primary (or auxiliary) DMA channel while the auxiliary (or primary) channel may still be running, halt the running channel by writing a halt signal to the START or AUX START bits. Before proceeding, check the STATUS or AUX STATUS bits of the running channel to ensure it has halted. This is necessary because the DMA halt takes place in read/write boundaries (depending on the type of halt issued), and the channel must wait for any ongoing read or write cycles to complete. When reinitializing this channel, be especially careful to restore its previous status exactly. For an example of how to deal with this situation, refer to the Designer Notebook Page, *split-mode DMA re-initialization*, available through the DSP hotline.

7.2 When a DMA Channel Finishes a Transfer

Many applications require that you perform certain tasks after a DMA channel has finished a block transfer.

You can program the DMA to interrupt the CPU when this happens (TCC or AUX TCC bits). You can also achieve this by polling if:

- **The corresponding IIF (DMA INTx) bit is set to 1 (interrupt polling).** This requires that the DMA control register TCC (or AUX TCC) bit be set first. This method does not cause any extra CPU/DMA access conflict. But its drawback, when using split mode, is that you cannot differentiate whether the primary or auxiliary channel has finished.
- **The transfer counter has a zero value.** This option is sometimes not reliable, because the DMA channel could be in the middle of an autoinitialization sequence.
- **The TCINT (or AUX TCINT flag) is set to 1.** This option is reliable, but the CPU is polled via the peripheral bus, potentially causing CPU/DMA access conflict if the DMA is operating to/from the peripheral bus. This is a good option if you do not foresee any problem with the additional access delay.
- **The START (AUX START) bits in the DMA channel control register are set to 10_2.** This option can also cause a CPU/DMA access conflict.

7.3 DMA Assembly Programming Examples

The DMA coprocessor is a memory-mapped peripheral that you can easily program from C as well as from assembly. Example 7–1 through Example 7–5 provide examples on programming the DMA coprocessor using assembly language. Example 7–6 through Example 7–11 provide examples on programming the DMA coprocessor from C. The source code for examples Example 7–6 through Example 7–11 can be found in the TI BBS (self-extracting file: C4xdmaex.exe).

Example 7–1 shows one way for setting up DMA channel 2 to initialize an array to zero. This DMA transfer is set up to have priority over a CPU operation and to generate an interrupt flag, DMA INT2, after the transfer is completed. The DMA control register is set to 00C4 0007h.

Example 7–1.Array initialization With DMA

```
*
*   TITLE ARRAY INITIALIZATION WITH DMA
*
*   THIS EXAMPLE INITIALIZES A 128 ELEMENTS ARRAY TO ZERO. THE DMA
*   TRANSFER IS SET UP TO HAVE HIGHER PRIORITY OVER CPU OPERATION.
*   THE DMA INT2 INTERRUPT FLAG IS SET TO 1 AFTER THE TRANSFER IS
*   COMPLETED.
*
          .data
DMA2      .word    001000C0H      ;DMA channel 2 map address
CONTROL   .word    00C40007H      ;DMA register initialization data
SOURCE    .word    ZERO
SRC_IDX   .word    0
COUNT     .word    128
DESTIN    .word    ARRAY
DES_IDX   .word    1
ZERO      .float   0.0            ;Array initialization value 0.0
          .bss     ARRAY,128
          .text
START     LDP      @DMA2          ;Load data page pointer
          LDA      @DMA2,AR0      ;Point to DMA channel 2 registers
          LDI      @SOURCE,R0     ;Initialize DMA source register
          STI      R0,*+AR0(1)
          LDI      @SRC_IDX,R0    ;Initialize DMA source index register
          STI      R0,*+AR0(2)
          LDI      @COUNT,R0      ;Initialize DMA count register
          STI      R0,*+AR0(3)
          LDI      @DESTIN,R0     ;Initialize DMA destination register
          STI      R0,*+AR0(4)
          LDI      @DES_IDX,R0    ;Initialize DMA destination index register
          STI      R0,*+AR0(5)
          LDI      @CONTROL,R0    ;Start DMA channel 2 transfer
          STI      R0,*AR0
          .end
```

The DMA transfer can be synchronized with external interrupts, communication-port ICRDY/OCRDY signals, and timer interrupts. In order to enable this feature, the SYNCH MODE field, bits 6–7, of the DMA-control register must be

configured to a proper value, and the corresponding bits of the DMA-interrupt enable (DIE) register must be set. Example 7–2 sets up DMA channel 4 read synchronization with the communication-port 4 ICRDY signal. The DMA continuously transfers data from the communication-port input register until the START field, bits 22–23 of the DMA control register, is changed by the CPU.

Example 7–2. DMA Transfer With Communication-Port ICRDY Synchronization

```
*
*   TITLE  DMA TRANSFER WITH COMMUNICATION PORT ICRDY
*             SYNCHRONIZATION
*
*   THIS EXAMPLE SETS UP DMA CHANNEL 4 TO TRANSFER DATA FROM
*   COMMUNICATION PORT INPUT REGISTER TO INTERNAL RAM WITH ICRDY
*   SIGNAL READ SYNCHRONIZATION. THE TRANSFER MODE OF THE DMA IS
*   SET TO 00. THEREFORE THE TRANSFER WON'T STOP UNTIL THE START
*   BITS OF THE DMA CONTROL REGISTER IS CHANGED.
*          .data
DMA4       .word    001000E0H       ;DMA channel 4 map address
CONTROL    .word    00C00040H       ;DMA register initialization data
SOURCE     .word    00100081H
SRC_IDX    .word    0
COUNT      .word    0               ;Transfer counter is set to largest value
DESTIN     .word    002FF800H
DES_IDX    .word    1
           .text
START      LDP      @DMA4           ;Load data page pointer
           LDA      @DMA4,AR0       ;Point to DAM channel 4 registers
           LDI      @SOURCE,R0      ;Initialize DMA source register
           STI      R0,*+AR0(1)
           LDI      @SRC_IDX,R0     ;Initialize DMA source index register
           STI      R0,*+AR0(2)
           LDI      @COUNT,R0       ;Initialize DMA count register
           STI      R0,*+AR0(3)
           LDI      @DESTIN,R0      ;Initialize DMA destination register
           STI      R0,*+AR0(4)
           LDI      @DES_IDX,R0     ;Initialize DMA destination index register
           STI      R0,*+AR0(5)
           LDI      @CONTROL,R0     ;Start DMA channel 4 transfer
           STI      R0,*AR0
           LDHI     010H,DIE        ;Enable ICRDY 4 read sync.
           .end
```

If external interrupt signals are used for DMA transfer synchronization, then pins IIOF0-3 must be configured as interrupt pins.

The ’C4x DMA split mode is another way besides memory-map address to transfer data from/to the communication port. When the split-mode bit of the DMA control register is set, the DMA is separated into primary and auxiliary channels. The primary channel transfers data from memory to the communication-port output register, and the auxiliary channel transfers data from the communication port to memory. The communication-port number is selected in bits15–17 of the DMA control register.

Example 7–3 shows how to set up DMA channel 1 into split mode. The DMA primary channel transfers data from internal RAM to communication port 3

through external interrupt INT2 synchronization and bit-reversed addressing. The DMA auxiliary channel transfers data from communication port 3 to internal RAM via external interrupt INT3 synchronization and linear addressing.

Example 7–3. DMA Split-Mode Transfer With External-Interrupt Synchronization

```
*
*   TITLE  DMA SPLIT-MODE TRANSFER WITH EXTERNAL INTERRUPT SYNCHRONIZATION
*
*   THIS EXAMPLE SETS UP DMA CHANNEL 1 TO SPLIT-MODE. THE PRIMARY CHANNEL TRANSFERS
*   DATA FROM INTERNAL RAM TO COMM PORT 3 OUTPUT REGISTER WITH EXTERNAL INTERRUPT
*   INT2 SYNCHRONIZATION AND BIT-REVERSED ADDRESSING. THE AUXILIARY CHANNEL TRANSFERS
*   DATA FROM COMMUNICATION PORT 3 INPUT REGISTER TO INTERNAL RAM WITH EXTERNAL
*   INTERRUPT INT3 SYNCHRONIZATION AND LINEAR ADDRESSING.
*
          .data
DMA1      .word     001000B0H       ;DMA channel 1 map address
CONTROL   .word     03CDD0D4H       ;DMA register initialization data
SOURCE    .word     002FFC00H
SRC_IDX   .word     08H             ;The same value as IR0 for bit-reversed
COUNT     .word     8
DESTIN    .word     002FF800H
DES_IDX   .word     1
AUX_CNT   .word     8         .text
STAR      LDP       @DMA1           ;Load data page pointer
          LDA       @DMA1,AR0       ;Point to DAM channel 1 registers
          LDI       @SOURCE,R0      ;Initialize DMA primary source register
          STI       R0,*+AR0(1)
          LDI       @SRC_IDX,R0     ;Initialize DMA primary source index register
          STI       R0,*+AR0(2)
          LDI       @COUNT,R0       ;Initialize DMA primary count register
          STI       R0,*+AR0(3)
          LDI       @DESTIN,R0      ;Initialize DMA aux destination register
          STI       R0,*+AR0(4)
          LDI       @DES_IDX,R0     ;Initialize DMA aux destination index register
          STI       R0,*+AR0(5)
          LDI       @AUC_CNT,R0     ;Initialize DMA auxiliary count register
          STI       R0,*+AR0(7)
          LDI       @CONTROL,R0     ;Start DMA channel 1 transfer
          STI       R0,*AR0
          LDI       01100H,IIF      ;Configure INT2 and INT3 as interrupt pins
          LDI       0A0H,DIE        ;Enable INT2 read and INT3 write sync.
          .end
```

An advantage of the ’C4x DMA is the autoinitialization feature. This allows you to set up the DMA transfer in advance and makes the DMA operation completely independent from the CPU. When the DMA operates in autoinitialization mode, the link pointer and auxiliary link pointer initialize the registers that control the DMA operation. The link pointer can be incremented (AUTOINIT STATIC = 0) during autoinitialization or held constant (AUTOINIT STATIC = 1) during autoinitialization. This option allows autoinitialization values to be stored in sequential memory locations or in stream-oriented devices such as the on-chip communication ports or external FIFOs. When DMA SYNC MODE is enabled, The DMA autoinitialization operation can be configured to synchronize with the same signal. Example 7–4 sets up DMA channel 0 to wait for the communication port to input the initialization value. After DMA autoinitializa-

tion is complete, the DMA channel starts transferring data from the communication port input register to internal RAM.

Example 7–4. DMA Autoinitialization With Communication Port ICRDY

```
*
*   TITLE DMA AUTOINITIALIZATION WITH COMMUNICATION PORT ICRDY
*
*   THIS EXAMPLE SETS UP DMA CHANNEL 0 TO WAIT FOR COMMUNICATION
*   PORT TO INPUT THE INITIALIZATION VALUE. THE DMA AUTOINITIAL-
*   IZATION AND TRANSFER ARE BOTH DRIVEN BY ICRDY 0 FLAG. AFTER
*   DMA AUTOINIT IS COMPLETED, THE DMA CHANNEL STARTS TRANSFERRING
*   DATA FROM COMM PORT INPUT REGISTER TO INTERNAL RAM WITH ICRDY
*   0 READ SYNCHRONIZATION. THE VALUES IN COMM PORT 0 INPUT FIFO
*   SHOULD BE:
*
*         SEQUENCE |  VALUE
*         ---------+-----------------------------------------------
*         1   |     00C40047H (STOP AFTER TRANSFER COMPLETED)
*             |     OR 00C4054BH (REPEAT AFTER TRANSFER COMPLETED)
*         2   |     00100041H
*         3   |     0H
*         4   |     20H
*         5   |     002FF800H
*         6   |     1H
*         7   |     00100041H
*
          .data
DMA0      .word     001000A0H       ;DMA channel 0 map address
DMA_INIT  .word     0004054BH       ;DMA initialization control word
LINK      .word     00100041H       ;Comm port input register address
DMA_START .word     00C4054BH       ;DMA start control word
          .text
START     LDP       @DMA0           ;Load data page pointer
          LDA       @DMA0,AR0       ;Point to DMA channel 0 registers
          LDI       @DMA_INIT,R0    ;Initialize DMA control register
          STI       R0,*AR0
          LDI       @LINK,R0        ;Initialize DMA link pointer
          STI       R0,*+AR0(6)
          LDI       @DMA_START,R0   ;Start DMA channel 0 transfer
          STI       R0,*AR0
          LDI       01H,DIE         ;Enable ICRDY 0 read sync.
          .end
```

The DMA autoinitialization and transfer continues executing if the DMA autoinitialization is still enabled. Therefore, a DMA setup like the one in Example 7–4 can make it possible for an external device to control the DMA operation through the communication port.

With the autoinitialization feature, the ’C4x DMA coprocessor can support a variety of DMA operations without slowing down CPU computation. A good example is a DMA transfer triggered by one interrupt signal. Usually, this is implemented by starting a DMA activity with a CPU interrupt service routine, but this utilizes CPU time. However, as shown in Example 7–5, you can set up a single interrupt-driven dummy DMA transfer with autoinitialization. When the inter-

rupt signal is set, the DMA will complete the dummy DMA transfer and start the autoinitialization for the desired DMA transfer.

Example 7–5. Single-Interrupt-Driven DMA Transfer

```
*
*   TITLE SINGLE INTERRUPT-DRIVEN DMA TRANSFER
*
*   THIS EXAMPLE SETS UP A DUMMY DMA TRANSFER FROM INTERNAL RAM
*   TO THE SAME MEMORY WITH EXTERNAL INT 0 SYNCHRONIZATION AND
*   AUTOINITIALIZATION FOR TRANSFERRING 64 DATA FROM LOCAL MEMORY
*   TO INTERNAL RAM. AFTER THE SECOND TRANSFER IS COMPLETED, THE
*   DMA IS RE-INITIALIZED TO FIRST DMA TRANSFER SETUP.
*
          .data
DMA5      .word    001000F0H      ;DMA channel 5 map address
DMA_INIT  .word    0000004BH      ;DMA initialization control word
LINK      .word    DMA1           ;1st DMA link list address
DMA_START .word    00C0004BH      ;DMA start control word
DMA1      .word    00C0004BH      ;1st dummy DMA transfer link list
          .word    002FF800H
          .word    00000000H
          .word    00000001H
          .word    002FF800H
          .word    00000000H
          .word    DMA2
DMA2      .word    00C4000BH      ;The desired DMA transfer link
          .word    00400000H      ;list
          .word    00000001H
          .word    00000040H
          .word    002FF800H
          .word    00000001H
          .word    DMA1
          .text
START     LDP      @DMA5          ;Load data page pointer
          LDA      @DMA5,AR0      ;Point to DMA channel 5 registers
          LDI      @DMA_INIT,R0   ;Initialize DMA control register
          STI      R0,*AR0
          LDI      @LINK,R0       ;Initialize DMA link pointer
          STI      R0,*+AR0(6)
          LDI      @DMA_START,R0  ;Start DMA channel 5 transfer
          STI      R0,*AR0
          LDI      01H,IIF        ;Configure INT0 as interrupt pins
          LDHI     0800H,DIE      ;Enable INT 0 read sync. for
                                  ;DMA channel 5
          .end
```

7.4 DMA C-Programming Examples

Example 7–6 to Example 7–11 includes DMA programing examples from C. These examples cover unified and Split mode, DMA autoinitialization and DMA synchronization operations. Descriptions of the examples presented are as follows:

- Example 7–6: Unified-mode DMA transfers data between commports using read sync.
- Example 7–7: Unified-mode DMA uses autoinitialization (method 1) to transfer 2 data blocks.
- Example 7–8: Unified-mode DMA uses autoinitialization (method 2) to transfer 2 data blocks.
- Example 7–9: Split-mode auxiliary DMA transfers data between commports using read sync.
- Example 7–10: Split-mode auxiliary and primary channel send/receive data to and from commport
- Example 7–11: Split-mode DMA autoinitializes both auxiliary and primary channels (auxiliary transfers 1 block and primary transfers 2 blocks)

Example 7–12 is the include file for all examples (dma.h).

Example 7–6.Unified-Mode DMA Using Read Sync

```
/*****************************************************************************
  EXAMPLE: Unified–mode
           Commport–to–commport transfer:
           DMA3 in unified mode transfers 8 words from commport 3 to commport 0.
           DMA3 source sync with ICRDY3 is used.
           Note: Writes cannot be synchronized with OCRDY0, because a DMA i can
           only be synchronized with signals coming commport i. You could sync
           on ICRDY3 or on OCRDY0, not both (the choice depends on the specific
           application to avoid deadlock).
           In this program, DMA3 expects data in commport 3 being sent by
           another processor/device. Otherwise no transfer will occur.
*****************************************************************************/
#include ”dma.h”
#define DMAADDR         0x001000d0
#define CTRLREG         0x00c40045 /* DMA sends interrupt to CPU when transfer
                                      finishes(TC=1),DMA–CPU rotating priority */
#define SRC             0x00100071 /* src = commport 0 input fifo */
#define SRC_IDX         0x0        /* src address does not increment */
#define COUNTER         0x08       /* number of words to transfer */
#define DST             0x00100042 /* dst = commport 3 output fifo */
#define DST_IDX         0x0        /* dst address does not increment */
#define DIEVAL          0x4000     /* set ICRDY3 read sync */
DMAUNIF *dma = (DMAUNIF *)DMAADDR;
int dieval = DIEVAL;

main() {

dma–>src     = (void *)SRC;
dma–>src_idx = SRC_IDX;
dma–>counter = COUNTER;
dma–>dst     = (void *)DST;
dma–>dst_idx = DST_IDX;
dma–>ctrl    = (void *)CTRLREG;
asm(” ldi @_dieval,die”);
PRIM_WAIT_DMA((volatile int *)dma);
}
```

Example 7–7.Unified-Mode DMA Using Autoinitialization (Method 1)

```
/*****************************************************************************
EXAMPLE: Unified Mode
         Autoinitialization method 1:
         DMA0 in unified mode transfers 8 words from 0x02ffC00 (index 1) to
         0x02ffd00 (index 1) and then it transfer 4 words from 0x02ffe00 (index 4)
         to to 0x02fff00 (index 1). No DMA sync transfer is used.
         Autoinitialization method 1 requires N autoinitialization memory blocks
        to transfer N blocks and starts with a DMA transfer counter equals to 0.
*****************************************************************************/
#include ”dma.h”
#define DMAADDR         0x001000a0

/* 1st transfer settings */
#define CTRLREG1        0x00c00009 /* DMA–CPU rotating priority and DMA
                                   autoinitializes when transfer counter = 0 */
#define SRC1            0x002ffc00 /* src address */
#define SRC1_IDX        0x1        /* src address increment */
#define COUNTER1        0x08       /* number of words to transfer */
#define DST1            0x002ffd00 /* dst address rt 3 output fifo */
#define DST1_IDX        0x1        /* dst address increment */

/* 2nd transfer settings */
#define CTRLREG2        0x00c40005 /* DMA sends interrupt to CPU when transfer
                                     finishes(TC=1),DMA–CPU rotating priority
                                     and DMA stops after transfer completes */
#define SRC2            0x002ffe00 /* src address */
#define SRC2_IDX        0x4        /* src address increment */
#define COUNTER2        0x4        /* number of words to transfer */
#define DST2            0x002fff00 /* dst address */
#define DST2_IDX        0x1        /* dst address increment */
DMAUNIF *dma = (DMAUNIF *)DMAADDR;
DMAUNIF autoini1;
DMAUNIF autoini2;

main() {

/* initialize 1st set of autoinitialization values    */
autoini1.src     = (void *)SRC1;
autoini1.src_idx = SRC1_IDX;
autoini1.counter = COUNTER1;
autoini1.dst     = (void *)DST1;
autoini1.dst_idx = DST1_IDX;
autoini1.linkp   = &autoini2;
autoini1.ctrl    = (void *)CTRLREG1;

/* initialize 2nd set of autoinitialization values    */
autoini2.src     = (void *)SRC2;
autoini2.src_idx = SRC2_IDX;
autoini2.counter = COUNTER2;
autoini2.dst     = (void *)DST2;
autoini2.dst_idx = DST2_IDX;
autoini2.ctrl    = (void *)CTRLREG2;

/* initialize DMA (link pointer pointing to 1st set of autoinit. values */
dma–>linkp       = &autoini1;
dma–>counter     = 0;
dma–>ctrl        = (volatile void *)CTRLREG1;

/* wait for DMA to finish transfer */
PRIM_WAIT_DMA((volatile int *)dma);        }
```

Example 7–8.Unified-Mode DMA Using Autoinitialization (Method 2)

```
/*************************************************************************
EXAMPLE: Unified Mode
         Autoinitialization method 2:
         DMA0 in unified mode transfers 8 words from 0x02ffC00 (index 1)
         to 0x02ffd00 (index 1) and then it transfer 4 words from 0x02ffe00
         (index 4) to to 0x02fff00 (index 1). No DMA sync transfer is used
         Autonitialization method 2 requires (N–1) autoinitialization memory
         blocks to transfer N blocks and starts with a DMA transfer counter
         different from 0.
*************************************************************************/
#include ”dma.h”
#define DMAADDR         0x001000a0

/* 1st transfer settings */
#define CTRLREG1        0x00c00009 /* DMA–CPU rotating priority and DMA
                                   autoinitializes when transfer counter = 0 */
#define SRC1            0x002ffc00 /* src address */
#define SRC1_IDX        0x1        /* src address increment */
#define COUNTER1        0x08       /* number of words to transfer */
#define DST1            0x002ffd00 /* dst address rt 3 output fifo */
#define DST1_IDX        0x1        /* dst address increment */

/* 2nd transfer settings */
#define CTRLREG2        0x00c40005 /* DMA sends interrupt to CPU when transfer
                                     finishes(TC=1),DMA–CPU rotating priority
                                     and DMA stops after transfer completes */
#define SRC2            0x002ffe00 /* src address */
#define SRC2_IDX        0x4        /* src address increment */
#define COUNTER2        0x4        /* number of words to transfer */
#define DST2            0x002fff00 /* dst address */
#define DST2_IDX        0x1        /* dst address increment */
DMAUNIF *dma = (DMAUNIF *)DMAADDR;
DMAUNIF autoini2;

main() {

/* initialize 2nd set of autoinitialization values    */
autoini2.src     = (void *)SRC2;
autoini2.src_idx = SRC2_IDX;
autoini2.counter = COUNTER2;
autoini2.dst     = (void *)DST2;
autoini2.dst_idx = DST2_IDX;
autoini2.ctrl    = (void *)CTRLREG2;

/* initialize DMA with 1st set of autoinitialization values    */
dma–>src     = (void *)SRC1;
dma–>src_idx = SRC1_IDX;
dma–>counter = COUNTER1;
dma–>dst     = (void *)DST1;
dma–>dst_idx = DST1_IDX;
dma–>linkp   = &autoini2;
dma–>ctrl    = (void *)CTRLREG1;

/* wait for DMA to finish transfer */
PRIM_WAIT_DMA((volatile int *)dma);
}
```

Example 7–9.Split-Mode Auxiliary DMA Using Read Sync

```
/*****************************************************************************
EXAMPLE: Split–mode (AUX only)
         Commport–to–commport transfer:
         DMA 3 Auxiliary channel transfers 8 words from commport 3 to
         commport 0. DMA3 source sync with ICRDY3 is used.
         This example is functionally equivalent to Example 7–7.
         In this program, DMA3 expects data in commport 3 being sent by
         another processor/device. Otherwise no transfer will occur.
*****************************************************************************
/
#include ”dma.h”
#define DMAADDR         0x001000d0
#define CTRLREG         0x0309c091 /* DMA Aux sends interrupt to CPU when
                                      transfer finishes(TC=1),DMA–CPU rotating
                                      priority */
#define DST             0x00100042 /* dst = commport 3 output fifo */
#define DST_IDX         0x0        /* dst address does not increment */
#define DIEVAL          0x4000     /* set ICRDY3 Auxiliar read sync */
#define ACOUNTER        0x08       /* auxiliar channle counter  */
DMASPLIT *dma = (DMASPLIT *)DMAADDR;
int dieval = DIEVAL;

main() {

dma–>dst      = (void *)DST;
dma–>dst_idx  = DST_IDX;
dma–>acounter = ACOUNTER;
dma–>ctrl     = (void *)CTRLREG;
asm(” ldi @_dieval,die”);
AUX_WAIT_DMA((volatile int *)dma);
}
```

Example 7–10. Split-Mode Auxiliary and Primary Channel DMA

```
/*************************************************************************
EXAMPLE: Split–mode (AUX and PRIMARY both running)
         Commport–to–commport transfer:
         DMA3 prim. channel sends 4 words from memory (0x02ffc00) to
         commport 3 (output FIFO).
         DMA3 aux.channel receives 8 words from commport 3 (input FIFO)
         to memory (0x02ffd00)
         DMA3 prim. channel uses OCRDY3 write sync.
         DMA3 aux. channel uses ICRDY3 read sync.
         In this program, DMA3 aux channel expects data in commport 3 being
         sent by another processor/device. Otherwise no aux channel transfer
         will occur.
*************************************************************************/
#include ”dma.h”
#define DMAADDR         0x001000d0
#define CTRLREG         0x03cdc0d5 /* DMA Aux/prim send interrupt to CPU when
                                      transfer finishes(TC=1),DMA–CPU rotating
                                      priority, read/write sync transfer */
#define DIEVAL          0x24000    /* set ICRDY3/OCRDY read/write sync */
#define DST             0x02ffd00  /* auxiliary channel settings */
#define DST_IDX         0x1
#define ACOUNTER        0x08
#define SRC             0x02ffc00  /* primary channel settings */
#define SRC_IDX         0x1
#define COUNTER         0x04
DMASPLIT *dma = (DMASPLIT *)DMAADDR;
int dieval = DIEVAL;

main() {

dma–>src      = (void *)SRC;       /* primary channel */
dma–>src_idx  = SRC_IDX;
dma–>counter  = COUNTER;
dma–>dst      = (void *)DST;       /* auxiliary channel */
dma–>dst_idx  = DST_IDX;
dma–>acounter = ACOUNTER;
dma–>ctrl     = (void *)CTRLREG;
asm(” ldi @_dieval,die”);
SPLIT_WAIT_DMA((volatile int *)dma);
}
```

Example 7–11. Split-Mode DMA Using Autoinitialization

```
/*****************************************************************************
EXAMPLE   : Split–mode (AUX and PRIMARY both running)
          Autoinitialization example:
          DMA3 aux .channel autoinitializes and THEN receives 4 words from
          commport 3 (input FIFO) to memory (0x02ffd00).
          DMA3 pri.channel sends 4 words from memory (0x02ffc00) to
          commport 3 (output FIFO)  and THEN other 2 words from memory
          (0x02ffc10) with index=2 to commport 3 (output FIFO).
          DMA3 prim. channel uses OCRDY3 write sync.
          DMA3 aux. channel uses ICRDY3 read sync.
          Autoinitialization method 1 is used in all cases.
          In this program, DMA3 aux channel expects data in commport 3 being
          sent by another processor/device. Otherwise no aux channel transfer
          will occur.
*****************************************************************************/
#include ”dma.h”
#define DMAADDR         0x001000d0
#define CTRLREG1        0x03cdc0e9 /* DMA aux/prim send interrupt to CPU when
                                      transfer finishes(TC=1),DMA–CPU rotating
                                      priority, read/write sync transfer */
#define CTRLREG2        0x03cdc0d5 /* same as above but transfer finishes */
#define DIEVAL          0x24000    /* set ICRDY3/OCRDY read/write sync */

/* Primary Channel */
#define SRC1            0x02ffc00       /* autoinitialization 1 */
#define SRC1_IDX        0x1
#define COUNTER1        0x04
#define SRC2            0x02ffc10       /* autoinitialization 2 */
#define SRC2_IDX        0x2
#define COUNTER2        0x02

/* Auxiliary channel */
#define DST1            0x02ffd00       /* autoinitialization 1 */
#define DST1_IDX        0x1
#define ACOUNTER1       0x04

DMASPLIT *dma = (DMASPLIT *)DMAADDR;
int dieval = DIEVAL;
DMAPRIM autoini1, autoini2;
DMAAUX autoiniaux;

main() {

/* PRIMARY CHANNEL : 1st autoinitialization values */
autoini1.ctrl    = (void *)CTRLREG1;
autoini1.src     = (void *)SRC1;
autoini1.src_idx = SRC1_IDX;
autoini1.counter = COUNTER1;
autoini1.linkp   = &autoini2;
```

Example 7–11. Split-Mode DMA Using Autoinitialization (Continued)

```
/* PRIMARY CHANNEL : 2nd autoinitialization values */
autoini2.ctrl    = (void *)CTRLREG2;
autoini2.src     = (void *)SRC2;
autoini2.src_idx = SRC2_IDX;
autoini2.counter = COUNTER2;

/* AUXILIARY CHANNEL : 1st autoinitialization values */
autoiniaux.ctrl    = (void *)CTRLREG2;
autoiniaux.dst     = (void *)DST1;
autoiniaux.dst_idx = DST1_IDX;
autoiniaux.acounter = ACOUNTER1;

/* initialize DMA */
dma–>linkp       = &autoini1;
dma–>alinkp      = &autoiniaux;
dma–>counter     = 0;
dma–>acounter    = 0;
dma–>ctrl        = (void *)CTRLREG1;
asm(" ldi @_dieval,die");

/* wait for DMA to finish transfer */
SPLIT_WAIT_DMA((volatile int *)dma);
}
```

Example 7–12. Include File for All C Examples (dma.h)

```
typedef struct dmaunif{
        volatile void *ctrl;            /* control register     */
        volatile void *src;             /* source address       */
        volatile int src_idx;           /* source address index */
        volatile int counter;           /* transfer counter     */
        volatile void *dst;             /* dest. address        */
        volatile int dst_idx;           /* dest. address index  */
        struct dmaunif *linkp;          /* link pointer         */
        }DMAUNIF;
typedef struct dmaprim{
        volatile void *ctrl;            /* control register     */
        volatile void *src;             /* prim. src address    */
        volatile int src_idx;           /* prim. index          */
        volatile int counter;           /* prim transfer counter*/
        struct dmaprim *linkp;          /* link pointer         */
        }DMAPRIM;
typedef struct dmaaux{
        volatile void *ctrl;            /* control register     */
        volatile void *dst;             /* aux. dst address     */
        volatile int dst_idx;           /* aux. index           */
        volatile int acounter;          /* aux. transfer counter*/
        struct dmaaux *alinkp;          /* aux. link pointer    */
        }DMAAUX;
typedef struct {
        volatile void *ctrl;            /* control register     */
        volatile void *src;             /* prim. src address    */
        volatile int src_idx;           /* prim. index          */
        volatile int counter;           /* prim transfer counter*/
        volatile void *dst;             /* aux. dst address     */
        volatile int dst_idx;           /* aux. index           */
        struct dmaprim *linkp;          /* link pointer         */
        volatile int acounter;          /* aux. transfer counter*/
        struct dmaaux *alinkp;          /* aux. link pointer    */
        } DMASPLIT;
#define PRIM_WAIT_DMA(x)   while ((0x00c00000 & *x)!=0x00800000)
#define AUX_WAIT_DMA(x)    while ((0x03000000 & *x)!=0x02000000)
#define SPLIT_WAIT_DMA(x)  while ((0x03c00000 & *x)!=0x02800000)
```

Chapter 8

Using the Communication Ports

The ’C4x communication ports are very high-speed data transmission circuits. Their speed and the close proximity of multiple data lines create special challenges. General design rules that are applicable to high-speed (<10ns) memory interface design are appropriate for ’C4x communication-port interconnections. This chapter provides guidelines for designing communication-port interfaces.

Topic **Page**

8.1 Communication Ports

To provide simple processor-to-processor communication, the ’C4x has six parallel bidirectional communication ports. Because these ports have port arbitration units to handle the ownership of the communication-port data bus between the processors, you should concentrate only on the internal operation of the communication ports. For software, these communication ports can be treated as 32-bit on-chip data I/O FIFO buffers. Processor read data from/write data to communication is simple:

```
LDI   @comm_port0_input,R0  ;Read data from comm. port 0
```

or

```
STI   R0,@comm_port0_output ;Write data to comm. port 1
```

If the CPU or DMA reads from or writes to the communication-port I/O FIFO and the I/O-FIFO is either empty (on a read) or full (on a write), the read/write execution will be extended either until the data is available in the input FIFO for a read, or until the space is available in the output FIFO for a write. Sometimes, you can use this feature to synchronize the devices. However, this can slow down the processing speed and even hang up the processor. Avoid such situations by synchronizing the CPU/DMA accesses with the following flags that indicate the status of the port:

ICRDY (input channel ready)
= 0, the input channel is empty and not ready to be read.
= 1, the input channel contains data and is ready to read.

ICFULL (input channel full)
= 0, the input channel is not full.
= 1, the input channel is full.

OCRDY (output channel ready)
= 0, the output channel is full and not ready to be written.
= 1, the output channel is not full and ready to be written.

OCEMPTY (output channel empty)
= 0, the output channel is not empty.
= 1, the output channel is empty.

Example 8–1 shows the reading of data from the communication port, eight data at a time using the CPU ICFULL interrupt. Example 8–2 shows the writing of data to a communication port, one datum at a time using the polling method. Both examples show DMA reads/writes. (DMA is discussed in subsection 7.3, *DMA Assembly Programming Examples* on page 7-4.

Example 8–1.Read Data from Communication Port With CPU ICFULL Interrupt

```
*
*   TITLE READ DATA FROM COMMUNICATION PORT WITH CPU
*   ICFULL INTERRUPT
*
*   THIS EXAMPLE ASSUMES THE ICFULL 0 INTERRUPT VECTOR IS SET IN THE CPU
*   INTERRUPT VECTOR TABLE. THE EIGHT DATA WORDS ARE READ IN
*   WHENEVER THE DATA IS FULL IN COMM PORT 0 INPUT FIFO.
*
          .
          .
          .
          LDA      @COMM_PORT0_CTL,AR2    ;Load comm port 0 control Reg. address
          LDA      @COMM_PORT0_INPUT,AR0  ;Load comm port 0 input FIFO address
          LDA      @INTERNAL_RAM,AR1      ;Load internal RAM address
          AND3     0F7H,*AR2,R9           ;Unhalt comm port 0 input channel
          STI      R9,*AR2
          OR       04H,IIE                ;Enable ICRDY 0 interrupt
          OR       02000H,ST              ;Enable CPU global interrupt
          .
          .
          .
ICFULL0   PUSH     ST
          PUSH     RS
          PUSH     RE
          PUSH     RC
          LDI      *AR0,R10               ;Read data from comm port 0 input
          RPTS     6                      ;Setup for loop READ
READ      LDI      *AR0,R10               ;Read data from comm port 0 input
||        STI      R10,*AR1++(1)          ;Store data into internal RAM
          STI      R10,*AR1++(1)          ;Store data into internal RAM
          POP      RC
          POP      RE
          POP      RS
          POP      ST
          RETI
```

Example 8–2. Write Data to Communication Port With Polling Method

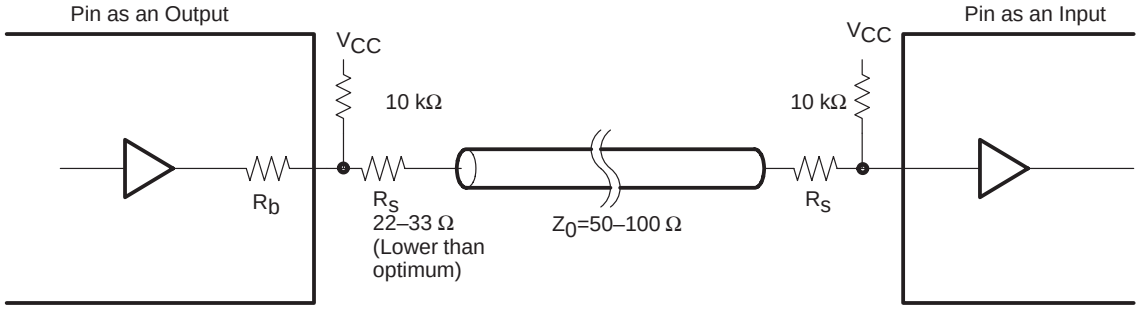
```
*
*   TITLE WRITE DATA TO COMMUNICATION PORT WITH POLLING METHOD
*
*   THE BIT 8 OF COMMUNICATION PORT 0 CONTROL REGISTER WILL BE
*   SET ONLY WHEN THE OUTPUT FIFO IS FULL. THIS EXAMPLE CHECKS
*   THIS BIT TO MAKE SURE THERE IS SPACE AVAILABLE IN
*   OUTPUT FIFO.
*
            .
            .
            .
            LDA     @COMM_PORT0_CTL,AR2    ;Load comm port 0 control reg address
            LDA     @COMM_PORT0_OUTPUT,AR0 ;Load comm port 0 output FIFO address
            LDA     @INTERNAL_RAM,AR1      ;Load internal RAM address
            AND3    0EFH,*AR2,R9           ;Unhalt comm port 0 output channel
            STI     R9,*AR2
            LDI     0100H,R9               ;Load mask for bit 8
WAIT:       TSTB    *AR2,R9                ;Check if output FIFO is full
            BZD     WAIT                   ;If yes, check again
WRITE_COMM  LDI     *AR1++(1),R10          ;Read data from internal RAM
            STI     R10,*AR0               ;Store data into comm port 0 output
            NOP
            .
            .
```

8.2 Signal Considerations

Because of the bidirectional high-speed protocol used in the ’C4x communication ports, signal quality is extremely important. Poor quality signals can potentially cause both ends of a communication-port link to become a master. If this occurs and one communication port drives a signal request, no response is received from the other communication port, and the link hangs. This condition remains until both ’C4x devices are reset. If this is not corrected, the communication-port drivers can be damaged.

If poor quality signals are a problem, use circuits to improve impedance matching. Because the ’C4x communication-port output buffer impedance can change during signal switching, a conventional parallel termination does not help. Serial matching resistors can be added at each end of all communication port lines (see Figure 8–1). Serial resistors help match the output buffer impedance to the line impedance and protect against signal contention caused by any potential fault condition. The resistor value, plus buffer output impedance, should match the line impedance. Results have shown that a lower than optimal serial resistor value provides better performance. A resistor value of 22–33 Ω is usually a reasonable start. Some experimentation may be needed to reduce ringing effects. A good received signal should have an undershoot of 0.5 to 1.0 V or less. A resistor value that is too high results in an underdamped falling edge that does not cross the zero logic level and should be avoided.

Figure 8–1. Impedance Matching for ’C4x Communication-Port Design



Even though pullup resistors do not help for impedance matching, they are recommended at each end to avoid unintended triggering after reset, when $\overline{\text{RESET}}$ going low is not received on all ’C4x devices at the same time.

A pulldown resistor is not desirable, because it increases power consumption, does not protect the device from a fault condition, and can cause token loss and byte slippage on reset.

For jumps to other boards or for long distances, a unidirectional data flow with buffering is the preferred method. In this case, use buffers with hysteresis for $\overline{\text{CSTRB}}$ and $\overline{\text{CRDY}}$ at each end with delays greater than those in the data bus. This has two advantages: it cleans up the signals and helps eliminate glitches that can be erroneously perceived as valid control; it also allows the data bits to settle before the receiver sees $\overline{\text{CSTRB}}$ going low.

8.3 Interfacing With a Non-'C4x Device

To guarantee a correct word transfer operation between a 'C4x communication port and a non-'C4x device, the non-'C4x device should mimic the handshaking operation between $\overline{\text{CSTRB}}$ and $\overline{\text{CRDY}}$ (word transfer), CREQ and CACK (token transfer). The token transfer operation is more complex than the word transfer operation. It requires tri-stating of pins after different events. Sections 8.6 and 8.7 offer examples on how to handle token transfers with non-'C4x devices. The word transfer operation is much simpler. The following sequence describes the word transfer operation:

Word transfer operation

CASE I: The non-'C4x has the token and transmits data. The 'C4x receives data.

1) The non-'C4x device drives the first byte (byte 0) into the CD data lines and then drops $\overline{\text{CSTRB}}$ low, indicating new data. There is no need to meet the maximum timing requirements, but the data should be valid before $\overline{\text{CSTRB}}$ goes low.
2) The non-'C4x device waits for the 'C4x to respond with $\overline{\text{CRDY}}$ low and then can immediately drive the next data byte and bring $\overline{\text{CSTRB}}$ high.
3) The non-'C4x device waits for $\overline{\text{CRDY}}$ to be high; then, steps 1, 2, and 3 repeat for bytes 1 – 3.
4) After byte 3 is transmitted, the non-'C4x device can leave the byte 3 value in the CD lines until a new word is sent.
5) In 'C4x device revisions lower than 3.0, $\overline{\text{CSTRB}}$ should go high after receiving $\overline{\text{CRDY}}$ low no later than one 'C4x H1/H3 cycle between word boundaries. See Section 8.9, *Implementing a $\overline{\text{CSTRB}}$ Shortener Circuit* on page 8-17, for an implementation of a $\overline{\text{CSTRB}}$ shortener circuit. In 'C4x device revisions 3.0 or higher, no $\overline{\text{CSTRB}}$ width restriction exists.
6) The non-'C4x device can drive $\overline{\text{CSTRB}}$ low for the next word at any time after receiving $\overline{\text{CRDY}}$ high from the last byte. There is no reason to wait for the internal 'C4x synchronizer between $\overline{\text{CRDY}}$ low and $\overline{\text{CSTRB}}$ low for the next word to finish.

CASE II: The 'C4x has the token and transmits data. The non-'C4x device receives data.

1) After receiving $\overline{\text{CRSTB}}$ low from the 'C4x, indicating new data valid, the non-'C4x device can immediately read the data byte and then drive $\overline{\text{CRDY}}$ low, indicating that the byte has been read. There is no maximum time limit between these two events.
2) The non-'C4x device then waits to receive $\overline{\text{CSTRB}}$ high and can immediately drive $\overline{\text{CRDY}}$ high, ending the byte transfer operation.

8.4 Terminating Unused Communication Ports

To avoid unintended communication port triggering, you can terminate unused communication-port control lines in one of the following ways:

- Use pullup resistors in all the communication-port control lines. Pullups in data lines of input communication ports are optional, but they lower power consumption. Pullups in data lines of output communication ports are not required; if used, they increase power consumption.
- Tie the control lines together on the same communication port, that is, $\overline{\text{CSTRB}}$ to $\overline{\text{CRDY}}$ and $\overline{\text{CREQ}}$ to $\overline{\text{CACK}}$. This holds the control inputs high without using external pullup resistors.

8.5 Design Tips

- Be careful with different voltage levels when running multiple ’C4x devices (or any other CMOS device) from different power supplies. This can create a CMOS latch-up that can permanently damage your device. Adding serial resistors to ’C4x communication ports connecting devices in different boards marginally helps to protect communication-port drivers. It is recommended that all ’C4x devices in the system remain in reset until power supplies are stable.

- Sometimes, it is beneficial to keep the line impedance as high as possible. This helps when interfacing to external cables. Typical ribbon cable impedance is about 100 Ω.

  Because it is sometimes difficult to route high-impedance lines (especially long ones) in a circuit board, use an external ribbon cable to jump over the length of a board. In this case, only two headers should be installed in the circuit board.

- Use an alternating signal and ground scheme. This helps control differential signal coupling and impedance variation. For quality signals, use a 26-wire ribbon ((4 control + 8 data + 1 shield) * 2 = 26). The shield is needed for the signal that is otherwise on the edge.

  Do not route signals on top of each other. When it is necessary to cross traces on adjacent layers, cross them at right angles to reduce coupling.

Note:

Because the ’C4x communication ports are very high-speed data transmission circuits, signal quality is very important. A poor quality signal can cause the missing or slipping of a byte. If this happens, the only solution is a ’C4x reset. Because at reset communication ports 0,1, and 2 are transmitters and 3, 4, and 5 are receivers, a safe reset requires resetting of every ’C4x connected to the ’C4x with the faulty condition. Global reset becomes a necessity.

8.6 Commport to Host Interface

A host interface between a ’C4x comport and a PC’s bidirectional printer port has many advantages including freeing up the DSP bus and treating the host PC as a virtual ’C4x node within a system of ’C4x devices.

This interface uses a bidirectional PC printer port interface. Logic circuits, buffers and resistors convert logic control levels driven from the printer port into ’C4x commport control signals. Signals driven from the ’C4x are converted into status signals, which can be polled in software by the PC. In addition, the PC’s printer port provides the byte-wide data path into and out of the PC.

You can use this I/O interface for host-data communication, bootloading, and debug operations. With proper buffering and software control, it is also possible to build long and reliable links. The speed is primarily dependent on the speed of the host. When using a PC as the host, the speed is limited by the PC’s I/O channel speed. If higher rates are needed, use a memory-mapped version of the printer port in the PC.

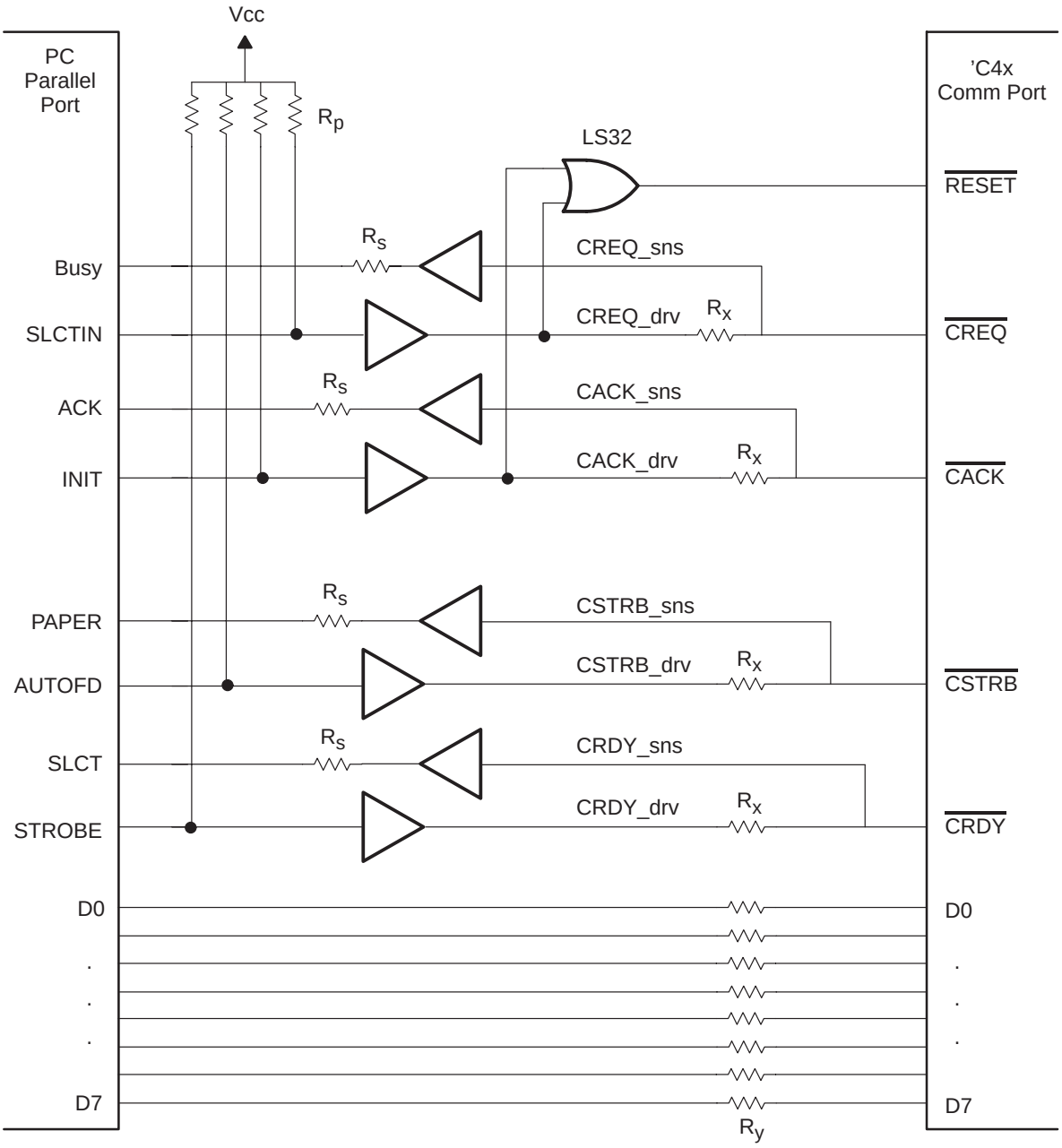
The printer port used to test this circuit was the DSP-550 from STB Systems, but there are other bidirectional printer ports on the market. Using the STB card in the bidirectional mode requires that a jumper be set (see your manual). Then, if a 1 is written to bits 5 or 7 of the control register (this depends on your printer port), data can be read back from the data register.

8.6.1 Simplified Hardware Interface for ’C40 PG $\geq$ 3.3, or ’C44 devices

Figure 8–2 shows a simplified commport signal splitter that splits each commport control signal into a simple *drive* and *sense* pair of signals. *Simplified*, in this case, means that, though the circuit is easy to follow functionally and will operate, it is not the preferred solution (see the improved driver in Figure 8–3). The signals in this circuit can be easily buffered without risk of driver conflicts. However, keep a few things in mind about the simplified design:

- Due to commport-control signal restrictions in earlier silicon revisions this circuit will not work with the TMS320C40 PG 3.0 or lower.
- This circuit requires a bidirectional printer port.
- Standard printer-port cables often do not provide ’clean’ signals
- A high value is needed for the isolation resistor in order to keep the current levels during signal opposition to a minimum. But, a low value is needed for the isolation resistor in order to insure reasonably fast rise and fall times of the commport control signals when they are inputs. This conflict can be overcome by carefully picking the correct resistor values or by adding additional biasing.

Figure 8–2. Better Commport Signal Splitter

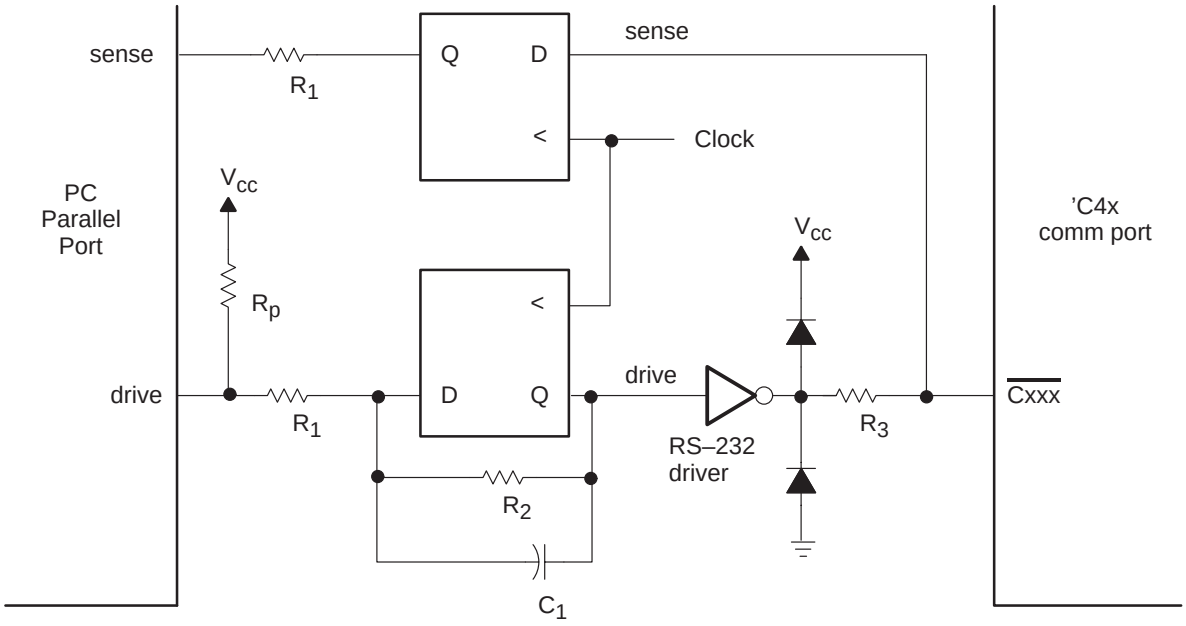


Legend: R_p = 470 ohms R_x = 180 ohms
R_s = 47 ohms R_y = 220 ohms

8.6.2 Improved Drive and Sense Amplifiers

Two improvements are suggested for the interface described above. The improvements are described in Figure 8–3.

Figure 8–3. Improved Interface Circuit



Legend: R_p = 470 ohms R_1 = 1 K ohms R_2 = 10 K ohms R_3 = 50 ohms C_1 = 100 pF

The first improvement is that the signals going to and from the printer port are synchronized using a clock and a simple data latch. By taking samples in time, noise which may be able to corrupt the first sample of a transistion will probably not be enough to corrupt the next sample. By adding a hysteris loop made from resistors R1 and R2, the noise immunity is improved more. Capacitor C1 is an additional analog filter that rejects high-frequency noise.

The next major improvement is the use of a current driver in place of the isolation resistor. In this case, an RS232 driver is used; this driver can drive beyond the supply rails of the DSP and has a built-in current limit of about 20mA. Diodes D1 and D2, along with R3, clamp the resulting signal to the supply rails of the DSP and latch to prevent excessive overdrive. The DSP and latch both have internal clamping diodes, but it is not recommended that you rely on them as the internal clamp diodes are not intended for this purpose.

8.6.3 How the Circuit Works

The PC can drive any value on the control lines, independent from the returned status. If a logic 1 is driven into the drive side of the isolation resistor and a logic 0 is observed on the sense side, the ’C4x commport signal under question is without a doubt an output.

By then driving levels and polling the returned status, it is possible to synchronize a host processor to the state machine of the ’C4x commport. The advantage of this design is that it can be easily ported to any smart processor with any basic I/O capability. For example, TMS320C31/32 devices have been used as slave devices that are bootloaded from a commport and then used as serial ports with internal memory and additional processing capabilities. Complicated and risky ASIC designs are not required and the solution is fully programmable.

You must include current limiting circuitry when designing any ’C4x interface. If the current is not limited, it can exceed 100 mA per pin, which can damage a device.

8.6.4 The Interface Software

The interface software for this host interface is available through the TI BBS (filename: M4x_2.exe). This file contains not only the low-level software drivers, but also extra code for the M4x (a multiprocessor ’C4x communication kernel) applications note. The following files are contained in this application:

- M4X Debugger (no source code)
- MEMVIEW memory and communications matrix view and edit utility
- MANDEL40 multiprocessor Mandelbrot demonstration program
- M4X.ASM multiprocessor TMS320C4x communications kernel
- DRIVER.CPP higher level system functions
- TARGET.CPP getmem, putmem, run, stop and singlestep commands
- OBJECT.CPP source code for using the printer port interface

8.7 An I/O Coprocessor–’C4x Interface

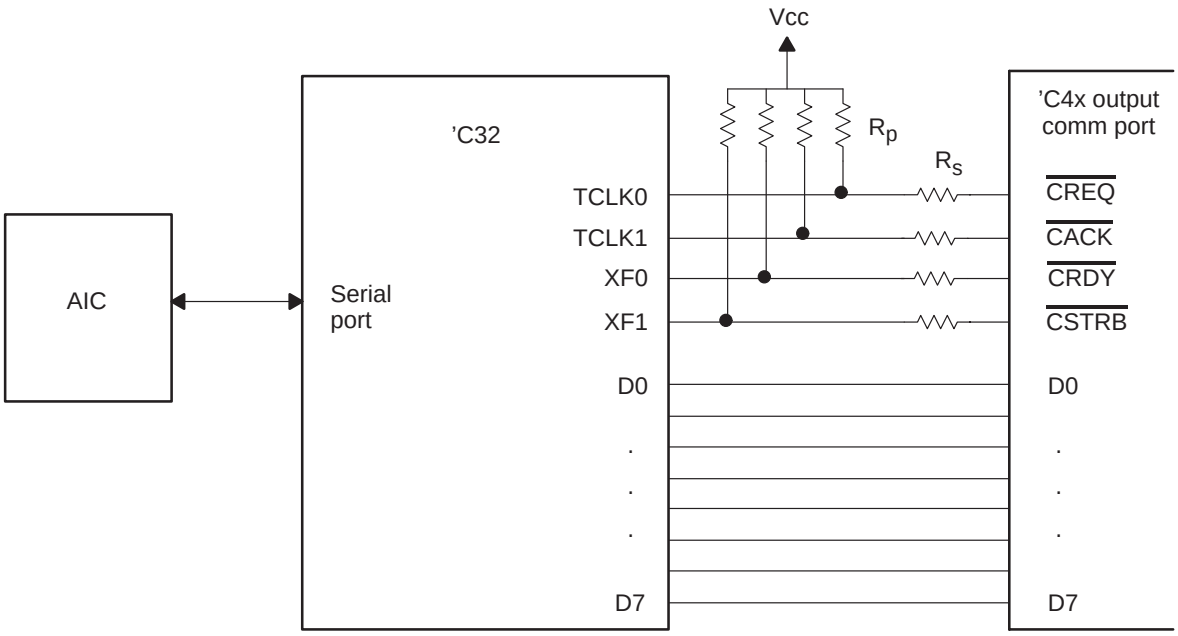
This section presents a software-based interface that provides a ’C4x with a flexible bidirectional interface to a TMS320C32. The ’C32 acts as a smart I/O coprocessor that can provide AIC interfacing and data preprocessing among others. The ’C32 is an inexpensive and flexible solution.

Some of the advantages of using an I/O coprocessor include:

- An I/O coprocessor can provide with data-processing.
- An I/O coprocessor allows for error correction and recovery from ’C4x commport interface problems.
- An I/O coprocessor can buffer data, allowing faster ’C4x data throughput.

Figure 8–4 shows the ’C32-to-’C4x interface. Through the interface, a ’C4x commport is memory-mapped to the ’C32 external memory bus. The interface uses four ’C32 I/O pins to drive the commport control signals.

Figure 8–4. A ’C32 to ’C4x Interface



Pullup resistors in the XF0, XF1, TCLK0 and TCLK1 lines are used to prevent undesired glitches due to temporary high-impedance conditions. Serial resistors are also used on the same pins for better impedance matching.

The interface software drivers and a more detailed explanation of the interface can be obtained from our TI BBS (filename 4xaic.exe). Token transfer and word transfer drivers are included with the software.

8.8 Implementing a Token Forcer

After system reset, half of the communication channels associated with a particular 'C4x have token ownership (communication ports 0, 1, 2), and the other half (communication ports 3, 4, 5) do not.

If, because of system configuration requirements, communication port direction must to be changed, the circuits shown in Figure 8–5 and Figure 8–6 can be used. The circuits force the token to be passed and communication port direction to remain changed.

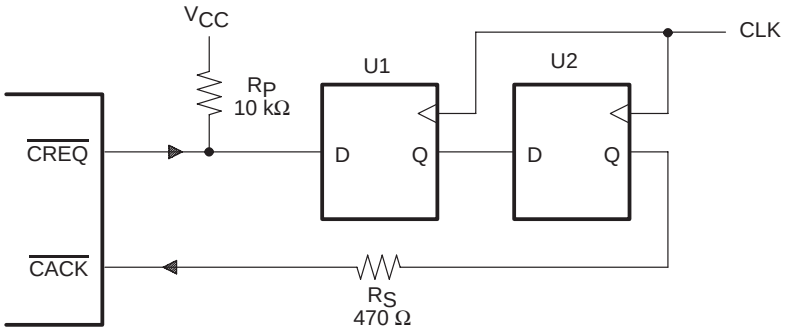
Even though these circuits are intended to force a change of the original communication port direction after reset, they can be used also to maintain the original direction. However, this can be more conveniently achieved using pullups in $\overline{\text{CACK}}$ and $\overline{\text{CREQ}}$. The pullups prevent any damage to the communication ports in the event of a program error that writes into a port configured as an input.

Forcing a communication port to become an output port

Figure 8–5 shows a circuit that forces a communication port to become an output port. In this circuit, driving the $\overline{\text{CACK}}$ line with the $\overline{\text{CREQ}}$ line reconfigures an input port as an output port. When a word is written to the FIFO, $\overline{\text{CREQ}}$ is driven low, indicating a token request. After a synchronizer delay of 1 to 2 cycles (U1 and U2), $\overline{\text{CACK}}$ is driven low, indicating a token acknowledge. $\overline{\text{CREQ}}$ then goes active high and then is held high by R_P as the line switches to an input. The CLK signal can be any clock with a frequency equal to or lower than the H1/H3 clock.

The synchronizer delay is important. If no delay is provided, the $\overline{\text{CREQ}}$ line will not be ready to change to an input high condition. As a result, the $\overline{\text{CACK}}$ line, which, at this point, is a delayed version of $\overline{\text{CREQ}}$, is inverted and applied to the $\overline{\text{CREQ}}$ line. This results in an oscillation until the synchronizer period has timed out.

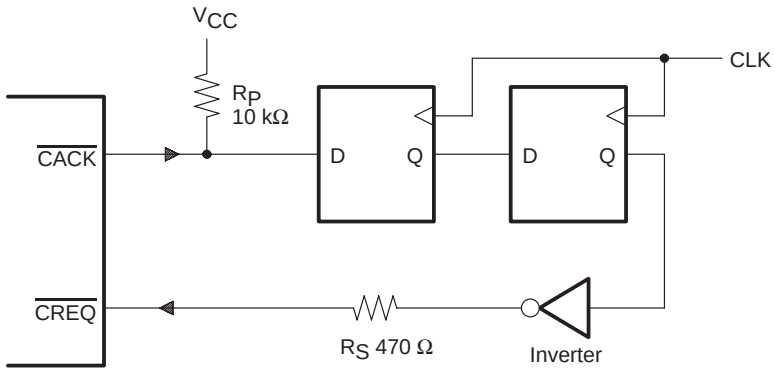
Figure 8–5. A Token Forcer Circuit (Output)



Forcing a communication port to become an input port

Figure 8–6 shows a circuit that forces a communication port to become an input port. In this circuit, driving the $\overline{\text{CREQ}}$ line with an inverted $\overline{\text{CACK}}$ reconfigures an input port as an output. If $\overline{\text{CREQ}}$ is an input, it is held low through R_S whenever $\overline{\text{CACK}}$ is high or floating high because of R_P. The port then responds to this request by driving $\overline{\text{CACK}}$ low, which, in turn, drives $\overline{\text{CREQ}}$ high, finishing the token acknowledge. As in Figure 8–5, synchronizer delays mimic the response of another ’C4x communication port to prevent oscillation.

Figure 8–6. Communication-Port Driver Circuit (Input)



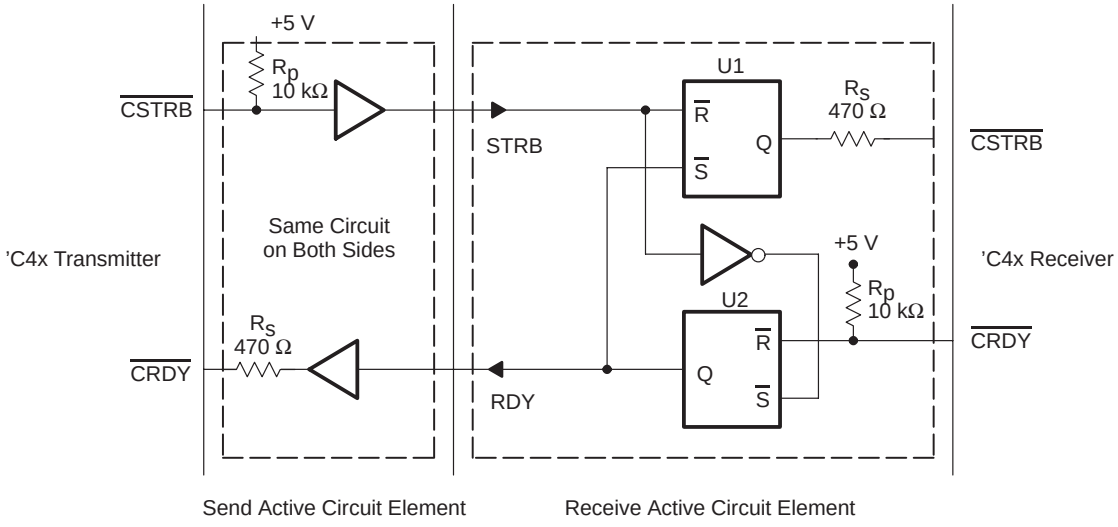
Note that after the port has been reconfigured as an input port, the $\overline{\text{CREQ}}$ line is active high while the output of the inverter is low. This causes a constant current flow from $\overline{\text{CREQ}}$ to the inverter.

8.9 Implementing a $\overline{\text{CSTRB}}$ Shortener Circuit

In ’C40 device revisions lower than 3.0, the width of the $\overline{\text{CSTRB}}$ low pulse between word boundaries should not exceed 1.0 H1/H3 at the receiving end. A $\overline{\text{CSTRB}}$ low beyond the synchronization period on a word boundary can be recognized as a new valid $\overline{\text{CSTRB}}$, resulting in an extra byte reception (byte slippage). For a short distance between two communicating ’C4x devices, byte slippage is not a problem. In ’C40 device revisions 3.0 or higher, or in any revision of the ’C44, no $\overline{\text{CSTRB}}$ width restriction exists.

The circuit shown in Figure 8–7 can reduce the width of $\overline{\text{CSTRB}}$ for very long distances when you are using ’C4x device revisions lower than 3.0. The circuit has buffers for $\overline{\text{CSTRB}}$ and $\overline{\text{CRDY}}$ on the transmitting end and two S-R flip-flops on the receiving end. On the receiving end, a low STRB incoming signal causes the Q signal of S-R flip-flop U1 to go low, forcing the $\overline{\text{CSTRB}}$ pin to go low. When $\overline{\text{CRDY}}$ responds with a low signal, S-R flip-flop U2 drives the RDY signal low. Because RDY is also tied to the $\overline{\text{S}}$ input of U1, and $\overline{\text{S}}$ has precedence over $\overline{\text{R}}$ in an S-R flip-flop, Q in U1 goes high. Also, STRB is inverted and drives the $\overline{\text{S}}$ input of U2. In this way, the width of the local $\overline{\text{CSTRB}}$ is shortened, regardless of the channel length. When the STRB signal goes back high, the S-R flip-flop pair is ready to receive another $\overline{\text{CSTRB}}$.

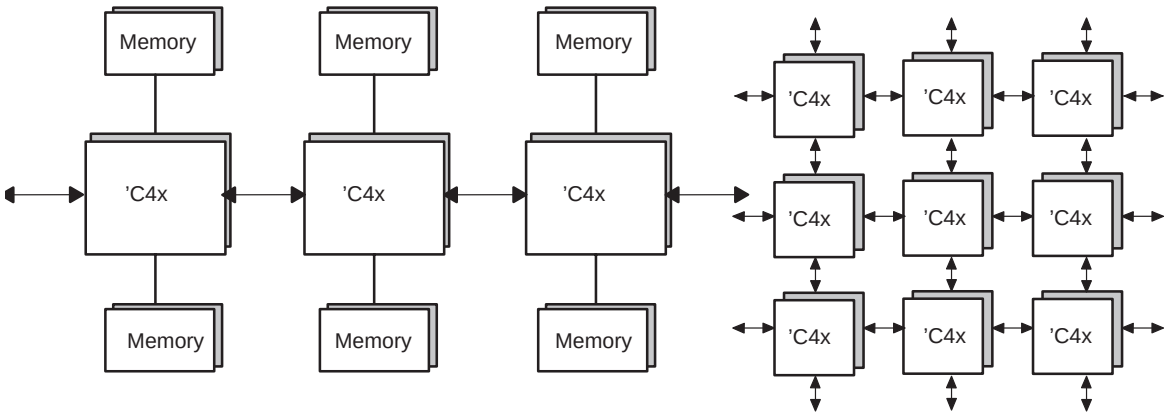
Figure 8–7. $\overline{\text{CSTRB}}$ Shortener Circuit



8.10 Parallel Processing Through Communication Ports

The ’C4x communication ports are key to parallel processing design flexibility. Many processors can be linked together in a wide variety of network configurations. In this section, Figure 8–8 illustrates ’C4x parallel processing connectivity networks that are used to fulfill many signal processing system needs.

Figure 8–8. ’C4x Parallel Connectivity Networks

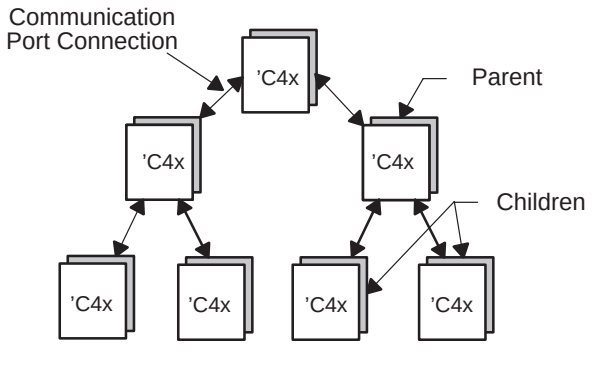


Pipelined Linear Array

For convolution and correlation and other pipelined operations in graphics and modem applications.

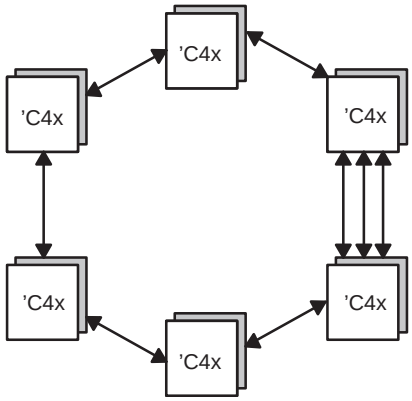
2D Array

Excellent for image processing.



Tree Structures

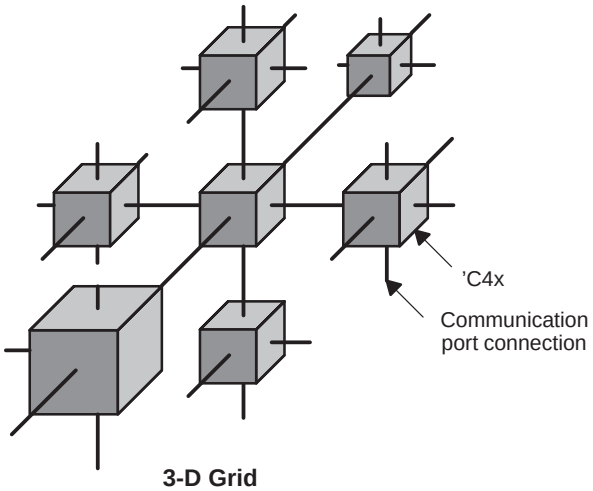
Supports broadcasting and data searches for speech and image recognition applications.



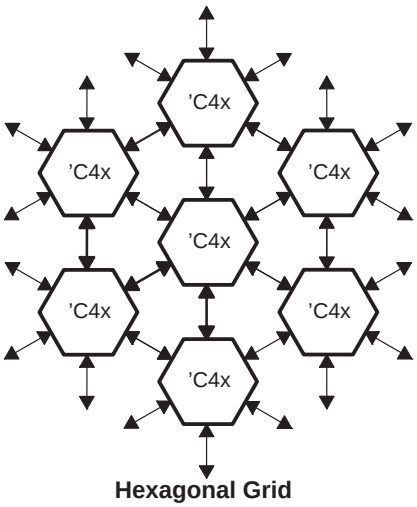
Bidirectional Ring

Clockwise and counterclockwise data flow. Group port for more I/O. Very effective for neural networks.

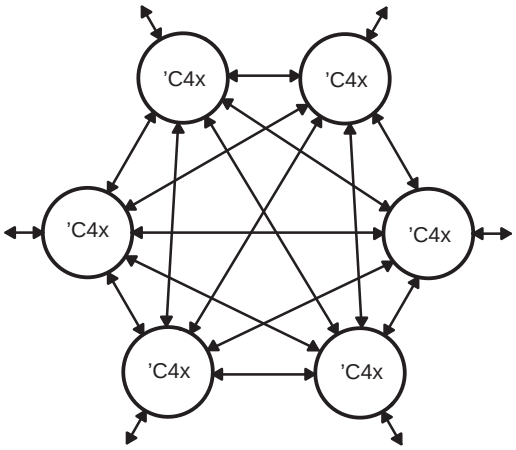
Figure 8–8. ’C4x Parallel Connectivity Networks (Continued)



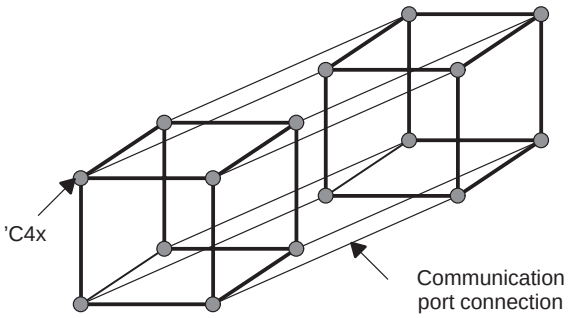
3-D Grid
For hierarchical processing such as image understanding and finite element analysis.



Hexagonal Grid
6 nearest neighbors connection. Useful in numerical analysis and image processing.



Fully-Connected Network



4-D Hypercube
A more general-purpose structure.

According to memory interface, ’C4x parallel system architecture can be classified in three basic groups:

- Shared-Memory Architecture: shares global memory among processors.
- Distributed-Memory Architecture: each processor has its own private local memory. Interprocessor communication is via ’C4x communication ports.
- Shared- and Distributed-Memory Architecture: each processor has its own local memory but also shares a global memory with other processors.

Figure 8–8 shows examples of these basic groups.

8.11 Broadcasting Messages From One ’C4x to Many ’C4x Devices

Message broadcasting from one ’C4x to many ’C4x devices requires a simple interface. However, try to avoid signal analog delays caused by distance differences between the ’C4x master and the ’C4x slave processor. These delays could create bus contention in the $\overline{\text{CSTRB}}$ and $\overline{\text{CRDY}}$ lines. Figure 8–9 shows the block diagram of a multiple processor system. In this design, one ’C4x is the dedicated transmitter, and three ’C4x devices are dedicated receivers. No reset circuitry is needed, because the transmitter is communication port 0, and the receivers are communication ports 3, 4, and 5. At reset, ’C4x communication ports 0, 1, and 2 are output ports, and communication ports 3, 4, and 5, are input ports.

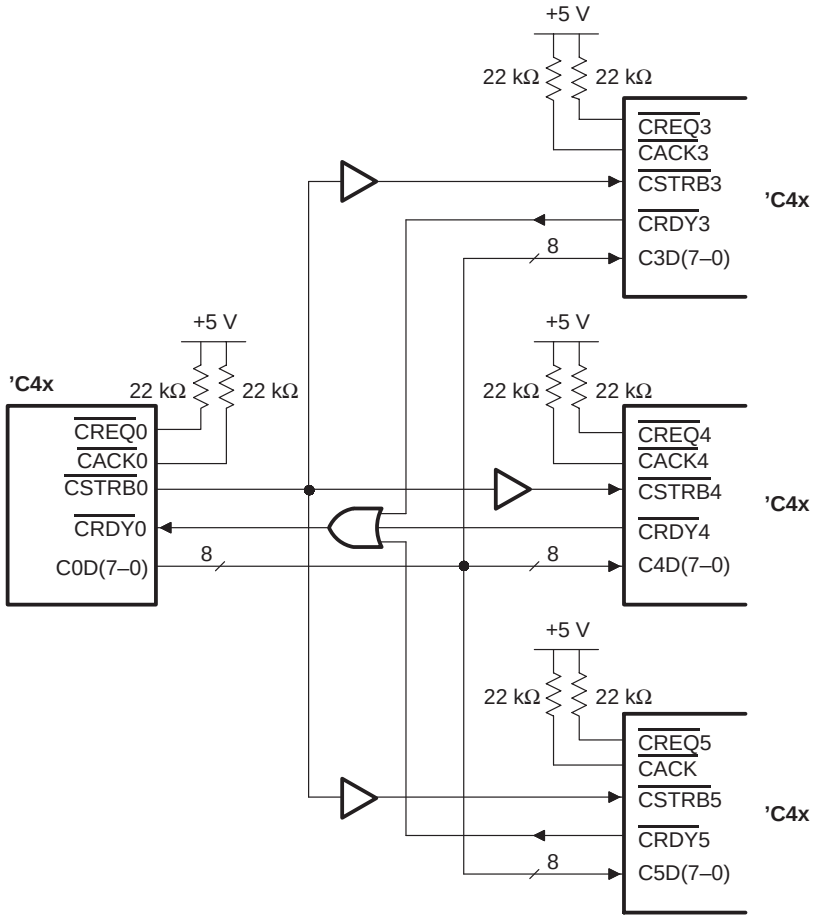
Because the communications configuration is fixed, no token transfer is needed; this allows the $\overline{\text{CREQ}}$ and $\overline{\text{CACK}}$ pins of all processors to be individually pulled up to 5 volts through 22-kΩ resistors.

In all cases, each $\overline{\text{CSTRB}}$ should be individually buffered to ensure that line reflections do not corrupt each received $\overline{\text{CSTRB}}$ signal. The data pins CD7–0 of intercommunicating ’C4x devices can be tied together. In general, for fewer than three receivers and distances shorter than six inches, data skew relative to $\overline{\text{CSTRB}}$ is not a problem, and data buffering is not needed. However, if more than three receivers must be driven by a single transmitter or the distance is more than six inches, both the $\overline{\text{CSTRB}}$ and CD7–0 lines must be buffered.

The $\overline{\text{CRDY}}$ signal input is generated by ORing the $\overline{\text{RDY}}$ outputs of all of the receiver communication ports. The transmitter should not receive a $\overline{\text{RDY}}$ signal until the receiver has received all data.

In addition, to ensure that the dedicated receiver ’C4x devices do not try to arbitrate for the communication-port bus, you should halt the output ports of the receiver ’C4x devices by setting bit four of their communication-port control registers to one.

Figure 8–9. Message Broadcasting by One ’C4x to Many ’C4x Devices



Chapter 9

’C4x Power Dissipation

The power-supply current requirement (I_{DD}) of the ’C4x vary with the specific application and the device program activity. The maximum power dissipation of a device can be calculated by multiplying I_{DD} with V_{DD} (power supply voltage requirement). Both parameters are provided in the ’C4x data sheet. Additionally, due to the inherent characteristics of CMOS technology, the current requirements depend on clock rates, output loadings, and data patterns.

This chapter presents the information you need to determine power-supply current requirements for the ’C4x under various operating conditions. After you make this determination, you can then calculate the device power dissipation, and, in turn, thermal management requirements.

Topic **Page**

9.1 Capacitive and Resistive Loading

In CMOS devices, the internal gates swing completely from one supply rail to the other. The voltage change on the gate capacitance requires a charge transfer, and therefore causes power consumption.

The required charge for a gate's capacitance is calculated by the following equation:

$$Q_{gate} = V_{DD} \times C_{gate} \text{ (coulombs)}$$

where:

Q_{gate} is the gate's charge,

V_{DD} is the supply voltage, and

C_{gate} is the gate's capacitance.

Since current is coulombs per second, the current can then be obtained from:

$$I = coul / s = V_{DD} \times C_{gate} \times Frequency$$

where:

I is the current.

For example, the current consumed by an 80-pF capacitor being driven by a 10-MHz CMOS level square wave is calculated as follows:

$$I = 5 \text{ (volts)} \times 80 \times 10^{-12} \text{(farads)} \times 10 \times 10^{6} \text{(charges/s)}$$

$$= 4 \text{ mA @ 10 MHz}$$

Furthermore, if the total number of gates in a device is known, the effective total capacitance can be used to calculate the current for any voltage and frequency. For a given CMOS device, the total number of gates is probably not known, but you can solve for a current at a particular frequency and supply voltage and later use this current to calculate for any supply voltage and operating frequency.

$$I_{device} = V_{DD} \times C_{total} \times f_{CLK}$$

where:

I_{device} is the current consumed by the device,

C_{total} is the total capacitance, and

f_{CLK} is the clock cycle.

Solving for power ($P = V \times I$), the equation becomes:

$$P_{device} = V_{DD}^2 \times C_{total} \times f_{CLK}$$

where:

P_{device} is the power consumed by the device.

In this case, C_{total} includes both internal and external capacitances. C_{total} can be effectively reduced by minimizing power-consuming internal operation and external bus cycles. Bipolar devices, pullup resistors and other devices consume DC power that adds a constant offset unaffected by f_{CLK}. The effect of these DC losses depends on data, not frequency. This document assumes an all-CMOS approach in which these effects are minimal.

Another source of power consumption is the current consumed by a CMOS gate when it is biased in the linear region. Typically, if a gate is allowed to float, it can consume current. Pullups and pulldowns of unused pins are therefore recommended.

9.2 Basic Current Consumption

Generally, power supply current requirements are related to the system—for example, operating frequency, supply voltage, temperature, and output load. In addition, because the current requirement for a CMOS device depends on the charging and discharging of node capacitance, factors such as clocking rate, output load capacitance, and data values can be important.

9.2.1 Current Components

The power supply current has four basic components:

- Quiescent
- Internal operations
- Internal bus operations
- External bus operations

9.2.2 Current Dependency

The power supply current consumption depends on many factors. Four are system related:

- Operation frequency
- Supply voltage
- Operating temperature
- Output load

Several others are related to TMS320C4x operation:

- Duty cycle of operations
- Number of buses used
- Wait states
- Cache usage
- Data value

You can calculate the total power supply current requirement for a ’C4x device by using the equation below, which comprises the four basic power supply current components and three system-related dependencies described above.

$$I_{total} = (I_q + I_{iops} + I_{ibus} + I_{xbus}) \times F \times V \times T$$

where:

I_{total} is the total supply current,

I_q is the quiescent current component,

I_{iops} is the current component due to internal operations,

I_{ibus} is the current component due to internal bus usage, including data value and cycle time dependency,

I_{xbus} is the current component due to external bus usage, including data value wait state, cycle time, and capacitive load dependency,

F is a scale factor for frequency,

V is a scale factor for supply voltage, and

T is a scale factor for operating temperature.

This report describes in detail the application of this equation and determination of all the dependencies. The power dissipation measurements in this report were taken using a ’C40 PG 3.X running at speeds up to 50 MHz and at a voltage level of 5 V.

The minimum power supply current requirement is 130 mA. The typical current consumption for most algorithms is 350 mA, as described in the TMS320C4x data sheet, unless excessive data output is being performed.

CAUTION

The maximum current requirement for a ’C4x running at 50 MHz is 850 mA and occurs only under worst case conditions: writing alternating data (AAAA AAAA to 5555 5555) out of both external buses simultaneously, every cycle, with 80 pF loads.

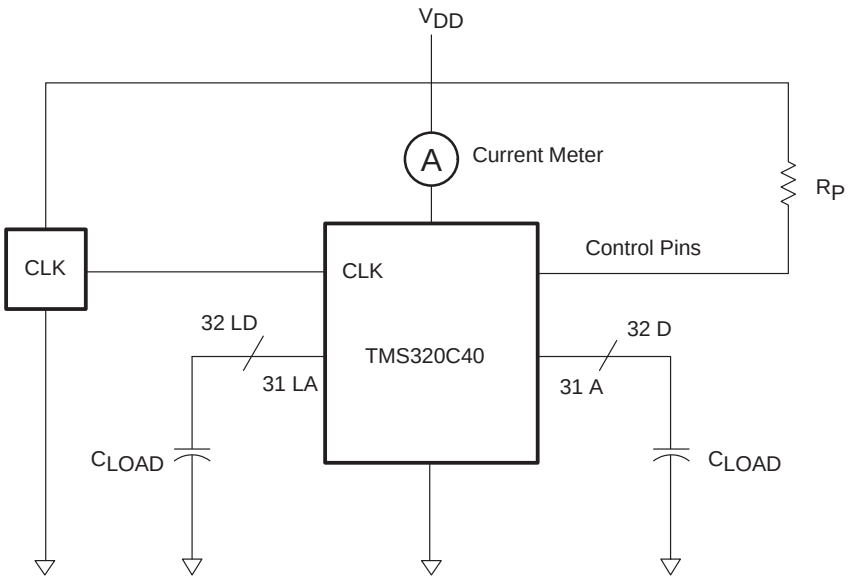
9.2.3 Algorithm Partitioning

Each part of an algorithm has its own pattern with respect to internal and external bus usage. To analyze the power supply current requirement, you must partition an algorithm into segments with distinct concentrations of internal or external bus usage. Analyze each program segment to determine its power-supply current requirement. You can then calculate the average power supply current requirement from the requirements of each segment of the algorithm.

9.2.4 Test Setup Description

All TMS320C4x supply current measurements were performed on the test setup shown in Figure 9–1. The test setup consists of a TMS320C40, capacitive loads on all data and address lines, but no resistive loads. A Tektronix digital multimeter measures the power supply current. Unless otherwise specified, all measurements are made at a supply voltage of 5 V, an input clock frequency of 50 MHz, a capacitive load of 80 pF, and an operating temperature of 25°C. Note that the current consumed by the oscillator and pullup resistors does not flow through the current meter. This current is considered part of the system’s resistive loss (see section 9.1, *Capacitive and Resistive Loading*).

Figure 9–1. Test Setup



9.3 Current Requirement of Internal Components

The power-supply current requirement for internal circuitry consists of three components: quiescent, internal operations, and internal bus operations. Quiescent and internal operations are constants, whereas the internal bus operations component varies with the rate of internal bus usage and the data values being transferred.

9.3.1 Quiescent

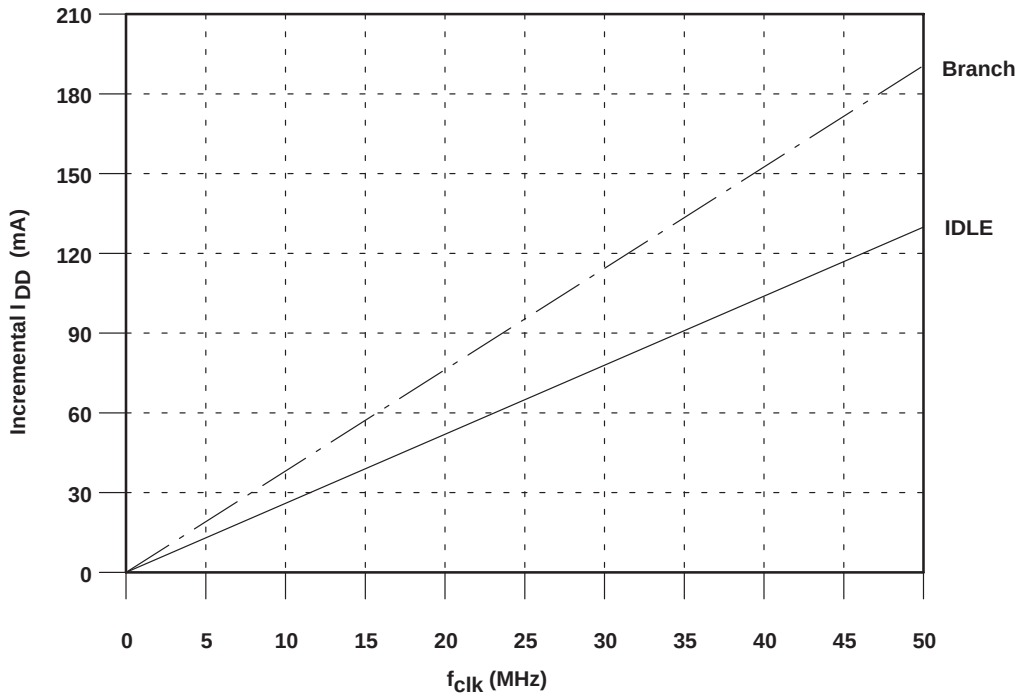
The quiescent requirement for the TMS320C4x is 130 mA while in IDLE. Quiescent refers to the baseline supply current drawn by the TMS320C4x during minimal internal activity. Examples of quiescent current include:

- Maintaining timer and oscillator
- Executing the IDLE instruction
- Holding the TMS320C4x in reset

9.3.2 Internal Operations

Internal operations include register-to-register multiplication, ALU operations, and branches, but not external bus usage or significant internal bus usage. Internal operations add a constant 60 mA above the quiescent requirement, so that the total contribution of quiescent and internal operation is 190 mA. Note, however, that internal and/or external program operations executed via an RPTS instruction do not contribute an internal operations power supply current component. During an RPTS instruction, program fetch activity other than the instruction being repeated is suspended; therefore, power-supply current is related only to the data operations performed by the instruction being executed.

Figure 9–2. Internal and Quiescent Current Components

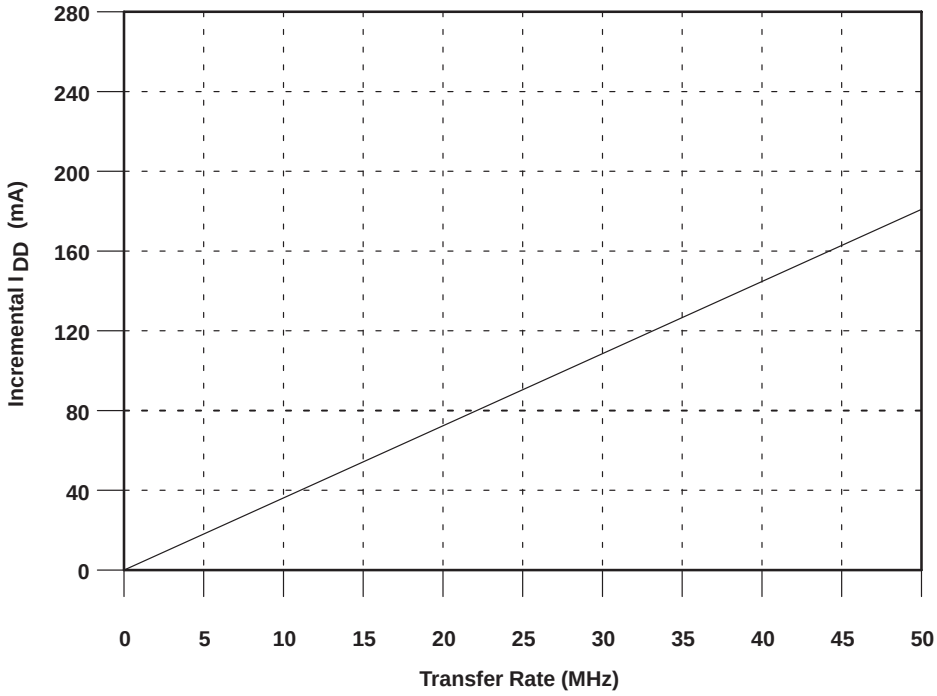


9.3.3 Internal Bus Operations

The internal bus operations include all operations that utilize the internal buses extensively, such as internal RAM accesses every cycle. No distinction is made between internal reads or writes, such as instruction or operand fetches from internal memory, because internally they are equal. Significant use of internal buses adds a data-dependent term to the equation for the power supply current requirement. Recall that switching requires more current. Hence, changing data at high rates requires higher power-supply current.

Pipeline conflicts, use of cache, fetches from external wait-state memory, and writes to external wait-state memory all affect the internal and external bus cycles of an algorithm executing on the TMS320C4x. Therefore, you must determine the algorithm's internal bus usage in order to accurately calculate power supply current requirements. The TMS320C4x software simulator and XDS emulator both provide benchmarking and timing capabilities that help you determine bus usage.

Figure 9–3. Internal Bus Current Versus Transfer Rate



The current resulting from internal bus usage varies linearly with transfer rates. Figure 9–3 shows internal bus-current requirements for transferring alternating data (AAAA AAAAh to 5555 5555h) at several frequencies. Note that transfer rates greater than the TMS320C4x's MIPS rating are possible because of internal parallelism.

The data set AAAA AAAAh to 5555 5555h exhibits the maximum internal bus current for data transfer operations. The current required for transferring other data patterns may be derated accordingly, as described later in this subsection.

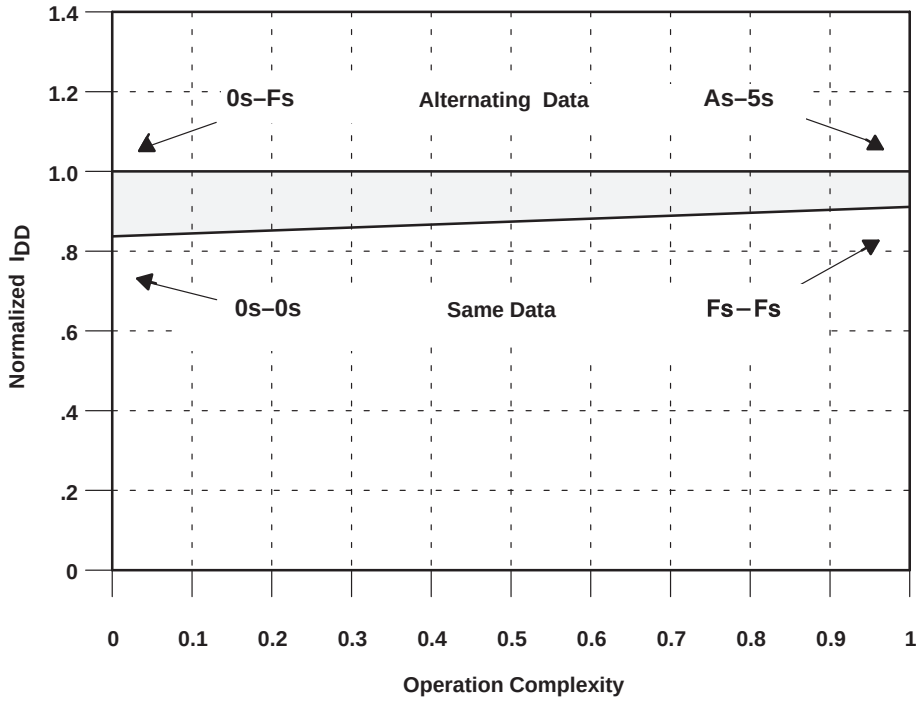
As the transfer rate decreases (that is, transfer-cycle time increases) the incremental I_{DD} approaches 0 mA. This figure represents the incremental I_{DD} due to internal bus operations and is added to quiescent and internal operations current values.

For example, the maximum transfer rate corresponds to three accesses every cycle (one program fetch and two data transfers) or an effective one-third H1 transfer cycle time. At this rate, 178 mA is added to the quiescent (130 mA) and internal operation (60 mA) current values for a total of 368 mA.

Figure 9–3 shows the internal bus current requirement when transferring As followed by 5s for various transfer rates. Figure 9–4 shows the data dependence of the internal bus-current requirement when the data is other than As followed by 5s. The trapezoidal region bounds all possible data values transferred. The lower line represents the scale factor for transferring the same data. The upper line represents the scale factor for transferring alternating data (all 0s to all Fs or all As to all 5s, etc.).

The possible permutation of data values is quite large. The term relative data complexity refers to a relative measure of the extent to which data values are changing and the extent to which the number of bits are changing state. Therefore, relative data complexity ranges from 0, signifying minimal variation of data, to a normalized value of 1, signifying greatest data variation.

Figure 9–4. Internal Bus Current Versus Data Complexity Derating Curve



If a statistical knowledge of the data exists, Figure 9–4 can be used to determine the exact power supply requirement on the basis of internal bus usage. For example, Figure 9–4 indicates a 89.5% scale factor when all Fs (FFFF FFFFh) are moved internally every cycle with two accesses per cycle (80 Mbytes per second). Multiplying this scale factor by 178 mA (from

Figure 9–3) yields 159 mA due to internal bus usage. Therefore, an algorithm running under these conditions requires about 349 mA of power supply current (130 + 60 + 159).

Since a statistical knowledge of the data may not be readily available, a nominal scale factor may be used. The median between the minimum and maximum values at 50% relative data complexity yields a value of 0.93 and can be used as an estimate of a nominal scale factor. Therefore, this nominal data scale factor of 93% can be used for internal bus data dependency, adding 165.5 mA to 130 mA (quiescent) and 60 mA (internal operations) to yield 355.5 mA. As an upper bound, assume worst case conditions of three accesses of alternating data every cycle, adding 178 mA to 130 mA (quiescent) and 60 mA (internal operations) to yield 368 mA.

9.4 Current Requirement of Output Driver Components

The output driver circuits on the TMS320C4x are required to drive significantly higher DC and capacitive loads than internal device logic drivers. Because of this, output drivers impose higher supply current requirements than other sections of circuitry in the device.

Accordingly, the highest values of supply current are exhibited when external writes are being performed at high speed. During read cycles, or when the external buses are not being used, the TMS320C4x is not driving the data bus; this eliminates a significant component of the output buffer current. Furthermore, in many typical cases, only a few address lines are changing, or the whole address bus is static. Under these conditions, an insignificant amount of supply current is consumed. Therefore, when no external writes are being performed or when writes are performed infrequently, current due to output buffer circuitry can be ignored.

When external writes are being performed, the current required to supply the output buffers depends on several considerations:

- Data pattern being transferred
- Rate at which transfers are being made
- Number of wait states implemented (because wait states affect rates at which bus signals switch)
- External bus DC and capacitive loading

External bus operations involve external writes to the device and constitute a major power-supply current component. The power supply current for the external buses, made up of four components, is summarized in the following equation:

$$I_{xbus} = (I_{base\ local} + I_{local}) + (I_{base\ global} + I_{global})$$

where:

$I_{base\ local/global}$ is the current consumed by the internal driver and pin capacitance,

I_{local} is the local bus current component, and

I_{global} is the global bus current component.

The remainder of this section describes in detail the calculation of external bus current requirements.

Note:

The DMA current component (I_{DMA}) and communication port current component (I_{CP}) should be included in the calculation of I_{xbus} if they are used in the operations.

9.4.1 Local or Global Bus

The current due to bus writes varies with write cycle time. As discussed in the previous section, to obtain accurate current values, you must first determine the rate and timing for write cycles to external buses by analyzing program activity, including any pipeline conflicts that may exist. To do this, you can use information from the TMS320C4x emulator or simulator as well as the *TMS320C4x User's Guide*. In your analysis, you must account for effects from the use of cache, because use of cache can affect whether or not instructions are fetched from external memory.

When evaluating external write activity in a given program segment, you must consider whether or not a particular level of external write activity constitutes significant activity. If writes are being performed at a slow enough rate, they do not impact supply current requirements significantly and can be ignored. This is the case, however, only if writes are being performed at very slow rates on either the local or global bus.

When bus-write cycle timing has been established, Figure 9–5 can be used to determine the contribution to supply current due to bus activity. Figure 9–5 shows values of current contribution from the local or global bus for various transfer rates. This data was gathered when alternating values of 5555 55555h and AAAA AAAAh were written at a capacitive load of 80 pF per output signal line. This condition exhibits the highest current values on the device. The values presented in the figure represent the incremental current contributed by the local or global bus output driver circuitry under the given conditions. Current values obtained from this graph are later scaled and added to several other current terms to calculate the total current for the device. As indicated in the figure, the lower limit $I_{base} = I_q + I_{iops} + I_{ibus}$ is essentially I_{total} for transfer rates less than 1 Mword/second.

Figure 9–5. Local/Global Bus Current Versus Transfer Rate and Wait States

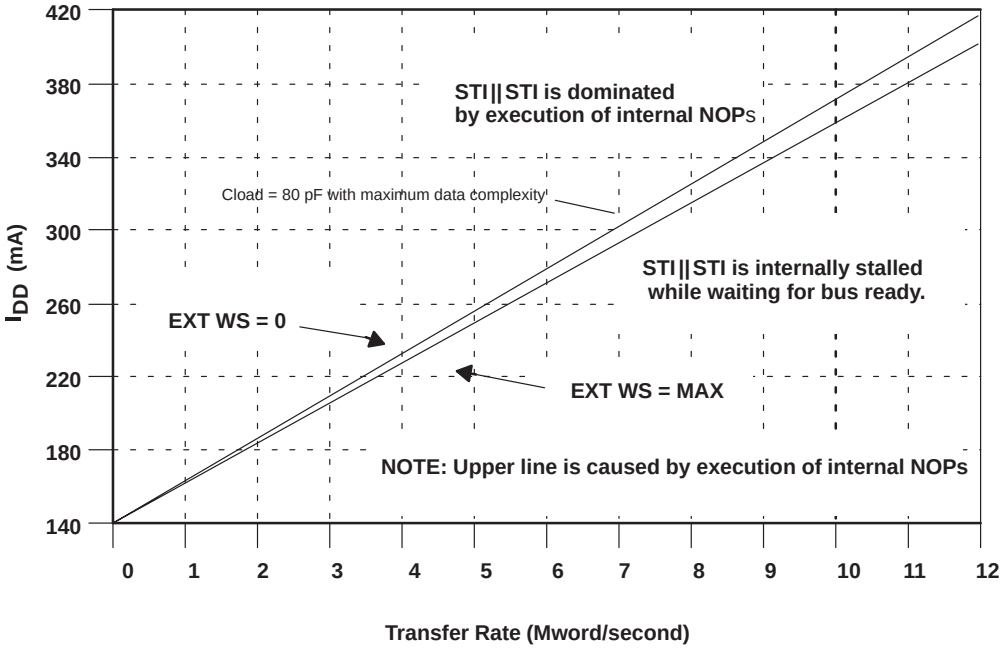
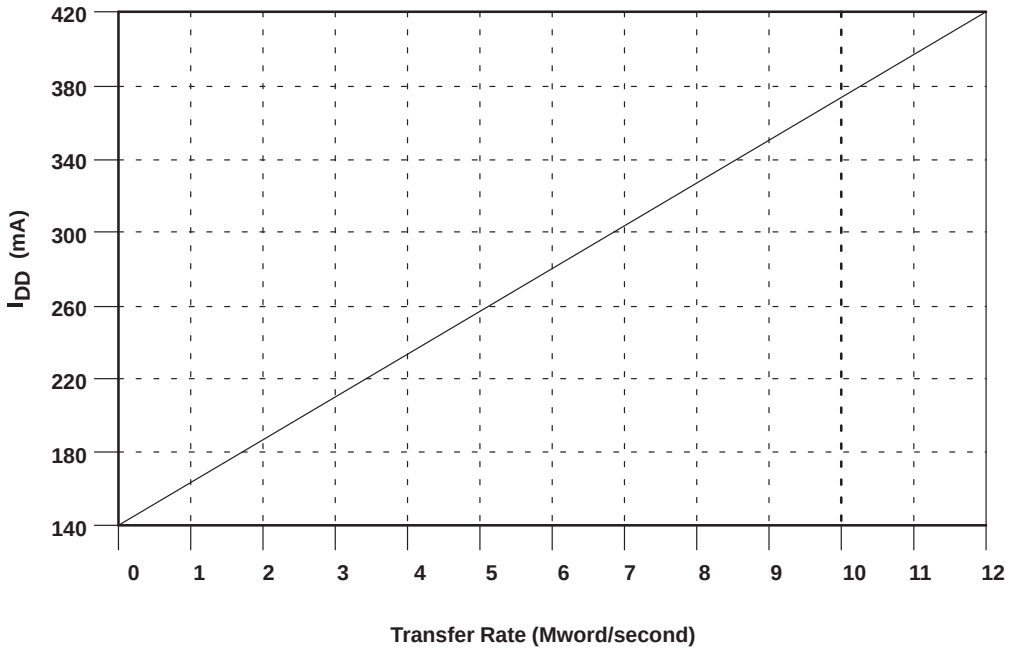


Figure 9–5 demonstrates a feature of the ’C4x’s external bus architecture known as a posted write. In general, data is written to a latch (or a one deep FIFO) and held by the bus until the bus cycle is complete. Since the CPU may not require that bus again for some time, the CPU is free to perform operations on other buses until a conflict occurs. Conflicts include DMA, a second write, or a read to the bus.

In Figure 9–5, the upper line is applicable when STI || STI is not dominated by execution of internal NOPs and the external wait state is equal to zero. The lower line shows when STI || STI is internally stalled while waiting for the external bus to go ready because of wait states. The addition of NOPs between successive STI || STI operations contributes to internal bus current and therefore does not result in the lowest possible current.

Figure 9–6. Local/Global Bus Current Versus Transfer Rate at Zero Wait States



To further illustrate the relationship of current and write cycle time, Figure 9–6 shows the characteristics of current for various numbers of cycles between writes for zero wait states. The information on this graph can be used to obtain more precise values of current whenever zero wait states are used. Table 9–1 lists the number of cycles used for software generated wait states.

Table 9–1. Wait State Timing Table

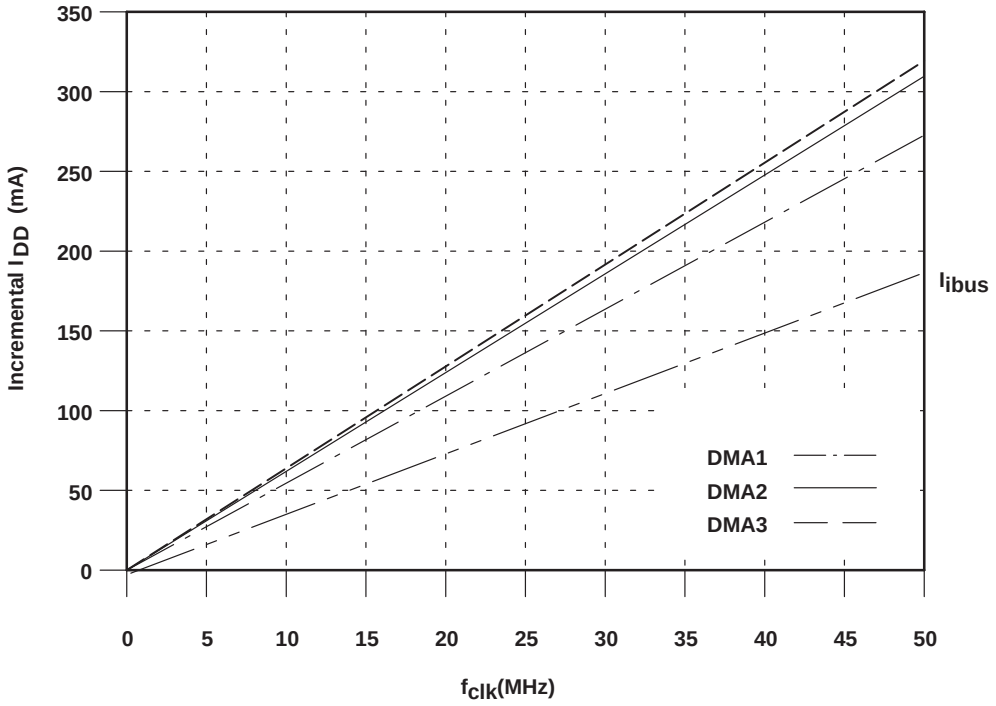
Wait State	Read Cycles	Write Cycles
0	1	2
1	2	3
2	3	4
3	4	5

Once a current value has been obtained from Figure 9–5 or Figure 9–6, this value can be scaled by a data dependency factor if necessary, as described on page 9-16. This scaled value is then summed along with several other current terms to determine the total supply current.

9.4.2 DMA

Using DMA to transfer data consumes power that is data dependent. The current resulting from DMA bus usage (I_{DMA}) varies linearly with the transfer rate. Figure 9–7 shows DMA bus current requirements for transferring alternating data (AAAA AAAAh to 5555 5555h) at several transfer rates; it also shows that current consumption increases when more DMA channels are used. However, as more DMA channels are used, the incremental change in current diminishes as the internal DMA bus becomes saturated. Note that DMA current is superimposed over I_{ibus} (internal bus) value.

Figure 9–7. DMA Bus Current Versus Clock Rate

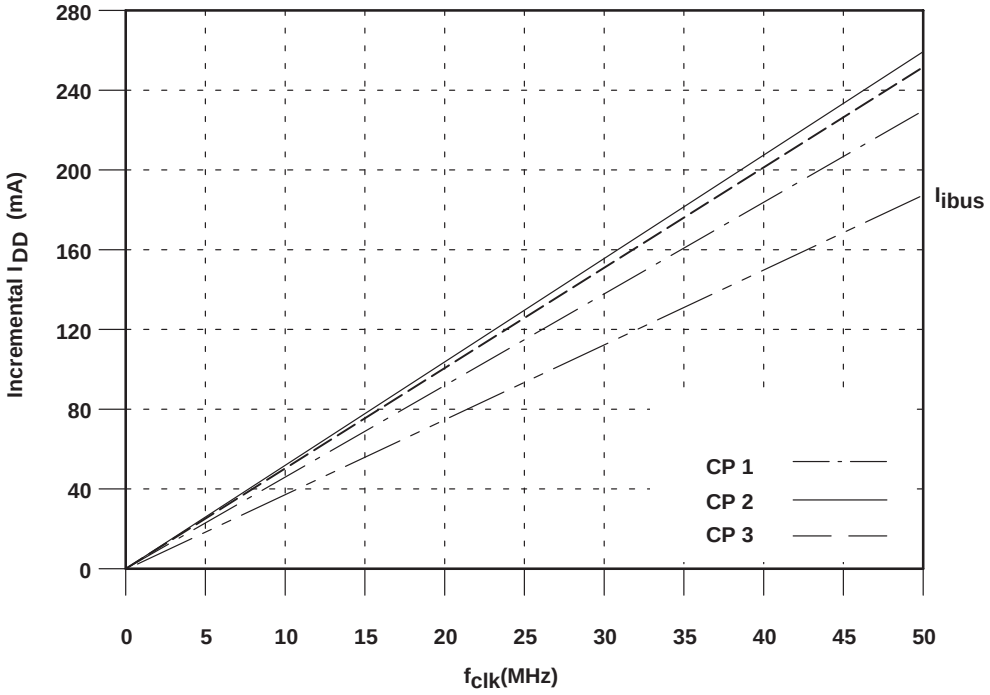


9.4.3 Communication Port

Communication port operations add a data-dependent term to the equation for the current requirement. The current resulting from communication port operation (I_{CP}) varies linearly with the transfer rate. Figure 9–8 shows communication port operation current requirements for transferring alternating data (AAAA AAAAh to 5555 5555h) at several transfer rates; it also shows that current consumption increases when more communication port channels are

used. Similar to the DMA bus current consumption, adding communication ports eventually saturates the peripheral bus as more channels are added.

Figure 9–8. Communication Port Current Versus Clock Rate



Note that since the communication ports are intended to communicate with other TMS320C4x communication ports over short distances, no additional capacitive loading was added. In this case, the transmission distance is about 6 inches without additional 80-pF loads. Note that communication port current is superimposed over I_{ibus} value.

9.4.4 Data Dependency

Data dependency of the current for the local and global buses is expressed as a scale factor that is a percentage of the maximum current exhibited by either of the two buses.

Figure 9–9. Local/Global Bus Current Versus Data Complexity

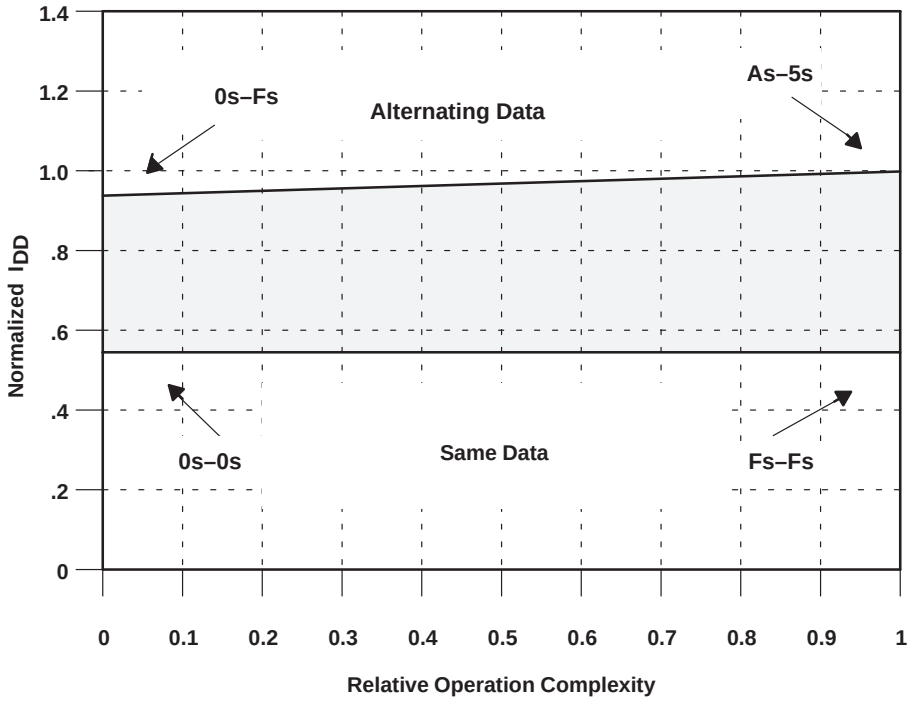


Figure 9–9 shows normalized weighting factors that can be used to scale current requirements on the basis of patterns in data being written on the external buses. The range of possible weighting factors forms a trapezoidal pattern bounded by extremes of data values. As the figure shows, the minimum current occurs when all zeros are written, while the maximum current occurs when alternating 5555 5555h and AAAA AAAAh are written. This condition results in a weighting factor of 1, which corresponds to using the values from Figure 9–5 and/or Figure 9–6 directly.

As with internal bus operations, data dependencies for the external buses are well defined, but accurate prediction of data patterns is often either impossible or impractical. Therefore, unless you have precise knowledge of data patterns, you should use an estimate of a median or average value for the scale factor. Assuming that data will be neither 5s and As nor all 0s and will be varying randomly, then a value of 0.80 is appropriate. Otherwise, if you prefer a conservative approach, you can use a value of 1.0 as an upper bound.

Regardless of the approach taken for scaling, once you determine the scale factor for the buses, apply this factor to the current values you determined with the graphs in section 9.4.1, *Local or Global Bus*.

For example, if a nominal scale factor of 0.80 for the buses is assumed, the current contribution from the two buses is as follows:

Local or Global : 0.80 $\times$ 133 mA = 106.4 mA

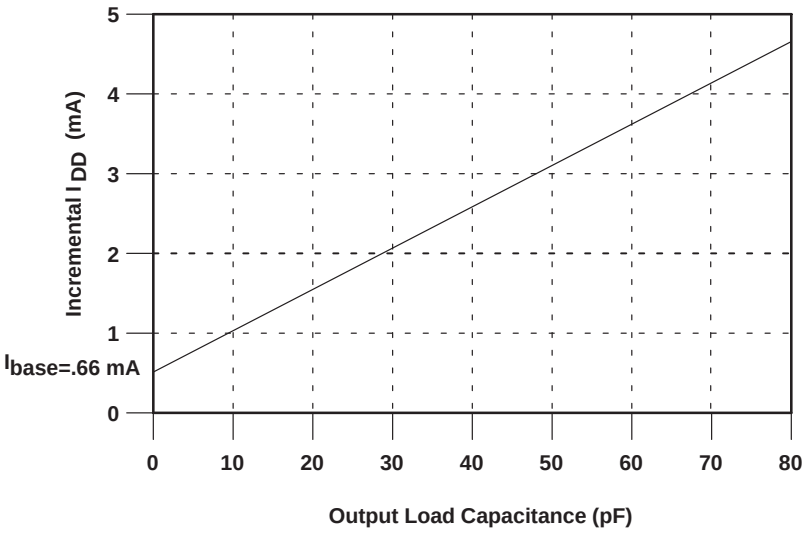
9.4.5 Capacitive Loading Dependence

Once cycle timing and data dependencies have been accounted for, capacitive loading effects should be calculated and applied. Figure 9–10 shows the current values obtained above as a function of actual load capacitance if the load capacitance presented to the buses is less than 80 pF.

In the previous example, if the load capacitance is 20 pF instead of 80 pF, the actual pin current would be 1.66 mA.

While the slope of the line in Figure 9–10 can be used to interpolate scale factors for loads greater than 80 pF, the TMS320C4x is specified to drive output loads less than 80 pF; interface timings cannot be guaranteed at higher loads. With data dependency and capacitive load scale factors applied to the current values for local and global buses, the total supply current required for the device for a particular application can be calculated, as described in the next section.

Figure 9–10. Pin Current Versus Output Load Capacitance (10 MHz)



9.5 Calculation of Total Supply Current

The previous sections have discussed currents contributed by different sources on the TMS320C4x. Because determinations of actual current values are unique and independent for each source, each current source was discussed separately. In an actual application, however, the sum of the independent contributions determines the total current requirement for the device. This total current value is exhibited as the total current supplied to the device through all of the V_{DD} inputs and returned through the V_{SS} connections.

Note that numerous V_{DD} and V_{SS} pins on the device are routed to a variety of internal connections, not all of which are common. Externally, however, all of these pins should be connected in parallel to 5 V and ground planes, providing very low impedance.

As mentioned previously, because of the inherent differences in operations between program segments, it is usually appropriate to consider current for each of the segments independently. In this way, peak current requirements are readily obtained. Further, you can make average current calculations to use in determining heating effects of power dissipation. These effects, in turn, can be used to determine thermal management considerations.

9.5.1 Combining Supply Current Due to All Components

To determine the total supply current requirements for any given program activity, calculate each of the appropriate components and combine them in the following sequence:

1) Start with 130 mA quiescent current requirement.

2) Add 60 mA for internal operations unless the device is dormant, such as when executing IDLE or using an RPTS instruction to perform internal and/or external bus operations (see *Internal Operations* section on page 9-7). Internal or external bus operations executed via RPTS do not contribute an internal operations power supply current component. Therefore, current components in the next two steps may still be required, even though the 60 mA is omitted.

3) If significant internal bus operations are being performed (see subsection 9.3.2, *Internal Bus Operations* on page 9-8), add the calculated current value.

4) If external writes are being performed at high speed (see Section 9.4, *Current Requirements of Output Driver Component*s on page 9-12), then add the values calculated for local and global bus current components.

5) Add DMA and communication port current requirements if they are used.

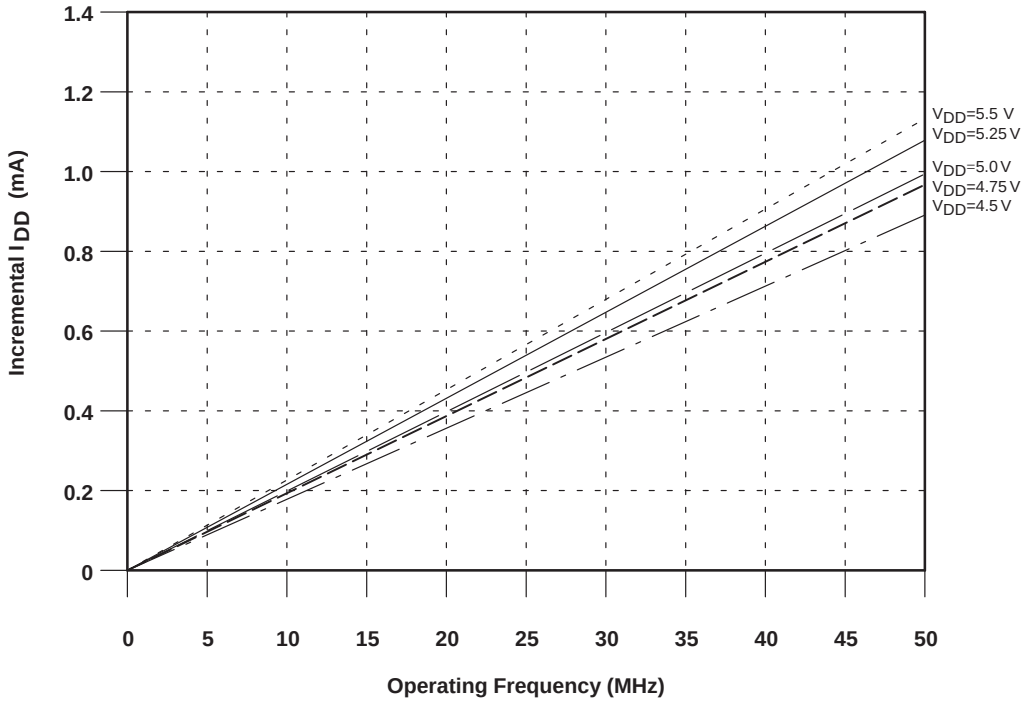
The current value resulting from summing these components is the total device current requirement for a given program activity.

9.5.2 Supply Voltage, Operating Frequency, and Temperature Dependencies

Three additional factors that affect current requirements are supply voltage level, operating temperature, and operating frequency. However, these considerations affect total supply current, not specific components (that is, internal or external bus operations). Note that supply voltages, operating temperature, and operating frequency must be maintained within required device specifications.

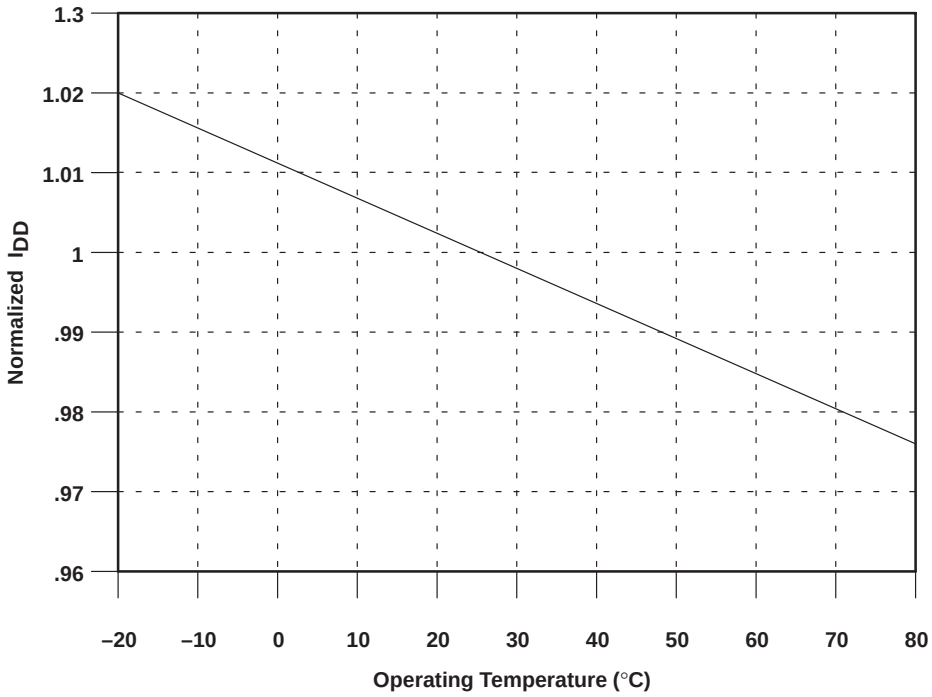
The scale factor for these dependencies is applied in the same manner as discussed in previous sections, once the total current for a particular program segment has been determined. Figure 9–11 shows the relative scale factors to be applied to the supply current values as a function of both V_{DD} and operating frequency.

Figure 9–11. Current Versus Frequency and Supply Voltage



Power-supply current consumption does not vary significantly with operating temperature. However, you can use a scale factor of 2% normalized I_{DD} per 50°C change in operating temperature to derate current within the specified range noted in the TMS320C4x data sheet.

Figure 9–12. Change in Operating Temperature (°C)



This temperature dependence is shown graphically in Figure 9–12. Note that a temperature scale factor of 1.0 corresponds to current values at 25°C, which is the temperature at which all other references in the document are made.

9.5.3 Design Equation

The procedure for determining the power-supply current requirement can be summarized in the following equation:

$$I_{total} = (I_{qidle} + I_{iops} + I_{ibus} + I_{xbusglobal} + I_{xbuslocal} + I_{DMA} + I_{cp}) \times F \times V \times T$$

where:

F is a scale factor for frequency

V is a scale factor for supply voltage

T is a scale factor for operating temperature

Table 9–2 describes the symbols used in the power-supply current equation and gives the value and the number from which the value is obtained.

Table 9–2. Current Equation Typical Values (F_{CLK} = 40 MHz)

Symbol	Value			Note	Reference
	Min	Typical	Max		
I_{qidle2}	–	20 μA	50 μA	Idle2 shutdown	Figure 9–2
I_{qidle}	130 mA	130 mA	130 mA	Internal idle	Figure 9–2
I_{iops}	60 mA	60 mA	60 mA	Branch to self internal	Figure 9–2
I_{ibus}	0 mA	50 mA	190 mA	Data dependent	Figure 9–3, Figure 9–4
$I_{xbusglobal}$ (max)	0 mA	50 mA	280 mA	Data and C_{load} dependent	Figure 9–5, Figure 9–6, Figure 9–9
$I_{xbuslocal}$ (max)	0 mA	50 mA	280 mA	Data and C_{load} dependent	Figure 9–5, Figure 9–6, Figure 9–9
I_{DMA}	0 mA	50 mA	300 mA	Data and source/ destination dependent	Figure 9–7
I_{CP}	0 mA	50 mA	250 mA	Data dependent	Figure 9–8

Notes: 1) All values are scaled by frequency and supply voltage. The nominal tested frequency is 40 MHz.
2) Externally-driven signals are capacitive-load dependent.
3) It is unrealistic to add all of the maximum values, since it is impossible to run at those levels.

9.5.4 Average Current

Over the course of an entire program, some segments typically exhibit significantly different levels of current for different durations. For example, a program may spend 80% of its time performing internal operations and draw a current of 250 mA; it may spend the remaining 20% of its time performing writes at full speed to both buses and drawing 790 mA.

While knowledge of peak current levels is important in order to establish power supply requirements, some applications require information about average current. This is particularly significant if periods o
f high peak current are short in duration. You can obtain average current by performing a weighted sum of the current due to the various independent program segments over time. You can calculate the average current for the example in the previous paragraph as follows:

$I = 0.8 \times 250\ mA + 0.2 \times 790\ mA = 358\ mA$

Using this approach, you can calculate average current for any number of program segments.

9.5.5 Thermal Management Considerations

Heating characteristics of the TMS320C4x are dependent upon power dissipation, which, in turn, is dependent upon power supply current. When mak-

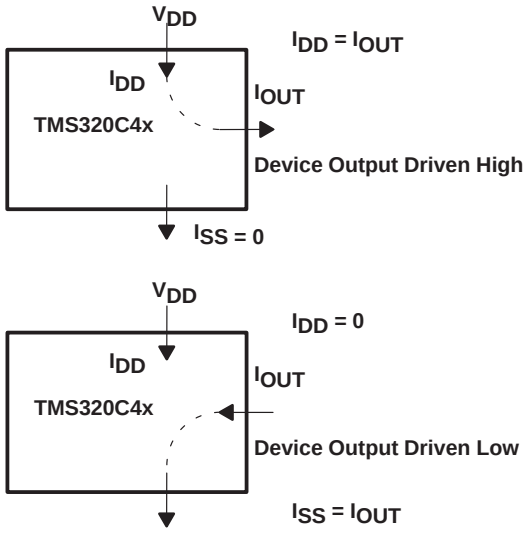
ing thermal management calculations, you must consider the manner in which power supply current contributes to power dissipation and to the TMS320C4x package thermal characteristics' time constant.

Depending on the sources and destinations of current on the device, some current contributions to I_{DD} do not constitute a component of power dissipation at 5 volts. That is to say, the TMS320C4x may be acting only as a switch, in which case, the voltage drop is across a load and not across the 'C4x. If the total current flowing into V_{DD} is used to calculate power dissipation at 5 volts, erroneously large values for package power dissipation will be obtained. The error occurs because the current resulting from driving a logic high level into a DC load appears only as a portion of the current used to calculate system power dissipation due to V_{DD} at 5 volts. Power dissipation is defined as:

$$P = V \times I$$

where P is power, V is voltage, and I is current. If device outputs are driving any DC load to a logic high level, only a minor contribution is made to power dissipation because CMOS outputs typically drive to a level within a few tenths of a volt of the power supply rails. If this is the case, subtract these current components out of the TMS320C4x supply current value and calculate their contribution to system power dissipation separately (see Figure 9–13).

Figure 9–13. Load Currents



Furthermore, external loads draw supply current (I_{DD}) only when outputs are driven high, because when outputs are in the logic zero state, the device is sinking current through V_{SS}, which is supplied from an external source. Therefore, the power dissipation due to this component will not contribute through I_{DD} but will contribute to power dissipation with a magnitude of:

$$P = V_{OL} \times I_{OL}$$

where V_{OL} is the low-level output voltage and I_{OL} is the current being sunk by the output, as shown in Figure 9–13. The power dissipation component due to outputs being driven low should be calculated and added to the total power dissipation.

When outputs with DC loads are being switched, the power dissipation components from outputs being driven high and outputs being driven low should be averaged and added to the total device power dissipation. Power components due to DC loading of the outputs should be calculated separately for each program segment before average power is calculated.

Note that unused inputs that are left unconnected may float to a voltage level that will cause the input buffer circuits to remain in the linear region, and therefore contribute a significant component to power supply current. Accordingly, if you want absolute minimum power dissipation, you should make any unused inputs inactive by either grounding or pulling them high. If several unused inputs must be pulled high, they can be pulled high together through one resistor to minimize component count and board space.

When you use power dissipation values to determine thermal management considerations, use the average power unless the time duration of individual program segments is long. The thermal characteristics of the TMS320C40 in the 325-pin PGA package are exponential in nature with a time constant on the order of minutes. Therefore, when subjected to a change in power, the temperature of the device package will require several minutes or more to reach thermal equilibrium.

If the duration of program segments exhibiting high power dissipation values is short (on the order of a few seconds) in comparison to the package thermal characteristics' time constant, use average power calculated in the same manner as average current described in the previous section. Otherwise, calculate maximum device temperature on the basis of the actual time required for the program segments involved. For example, if a particular program segment lasts for 7 minutes, the device essentially reaches thermal equilibrium due to the total power dissipation during the period of device activity.

Note that the average power should be determined by calculating the power for each program segment (including all considerations described above) and performing a time average of these values, rather than simply multiplying the average current by V_{DD}, as determined in the previous subsection.

Calculate specific device temperature by using the TMS320C4x thermal impedance characteristics included in the TMS320C4x data sheet.

9.6 Example Supply Current Calculations

An FFT represents a typical DSP algorithm. The FFT code used in this calculation processes data in the RAM blocks. The entire algorithm consists mainly of internal bus operations and hence includes quiescent and, in general, internal operations. At the end of the processing, the results are written out on the global and local bus. Therefore, the algorithm exhibits a higher current requirement during the write portion where the external bus is being used significantly.

9.6.1 Processing

The processing portion of the algorithm is 95% of the total algorithm. During this portion, the power-supply current is required for the internal circuitry only. Data is processed in several loops that make up the majority of the algorithm. During these loops, two operands are transferred on every cycle. The current required for internal bus usage, then, is 60 mA (from Figure 9–3). The data is assumed to be random. A data value scale factor of 0.93 is used (from Figure 9–4). This value scales 60 mA, yielding 55.8 mA for internal bus operations. Adding 55.8 mA to the quiescent current requirement and internal operations current requirement yields a current requirement of 245.8 mA for the major portion of the algorithm.

$I = I_q + I_{iops} + I_{ibus}$
$I = 130\ mA + 60\ mA + (60\ mA)\ (0.93)$
$= 245.8\ mA$

9.6.2 Data Output

The portion of the algorithm corresponding to writing out data is approximately 5% of the total algorithm. Again, the data that is being written is assumed to be random. From Figure 9–4 and Figure 9–10, scale factors of 0.93 and 0.8 are used for derating due to data value dependency for internal and local buses, respectively. During the data dump portion of the code, a load and a store are performed every cycle; however, the parallel load/store instruction is in an RPTS loop. Therefore, there is no contribution due to internal operations, because the instruction is fetched only once. The only internal contributions are due to quiescent and internal bus operations. Figure 9–5 indicates a 23-mA current contribution due to writes every available cycle. Therefore, the total contribution due to this portion of the code is:

$I = I_q + I_{ibus} + I_{xbus}$

or

$I = 130\ mA + (60\ mA)\ (0.93) + 85\ mA + (23\ mA)\ (0.8)$

$= 289.2\ mA$

9.6.3 Average Current

The average current is derived from the two portions of the algorithm. The processing portion took 95% of the time and required about 245.8 mA; the data dump portion took the other 5% and required about 411.6 mA. The average is calculated as:

I_{avg} = (0.95) (245.8 mA) + (0.05) (289.2 mA)

= 247.97 mA

From the thermal characteristics specified in the *TMS320C4x User's Guide*, it can be shown that this current level corresponds to a case temperature of 28°C. This temperature meets the maximum device specification of 85°C and hence requires no forced air cooling.

9.6.4 Experimental Results

A photograph of the power-supply current for the FFT, using a 40-MHz system clock, is shown in Appendix A. During the FFT processing, the current varied between 190 and 220 mA. The current during external writes had a peak of 230 mA, and the average current requirement as measured on a digital multimeter was 205 mA. Scaling those results to the 50-MHz calculations yielded results that were close to the actual measured power-supply current.

9.7 Design Considerations

Designing systems for minimum power dissipation involves reducing device operating current requirements due to signal switching rate, capacitive loading, and other effects. Selective consideration of these effects makes it possible to optimize system performance while minimizing power consumption. This section describes current reduction techniques based on operating current dependencies of the device as discussed in previous sections of this document.

9.7.1 System Clock and Signal Switching Rates

Since current (and therefore, power) requirements of CMOS devices are directly proportional to switching frequency, one potential approach to minimizing operating power is to minimize system clock frequency and signal switching rates. Although performance is often directly proportional to system clock and signal switching rates, tradeoffs can be made in both areas to achieve an optimal balance between power usage and performance in the design of a system.

If reducing power is a primary goal, and a given system design does not have particularly demanding performance requirements, the system clock rate can be reduced with the corresponding savings in power. Minimum power is realized when system clock rates are only as fast as necessary to achieve required system performance. Additionally, if overall system clock rates cannot be reduced, an alternative approach to power reduction is to reduce clock speed wherever possible during periods of inactivity.

Also, the appropriate choice of clock generation approach will ensure minimum system power dissipation. The use of an external oscillator rather than the on-chip oscillator can result in lower power device and system power dissipation levels. As described previously, the internal oscillator can require as much as 10 mA when operating at 40 MHz. If you use an external oscillator that requires less than 10 mA for clock generation, overall system power is reduced.

When considering switching rates of signals other than the system clock, the main consideration is to minimize switching. Specifically, any unnecessary switching should be avoided. Outputs or inputs that are unused should either be disabled, tied high, or grounded, whichever is appropriate. Additionally, outputs connected to external circuitry should drive other power dissipation elements only when absolutely necessary.

9.7.2 Capacitive Loading of Signals

Current requirements are also directly proportional to capacitive loading. Therefore, all capacitive loading should be minimized. This is especially significant for device outputs.

The approaches to minimize capacitive loading are consistent with efficient PC board layout and construction practices. Specifically, signal runs should be as short as possible, especially for signals with high switching rates. Also, signals should not run long distances across PC boards to edge connectors unless absolutely necessary.

Note that the buffering of device outputs that must drive high capacitive loads reduces supply current for the TMS320C40, but this current is translated to the buffering device. Whether or not this is a valid tradeoff must be determined at the system level. The two main considerations are: 1) whether the power required by the buffers is more or less than the power required from the 'C40 to drive the load in question, and 2) whether or not off-loading the power to the buffers has any implications with respect to system power-down modes. It may be desirable to use buffers to drive high capacitive loads, even though they may require more current than the TMS320C40, especially in cases where part of the system may be powered down but the TMS320C40 is still required to interface to other low capacitance loads.

9.7.3 DC Component of Signal Loading

In order to achieve lowest device current requirements, the internal and external DC load component of device input and output signal loading must also be minimized .

Any device inputs that are unused and left floating may cause excessively high DC current to be drawn by their input buffer circuitry. This occurs because if an input is left unconnected, the voltage on the input may float to a level that causes the input buffer to be biased at a point within its range of linear operation. This can cause the input buffer circuit to draw a significant DC current directly from V_{DD} to ground. Therefore, any unused device inputs should be pulled up to V_{DD} via a resistor pullup of nominally 20 kΩ, or driven high with an unused gate. Input-only pins that are not used can be pulled up in parallel with other inputs of the same type with a single gate or resistor to minimize system component count. In this case, up to 15 or more standard device inputs can be pulled up with a single resistor.

Any device I/O pins that are unused should be selected as outputs. This avoids the requirement for pull-ups (to ensure that the I/O input stage is not biased in the linear region) and therefore eliminates an unnecessary current component.

For any device output, any DC load present is directly reflected in the system's power-supply current. Therefore, DC loading of outputs should be reduced to a minimum. If DC currents are being sourced from the address bus outputs, the address bus should be set to a level that minimizes the current through the external load. This can be accomplished by performing a dummy read from an external address.

For I/O pins that must be used in both the input and output modes, individual pullup resistors of nominally 20 kΩ should be used to ensure minimum power dissipation if these pins are not always driven to a valid logic state. This is particularly true of the data-bus pins. When the bus is not being driven explicitly, it is left floating, which can cause excessively high currents to be drawn on the input buffer section of all 64 bits of the bus. In this case, because all 64 data bus bits are normally used independently in most applications, each data-bus pin should be pulled up with a separate resistor for minimum power.

Chapter 10

Development Support and Part Order Information

This chapter provides development support information, socket descriptions, device part numbers, and support tool ordering information for the ’C4x.

Each ’C4x support product is described in the *TMS320 Family Development Support Reference Guide* (literature number SPRU011). In addition, more than 100 third-party developers offer products that support the TI TMS320 family. For more information, refer to the *TMS320 Third-Party Reference Guide* (literature number SPRU052).

For information on pricing and availability, contact the nearest TI Field Sales Office or authorized distributor. See the list at the back of this book.

Topic **Page**

10.1 Development Support

Texas Instruments offers an extensive line of development tools for the TMS320C4x generation of DSPs, including tools to evaluate the performance of the processors, generate code, develop algorithm implementations, and fully integrate and debug software and hardware modules.

The following products support the development of ’C4x applications:

Code Generation Tools

- The optimizing ANSI C compiler translates ANSI C language directly into highly optimized assembly code. You can then assemble and link this code with the TI assembler/ linker, which is shipped with the compiler. It supports both ’C3x and ’C4x assembly code. This product is currently available for PCs (DOS, DOS extended memory, OS/2), VAX/VMS and SPARC workstations. See the *TMS320 Floating-Point DSP Optimizing C Compiler User’s Guide* (SPRU034) for detailed information about this tool.
- The assembler/linker converts source mnemonics to executable object code. It supports both ’C3x and ’C4x assembly code. This product is currently available for PCs (DOS, DOS extended memory, OS/2). The ’C3x/’C4x assembler for the VAX/VMS and SPARC workstations is only available as part of the optimizing ’C3x/’C4x compiler. See the *TMS320 Floating-Point DSP Assembly Language Tools User’s Guide* (SPRU035) for detailed information about available assembly-language tools.
- The digital filter design package helps you design digital filters.

System Integration and Debug Tools

- The simulator simulates (via software) the operation of the ’C4x and can be used in C and assembly software development. This product is currently available for PCs (DOS, Windows) and SPARC workstations. See the *TMS320C4x C Source Debugger User’s Guide* (SPRU054) for detailed information about the debugger.
- The XDS510 emulator performs full-speed in-circuit emulation with the ’C4x, providing access to all registers as well as to internal and external memory of the device. It can be used in C and assembly software development and has the capability to debug multiple processors. This product is currently available for PCs (DOS, Windows, OS/2) and SPARC workstations. This product includes the emulator board (emulator box, power supply, and SCSI connector cables in the SPARC version), the ’C4x C Source Debugger and the JTAG cable.

  Because ’C3x and ’C5x XDS510 emulators also come with the same emulator board (or box) as the ’C4x, you can buy the ’C4x C Source Debugger

Software as a separate product called ’C4x C Source Debugger Conversion Software. This enables you to debug ’C3x/’C4x applications with the same emulator board. The emulator cable that comes with the ’C3x XDS510 emulator cannot be used with the ’C4x. A JTAG emulation conversion cable (see Section 10.3) is needed instead. The emulator cable that comes with the ’C5x XDS510 emulator can also be used for the ’C4x without any restriction. See the *TMS320C4x C Source Debugger User’s Guide* (SPRU054) for detailed information about the ’C4x emulator.

- The parallel processing development system (PPDS) is a stand-alone board with four ’C4xs directly connected to each other via their communication ports. Each ’C4x has 64K-words SRAM and 8K-byte EPROM as local memory, and they all share a 128K-word global SRAM. See the *TMS320C4x Parallel Processing Development System Technical Reference* (SPRU075) for detailed information about the PPDS.
- The emulation porting kit (EPK) enables you to integrate emulation technology directly into your system without the need of an XDS510 board. This product is intended to be used by third parties and high-volume board manufacturers and requires a licensing agreement with Texas Instruments.

10.1.1 Third-Party Support

The TMS320 family is supported by products and services from more than 100 independent third-party vendors and consultants. These support products take various forms (both as software and hardware), from cross-assemblers, simulators, and DSP utility packages to logic analyzers and emulators. The expertise of those involved in support services ranges from speech encoding and vector quantization to software/hardware design and system analysis.

See the *TMS320 Third-Party Support Reference Guide* (literature number SPRU052) for a more detailed description of services and products offered by third parties.

10.1.2 The DSP Hotline

For answers to TMS320 technical questions on device problems, development tools, documentation, upgrades, and new products, you can contact the DSP hotline via:

- **Phone**: (713)274–2320 Monday through Friday from 8:30 a.m. to 5:00 p.m. central time
- **Fax**: (713)274–2324. (US DSP Hotline), +33–1–3070–1032 (European DSP hotline)

- **Electronic Mail**: 4389750@mcimail.com

To ask about third-party applications and algorithm development packages, contact the third party directly. Refer to the *TMS320 Third-Party Support Reference Guide* (SPRU052) for addresses and phone numbers.

Extensive DSP documentation is available; this includes data sheets, user's guides, and application reports. Contact the hotline for information on **literature** that you can request from the Literature Response Center, (800)477–8924.

The DSP hotline does not provide pricing information. Contact the nearest TI Field Sales Office for prices and availability of TMS320 devices and support tools.

10.1.3 The Bulletin Board Service (BBS)

The TMS320 DSP Bulletin Board Service (BBS) is a telephone-line computer service that provides information on TMS320 devices, specification updates for current or new devices and development tools, silicon and development tool revisions and enhancements, new DSP application software as it becomes available, and source code for programs from any TMS320 user's guide.

You can access the BBS via:

- **Modem**: (300-, 1200-, or 2400-bps) dial (713)274–2323. Set your modem to 8 data bits,1 stop bit, no parity.

To find out more about the BBS, refer to the *TMS320 Family Development Support Reference Guide* (literature number SPRU011).

10.1.4 Internet Services

Texas Instruments offers two Internet-accessible services for DSP support: an ftp site, and a www site.

- **World-wide web**: Point your browser at *http://www.ti.com* to access TI's web site. At the site, you can follow links to find product information, online literature, an online lab, and the *320 Hotline online*.

- **FTP**: Use anonymous *ftp* to *ti.com* (Internet port address 192.94.94.1) to access copies of the files found on the BBS. The BBS files are located in the subdirectory called *mirrors*.

10.1.5 Technical Training Organization (TTO) TMS320 Workshops

'C4x DSP Design Workshop. This workshop is tailored for hardware and software design engineers and decision-makers who will be designing and utilizing the 'C4x generation of DSP devices. Hands-on exercises throughout the course give participants a rapid start in developing 'C4x design skills. Microprocessor/assembly language experience is required. Experience with digital design techniques and C language programming experience is desirable.

These topics are covered in the 'C4x workshop:

- 'C4x architecture/instruction set
- Use of the PC-based software simulator
- Use of the 'C3x/'C4x assembler/linker
- C programming environment
- System architecture considerations
- Memory and I/O interfacing
- Development support

For registration information, pricing, or to enroll, call (800)336–5236, ext. 3904.

10.2 Sockets

Table 10–1 contains available sockets that accept the 325-pin ’C40 pin grid array (PGA) and the 304–pin ’C44 Plastic Quad Flatpack (PQF). Table 10–2 lists the phone numbers of the manufacturers listed in Table 10–1.

Table 10–1. Sockets that Accept the 325-pin ’C40 and the 304-pin ’C44

Manufacturer	Type	Part Number
Advanced Interconnections	C40-wire-wrap socket	3919
AMP	C40-tool-activated ZIF socket	AMP 382533–9
AMP	Actuation tool for AMP382533–9	AMP 854234–1
AMP	C40-handle-activated ZIF socket	AMP 382320–9
AMP	C40-PGA ZIF	AMP 55291–2
Emulation Technology	C40-logic analyzer socket	BZ6–325–H6A35–TMS320C40Z
Emulation Technology	C40-wire-wrap socket	AB–325–H6A35Z–P13–M
Mark Eyelet	C40-wire-wrap socket	MP325–73311D16
Yamaichi	TMS320C44 PDB Socket (304 pins)	ic201–3044–004

Table 10–2. Manufacturer Phone Numbers

Manufacturer	Phone Number
AMP	(717) 564–0100
Advanced Interconnections	(401) 823–5200
Emulation Technology	(408) 982–0660
Mark Eyelet	(203) 756–8847
Yamaichi	(408) 456–0797

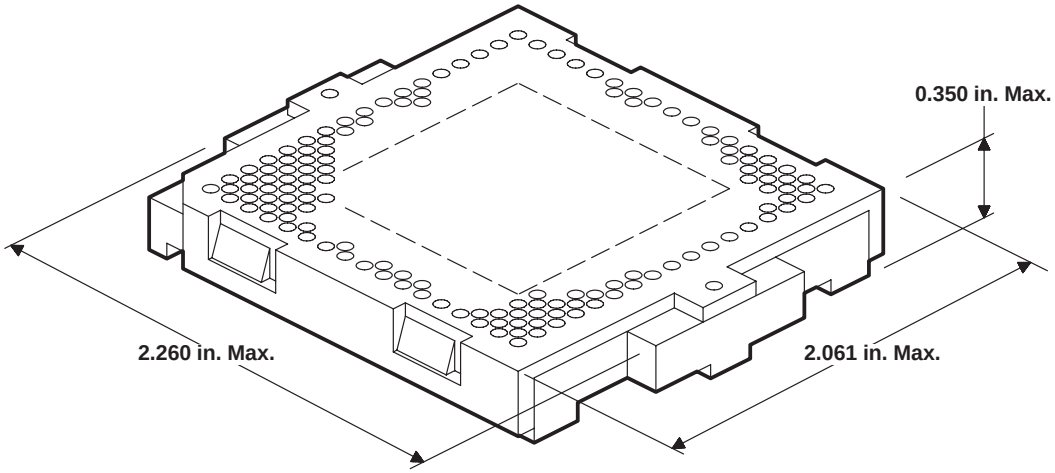
The remainder of this section describes two available sockets that accept the ’C4x pin grid array (PGA). Both sockets feature zero insertion force (ZIF):

- A tool-activated ZIF socket (TAZ)
- A handle-activated ZIF socket (HAZ)

The sockets described herein are manufactured by AMP Incorporated.

10.2.1 Tool-Activated ZIF PGA Socket (TAZ)

Figure 10–1. Tool-Activated ZIF Socket



Description:

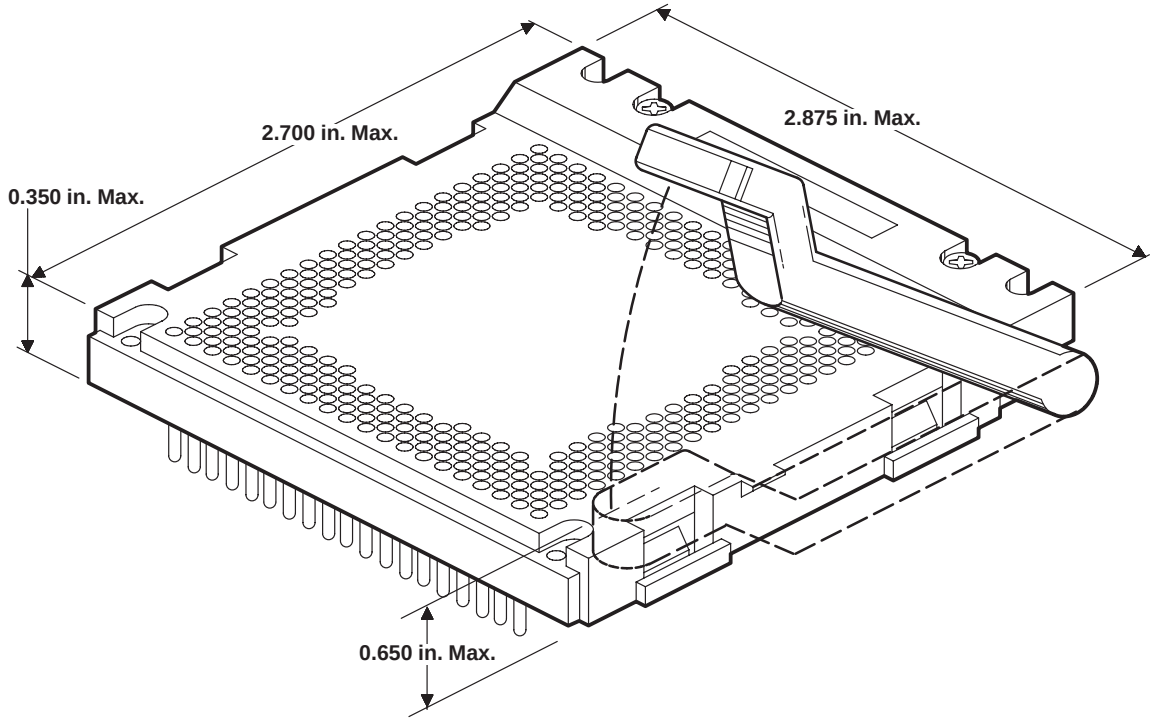
AMP part number:	382533–9
Pin positions:	325
Soldertail length:	0.170 in. for PC boards 0.125 in. thick (other tail lengths available)
Actuator tool	354234–1

Features:

- ❑ Slightly larger than a PGA device
- ❑ Easy package loading because of large funnel entry
- ❑ Zero insertion force
- ❑ Contact wiping action during insertion ensures clean contact points
- ❑ Spring-loaded cover ensures proper loading
- ❑ Can be used with robotic insertion and removal
- ❑ Horizontal vs. vertical socket forces prevent damage to the device

10.2.2 Handle-Activated ZIF PGA Socket (HAZ)

Figure 10–2. Handle-Activated ZIF Socket



Description:

AMP part number:	382320–9
Pin positions:	325
Solder tail length:	0.170 in. for PC boards 0.125 in. thick (other tail lengths available)

Features:

- ❑ Can be used for test and burn-in
- ❑ Spring contacts are normally closed
- ❑ Easy package loading because of large funnel entry
- ❑ Zero insertion force
- ❑ Contact wiping action during socket closing ensures clean contact points
- ❑ Maximum Operating temperature is 160° C (to allow burn-in capability)

10.3 Part Order Information

This section describes the part numbers of 'C4x devices, development support hardware, and software tools.

10.3.1 Nomenclature

To designate the stages in the product development cycle, Texas Instruments assigns prefixes to the part numbers of all TMS320 devices and support tools. Each TMS320 device has one of three prefixes: TMX, TMP, or TMS. Each support tool has one of two possible prefix designators: TMDX or TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices and tools (TMS/TMDS). This development flow is defined below.

Device Development Evolutionary Flow:

TMX The part is an experimental device that is not necessarily representative of the final device's electrical specifications.

TMP The part is a device from a final silicon die that conforms to the device's electrical specifications but has not completed quality and reliability verification.

TMS The part is a fully qualified production device.

Support Tool Development Evolutionary Flow:

TMDX The development-support product that has not yet completed Texas Instruments internal qualification testing.

TMDS The development-support product is a fully qualified development support product.

TMX and TMP devices and TMDX development support tools are shipped with the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

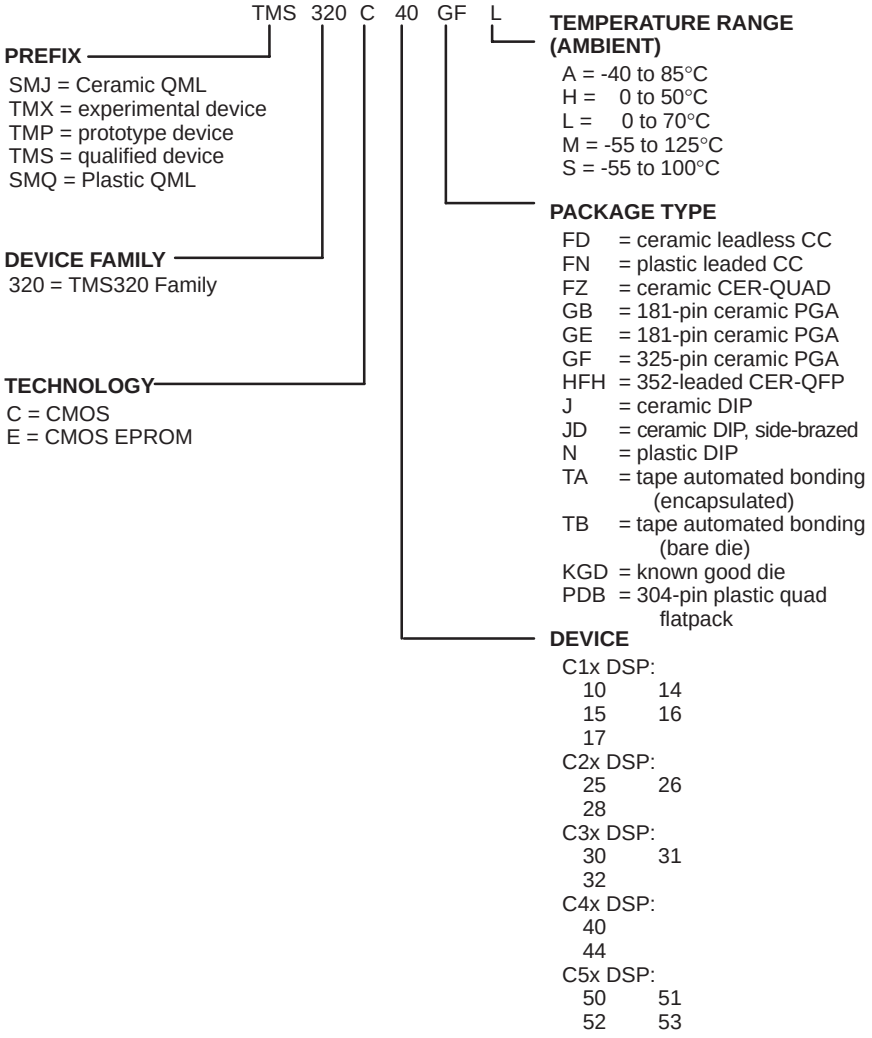
TMS devices and TMDS development support tools have been fully characterized, and the quality and reliability of the device has been fully demonstrated. Texas Instruments standard warranty applies to these products.

Note:

It is expected that prototype devices (TMX or TMP) have a greater failure rate than standard production devices. Texas Instruments recommends that these devices *not* be used in any production system, because their expected end-use failure rate is still undefined. Only qualified production devices should be used.

TI device nomenclature also includes the device family name and a suffix. This suffix indicates the package type (for example, N, FN, or GB) and temperature range (for example, L). Figure 10–3 provides a legend for reading the complete device name for any TMS320 family member.

Figure 10–3. Device Nomenclature



10.3.2 Device and Development Support Tools

Table 10–3 lists ’C4x device part numbers. Table 10–4 lists the development support tools available for the ’C4x DSP, their part numbers, and the platform on which they run.

Table 10–3. Device Part Numbers

Device Part Number	Voltage	Operating Frequency	Comm Ports	Package
TMS320C40GFL	5V	50 MHz/40 ns	6	325-pin ceramic PGA
TMS320C40GFL60	5V	60 MHz/33 ns	6	325-pin ceramic PGA
TMS320C44PDB50	5V	50 MHz/40 ns	4	304-pin PQFP
TMS320C44PDB60	5V	60 MHz/33 ns	4	304-pin PQFP
SMJ320C40GFM40	5V	40MHz/50 ns	6	325-pin ceramic PGA
SMJ320C40GFM50	5V	50MHz/40 ns	6	325-pin ceramic PGA
SMJ320C40HFHM40	5V	40MHz/50 ns	6	352-lead ceramic PGA
SMJ320C40HFHM50	5V	50MHz/40 ns	6	352-lead ceramic PGA
SMJ320C40TAM40	5V	40MHz/50ns	6	324 pad TAB tape (encapsulated)
SMJ320C40TBM40	5V	40MHz/50ns	6	324 pad TAB tape (bare die)
TMS320C40TAL50	5V	50MHz/40ns	6	324 pad TAB tape (encapsulated)
SMJ320C40TAM50	5V	50MHz/40ns	6	324 pad TAB tape (encapsulated)
SMJ320C40TBM50	5V	50MHz/40ns	6	324 pad TAB tape (bare die)
TMS320C40TAL60	5V	60MHz/33ns	6	324 pad TAB tape (encapsulated)
SMJ320C40KGDM40	5V	40MHz/50ns	6	Known Good Die
SMJ320C40KGDM50	5V	50MHz/40ns	6	Known Good Die
TMS320C40KGDL50	5V	50MHz/40ns	6	Known Good Die
TMS320C40KGDL60	5V	60MHz/33ns	6	Known Good Die

Table 10–4. Development Support Tools Part Numbers

Development Tool	Part Number	Platform
C Compiler/Assembler/Linker	TMDS3243855-02	PC (DOS, OS/2)
C Compiler/Assembler/Linker	TMDS3243255-08	VAX (VMS)
C Compiler/Assembler/Linker	TMDS3243555-08	SPARC (Sun OS)
Assembler/Linker	TMDS3243850-02	PC (DOS)
Simulator (C language)	TMDS3244851-02	PC (DOS, Windows)
Simulator (C language)	TMDS3244551-09	SPARC (Sun OS)
Tartan Floating Point Library	320FLO-PC-C40	PC (DOS)
Tartan Floating Point Library	320FLO-SUN-C40	SPARC (Sun OS)
Digital Filter Design Package	DFDP	PC (DOS)
C Source Debugger Conversion Software	TMDS3240140	PC (XDS510)
C Source Debugger Conversion Software	TMDS3240640	Sun (XDS510WS)
Emulation Porting Kit	TMDX3240040†	—
’C3x/’C4x Tartan C/C++ Compiler/Assembler/Linker	TAR–CCM–PC	PC (DOS)
’C3x/’C4x Tartan C/C++ Compiler/Assembler/Linker	TAR–CCM–SP	SPARC
’C3x/’C4x Tartan C/C++ Compiler/Assembler/Linker/ Simulator	TAR–SIM–PC	PC (DOS)
’C3x/’C4x Tartan C/C++ Compiler/Assembler/Linker/ Simulator	TAR–SIM–SP	SPARC
’C3x/’C4x Tartan C/C++ XDS510 Debugger	TAR–DEG–XDS–PC	PC (DOS, Windows)
’C3x/’C4x Tartan C/C++ XDS510 Debugger	TAR–DEG–XDS–SP	SPARC (Sun OS)
XDS510 Emulator‡	TMDS3260140	PC (DOS, OS/2, Windows)
XDS510WS Emulator§	TMDS3260640	Sun (SPARC SCSI)
PC/Sparc JTAG Emulation Cable	TMDS3080001	XDS510/XDS510WS
Parallel Processing Development System	TMDX3261040	XDS510/XDS510WS

† Requires licensing agreement.
‡ Includes XDS510WS box, SCSI cable, power supply, and JTAG cable. TMDS3240640 C-source debugger software not included.
§ Includes XDS510 board and JTAG cable. TMDS3240140 C-source debugger software not included.

Chapter 11

XDS510 Emulator Design Considerations

This chapter explains the design requirements of the XDS510 emulator with respect to JTAG designs, and discusses the XDS510 cable (manufacturing part number 2617698-0001). This cable is identified by a label on the cable pod marked **JTAG 3/5V** and supports both standard 3-volt and 5-volt target system power inputs.

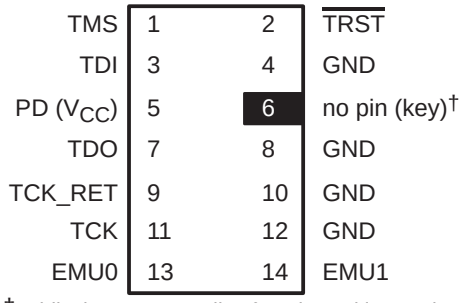
The term *JTAG,* as used in this book, refers to TI scan-based emulation, which is based on the IEEE 1149.1 standard.

Topic **Page**

11.1 Designing Your Target System's Emulator Connector (14-Pin Header)

JTAG target devices support emulation through a dedicated emulation port. This port is a superset of the IEEE 1149.1 standard and is accessed by the emulator. To communicate with the emulator, *your target system must have a 14-pin header* (two rows of seven pins) with the connections that are shown in Figure 11–1. Table 11–1 describes the emulation signals.

Figure 11–1. 14-Pin Header Signals and Header Dimensions



Header Dimensions:
Pin-to-pin spacing, 0.100 in. (X,Y)
Pin width, 0.025-in. square post
Pin length, 0.235-in. nominal

† While the corresponding female position on the cable connector is plugged to prevent improper connection, the cable lead for pin 6 is present in the cable and is grounded, as shown in the schematics and wiring diagrams in this document.

Table 11–1. 14-Pin Header Signal Descriptions

Signal	Description	Emulator† State	Target† State
TMS	Test mode select	O	I
TDI	Test data input	O	I
TDO	Test data output	I	O
TCK	Test clock. TCK is a 10.368-MHz clock source from the emulation cable pod. This signal can be used to drive the system test clock	O	I
$\overline{\text{TRST}}$‡	Test reset	O	I
EMU0	Emulation pin 0	I	I/O
EMU1	Emulation pin 1	I	I/O
PD(V_{CC})	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to V_{CC} in the target system.	I	O
TCK_RET	Test clock return. Test clock input to the emulator. May be a buffered or unbuffered version of TCK.	I	O
GND	Ground		

† I = input; O = output

‡ Do not use pullup resistors on $\overline{\text{TRST}}$: it has an internal pulldown device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pulldown resistor may be needed. (The size of this resistor should be based on electrical current considerations.)

Although you can use other headers, recommended parts include:

straight header, unshrouded	DuPont Connector Systems part numbers:	65610–114
		65611–114
		67996–114
		67997–114

11.2 Bus Protocol

The IEEE 1149.1 specification covers the requirements for the test access port (TAP) bus slave devices and provides certain rules, summarized as follows:

- The TMS/TDI inputs are sampled on the rising edge of the TCK signal of the device.
- The TDO output is clocked from the falling edge of the TCK signal of the device.

When these devices are daisy-chained together, the TDO of one device has approximately a half TCK cycle setup to the next device's TDI signal. This type of timing scheme minimizes race conditions that would occur if both TDO and TDI were timed from the same TCK edge. The penalty for this timing scheme is a reduced TCK frequency.

The IEEE 1149.1 specification does not provide rules for bus master (emulator) devices. Instead, it states that it expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules.

11.3 IEEE 1149.1 Standard

For more information concerning the IEEE 1149.1 standard, contact IEEE Customer Service:

Address: IEEE Customer Service
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331

Phone: (800) 678–IEEE in the US and Canada
(908) 981–1393 outside the US and Canada

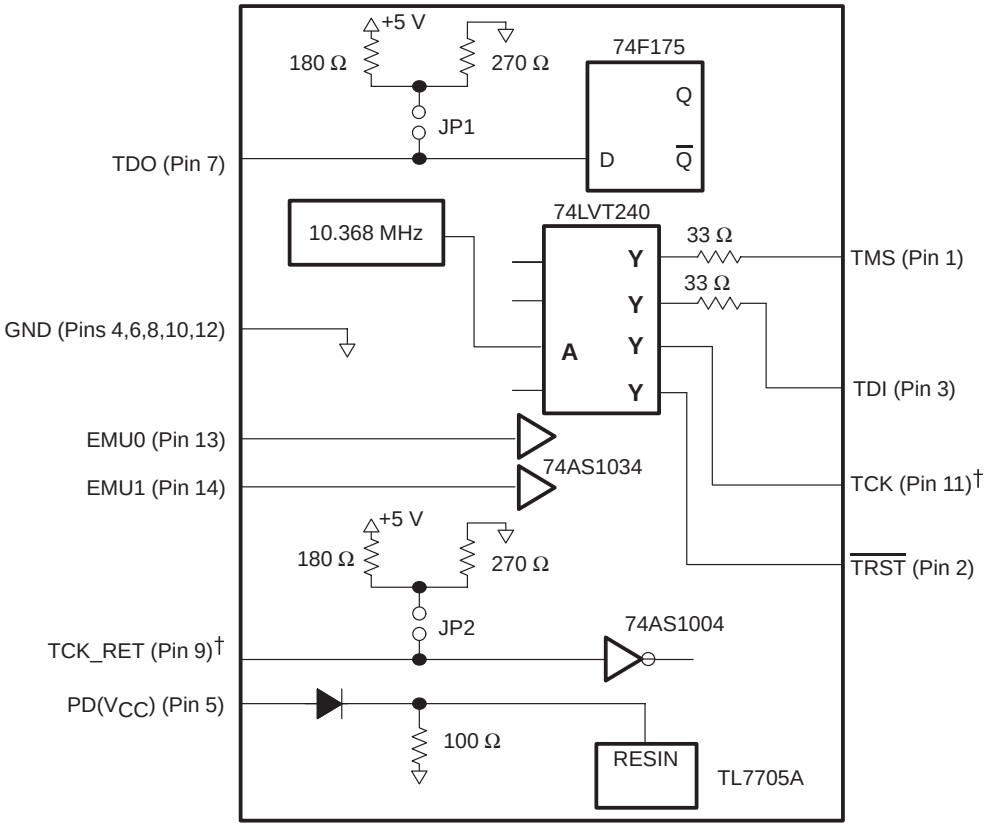
FAX: (908) 981–9667 Telex: 833233

11.4 JTAG Emulator Cable Pod Logic

Figure 11–2 shows a portion of the emulator cable pod. These are the functional features of the pod:

- Signals TDO and TCK_RET can be parallel-terminated inside the pod if required by the application. By default, these signals are not terminated.
- Signal TCK is driven with a 74LVT240 device. Because of the high-current drive (32 mA I_{OL}/I_{OH}), this signal can be parallel-terminated. If TCK is tied to TCK_RET, then you can use the parallel terminator in the pod.
- Signals TMS and TDI can be generated from the falling edge of TCK_RET, according to the IEEE 1149.1 bus slave device timing rules.
- Signals TMS and TDI are series-terminated to reduce signal reflections.
- A 10.368-MHz test clock source is provided. You may also provide your own test clock for greater flexibility.

Figure 11–2. JTAG Emulator Cable Pod Interface



† The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

11.5 JTAG Emulator Cable Pod Signal Timing

Figure 11–3 shows the signal timings for the emulator cable pod. Table 11–2 defines the timing parameters. These timing parameters are calculated from values specified in the standard data sheets for the emulator and cable pod and are for reference only. Texas Instruments does not test or guarantee these timings.

The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

Figure 11–3. JTAG Emulator Cable Pod Timings

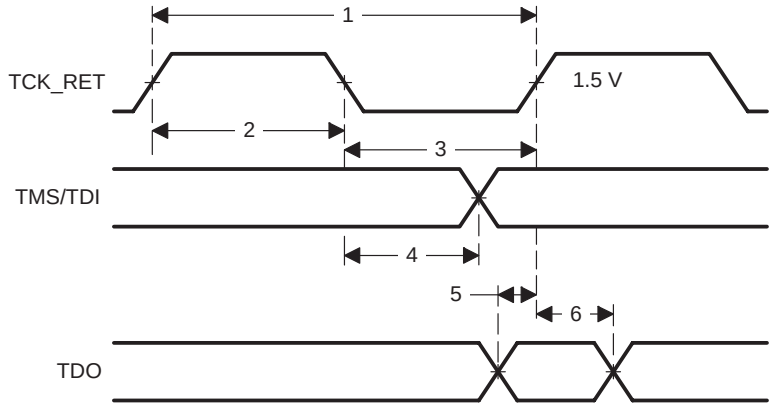


Table 11–2. Emulator Cable Pod Timing Parameters

No.	Reference	Description	Min	Max	Units
1	$t_{c(TCK)}$	TCK_RET period	35	200	ns
2	$t_{w(TCKH)}$	TCK_RET high-pulse duration	15		ns
3	$t_{w(TCKL)}$	TCK_RET low-pulse duration	15		ns
4	$t_{d(TMS)}$	Delay time, TMS/TDI valid from TCK_RET low	6	20	ns
5	$t_{su(TDO)}$	TDO setup time to TCK_RET high	3		ns
6	$t_{h(TDO)}$	TDO hold time from TCK_RET high	12		ns

11.6 Emulation Timing Calculations

The following examples help you calculate emulation timings in your system. For actual target timing parameters, see the appropriate device data sheets.

Assumptions:

$t_{su(TTMS)}$	Target TMS/TDI setup to TCK high	10 ns
$t_{d(TTDO)}$	Target TDO delay from TCK low	15 ns
$t_{d(bufmax)}$	Target buffer delay, maximum	10 ns
$t_{d(bufmin)}$	Target buffer delay, minimum	1 ns
$t_{(bufskew)}$	Target buffer skew between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{(TCKfactor)}$	Assume a 40/60 duty cycle clock	0.4 (40%)

Given in Table 11–2 (on page 11-5):

$t_{d(TMSmax)}$	Emulator TMS/TDI delay from TCK_RET low, maximum	20 ns
$t_{su(TDOmin)}$	TDO setup time to emulator TCK_RET high, minimum	3 ns

There are two key timing paths to consider in the emulation design:

- ❑ The TCK_RET-to-TMS/TDI path, called $t_{pd(TCK\_RET–TMS/TDI)}$
- ❑ The TCK_RET-to-TDO path, called $t_{pd(TCK\_RET–TDO)}$

Of the following two cases, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Case 1: Single processor, direct connection, TMS/TDI timed from TCK_RET low.

$$t_{pd\,(TCK\_RET–TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[20ns + 10ns]}{0.4}$$

$$= 75ns\ (13.3\ MHz)$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[15ns + 3ns]}{0.4}$$

$$= 45ns\ (22.2\ MHz)$$

In this case, the TCK_RET-to-TMS/TDI path is the limiting factor.

Case 2: Single/multiprocessor, TMS/TDI/TCK buffered input, TDO buffered output, TMS/TDI timed from TCK_RET low.

$$t_{pd\,(TCK\_RET-TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)} + t_{(bufskew)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{\left[20ns + 10ns + 1.35ns\right]}{0.4}$$

$$= 78.4ns\ (12.7\ MHz)$$

$$t_{pd\,(TCK\_RET-TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)} + t_{d\,(bufmax)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[15ns + 3ns + 10ns]}{0.4}$$

$$= 70ns\ (14.3\ MHz)$$

In this case, the TCK_RET-to-TMS/TDI path is the limiting factor.

In a multiprocessor application, it is necessary to ensure that the EMU0–1 lines can go from a logic low level to a logic high level in less than 10 μs. This can be calculated as follows:

$$t_r = 5(R_{pullup} \times N_{devices} \times C_{load\_per\_device})$$
$$= 5(4.7\ k\Omega \times 16 \times 15\ pF)$$
$$= 5.64\ \mu s$$

11.7 Connections Between the Emulator and the Target System

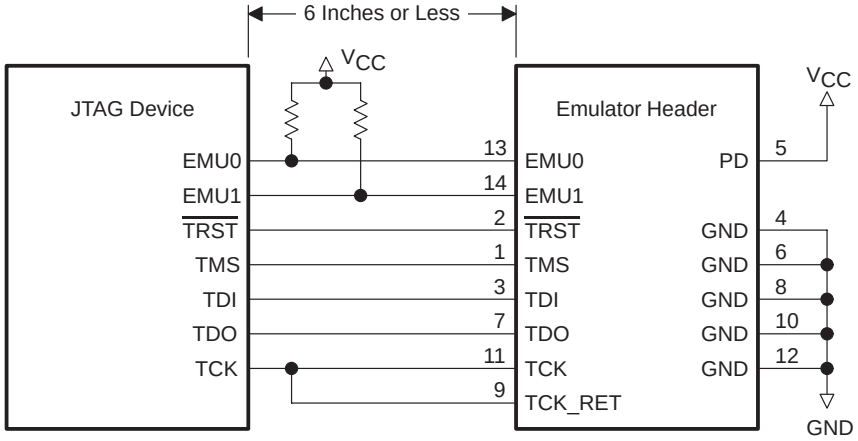
It is extremely important to provide high-quality signals between the emulator and the JTAG target system. Depending upon the situation, you must supply the correct signal buffering, test clock inputs, and multiple processor interconnections to ensure proper emulator and target system operation.

Signals applied to the EMU0 and EMU1 pins on the JTAG target device can be either input or output (I/O). In general, these two pins are used as both input and output in multiprocessor systems to handle global run/stop operations. EMU0 and EMU1 signals are applied only as inputs to the XDS510 emulator header.

11.7.1 Buffering Signals

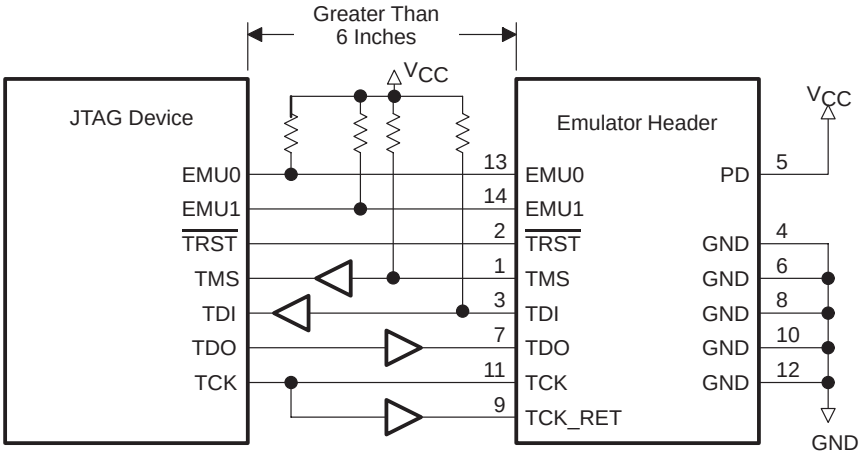
If the distance between the emulation header and the JTAG target device is greater than six inches, the emulation signals must be buffered. If the distance is less than six inches, no buffering is necessary. The following illustrations depict these two situations.

- **No signal buffering.** In this situation, the distance between the header and the JTAG target device should be no more than six inches.



The EMU0 and EMU1 signals must have pullup resistors connected to V_{CC} to provide a signal rise time of less than 10 μs. A 4.7-kΩ resistor is suggested for most applications.

❑ **Buffered transmission signals.** In this situation, the distance between the emulation header and the processor is greater than six inches. Emulation signals TMS, TDI, TDO, and TCK_RET are buffered through the same package.

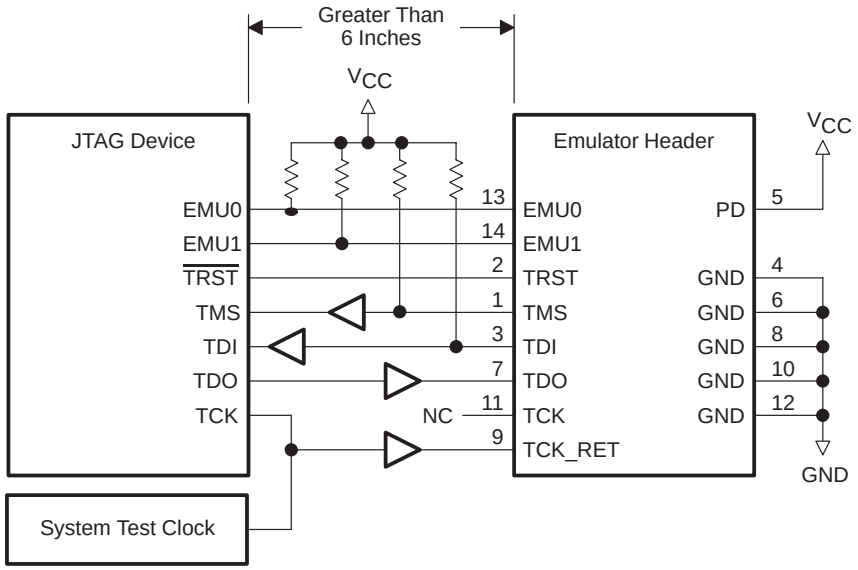


- The EMU0 and EMU1 signals must have pullup resistors connected to V_{CC} to provide a signal rise time of less than 10 μs. A 4.7-kΩ resistor is suggested for most applications.
- The input buffers for TMS and TDI should have pullup resistors connected to V_{CC} to hold these signals at a known value when the emulator is not connected. A resistor value of 4.7 kΩ or greater is suggested.
- To have high-quality signals (especially the processor TCK and the emulator TCK_RET signals), you may have to employ special care when routing the PWB trace. You also may have to use termination resistors to match the trace impedance. The emulator pod provides optional internal parallel terminators on the TCK_RET and TDO. TMS and TDI provide fixed series termination.
- Since $\overline{\text{TRST}}$ is an asynchronous signal, it should be buffered as needed to insure sufficient current to all target devices.

11.7.2 Using a Target-System Clock

Figure 11–4 shows an application with the system test clock generated in the target system. In this application, the TCK signal is left unconnected.

Figure 11–4. Target-System-Generated Test Clock



Note: When the TMS/TDI lines are buffered, pullup resistors should be used to hold the buffer inputs at a known level when the emulator cable is not connected.

There are two benefits to having the target system generate the test clock:

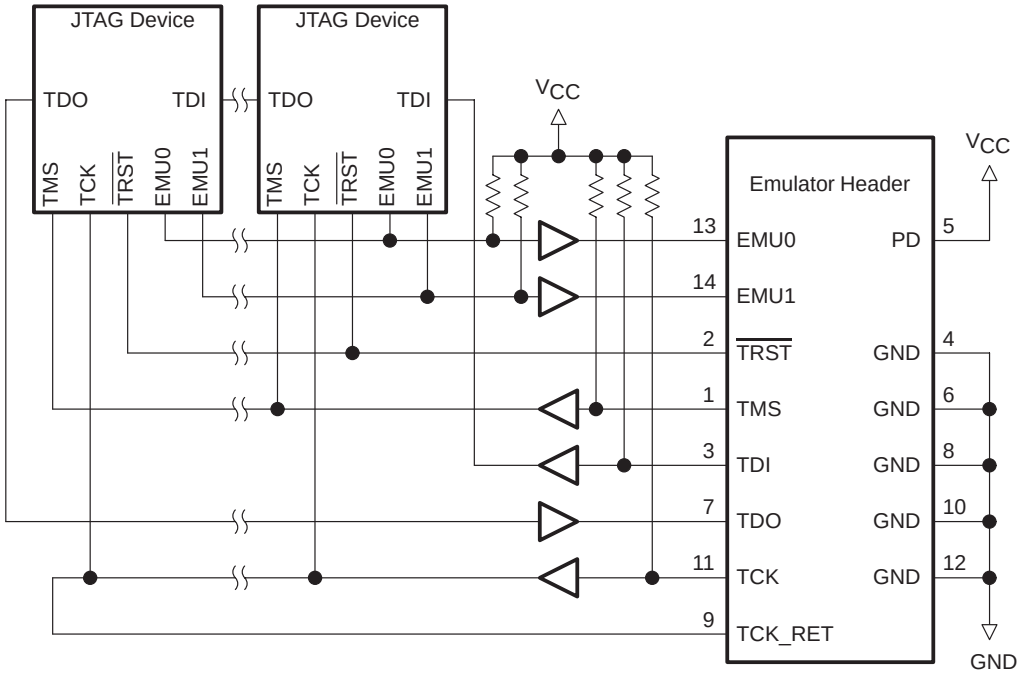
- The emulator provides only a single 10.368-MHz test clock. If you allow the target system to generate your test clock, you can set the frequency to match your system requirements.
- In some cases, you may have other devices in your system that require a test clock when the emulator is not connected. The system test clock also serves this purpose.

11.7.3 Configuring Multiple Processors

Figure 11–5 shows a typical daisy-chained multiprocessor configuration, which meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals in this example are buffered to isolate the processors from the emulator and provide adequate signal drive for the target system. One of the benefits of this type of interface is that you can generally slow down the test clock to eliminate timing problems. You should follow these guidelines for multiprocessor support:

- The processor TMS, TDI, TDO, and TCK signals should be buffered through the same physical package for better control of timing skew.
- The input buffers for TMS, TDI, and TCK should have pullup resistors connected to V_{CC} to hold these signals at a known value when the emulator is not connected. A resistor value of 4.7 kΩ or greater is suggested.
- Buffering EMU0 and EMU1 is optional but highly recommended to provide isolation. These are not critical signals and do not have to be buffered through the same physical package as TMS, TCK, TDI, and TDO. Unbuffered and buffered signals are shown in this section (page 11-8 and page 11-9).

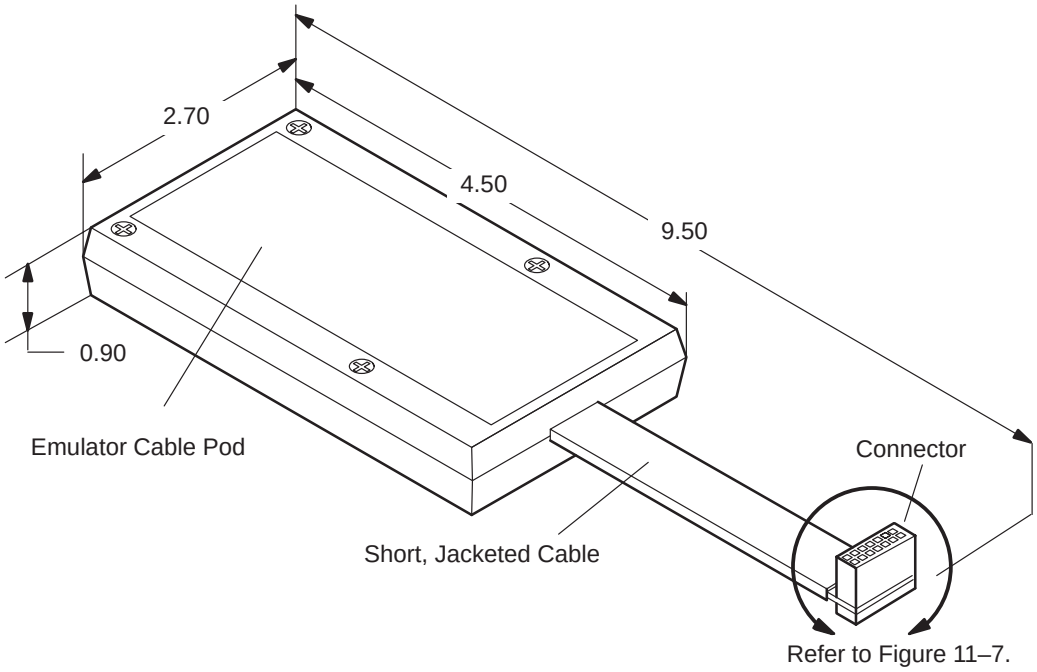
Figure 11–5. Multiprocessor Connections



11.8 Mechanical Dimensions for the 14-Pin Emulator Connector

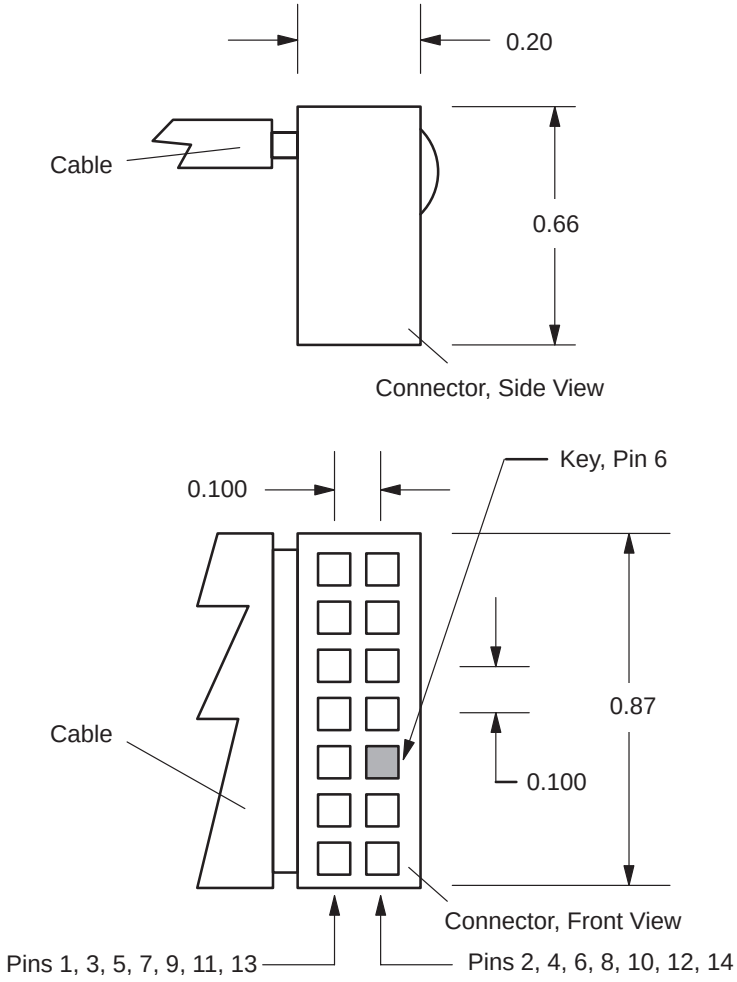
The JTAG emulator target cable consists of a 3-foot section of jacketed cable, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet 10 inches. Figure 11–6 and Figure 11–7 (page 11-13) show the mechanical dimensions for the target cable pod and short cable. Note that the pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes. The cable pod box is nonconductive plastic with four recessed metal screws.

Figure 11–6. Pod/Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified.

Figure 11–7. 14-Pin Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified.

11.9 Emulation Design Considerations

This section describes the scan path linker (SPL), which can simultaneously add all four secondary JTAG scan paths to the main scan path. It also describes how to use the emulation pins and configure multiple processors.

11.9.1 Using Scan Path Linkers

You can use the TI ACT8997 scan path linker (SPL) to divide the JTAG emulation scan path into smaller, logically connected groups of 4 to 16 devices. As described in the *Advanced Logic and Bus Interface Logic Data Book* (literature number SCYD001), the SPL is compatible with the JTAG emulation scanning. The SPL is capable of adding any combination of its four secondary scan paths into the main scan path.

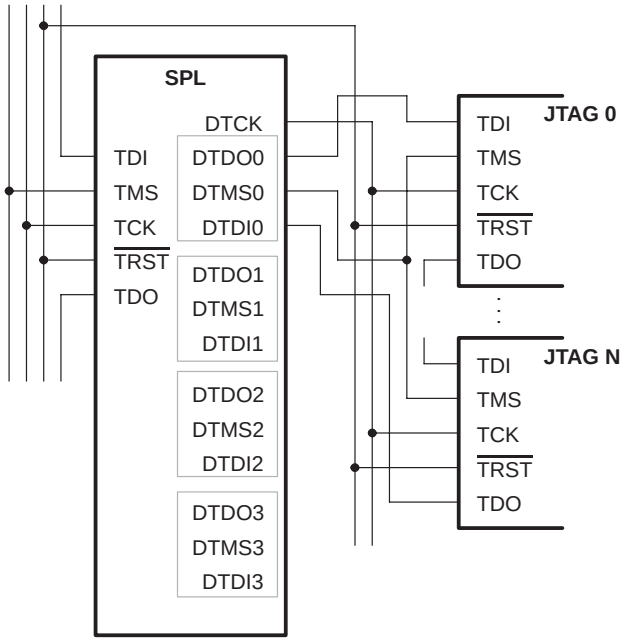
A system of multiple, secondary JTAG scan paths has better fault tolerance and isolation than a single scan path. Since an SPL has the capability of adding all secondary scan paths to the main scan path simultaneously, it can support global emulation operations, such as starting or stopping a selected group of processors.

TI emulators do not support the nesting of SPLs (for example, an SPL connected to the secondary scan path of another SPL). However, you can have multiple SPLs on the main scan path.

Although the ACT8999 scan path selector is similar to the SPL, it can add only one of its secondary scan paths at a time to the main JTAG scan path. Thus, global emulation operations are not assured with the scan path selector. For this reason, scan path selectors are not supported.

You can insert an SPL on a backplane so that you can add up to four device boards to the system without the jumper wiring required with nonbackplane devices. You connect an SPL to the main JTAG scan path in the same way you connect any other device. Figure 11–8 shows you how to connect a secondary scan path to an SPL.

Figure 11–8. Connecting a Secondary JTAG Scan Path to an SPL



The $\overline{\text{TRST}}$ signal from the main scan path drives all devices, even those on the secondary scan paths of the SPL. The TCK signal on each target device on the secondary scan path of an SPL is driven by the SPL's DTCK signal. The TMS signal on each device on the secondary scan path is driven by the respective DTMS signals on the SPL.

DTDO on the SPL is connected to the TDI signal of the first device on the secondary scan path. DTDI on the SPL is connected to the TDO signal of the last device in the secondary scan path. Within each secondary scan path, the TDI signal of a device is connected to the TDO signal of the device before it. If the SPL is on a backplane, its secondary JTAG scan paths are on add-on boards; if signal degradation is a problem, you may need to buffer both the $\overline{\text{TRST}}$ and DTCK signals. Although less likely, you may also need to buffer the DTMS*n* signals for the same reasons.

11.9.2 Emulation Timing Calculations for SPL

The following examples help you to calculate the emulation timings in the SPL secondary scan path of your system. For actual target timing parameters, see the appropriate device data sheets.

Assumptions:

$t_{su(TTMS)}$	Target TMS/TDI setup to TCK high	10 ns
$t_{d(TTDO)}$	Target TDO delay from TCK low	15 ns
$t_{d(bufmax)}$	Target buffer delay, maximum	10 ns
$t_{d(bufmin)}$	Target buffer delay, minimum	1 ns
$t_{(bufskew)}$	Target buffer skew between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{(TCKfactor)}$	Assume a 40/60 duty cycle clock	0.4 (40%)

Given in the SPL data sheet:

$t_{d(DTMSmax)}$	SPL DTMS/DTDO delay from TCK low, maximum	31 ns
$t_{su(DTDLmin)}$	DTDI setup time to SPL TCK high, minimum	7 ns
$t_{d(DTCKHmin)}$	SPL DTCK delay from TCK high, minimum	2 ns
$t_{d(DTCKLmax)}$	SPL DTCK delay from TCK low, maximum	16 ns

There are two key timing paths to consider in the emulation design:

- ❑ The TCK-to-DTMS/DTDO path, called $t_{pd(TCK–DTMS)}$
- ❑ The TCK-to-DTDI path, called $t_{pd(TCK–DTDI)}$

Of the following two cases, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Case 1: Single processor, direct connection, DTMS/DTDO timed from TCK low.

$$t_{pd\,(TCK–DTMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{d\,(DTCKHmin)} + t_{su\,(TTMS)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[31ns + 2ns + 10ns]}{0.4}$$

$$= 107.5ns\ (9.3\ MHz)$$

$$t_{pd\,(TCK–DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[15ns + 16ns + 7ns]}{0.4}$$

$$= 9.5ns\ (10.5\ MHz)$$

In this case, the TCK-to-DTMS/DTDL path is the limiting factor.

Case 2: Single/multiprocessor, DTMS/DTDO/TCK buffered input, DTDI buffered output, DTMS/DTDO timed from TCK low.

$$t_{pd\,(TCK–TDMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{(DTCKHmin)} + t_{su\,(TTMS)} + t_{(bufskew)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[31ns + 2ns + 10ns + 1.35ns]}{0.4}$$

$$= 110.9ns\ (9.0\ MHz)$$

$$t_{pd\,(TCK–DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)} + t_{d\,(bufskew)}\right]}{t_{(TCKfactor)}}$$

$$= \frac{[15ns + 15ns + 7ns + 10ns]}{0.4}$$

$$= 120ns\ (8.3\ MHz)$$

In this case, the TCK-to-DTDI path is the limiting factor.

11.9.3 Using Emulation Pins

The EMU0/1 pins of TI devices are bidirectional, three-state output pins. When in an inactive state, these pins are at high impedance. When the pins are active, they function in one of the two following output modes:

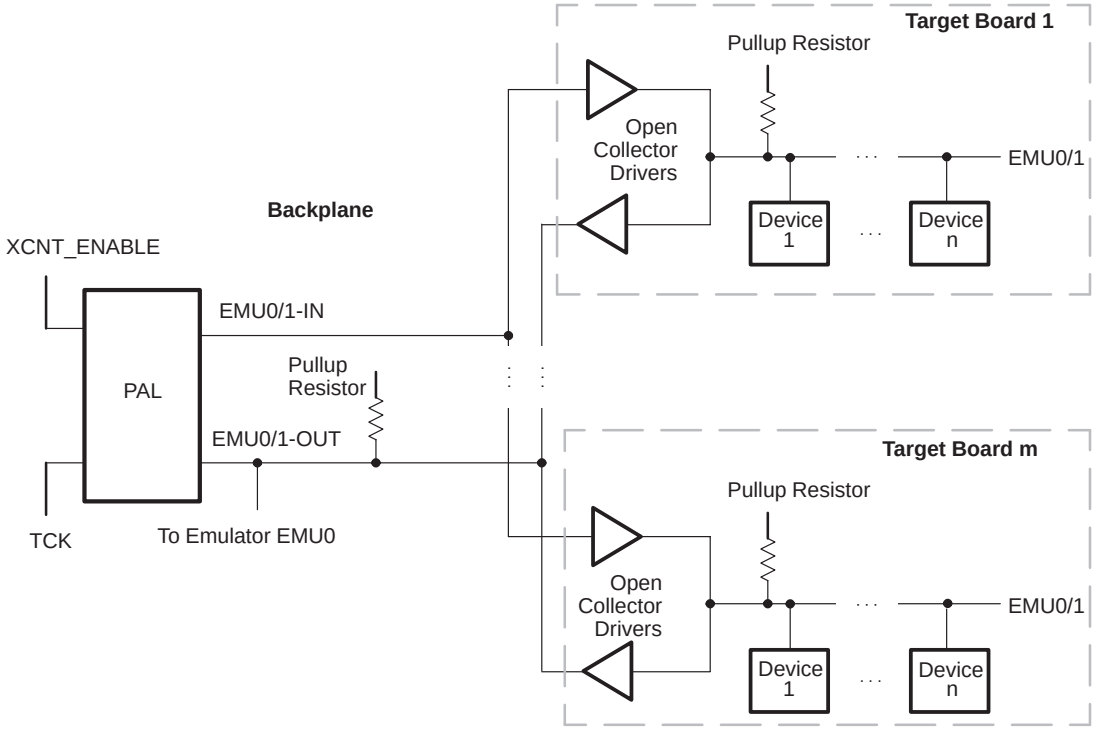
- **Signal Event**
  The EMU0/1 pins can be configured via software to signal internal events. In this mode, driving one of these pins low can cause devices to signal such events. To enable this operation, the EMU0/1 pins function as open-collector sources. External devices such as logic analyzers can also be connected to the EMU0/1 signals in this manner. If such an external source is used, it must also be connected via an open-collector source.

- **External Count**
  The EMU0/1 pins can be configured via software as totem-pole outputs for driving an external counter. These devices can be damaged if the output of more than one device is configured for totem-pole operation. The emulation software detects and prevents this condition. However, the emulation software has no control over external sources on the EMU0/1 signal. Therefore, all external sources must be inactive when any device is in the external count mode.

TI devices can be configured by software to halt processing if their EMU0/1 pins are driven low. This feature, in combination with the use of the signal event output mode, allows one TI device to halt all other TI devices on a given event for system-level debugging.

If you route the EMU0/1 signals between boards, they require special handling because these signals are more complex than normal emulation signals. Figure 11–9 shows an example configuration that allows any processor in the system to stop any other processor in the system. Do not tie the EMU0/1 pins of more than 16 processors together in a single group without using buffers. Buffers provide the crisp signals that are required during a RUNB (run benchmark) debugger command or when the external analysis counter feature is used.

Figure 11–9. EMU0/1 Configuration



Notes: 1) The low time on EMUx-IN should be at least one TCK cycle and less than 10 μs. Software will set the EMUx-OUT pin to a high state.

2) To enable the open-collector driver and pullup resistor on EMU1 to provide rising/falling edges of less than 25 ns, the modification shown in this figure is suggested. Rising edges slower than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used.

These seven important points apply to the circuitry shown in Figure 11–9 and the timing shown in Figure 11–10:

- Open-collector drivers isolate each board. The EMU0/1 pins are tied together on each board.
- At the board edge, the EMU0/1 signals are split to provide IN/OUT. This is required to prevent the open-collector drivers from acting as a latch that can be set only once.
- The EMU0/1 signals are bused down the backplane. Pullup resistors are installed as required.
- The bused EMU0/1 signals go into a PAL® device whose function is to generate a low pulse on the EMU0/1-IN signal when a low level is detected

on the EMU0/1-OUT signal. This pulse must be longer than one TCK period to affect the devices, but less than 10 μs to avoid possible conflicts or retriggering, once the emulation software clears the device's pins.

- During a RUNB debugger command or other external analysis count, the EMU0/1 pins on the target device become totem-pole outputs. The EMU1 pin is a ripple carry-out of the internal counter. EMU0 becomes a *processor-halted* signal. During a RUNB or other external analysis count, the EMU0/1-IN signal to all boards must remain in the high (disabled) state. You must provide some type of external input (XCNT_ENABLE) to the PAL to disable the PAL from driving EMU0/1-IN to a low state.

- If sources other than TI processors (such as logic analyzers) are used to drive EMU0/1, their signal lines must be isolated by open-collector drivers and be inactive during RUNB and other external analysis counts.

- You must connect the EMU0/1-OUT signals to the emulation header or directly to a test bus controller.

Figure 11–10. Suggested Timings for the EMU0 and EMU1 Signals

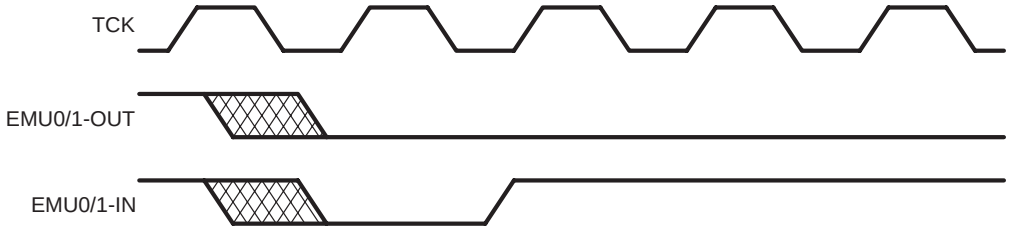
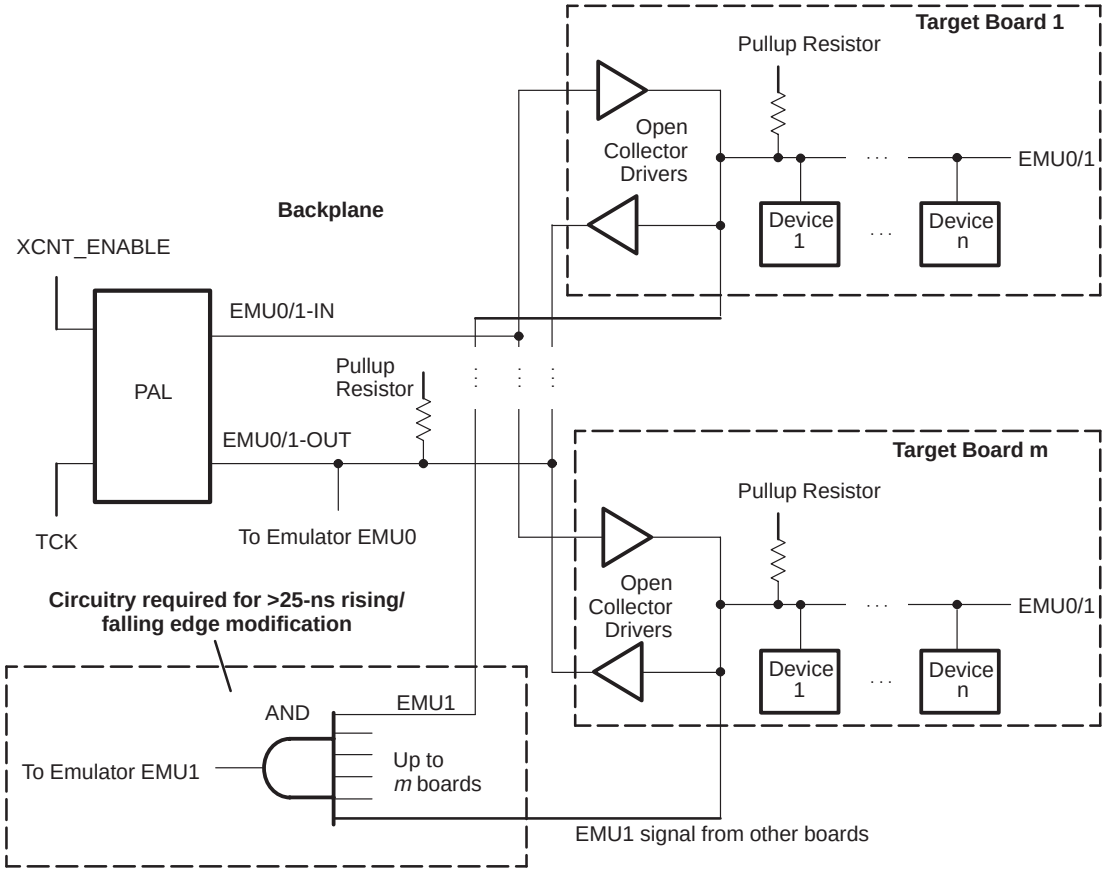


Figure 11–11. EMU0/1 Configuration With Additional AND Gate to Meet Timing Requirements

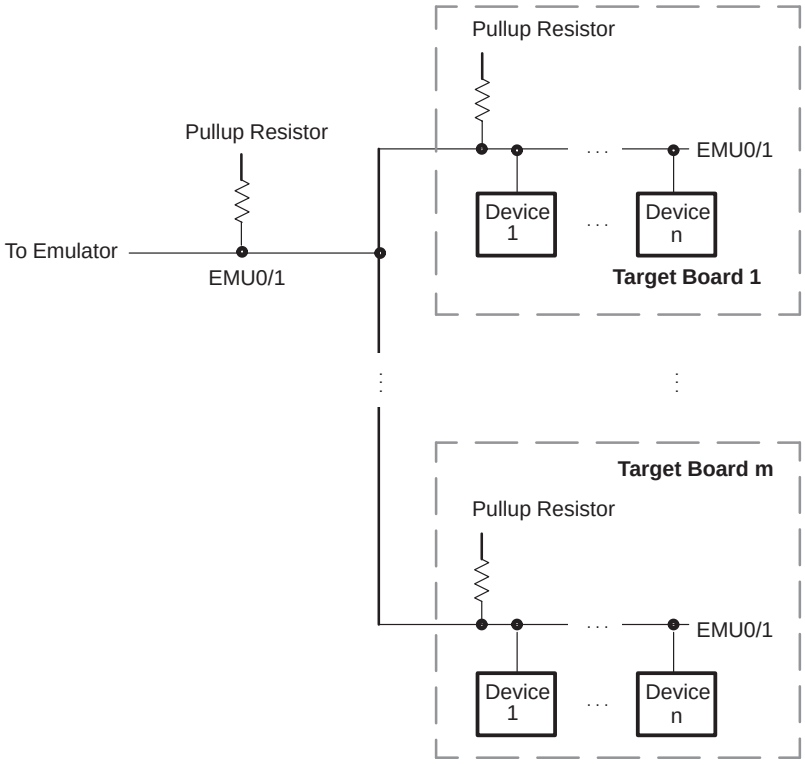


Notes: 1) The low time on EMUx–IN should be at least one TCK cycle and less than 10 μs. Software will set the EMUx–OUT pin to a high state.

2) To enable the open-collector driver and pullup resistor on EMU1 to provide rising/falling edges of less than 25 ns, the modification shown in this figure is suggested. Rising edges slower than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used.

If it is not important that the devices on one target board are stopped by devices on another target board via the EM0/1, then the circuit in Figure 11–12 can be used. In this configuration, the global-stop capability is lost. It is important not to overload EMU0/1 with more than 16 devices.

Figure 11–12. EMU0/1 Configuration Without Global Stop

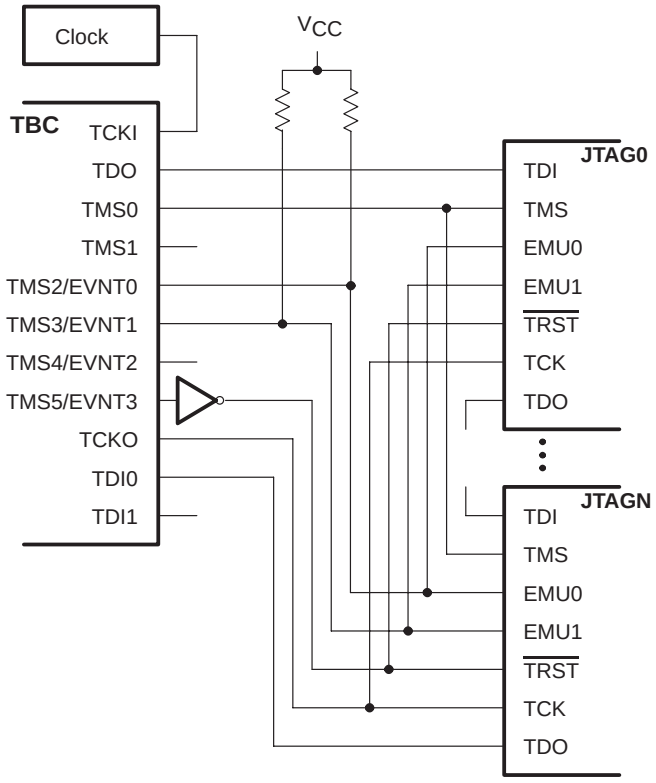


Note: The open-collector driver and pullup resistor on EMU1 must be able to provide rising/falling edges of less than 25 ns. Rising edges slower than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used. If this condition cannot be met, then the EMU0/1 signals from the individual boards should be ANDed together (as shown in Figure 1-11) to produce an EMU0/1 signal for the emulator.

11.9.4 Performing Diagnostic Applications

For systems that require built-in diagnostics, it is possible to connect the emulation scan path directly to a TI ACT8990 test bus controller (TBC) instead of the emulation header. The TBC is described in the Texas Instruments *Advanced Logic and Bus Interface Logic Data Book* (literature number SCYD001). Figure 11–13 shows the scan path connections of *n* devices to the TBC.

Figure 11–13. TBC Emulation Connections for n JTAG Scan Paths



In the system design shown in Figure 1–13, the TBC emulation signals TCKI, TDO, TMS0, TMS2/EVNT0, TMS3/EVNT1, TMS5/EVNT3, TCKO, and TDI0 are used, and TMS1, TMS4/EVNT2, and TDI1 are not connected. The target devices' EMU0 and EMU1 signals are connected to V_{CC} through pullup resistors and tied to the TBC's TMS2/EVNT0 and TMS3/EVNT1 pins, respectively. The TBC's TCKI pin is connected to a clock generator. The TCK signal for the main JTAG scan path is driven by the TBC's TCKO pin.

On the TBC, the TMS0 pin drives the TMS pins on each device on the main JTAG scan path. TDO on the TBC connects to TDI on the first device on the main JTAG scan path. TDI0 on the TBC is connected to the TDO signal of the last device on the main JTAG scan path. Within the main JTAG scan path, the TDI signal of a device is connected to the TDO signal of the device before it. $\overline{\text{TRST}}$ for the devices can be generated either by inverting the TBC's TMS5/EVNT3 signal for software control or by logic on the board itself.

Appendix A

Glossary

A

A0–A30: External address pin*s* for data/program memory or I/O devices. These pins are on the global bus. *See also LA0–LA30*.

address: The location of program code or data stored in memory.

addressing mode: The method by which an instruction interprets its operands to acquire the data it needs.

ALU: *See Arithmetic logic unit.*

analog-to-digital (A/D) converter: A successive-approximation converter with internal sample-and-hold circuitry used to translate an analog signal to a digital signal.

ARAU: See auxiliary register arithmetic unit.

arithmetic logic unit (ALU): The part of the CPU that performs arithmetic and logic operations.

auxiliary registers (ARn): A set of registers used primarily in address generation.

auxiliary register arithmetic unit (ARAU): *Auxiliary register arithmetic unit.* A16-bit arithmetic logic unit (ALU) used to calculate indirect addresses using the auxiliary registers as inputs and outputs.

B

bit-reversed addressing: Addressing in which several bits of an address are reversed in order to speed processing of algorithms, such as Fourier transforms.

BK: See block-size register.

block-size register: A register used for defining the length of a program block to be repeated in repeat mode.

bootloader: A built-in segment of code that transfers code from an external memory or from a communication port to RAM at power-up.

C

carry bit: A bit in status register ST1 used by the ALU for extended arithmetic operations and accumulator shifts and rotates. The carry bit can be tested by conditional instructions.

circular addressing: An addressing mode in which an auxiliary register is used to cycle through a range of addresses to create a circular buffer in memory.

context save/restore: A save/restore of system status (status registers, accumulator, product register, temporary register, hardware stack, and auxiliary registers, etc.) when the device enters/exits a subroutine such as an interrupt service routine.

CPU: *Central processing unit*. The unit that coordinates the functions of a processor.

CPU cycle: The time it takes the CPU to go through one logic phase (during which internal values are changed) and one latch phase (during which the values are held constant).

cycle: See CPU cycle.

D

D0–D31: External data bus pins that transfer data between the processor and external data/program memory or I/O devices. *See also LD0–LD31*.

data-address generation logic: Logic circuitry that generates the addresses for data memory reads and writes. This circuitry can generate one address per machine cycle. See also *program-address generation logic.*

data-page pointer: A seven-bit register used as the seven MSBs in addresses generated using direct addressing.

decode phase: The phase of the pipeline in which the instruction is decoded.

DIE: See DMA interrupt enable register.

DMA coprocessor: A peripheral that transfers the contents of memory locations independently of the processor (except for initialization).

DMA controller: See DMA coprocessor.

DMA interrupt enable register (DIE): A register (in the CPU register file) that controls which interrupts the DMA coprocessor responds to.

DP: See data-page pointer.

dual-access RAM: Memory that can be accessed twice in a single clock cycle. For example, your code can read from and write to a dual-access RAM in one clock cycle.

E

external interrupt: A hardware interrupt triggered by a pin.

extended-precision floating-point format: A 40-bit representation of a floating-point number with a 32-bit mantissa and an 8-bit exponent.

extended-precision register: A 40-bit register used primarily for extended-precision floating-point calculations. Floating-point operations use bits 39–0 of an extended-precision register. Integer operations, however, use only bits 31–0.

F

FIFO buffer: *First-in, first-out buffer.* A portion of memory in which data is stored and then retrieved in the same order in which it was stored. Thus, the first word stored in this buffer is retrieved first. The 'C4x's communication ports each have two FIFOs: one for transmit operations and one for receive operations.

H

hardware interrupt: An interrupt triggered through physical connections with on-chip peripherals or external devices.

hit: A condition in which, when the processor fetches an instruction, the instruction is available in the cache.

I

$\overline{\text{IACK}}$: *Interrupt acknowledge signal*. An output signal that indicates that an interrupt has been received and that the program counter is fetching the interrupt vector that will force the processor into an interrupt service routine.

IIE: See internal interrupt enable register.

IIF: See IIOF flag register.

IIOF flag register (IIF): Controls the function (general-purpose I/O or interrupt) of the four external pins ($\overline{\text{IIOF0}}$ to $\overline{\text{IIOF3}}$). It also contains timer/DMA interrupt flags.

index registers: Two registers (IR0 and IR1) that are used by the ARAU for indexing an address.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

internal interrupt enable register: A register (in the CPU register file) that determines whether or not the CPU will respond to interrupts from the communication ports, the timers, and the DMA coprocessor.

interrupt: A signal sent to the CPU that (when not masked) forces the CPU into a subroutine called an interrupt service routine. This signal can be triggered by an external device, an on-chip peripheral, or an instruction (TRAP, for example).

interrupt acknowledge ($\overline{\text{IACK}}$): A signal that indicates that an interrupt has been received, and that the program counter is fetching the interrupt vector location.

interrupt vector table (IVT): An ordered list of addresses which each correspond to an interrupt; when an interrupt occurs and is enabled, the processor executes a branch to the address stored in the corresponding location in the interrupt vector table.

interrupt vector table pointer (IVTP): A register (in the CPU expansion register file) that contains the address of the beginning of the interrupt vector table.

ISR: *Interrupt service routine*. A module of code that is executed in response to a hardware or software interrupt.

IVTP: See *interrupt vector table pointer*.

LA0–LA30: External address pin*s* for data/program memory or I/O devices. These pins are on the local bus. *See also A0–A30*.

LD0–LD31: External data-bus pins that transfer data between the processor and external data/program memory or I/O devices. *See also D0–D31*.

LSB: *Least significant bit.* The lowest order bit in a word.

M

machine cycle: See *CPU cycle*.

mantissa: A component of a floating-point number consisting of a fraction and a sign bit. The mantissa represents a normalized fraction whose binary point is shifted by the exponent.

maskable interrupt: A hardware interrupt that can be enabled or disabled through software.

memory-mapped register: One of the on-chip registers mapped to addresses in memory. Some of the memory-mapped registers are mapped to data memory, and some are mapped to input/output memory.

MFLOPS: *Millions of floating-point operations per second*. A measure of floating-point processor speed that counts of the number of floating-point operations made per second.

microcomputer mode: A mode in which the on-chip ROM is enabled. This mode is selected via the MP/$\overline{\text{MC}}$ pin. See also *MP/$\overline{\text{MC}}$ pin; microprocessor mode*.

microprocessor mode: A mode in which the on-chip ROM is disabled. This mode is selected via the MP/$\overline{\text{MC}}$ pin. See also *MP/$\overline{\text{MC}}$ pin; microcomputer mode*.

MIPS: Million instructions-per-second.

miss: A condition in which, when the processor fetches an instruction, it is not available in the cache.

MSB: *Most significant bit.* The highest order bit in a word.

multiplier: A device that generates the product of two numbers.

N

$\overline{\text{NMI}}$: *See Nonmaskable interrupt*.

nonmaskable interrupt (NMI): A hardware interrupt that uses the same logic as the maskable interrupts, but cannot be masked. It is often used as a soft reset.

O

overflow flag (OV) bit: A status bit that indicates whether or not an arithmetic operation has exceeded the capacity of the corresponding register.

P

PC: See *program counter*.

peripheral bus: A bus that the CPU uses to communicate the DMA coprocessor, communication ports, and timers.

pipeline: A method of executing instructions in an assembly-line fashion.

program counter: A register that contains the address of the next instruction to be fetched.

R

RC: See repeat counter register.

read/write (R/$\overline{W}$) pin: This memory-control signal indicates the direction of transfer when communicating to an external device.

register file: A bank of registers.

repeat counter register: A register (in the CPU register file) that specifies the number of times minus one that a block of code is to be repeated when a block repeat is performed.

repeat mode: A zero-overhead method for repeating the execution of a block of code.

reset: A means to bring the central processing unit (CPU) to a known state by setting the registers and control bits to predetermined values and signaling execution to fetch the reset vector.

reset pin: This pin causes the device to reset.

ROMEN: *ROM enable*. An external pin that determines whether or not the the on-chip ROM is enabled.

R/$\overline{W}$: *See read/write pin*.

S

short-floating-point format: A 16-bit representation of a floating-point number with a 12-bit mantissa and a 4-bit exponent.

short-integer format: A twos-complement 16-bit format for integer data.

short-unsigned-integer format: A 16-bit unsigned format for integer data.

sign extend: Fill the high order bits of a number with the sign bit.

single-access RAM: *SARAM*. Memory that can be read from or written to only once in a single CPU cycle.

single-precision floating-point format: A 32-bit representation of a floating point number with a 24-bit mantissa and an 8-bit exponent.

single-precision integer format: A twos-complement 32-bit format for integer data.

single-precision unsigned-integer format: A 32-bit unsigned format for integer data.

software interrupt: An interrupt caused by the execution of a TRAP instruction.

split mode: A mode of operation of the DMA coprocessor. This mode allows one DMA channel to service both the receive and transmit portions of a communication port.

ST: See status register.

stack: A block of memory reserved for storing and retrieving data on a first-in last-out basis. It is usually used for storing return addresses and for preserving register values.

status register: A register (in the CPU register file) that contains global information related to the CPU.

T

Timer: A programmable peripheral that can be used to generate pulses or to time events.

Timer-Period Register: *Timer-period register*. A 32-bit memory-mapped register that specifies the period for the on-chip timer.

trap vector table (TVT): An ordered list of addresses which each correspond to an interrupt; when a trap is executed, the processor executes a branch to the address stored in the corresponding location in the trap vector table.

trap vector table pointer (TVTP): A register (in the CPU expansion-register file) that contains the address of the beginning of the trap vector table.

TVTP: See *trap vector table pointer*.

U

unified mode: A mode of operation of the DMA coprocessor. The mode is used mainly for memory-to-memory transfers. This is the default mode of operation for a DMA channel. See also *split mode*.

W

wait state: A period of time that the CPU must wait for external program, data, or I/O memory to respond when reading from or writing to that external memory. The CPU waits one extra cycle for every wait state.

wait-state generator: A program that can be modified to generate a limited number of wait states for a given off-chip memory space (lower program, upper program, data, or I/O).

Z

zero fill: Fill the low or high order bits with zeros when loading a number into a larger field.

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

X

Z