TEXAS INSTRUMENTS

TMS320C3x General-Purpose Applications

User's Guide

1998

Digital Signal Processing Solutions

Texas Instruments

Printed in U.S.A., January 1998
SDS

SPRU194

User's Guide

TMS320C3x General-Purpose Applications

1998

TMS320C3x General-Purpose Applications User's Guide

Literature Number: SPRU194
January 1998

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

About This Manual

This user's guide serves as a reference book for the TMS320C3x generation of digital signal processors, which includes the TMS320C30, TMS320C31, TMS320LC31 and TMS320C32. Throughout the book, all references to 'C3x refer collectively to 'C30, 'C31, and 'C32 and the TMS320C30, TMS320C31, and TMS320C32 refer to all speed variations unless an exception is noted. This document provides information to assist managers and hardware/software engineers in application development.

Specifically, this book complements the TMS320C3x *User's Guide* by providing information to assist you in application development. It includes example code and hardware connections for various appliances.

This guide presents examples of 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 a detailed explanation of the underlying theory is too extensive to be included in this manual, appropriate references are given for further information.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface` that is similar to that of a typewriter. Examples use a **`bold version`** of the special typeface for emphasis. Interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  The following is a sample program listing:

```
0011  0005  0001          .field    1, 2
0012  0005  0003          .field    3, 4
0013  0005  0006          .field    6, 3
0014  0006                .even
```

  The following is an example of a system prompt and a command you might enter:

```
C:  csr –a /user/ti/simuboard/utilities
```

- Any string within angle brackets is considered to be a variable. In syntax descriptions, the variable is written in a typeface similar to that of the text. The following is an example of a variable syntax:

  <file name> Path name of a UNIX file
  <signal> Name of a signal

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown below. Portions of a syntax that are in *italics* describe the type of information that should be entered. The following is an example of a directive syntax:

  .asect "*section name***",** *address*

  In the preceding example, ".asect" is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use ".asect," the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you must specify the information within the brackets; you must not enter the brackets themselves. The following is an example of an instruction that has an optional parameter:

  LALK *16–bit constant [, shift]*

The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

Square brackets are also used as part of the pathname specification for VMS pathnames. In this case, the brackets are actually part of the pathname (they are not optional).

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If it is a required parameter, it is shown starting against the left margin of the shaded box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in column 1.

  symbol **.usect** **"***section name***",** *size in bytes* [**,** *alignment*]

  The *symbol* is required for the .usect directive and must begin in column 1. The *section name* must be enclosed in quotes and the parameter *size in bytes* must be separated from the *section name* by a comma. The *alignment* is optional and, if used, must be separated by a comma.

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. The following is an example of a list:

  ```
  { * | *+ | *- }
  ```

  This provides three choices: *, *+, or *–.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte $value_1$ [, ... , $value_n$]

Note that **.byte** does not begin in column one.

This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, each separated from the previous one by a comma.

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.

JTAG/MPSD Emulation Technical Reference (literature number SPDU079) provides the design requirements of the XDS510™ emulator controller, discusses JTAG designs (based on the IEEE 1149.1 standard), and modular port scan device (MPSD) designs.

Setting Up TMS320 DSP Interrupts in C Application Report (literature number SPRA036) describes methods of setting up interrupts for the TMS320 family of processors in C programming language. Sample code segments are provided, along with complete examples of how to set up interrupt vectors.

TLC32040C, TLC32040I, TLC32041C, TLC32041I Analog Interface Circuits
(literature number SLAS014E) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

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.

TMS320C3x/C4x Code Generation Tools Getting Started Guide (literature number SPRU119) describes how to install the TMS320C3x/C4x assembly language tools and the C compiler. Installation instructions are included for MS–DOS™, Windows 3.x, Windows NT, Windows 95, SunOS™, Solaris, and HP–UX™ systems.

TMS320C3x/C4x 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.

TMS320C3x C Source Debugger (literature number SPRU053) describes the 'C3x debugger for the emulator, evaluation module, and simulator. 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.

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.

TMS320C3x User's Guide (literature number SPRU031) describes the 'C3x 32-bit floating-point microprocessor (developed for digital signal processing as well as general applications), its architecture, internal register structure, instruction set, pipeline, specifications, and DMA and serial port operation. Software and hardware applications are included.

TMS320C3x/C4x Code Generation Tools Getting Started Guide (literature number SPRU119) describes how to install the TMS320C3x/C4x assembly language tools and the C compiler. Installation instructions are included for MS–DOS™, Windows 3.x, Windows NT, Windows 95, SunOS™, Solaris, and HP–UX™ systems.

TMS320C30 Digital Signal Processor (literature number SPRS032A) data sheet contains the electrical and timing specifications for this device, as well as signal descriptions and pinouts for all of the available packages.

TMS320C31, TMS320LC31 Digital Signal Processors (literature number SPRS035) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

TMS320C32 Digital Signal Processor (literature number SPRS027C) data sheet contains the electrical and timing specifications for this device, as well as signal descriptions and pinouts for all of the available packages.

TMS320 DSP Development Support Reference Guide (literature number SPRU011) describes the TMS320 family of digital signal processors and the tools that support these devices. Included are code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). Also covered are available documentation, seminars, the university program, and factory repair and exchange.

TMS320 Family Development Support Reference Guide (literature number SPRU011E) describes the TMS320 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 provides factory repair and exchange information.

TMS320 Third-Party Support Reference Guide (literature number SPRU052C) alphabetically lists over 100 third parties who supply various products that serve the family of TMS320 digital signal processors, including software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

References

The publications in the following reference list contain useful information regarding functions, operations, and applications of digital signal processing (DSP). These books also provide other references to many useful technical papers. The reference list is organized into categories of general DSP, speech, image processing, and digital control theory and is alphabetized by author.

❑ **General Digital Signal Processing:**

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

Bateman, A., and Yates, W., *Digital Signal Processing Design*. Salt Lake City, Utah: W. H. Freeman and Company, 1990.

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

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

Chassaing, R., and Horning, D., *Digital Signal Processing with the TMS320C25.* New York, NY: John Wiley and Sons, Inc., 1990.

Digital Signal Processing Applications with the TMS320 Family, Vol. I. Texas Instruments, 1986; Prentice-Hall, Inc., 1987.

Digital Signal Processing Applications with the TMS320 Family, Vol. II. Texas Instruments, 1990; Prentice-Hall, Inc., 1990.

Digital Signal Processing Applications with the TMS320 Family, Vol. III. Texas Instruments, 1990; Prentice-Hall, Inc., 1990.

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

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

Hutchins, B., and Parks, T., *A Digital Signal Processing Laboratory Using the TMS320C25*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

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

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

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

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

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

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

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

Oppenheim, Alan V., and Schafer, R.W., *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.

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

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

Rabiner, Lawrence R., and Gold, Bernard, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Sorensen, H. V., et al, *Real-Valued Fast Fourier Transform Algorithms*, IEEE Transform on ASSP, June 1987.

Treichler, J.R., Johnson, Jr., C.R., and Larimore, M.G., *Theory and Design of Adaptive Filters*. New York, NY: John Wiley and Sons, Inc., 1987.

❑ **Speech:**

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

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

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

Parsons, Thomas., *Voice and Speech Processing.* New York, NY: McGraw Hill Company, Inc., 1987.

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

Shaughnessy, Douglas., *Speech Communication.* Reading, MA: Addison-Wesley, 1987.

❑ **Image Processing:**

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

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

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

❑ **Multirate DSP:**

Crochiere, R.E., and Rabiner, L.R., *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

Vaidyanathan, P.P., *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

❑ **Digital Control Theory:**

Dote, Y., *Servo Motor and Motion Control Using Digital Signal Processors*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

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

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

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

Moroney, P., *Issues in the Implementation of Digital Feedback Compensators.* Cambridge, MA: The MIT Press, 1983.

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

❑ **Adaptive Signal Processing:**

Haykin, S., *Adaptive Filter Theory.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

Widrow, B., and Stearns, S.D. *Adaptive Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

❑ **Array Signal Processing:**

Haykin, S., Justice, J.H., Owsley, N.L., Yen, J.L., and Kak, A.C. *Array Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

Hudson, J.E. *Adaptive Array Principles.* New York, NY: John Wiley and Sons, 1981.

Monzingo, R.A., and Miller, J.W. *Introduction to Adaptive Arrays.* New York, NY: John Wiley and Sons, 1980.

Trademarks

ABEL is a trademark of DATA I/O.

CodeView, MS, MS-DOS, MS-Windows, and Presentation Manager are registered trademarks of Microsoft Corporation.

DEC, Digital DX, Ultrix, VAX, and VMS are trademarks of Digital Equipment Corporation.

HPGL is registered trademark of Hewlett Packard Company.

Macintosh and MPW are trademarks of Apple Computer Corp.

Micro Channel, OS/2, PC-DOS, and PGA are trademarks of International Business Machines Corporation.

SPARC, Sun 3, Sun 4, Sun Workstation, SunView, and SunWindows are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

Processor Initialization

Before you execute a DSP algorithm, you must initialize the processor. Initialization brings the processor to a known state. Generally, this occurs anytime after the processor is reset. This chapter reviews the concepts of processor initialization explained in the user's guide and provides examples.

Topic **Page**

1.1 Reset Process

You can reset the processor by applying a low level to the $\overline{\text{RESET}}$ input for at least ten H_1 cycles. The ’C3x terminates execution and puts the reset vector (the contents of memory location 0) in the program counter. The reset vector normally contains the address of the system-initialization routine. The hardware reset also initializes various registers and status bits.

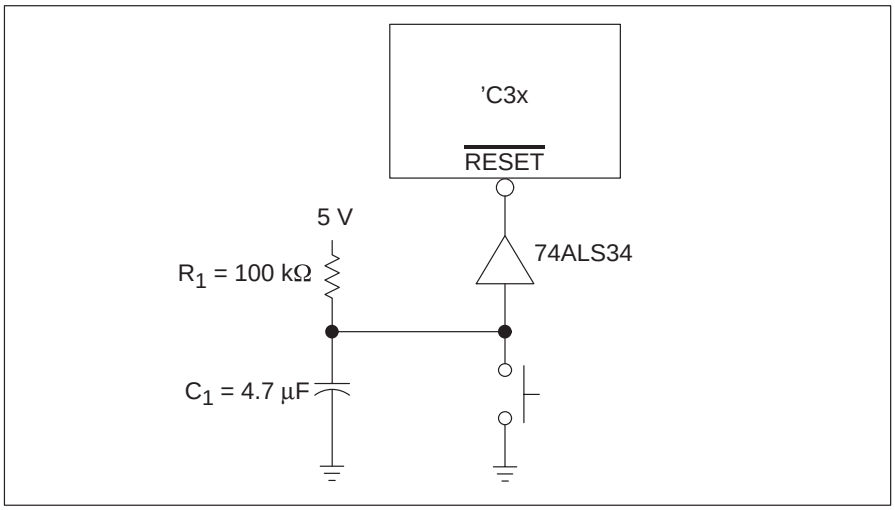
In order to reset the ’C3x correctly, you need to comply with several hardware and software requirements:

- ❑ If the ’C31 or ’C32 is in microcomputer mode, set the $\overline{\text{INTx}}$ pins (as discussed in *Using the TMS320C31 and TMS320C32 Boot Loaders* chapter of the *TMS320C3x User’s Guide*) so that the boot loader 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 on page 1-5 shows how you can initialize that vector.
- ❑ Apply a low level to the $\overline{\text{RESET}}$ input (see section 1.2).

1.2 Reset Signal Generation

The reset input controls the initialization of internal ’C3x logic and also causes the execution of the system initialization software. For proper system initialization, the reset signal must be applied for at least ten H1 cycles, that is, 600 ns for a ’C3x operating at 33.33 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 reset line for 100 to 200 ms. Once a proper reset pulse has been applied, the processor fetches the reset vector from location 0, which contains the address of the system initialization routine, Figure 1–1 shows a circuit that generates an appropriate power-up reset circuit.

Figure 1–1. Reset Circuit



1.3 How to Initialize the Processor

After reset, the ’C3x 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 typically performs several tasks:

- Sets the data-page pointer (DP) register
- Sets the stack pointer
- Sets the interrupt vector table
- Sets the trap vector table
- Sets the external memory control register
- Clears/enables cache

Note:

When running under microcomputer mode (MCBL/$\overline{\text{MP}}$=1), the on-chip bootloader automatically initializes the external memory-control register values from the bootloader table.

The ’C3x can be initialized using assembly language or C.

1.3.1 Processor Initialization Under Assembly Language

If you are running under an assembly-only environment, Example 1–1 on page 1-5 provides a basic initialization routine. This example shows code for initializing the ’C3x to the following machine state:

- All interrupts are enabled.
- The overflow mode is disabled.
- The program cache is enabled.
- The DP register is initialized to 0.
- The memory-mapped control registers are initialized.
- The internal memory is filled with 0s.

Example 1–1. TMS320C3x Processor Initialization

```
*
*   TITLE  PROCESSOR INITIALIZATION
*
    .global   RESET,INIT,BEGIN
    .global   INT0,INT1,INT2,INT3
    .global   ISR0,ISR1,ISR2,ISR3
    .global   DINT,DMA
    .global   TINT0,TINT1,XINT0,RINT0,XINT1,RINT1
    .global   TIME0,TIME1,XMT0,RCV0,XMT1,RCV1
    .global   TRAP0,TRAP1,TRAP2,TRP0,TRP1,TRP2
*
*   PROCESSOR INITIALIZATION FOR THE TMS320C3x
*
*   RESET AND INTERRUPT VECTOR SPECIFICATION. THIS
*   ARRANGEMENT ASSUMES THAT DURING LINKING, THE FOLLOWING
*   TEXT SEGMENT WILL BE PLACED TO START AT MEMORY
*   LOCATION 0.
*
    .sect  “init”           ;  Named section
RESET  .word  INIT          ;  RS± load address INIT to PC
INT0   .word  ISR0          ;  INT0±  loads address ISR0 to PC
INT1   .word  ISR1          ;  INT1±  loads address ISR1 to PC
INT2   .word  ISR2          ;  INT2±  loads address ISR2 to PC
INT3   .word  ISR3          ;  INT3±  loads address ISR3 to PC

XINT0  .word  XMT0          ;  Serial port 0 transmit interrupt processing
RINT0  .word  RCV0          ;  Serial port 0 receive interrupt processing
XINT1  .word  XMT1          ;  Serial port 1 transmit interrupt processing
RINT1  .word  RCV1          ;  Serial port 1 receive interrupt processing
TINT0  .word  TIME0         ;  Timer 0 interrupt processing
TINT1  .word  TIME1         ;  Timer 1 interrupt processing
DINT   .word  DMA           ;  DMA interrupt processing
       .space 20            ;  Reserved space
TRAP0  .word  TRP0          ;  Trap 0 vector processing begins
TRAP1  .word  TRP1          ;  Trap 1 vector processing begins
TRAP2  .word  TRP2          ;  Trap 2 vector processing begins
       .space 29            ;  Leave space for the other 29 traps
*
*   IN THE FOLLOWING SECTION, CONSTANTS THAT CANNOT BE REPRESENTED
*   IN THE SHORT FORMAT ARE INITIALIZED. THE NUMBERS IN PARENTHESES
*   AT THE END OF EACH COMMENT REPRESENT THE OFFSET OF THE
*   REGISTER FROM 808000H (CTRL)
```

Example 1–1. TMS320C3x Processor Initialization (Continued)

```
          .data
MASK      .word  0FFFFFFFFH
BLK0      .word  0809800H  ;  Beginning address of RAM block 0
BLK1      .word  0809C00H  ;  Beginning address of RAM block 1
STCK      .word  0809F00H  ;  Beginning of stack
CTRL      .word  0808000H  ;  Pointer for peripheral±bus memory map
DMACTL    .word  0000000H  ;  Init for DMA control (0)
TIM0CTL   .word  0000000H  ;  Init of timer 0 control (32)
TIM1CTL   .word  0000000H  ;  Init of timer 1 control (48)
SERGLOB0  .word  0000000H  ;  Init of serial 0 glbl control (64)
SERPRTX0  .word  0000000H  ;  Init of serial 0 xmt port control (66)
SERPRTR0  .word  0000000H  ;  Init of serial 0 rcv port control (67)
SERTIM0   .word  0000000H  ;  Init of serial 0 timer control (68)
SERGLOB1  .word  0000000H  ;  Init of serial 1 glbl control (80)
SERPRTX1  .word  0000000H  ;  Init of serial 1 xmt port control (82)
SERPRTR1  .word  0000000H  ;  Init of serial 1 rcv port control (83)
SERTIM1   .word  0000000H  ;  Init of serial 1 timer control (84)
PARINT    .word  0000000H  ;  Init of parallel interface control (100)
IOINT     .word  0000000H  ;  Init of I/O interface control (96)
*
          .text
*
*   THE ADDRESS AT MEMORY LOCATION 0 DIRECTS EXECUTION TO BEGIN HERE
*   FOR RESET PROCESSING THAT INITIALIZES THE PROCESSOR. WHEN RESET
*   IS APPLIED, THE FOLLOWING REGISTERS ARE INITIALIZED TO 0:
*

*   ST —— CPU STATUS REGISTER
*   IE —— CPU/DMA INTERRUPT ENABLE FLAGS
*   IF —— CPU INTERRUPT FLAGS
*   IOF—— I/O FLAGS
*
*   THE STATUS REGISTER HAS THE FOLLOWING ARRANGEMENT:

*   BITS:     31–14 13 12 11 10   9   8   7   6   5  4 3  2  1  0

*   FUNCTION: RESRV GIE CC CE CF RESRV RM  OVM  LUF  LV UF N  Z  V  C
*

INIT   LDP    0,DP      ;  Point the DP register to page 0
       LDI    1800H,ST  ;  Clear and enable cache, and disable OVM
       LDI    @MASK,IE  ;  Unmask all interrupts

*
   INTERNAL DATA MEMORY INITIALIZATION TO FLOATING POINT 0
*

   LDI    @BLK0,AR0       ;  AR0 points to block 0
   LDI    @BLK1,AR1       ;  AR1 points to block 1
   LDF    0.0,R0          ;  0 register R0
   RPTS   1023            ;  Repeat 1024 times ...
   STF    R0,*AR0++(1)    ;  Zero out location in RAM block 0 and ...
|| STF    R0,*AR1++(1)    ;  Zero out location in RAM block 1
```

Example 1–1. TMS320C3x Processor Initialization (Continued)

```
*
*   THE PROCESSOR IS INITIALIZED. THE REMAINING APPLICATION–
*   DEPENDENT PART OF THE SYSTEM (BOTH ON– AND OFF–CHIP) SHOULD
*   NOW BE INITIALIZED.
*
*   FIRST, INITIALIZE THE CONTROL REGISTERS. IN THIS EXAMPLE,
*   EVERYTHING IS INITIALIZED TO 0, SINCE THE ACTUAL INITIALIZATION IS
*   APPLICATION-DEPENDENT.
*
    LDI    @CTRL,AR0        ;  Load in AR0 the pointer to control
*                           ;  registers
    LDI    @DMACTL,R0
    STI    R0,*+AR0(0)      ;  Init DMA control


    LDI    @TIM0CTL,R0
    STI    R0,*+AR0(32)     ;  Init timer 0 control
    LDI    @TIM1CTL,R0
    STI    R0,*+AR0(48)     ;  Init timer 1 control
    LDI    @SERGLOB0,R0
    STI    R0,*+AR0(64)     ;  Init serial 0 global control
    LDI    @SERPRTX0,R0
    STI    R0,*+AR0(66)     ;  Init serial 0 xmt control
    LDI    @SERPRTR0,R0
    STI    R0,*+AR0(67)     ;  Init serial 0 rcv control
    LDI    @SERTIM0,R0
    STI    R0,*+AR0(68)     ;  Init serial 0 timer control
    LDI    @SERGLOB1,R0
    STI    R0,*+AR0(80)     ;  Init serial 1 global control
    LDI    @SERPRTX1,R0
    STI    R0,*+AR0(82)     ;  Init serial 1 xmt control
    LDI    @SERPRTR1,R0
    STI    R0,*+AR0(83)     ;  Init serial 1 rcv control
    LDI    @SERTIM1,R0
    STI    R0,*+AR0(84)     ;  Init serial 1 timer control
    LDI    @PARINT,R0
    STI    R0,*+AR0(100)    ;  Init parallel interface
                            ;     control (C30 only)
    LDI    @IOINT,R0
    STI    R0,*+AR0(96)     ;  Init I/O interface control
*
    LDI    @STCK,SP         ;  Init the stack pointer
    OR     2000H,ST         ;  Global interrupt enable
*
    BR     BEGIN            ;  Branch to the beginning of application

    .end
```

1.3.2 Processor Initialization Under C Language

If you are running under a C environment, your initialization routine is typically boot.asm (from the RTS30.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 stack pointer (SP) register (pointing to the .stack section). You must enable the cache, as shown in Example 1–2, and set up your interrupts inside your main routine before you enable interrupts. See the application report, *Setting Up TMS320 DSP Interrupts in C*, for more information.

Example 1–2. Enabling the Cache

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

1.4 Low-Power Mode Interrupt

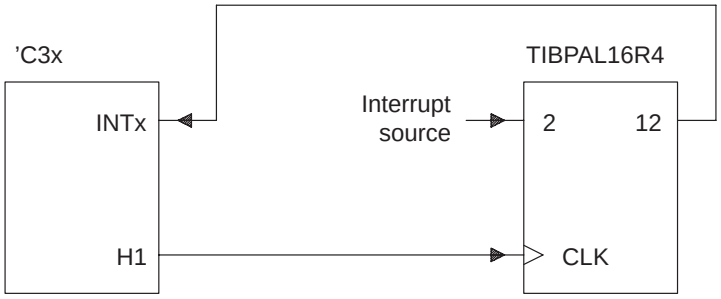
This section explains how to generate interrupts when the IDLE2 power-down mode is used.

The execution of the IDLE2 instruction causes the H1 and H3 processor clocks to be held at a constant level until the occurrence of an external interrupt. To use the IDLE2 power management feature effectively, interrupts must be generated with or without the presence of the H1 clock. For normal (non-IDLE2) operation, however, the interrupt inputs must be synchronized with the falling edge of the H1 clock. An interrupt must satisfy the following conditions:

- It must meet the setup time on the falling edge of H1.
- It must be at least one cycle and less than two cycles in duration.

For an interrupt to be recognized during IDLE2 operation and to turn the clocks back on, it must first be held low for one H1 cycle. The logic in Figure 1–2 can be used to generate an interrupt signal to the ’C3x with the correct timing during non-IDLE2 and IDLE2 operation. Figure 1–2 shows the interrupt circuit, which uses a 16R4 programmable logic device (PLD) to generate the appropriate interrupt signal.

Figure 1–2. Interrupt Generation Circuit for Use With IDLE2 Operation



Example 1–3 shows the PLD equations for the 16R4 using the ABEL™ language. This implementation makes the following assumptions regarding the interrupt source:

- The interrupt source is a low-going pulse or a falling edge. If the interrupt source stays active for more than one H1 cycle, it is regarded as the same interrupt request and not a new one.

- The interrupt source is at least one H1 cycle in duration. One H1 cycle is required to turn the H1 clock on again.

The interrupt is driven active as soon as the interrupt source goes active. It goes inactive again on detection of two H3 rising edges. These two rising edges ensure that the interrupt is recognized during normal operation and after the end of IDLE2 operation (when the clocks turn on again). The interrupt goes inactive after the two H3 clocks are counted and does not go inactive again until after the interrupt source again goes inactive and returns to active.

Example 1–3. State Machine and Equations for the Interrupt Generation 16R4 PLD

```
MODULE INTERRUPT_GENERATION
TITLE' INTERRUPT_GENERATION FOR IDLE2 AND NON-IDLE2 TMS320C31A

TMS320C31'

c3xu5  device 'P16R4';

"inputs
h3  Pin 1;
intsrc_Pin 2;  "Interrupt source

"output
intx_  Pin 12; "Interrupt input signal to the TMS320C31

sync_src_Pin 14; "Internal signal used to synchronize the
           "input to the H1 clock
same_  Pin 15; "Keeps track if the new interrupt source
           "has occurred. If active, no new interrupt
           "has occurred.

"This logic makes the following assumptions:
"The duration of the interrupt source is at least one H1
"cycle in duration. It takes one H1 cycle to turn the H1
"clock on again.

"The interrupt source is pulse- or level-triggered. If the
"source stays active after being asserted, it is regarded
"as the same interrupt request and not a new one.


"Name Substitutions for Test Vectors and Equations

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

source = !intsrc_;
sync   = !sync_src_;
samesrc= !same_;
c3xint = !intx_;

"state bits
outstate   = [samesrc,sync];

idle   = ^b00;
sync_st= ^b01;"synchronize state
wait   = ^b10;"wait for interrupt source to go inactive


state_diagram outstate*
```

Example 1–3.State Machine and Equations for the Interrupt Generation 16R4 PLD (Continued)

```
state idle:
          if (source) then sync_st
          else      idle;


state sync_st:
          if (source) then wait
          else      idle;


state wait:
          if (source) then wait
          else      idle;



equations
          !intx_ =  (source # sync) & !samesrc;

@page

"Test interrupt generation logic
test_vectors
([he,  source] -> [outstate,c3xint])
[  c,  L   ]  -> [idle,    L   ]; "check start from idle
[  L,  H   ]  -> [idle,    H   ]; "test normal interrupt operation
[  c,  H   ]  -> [sync_st, H   ];
[  c,  L   ]  -> [idle,    L   ];
[  c,  L   ]  -> [idle,    L   ];
[  L,  H   ]  -> [idle,    H   ]; "test coming out of idle2 operation
[  L,  H   ]  -> [idle,    H   ];
[  c,  H   ]  -> [sync_st, H   ];
[  c,  L   ]  -> [idle,    L   ];
[  c,  H   ]  -> [sync_st, H   ]; "test same source
[  c,  H   ]  -> [wait,    L   ];
[  c,  H   ]  -> [wait,    L   ];
[  c,  L   ]  -> [idle,    L   ];
[  L,  H   ]  -> [idle,    H   ]; "test idle2 operation
[  L,  H   ]  -> [idle,    H   ];
[  L,  H   ]  -> [idle,    H   ];
end    interrupt_generation
```

Chapter 2

Program Control

This chapter discusses a group of ’C3x instructions that provide program control and facilitate all types of high-speed processing. These instructions handle:

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

Topic **Page**

2.1 Subroutines

The ’C3x has a 24-bit program counter (PC) and a practically unlimited software stack. The CALL and CALL*cond* instructions cause the stack pointer to increment and store the contents of the next value of the program counter on the stack. At the end of the subroutine, the RETS*cond* instruction performs a conditional return.

Example 2–1 illustrates how to use a subroutine to determine the dot product between 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 at which 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 through the RETS instruction when execution has completed. For Example 2–1, it would suffice to save only register R2. However, many registers are saved for demonstration purposes. The saved registers are stored on the system stack. This stack must be large enough to accommodate the maximum anticipated storage requirements. You can use other methods of saving registers, also.

Example 2–1. Subroutine Call (Dot Product)

```
*
*   TITLE  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     ;  Lower 32 and upper 32 bits
        PUSH   AR0    ;  Save AR0
        PUSH   AR1    ;  Save AR1
        PUSH   RC     ;  Save RC
```

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

```
*                                       ;  Initialize R0:
      MPYF3  *AR0,*AR1,R0               ;  a(0) * b(0) ±> R0
      LDF    0.0,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       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
```

2.2 Stacks and Queues

The ’C3x provides a dedicated stack pointer (SP) register 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

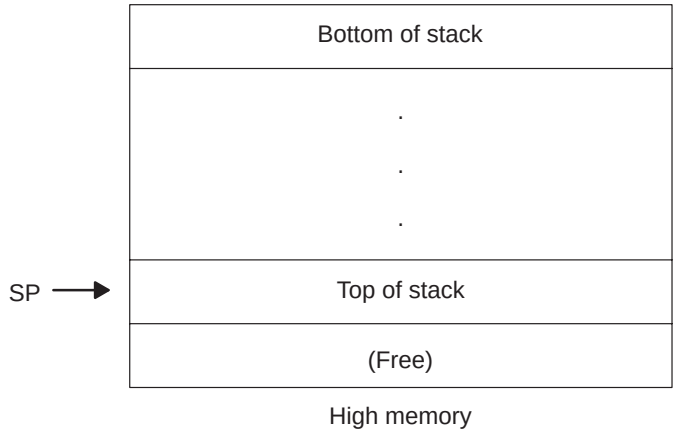
Queue A linear list for which all insertions are made at one end of the list, and all deletions are made at the other end.

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

2.2.1 System Stacks

A stack in the ’C3x fills from a low-memory address to a high-memory address, as shown in Figure 2–1. A system stack stores addresses and data during subroutine calls, traps, and interrupts.

Figure 2–1. System Stack Configuration



The stack pointer 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. Make provisions to accommodate your software’s anticipated storage requirements.

The stack pointer 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 it points to a predetermined memory location. Example 1–1 on page 1-5 shows how to initialize the SP. You must initialize the stack to a valid free memory space. Otherwise, use of the stack can 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 subroutines for temporary storage of registers, as 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–R7. 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 0.

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).

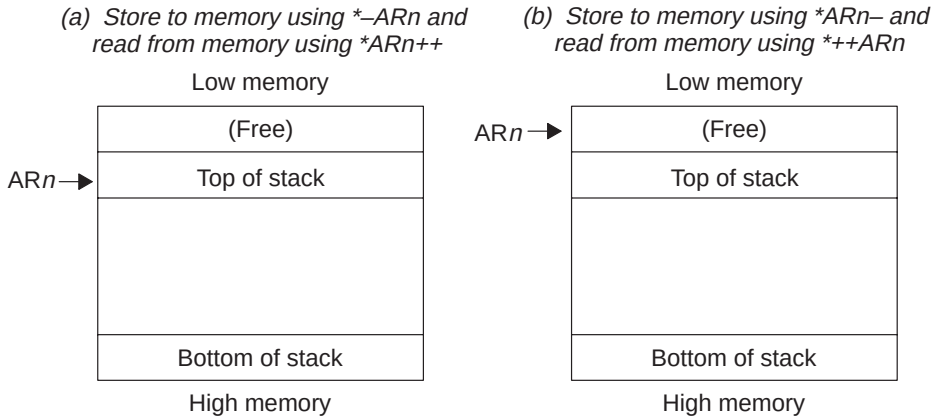
You can implement stack growth from high to low memory in two ways:

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.

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 Figure 2–2 (a), the AR always points to the top of the stack, and in Figure 2–2 (b), the AR always points to the next free location on the stack.

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



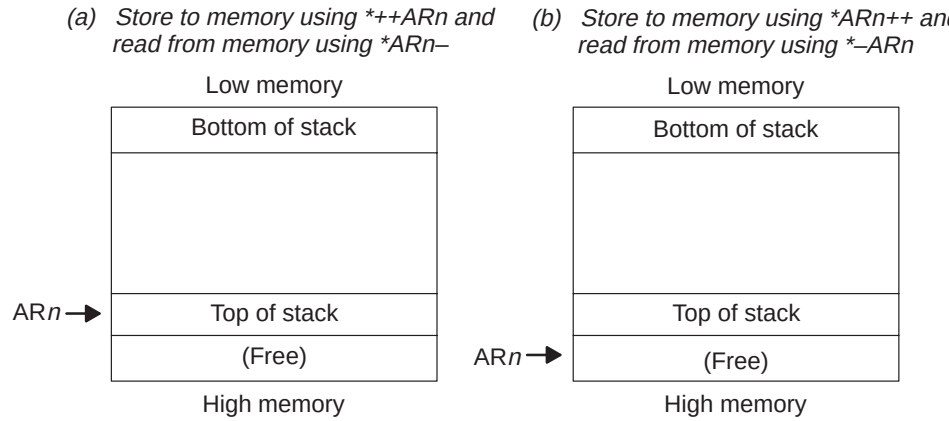
You can implement stack growth from low to high memory in two ways:

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.

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 2–3 illustrates these two cases. In Figure 2–3 (a), the AR always points to the top of the stack, and in Figure 2–3 (b), 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 implementation of queues and double-ended queues is based on 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 Service Routines

Interrupts on the ’C3x are prioritized and vectored. When an interrupt occurs, the corresponding flag is set in the interrupt flag (IF) register. If the corresponding bit in the interrupt enable (IE) register is set and interrupts are enabled by having the global interrupt enable (GIE) bit in the status register set to 1, interrupt processing begins. You can also write to the IF register, allowing you to force an interrupt by software or to clear interrupts without processing them.

2.3.1 Correct Interrupt Programming

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

1) Create and place an interrupt-vector table in the appropriate memory location.
2) Initialize the ITTP bit field (’C32 only).
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 2–2 shows the case in which a subroutine is called when external interrupt 1 has not occurred.

Example 2–2. Use of Interrupts for Software Polling

```
*   TITLE INTERRUPT POLLING

    .
    .
    .
    TSTB 40H,IF        ; Test if interrupt 1 has occurred
    CALLZ SUBROUTINE   ; If not, call subroutine
    .
    .
    .
```

When interrupt processing begins, the program counter (PC) is pushed onto the stack, and the interrupt vector is loaded into the PC. Interrupts are then disabled by clearing the GIE bit to 0, and the program continues from the address loaded in the PC. Since all interrupts are disabled, interrupt processing can proceed without further interruption, unless the interrupt service routine reenables interrupts.

2.3.3 Interrupt Priority

Interrupts on the ’C3x are automatically prioritized. This allows interrupts that occur simultaneously to be serviced in a predefined order. Infrequent (but lengthy) interrupt service routines (ISRs) might need to be interrupted by more frequently occurring interrupts. In Example 2–3, the ISR for INT2 temporarily modifies the IE register to permit interrupt processing when an interrupt to INT0 (but no other interrupt) occurs. When the routine finishes processing, the IE register is restored to its original state. The RETI*cond* instruction not only pops the next program counter address from the stack, but also sets the GIE bit of the status register. This enables all interrupts that have their interrupt enable bit set.

Example 2–3. Interrupt Service Routine

```
*  TITLE INTERRUPT SERVICE ROUTINE
*  .global   ISR2
ENABLE .set   2000h
MASK   .set   1
*
*  INTERRUPT PROCESSING FOR EXTERNAL INTERRUPT INT2±
*
ISR2:
       PUSH   ST         ;  Save status register
       PUSH   DP         ;  Save data page pointer
       PUSH   IE         ;  Save interrupt enable register
       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    MASK,IE    ;  Unmask only INT0
       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    IE         ;  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 during the processing of subroutine calls or interrupts. It can be extensive or simple, depending on system requirements. On the ’C3x, the program counter is automatically pushed onto the stack. Important information in other ’C3x registers, such as the status, auxiliary, or extended-precision registers, must be saved by special commands. To preserve the state of the status register, push it first and pop it last. This keeps the restoration of the extended-precision registers from affecting the status register.

Example 2–4 on page 2-13 and Example 2–5 on page 2-15 show saving and restoring the context of the ’C3x. In both examples, the stack expands towards higher addresses and is used for saving the registers. If you do not want to use the stack pointed at by SP, you can create a separate stack by using an auxiliary register as the stack pointer. Registers saved in these examples are:

- Extended-precision registers (R7 through R0)
- Auxiliary registers (AR7 through AR0)
- Data-page pointer (DP)
- Index registers (IR0 and IR1)
- Block-size register (BK)
- Status register (ST)
- Interrupt-related registers (IE and IF)
- I/O flag (IOF)
- Repeat-related registers (RS, RE, and RC)

You must 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.

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 as follows:

Save as Integers		Save as Floating-Point	
R4	RS	R6	R7
AR4	AR5		
AR6	AR7		
FP	DP (small model only)		
SP			

- If the program is in an interrupt service routine, it must preserve all of the 'C3x registers (see Example 2–6 on page 2-17).

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

Note:

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

Example 2–4. Context Save for the TMS320C3x

```
*   TITLE CONTEXT SAVE FOR THE TMS320C3x
*
*
       .global    SAVE
*
*   CONTEXT SAVE ON SUBROUTINE CALL OR INTERRUPT
*
SAVE:
       PUSH   ST        ;  Save status register
*
*   SAVE THE EXTENDED PRECISION REGISTERS
*
       PUSH   R0        ;  Save the lower 32 bits
       PUSHF  R0        ;     and the upper 32 bits of R0
       PUSH   R1        ;  Save the lower 32 bits
       PUSHF  R1        ;     and the upper 32 bits of R1
       PUSH   R2        ;  Save the lower 32 bits
       PUSHF  R2        ;     and the upper 32 bits of R2
       PUSH   R3        ;  Save the lower 32 bits
       PUSHF  R3        ;     and the upper 32 bits of R3
       PUSH   R4        ;  Save the lower 32 bits
       PUSHF  R4        ;     and the upper 32 bits of R4
       PUSH   R5        ;  Save the lower 32 bits
       PUSHF  R5        ;     and the upper 32 bits of R5
       PUSH   R6        ;  Save the lower 32 bits
       PUSHF  R6        ;     and the upper 32 bits of R6
       PUSH   R7        ;  Save the lower 32 bits
       PUSHF  R7        ;     and the upper 32 bits of R7
*
*   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 2–4.Context Save for the TMS320C3x (Continued)

```
*   SAVE THE REST 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   IE         ;  Save interrupt enable register
        PUSH   IF         ;  Save interrupt flag register
        PUSH   IOF        ;  Save I/O flag register
        PUSH   RS         ;  Save repeat start address
        PUSH   RE         ;  Save repeat end address
        PUSH   RC         ;  Save repeat counter
*
*   SAVE IS COMPLETE
*
```

Example 2–5. Context Restore for the TMS320C3x

```
*
*   TITLE CONTEXT RESTORE FOR THE TMS320C3x
*
       .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    IOF          ;  Restore I/O flag register
       POP    IF           ;  Restore interrupt flag register
       POP    IE           ;  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
*
*   RESTORE THE EXTENDED PRECISION REGISTERS
*
```

Example 2–5. Context Restore for the TMS320C3x (Continued)

```
        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
*
```

2.5 Delayed Branches

The ’C3x uses delayed branches to create single-cycle branching. The delayed branches operate like regular branches but do not flush the pipeline. Instead, the three instructions following a delayed branch are also executed. As discussed in the *Program Flow Control* chapter of the *TMS320C3x User’s Guide*, the only limitations are that none of the three instructions following a delayed branch may be a:

- Branch (standard or delayed)
- Call to a subroutine
- Return from a subroutine
- Return from an interrupt
- Repeat instruction
- TRAP instruction
- IDLE instruction

Conditional delayed branches use the conditions that exist at the end of the instruction immediately preceding the delayed branch. Sometimes a branch is necessary in the flow of a program, but fewer than three instructions can be placed after a delayed branch. For faster execution, it is still advantageous to use a delayed branch. This is shown in Example 2–6, with no operations performed (NOPs) taking the place of the unused instructions. The trade-off is more instruction words for less execution time.

Example 2–6. Delayed Branch Execution

```
*   TITLE DELAYED BRANCH EXECUTION      .
        .
        .
        .
        LDF    *+AR1(5),R2      ;  Load contents of memory to R2
        BGED   SKIP             ;  If loaded number >=0, branch (delayed)
        LDFN   R2,R1            ;  If loaded number <0, load it to R1
        SUBF   3.0,R1           ;  Subtract 3 from R1
        NOP                     ;  Dummy operation to complete delayed
*                               ;     branch
        MPYF   1.5,R1           ;  Continue here if loaded number <0
        .
        .
        .
SKIP    LDF    R1,R3            ;  Continue here if loaded number >=0
```

2.6 Repeat Modes

The ’C3x supports looping without any overhead. For that purpose, there are two instructions: RPTB, which repeats a block of code, and RPTS, which repeats a single instruction. There are three control registers: repeat start-address (RS), repeat end-address (RE), and repeat counter (RC). These contain the parameters that specify loop execution. See the *Program Flow Control* chapter in the *TMS320C3x User’s Guide* for a complete description of RPTB and RPTS. The code automatically sets RS and RF registers RPTB and RPTS when instructions are excluded; however, you must set the repeat counter register.

2.6.1 Block Repeat

Example 2–7 shows an application of the 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, the original array is:

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

The final array after the rearrangement is as follows:

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

Because the exchange operation is performed on two elements simultaneously, it requires 32 operations. The repeat counter register is initialized to 31. In general, if RC contains the number N, the loop is executed N + 1 times. The loop is defined by the RPTB instruction and the EXCH label.

Example 2–7. Loop Using Block Repeat

```
*  TITLE  LOOP USING BLOCK REPEAT
*
*  THIS CODE SEGMENT EXCHANGES THE VALUES OF ARRAY ELEMENTS THAT ARE
*  SYMMETRIC AROUND THE MIDDLE OF THE ARRAY.
*
       .
       .
       .
       LDI    @ADDR,AR0     ;  AR0 points to the beginning of the array
       LDI    AR0,AR1
       ADDI   63,AR1        ;  AR1 points to the end of the
*                           ;  64±element array
       LDI    31,RC         ;  Initialize repeat counter
*
       RPTB   EXCH          ;  Repeat RC+1 times between here and
*                           ;     EXCH
       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)
       .
       .
       .
```

The *Program Flow Control* chapter in the *TMS320C3x User's Guide* discusses restrictions in the block-repeat construct. According to the contents of registers RS, RE, and RC, the program counter is modified at the end of the loop. Therefore, no operation should attempt to modify the repeat counter or the program counter at the end of the loop.

It is possible to nest repeat blocks; however, there is only one set of control registers: RS, RE, and RC. It is necessary to save these registers before entering an inside loop. You can implement a nested loop by using a register as a counter and then using a delayed branch, rather than using the nested repeat block approach.

Example 2–8 shows how to use the block repeat to find a maximum of 147 numbers.

Example 2–8. Use of Block Repeat to Find a Maximum

```
*
*
*  TITLE USE OF BLOCK REPEAT TO FIND A MAXIMUM
*
*  THIS ROUTINE FINDS THE MAXIMUM OF N = 147 NUMBERS.
*
       .
       .
       .
       LDI    146,RC       ;  Initialize repeat counter to 147±1
       LDI    @ADDR,AR0    ;  AR0 points to beginning of array
       LD     *AR0++(1),R0 ;  Initialize MAX to the first value
*
       RPTB   LOOP
       CMPF   *AR0++(1),R0 ;  Compare number to the maximum
LOOP   LDFLT  *±AR0(1),R0  ;  If greater, this is a new maximum
                     .
                     .
                     .
```

2.6.2 Single-Instruction Repeat

The single-instruction repeat uses the control registers RS, RE, and RC in the same way as the block repeat. The advantage over the block repeat is that the instruction is fetched only once, and then the buses are available for moving operands. The single-instruction repeat construct is not interruptible; the block repeat is interruptible.

Example 2–9 shows an application of the single-repeat 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), each of length N = 512, then register R0 contains this quantity after computation:

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

The value of the RC is specified to be 511 in the instruction. If RC contains the number N, the loop is executed N + 1 times.

Example 2–9. Loop Using Single Repeat

```
*   TITLE LOOP USING SINGLE REPEAT
*
*   THIS CODE SEGMENT COMPUTES    SUM[a(i)b(i)] FOR i = 1 to N.
*
*

        .
        .
        .
        LDI    @ADDR1,AR0                ;  AR0 points to array a(i)
        LDI    @ADDR2,AR1                ;  AR1 points to array b(i)
*
        LDF    0.0,R0                    ;  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
||      ADDF3  R1,R0,R0                  ;      and accumulate the
                                         ;      previous one
*
        ADDF   R1,R0                     ;  One final addition
        .
        .
        .
```

2.7 Computed GOTOs

It is occasionally convenient to select the subroutine to be executed during run time (and not during assembly). The ’C3x’s computed GOTO instruction supports this selection. The computed GOTO is implemented 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 a computed GOTO for 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 TASK APPROPRIATE FOR THE CURRENT CYCLE,
*   CALLS THE TASK AS A SUBROUTINE, AND BRANCHES BACK TO THE IDLE
*   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.
*
*
       LDI    5,R0          ;  Initialize R0
       LDI    @ADDR,AR1     ;  AR1 holds base address of the table
WAIT   IDLE                 ;  Wait for the next interrupt
       ADDI3  *AR1,R0,AR2   ;  Add the base address to the table
*                           ;  Entry number
       SUBI   1,R0          ;  Decrement R0
       LDILT  5,R0          ;  If R0<0, reinitialize it to 5
       LDI    *AR2,R1       :  Load the task address
       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

This chapter describes the ’C3x instruction set, which supports both integer and floating-point arithmetic and logical operations. These instructions can be combined to form more complex operations.

Topic **Page**

3.1 Bit Manipulation

Instructions for logical operations, such as AND, OR, NOT, ANDN, and XOR, can be used with the shift instructions for bit manipulation. A special instruction called TSTB tests bits. TSTB performs the same operation as AND, but the result of the logical AND is only used to set the condition flags and is not written anywhere. Example 3–1 and Example 3–2 demonstrate the use of these 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      0100b,IF  ;  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.
*
*                                I
*                                ↓
*
*                 ┌──────────────┬──┬──────────────┐
*                 │              │  │              │      R1
*                 └──────────────┴──┴──────────────┘
*
*                                      J
*                                      ↓
*                 ┌─────────────────────┬──┬──────┐
*                 │                     │  │      │      R2
*                 └─────────────────────┴──┴──────┘
*
*                 ┌─────────────────────┐
*                 │          I          │              *AR0
*                 └─────────────────────┘
*
*                 ┌─────────────────────┐
*                 │          J          │              *(AR0+1)
*                 └─────────────────────┘
*
*
        .
        .
        .
        LDI    1,R0
        LSH    *AR0,R0          ;  Shift 1 to align it with bit I
        TSTB   R1,R0            ;  Test the Ith bit of R1
        BZD    CONT             ;  If bit = 0, branch delayed
        LDI    1,R0
        LSH    *+AR0(1),R0      ;  Align 1 with Jth location
        ANDN   R0,R2            ;  If bit = 0, reset Jth bit of R2
        OR     R0,R2            ;  If bit = 1, set Jth bit of R2
CONT    .
        .
        .
        .
```

3.2 Block Moves

Since the ’C3x 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 to off-chip memory for storage or for multiprocessor data transfers.

You can use direct memory access (DMA) in parallel with CPU operations to accomplish such data transfers. The DMA operation is explained in detail in *Programming the DMA Coprocessor* chapter later in the book. An alternative to DMA is to perform data transfers under program control using load and store instructions in a repeat mode. Example 3–3 shows the transfer of a block of 512 floating-point numbers from external memory to block 1 of the on-chip RAM.

Example 3–3. Block Move Under Program Control

```
*   TITLE BLOCK MOVE UNDER PROGRAM CONTROL
*
extern .word     01000H
block1 .word     0809C00H
       .
       .
       .
       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 Bit-Reversed Addressing

The ’C3x can implement fast Fourier transforms (FFTs) with bit-reversed addressing. If the data to be transformed is in the correct order, the final result of the FFT is presented in bit-reversed order. To recover the frequency-domain data in the correct order, you must swap certain memory locations. The bit-reversed addressing mode makes swapping unnecessary. The next time data needs to be accessed, the access is performed in a bit-reversed manner rather than sequentially. The base address of bit-reversed addressing must be located on a boundary the size of the table. For example, if IR0 = 2^{n-1}, the *n* least significant bits (LSBs) of the base address must be 0.

In bit-reversed addressing, IR0 holds a value equal to one half the size of the FFT if real and imaginary data are stored in separate arrays. During accessing, the auxiliary register is indexed by IR0, but with reverse carry propagation. Example 3–4 illustrates a 512-point complex FFT being moved from the place of computation (pointed at by AR0) to a location pointed at by AR1. In this example, real and imaginary parts, XR(i) and XI(i), of the data are not stored in separate arrays. They are interleaved as XR(0), XI(0), XR(1), XI(1), ..., XR(N-1), XI(N-1). Because of this arrangement, the length of the array is 2N instead of N, and IR0 is set to 512 instead of 256.

Example 3–4. 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    512,IR0
        LDI    2,IR1
        LDI    511,RC              ;  Repeat 511+1 times
        LDF    *+AR0(1),R1         ;  Load first imaginary point
        RPTB   LOOP
*
        LDF    *AR0++(IR0)B,R0     ;  Load real value (and point
||      STF    R1,*+AR1(1)         :     to next 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 Integer and Floating-Point Division

Although division is not implemented as a single instruction in the ’C3x, the instruction set can perform an efficient division routine. Integer and floating-point division are examined separately because a different algorithm is used for each.

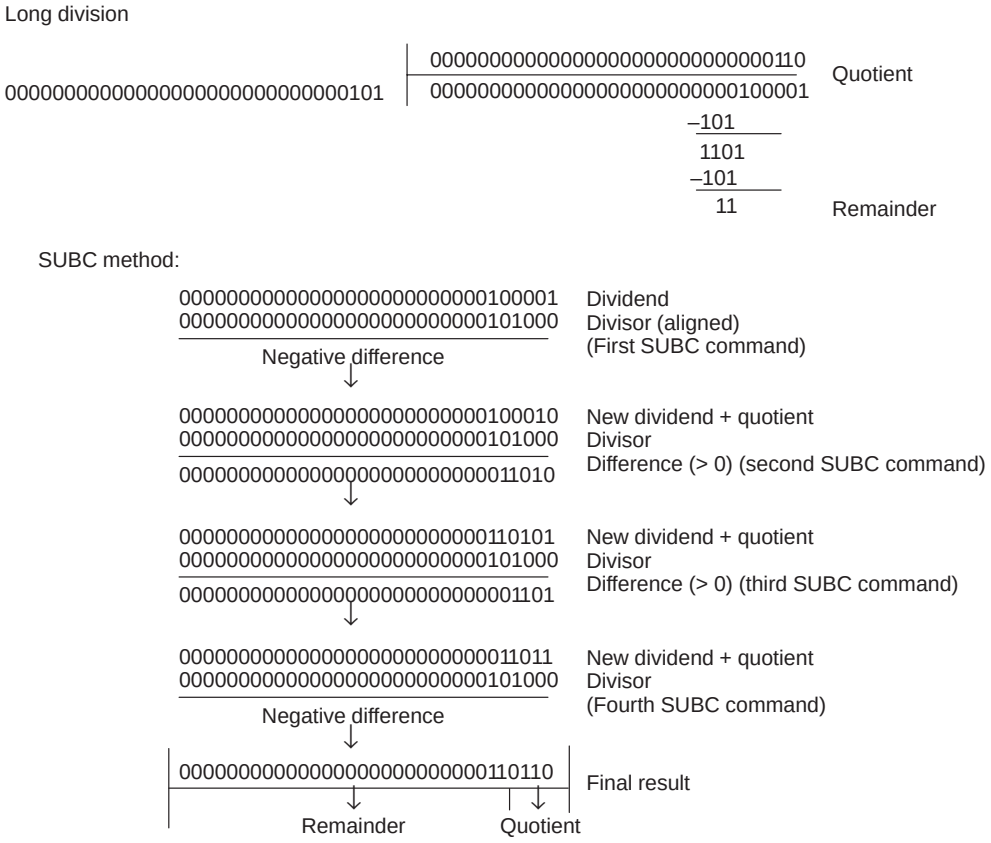
3.4.1 Integer Division

Division is implemented on the ’C3x by repeated subtractions using SUBC, a special conditional subtract instruction. Consider the case of a 32-bit positive dividend with i significant bits (and 32 – i sign bits), as well as 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, which is assumed to be smaller than the dividend, is shifted left i – j times to align it with the dividend. Using SUBC, the shifted divisor is subtracted from the dividend. For each subtraction that does not produce a negative answer, the dividend is replaced by the difference. It is then shifted to the left, and a 1 is put in the LSB. If the difference is negative, the dividend is simply shifted left by 1, leaving a zero in the LSB. 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 (see Figure 3–1). In this case, i = 6 and j = 3, so that the SUBC operation is repeated 6 – 3 + 1 = 4 times.

Figure 3–1. Long Division and SUBC Method



When the SUBC command is used, both the dividend and the divisor must be positive. Example 3–5 shows an example of 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 3–5. 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
```

Example 3–5. Integer Division (Continued)

```
        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
*
*   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 0.
*
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, shift the dividend left by k positions. Then apply the algorithm described above, with i replaced by i + k. It is assumed that i + k is less than 32.

3.4.2 Floating-Point Inverse and Division

This section explains how to implement floating-point division on the ’C3x. Since the algorithm outlined here computes the inverse of a number v, to perform y / v, multiply y by the inverse of v.

The computation of 1 / v is based on the following iterative algorithm. At the ith iteration, the estimate x [i] of 1 / v is computed from v and the previous estimate x [i–1] according to the following formula:

$$x[i] = x[i-1] \times (2.0 - v \times x[i-1])$$

To start the operation, an initial estimate x [0] is needed. If $v = a \times 2^e$, a good initial estimate is:

$$x[0] = 1.0 \times 2^{-e-1}$$

Example 3–6 shows the implementation of this algorithm on the ’C3x, where the iteration has been applied five times. Both accuracy and speed are affected by the number of iterations. The accuracy offered by the single-precision floating-point format is $2^{-23} = 1.192E-7$. If you want more accuracy, use more iterations. If you want less accuracy, reduce the number of iterations to decrease the execution time.

This algorithm properly treats the boundary conditions when the input number either is 0 or has a very large value. When the input is 0, the exponent e = –128. Then the calculation of x[0] yields an exponent that is equal to – (–128) –1 = 127, and the algorithm overflows and saturates. On the other hand, in the case of a very large number with e = 127, the exponent of x[0] is –127 – 1 = –128. This causes the algorithm to yield 0, which is reasonable for handling that boundary condition.

Example 3–6. Inverse of a Floating-Point Number

```
*
*   TITLE INVERSE OF A FLOATING±POINT NUMBER
*
*
*   SUBROUTINE INVF
*
*
*   THE FLOATING-POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, 1/v IS ALSO STORED IN R0.
*
*   TYPICAL CALLING SEQUENCE:
*      LDF    v,R0
*      CALL   INVF
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   v = NUMBER TO FIND THE RECIPROCAL OF (UPON THE CALL)
*   R0        |   1/v (UPON THE RETURN)
*
*   REGISTER USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, R3
*   REGISTER CONTAINING RESULT: R0
*
*      CYCLES: 35   WORDS: 32
*
*
       .global   INVF
*
INVF:  LDF    R0,R3     ;  v is saved for later
       ABSF   R0        ;  The algorithm uses v = |v|
*
*   EXTRACT THE EXPONENT OF v.
*
       PUSHF  R0
       POP    R1
       ASH    ±24,R1    ;  The 8 LSBs of R1 contain the exponent
*                       ;  of v
*
*   x[0] FORMATION IS GIVEN THE EXPONENT OF v.
*
       NEGI   R1
       SUBI   1,R1      ;  Now we have ±e±1, the exponent of x[0]
       ASH    24,R1
       PUSH   R1
       POPF   R1        ;  Now R1 = x[0] = 1.0 * 2**(±e±1)
*
```

Example 3–6. Inverse of a Floating-Point Number (Continued)

```
*   NOW THE ITERATIONS BEGIN.
*
        MPYF   R1,R0,R2  ;  R2 = v * x[0]
        SUBRF  2.0,R2    ;  R2 = 2.0 ± v * x[0]
        MPYF   R2,R1     ;  R1 = x[1] = x[0] * (2.0 ± v * x[0])
*
        MPYF   R1,R0,R2  ;  R2 = v * x[1]
        SUBRF  2.0,R2    ;  R2 = 2.0 – v * x[1]
        MPYF   R2,R1     ;  R1 = x[2] = x[1] * (2.0 ± v * x[1])
*
        MPYF   R1,R0,R2  ;  R2 = v * x[2]
        SUBRF  2.0,R2    ;  R2 = 2.0 ± v * x[2]
        MPYF   R2,R1     ;  R1 = x[3] = x[2] * (2.0 ± v * x[2])
*
        MPYF   R1,R0,R2  ;  R2 = v * x[3]
        SUBRF  2.0,R2    ;  R2 = 2.0 ± v * x[3]
        MPYF   R2,R1     ;  R1 = x[4] = x[3] * (2.0 ± v * x[3])
*
        RND    R1        ;  This minimizes error in the LSBs
*
*   FOR THE LAST ITERATION WE USE THE FORMULATION:
*   x[5] = (x[4] * (1.0 ± (v * x[4]))) + x[4]
*
        MPYF   R1,R0,R2  ;  R2 =  v * x[4] = 1.0..01.. => 1
        SUBRF  1.0,R2    ;  R2 =  1.0 ± v * x[4] = 0.0..01... => 0
        MPYF   R1,R2     ;  R2 =  x[4] * (1.0 ± v * x[4])
        ADDF   R2,R1     ;  R2 =  x[5] = (x[4]*(1.0±(v*x[4])))+x[4]
*
        RND    R1,R0     ;  Round since this is followed by a MPYF
*
*   NOW THE CASE OF v < 0 IS HANDLED.
*
        NEGF   R0,R2
        LDF    R3,R3     ;  This sets condition flags
        LDFN   R2,R0     ;  If v < 0, then R0 = ±R0
*
        RETS
*
*       END
*
        .end
```

3.5 Square Root Computation

An iterative algorithm is used to compute a square root on the ’C3x and is similar to the one used for computation of the inverse. This algorithm computes the inverse of the square root of a number v, 1 / SQRT(v). To derive SQRT(v), multiply this result by v. Since in many applications division by the square root of a number is desirable, the output of the algorithm saves the effort to compute the inverse of the square root.

At the ith iteration, the estimate x[i] of 1 / SQRT(v) is computed from v and the previous estimate x[i-1] according to this formula:

$$x[i] = x[i-1] \times (1.5 - (v / 2) \times x[i-1] \times x[i-1])$$

To start the operation, an initial estimate x[0] is needed. If $v = a \times 2^{e}$, a good initial estimate is:

$$x[0] = 1.0 \times 2^{-e/2}$$

Example 3–7 shows the implementation of this algorithm on the ’C3x, where the iteration is applied five times. Both accuracy and speed are affected by the number of iterations. If you want more accuracy and less speed, increase the number of iterations. If you want less accuracy and more speed, reduce the number of iterations.

Example 3–7. Square Root of a Floating-Point Number

```
*
*   TITLE  SQUARE ROOT OF A FLOATING±POINT NUMBER
*
*
*   SUBROUTINE SQRT
*
*   THE FLOATING POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, SQRT(v) IS ALSO STORED IN R0. NOTE
*   THAT THE ALGORITHM ACTUALLY COMPUTES 1/SQRT(v).
*
*
*   TYPICAL CALLING SEQUENCE:
*
*      LDF v, R0
*      CALL SQRT
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   v = NUMBER TO FIND THE SQUARE ROOT OF
*             |   (UPON THE CALL)
*   R0        |   SQRT(v) (UPON THE RETURN)
*
*   REGISTER USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, R3
*   REGISTER CONTAINING RESULT: R0
*
*   CYCLES: 50 WORDS: 39
*
          .global SQRT
*
*   EXTRACT THE EXPONENT OF v.
*
SQRT:  LDF    R0,R3     ;  Save v
       RETSLE           ;  Return if number is non±positive
       PUSHF  R0
       POP    R1
       ASH    ±24,R1    ;  The 8 LSBs of R1 contain exponent of v
       ADDI   1,R1      ;  Add a rounding bit in the exponent
       ASH    –1,R1     ;  e/2
*
*   X[0] FORMATION GIVEN THE EXPONENT OF v.
*
       NEGI   R1
       ASH    24,R1
       PUSH   R1
       POPF   R1        ;  Now R1 = x[0] = 1.0 * 2**(±e/2)
```

Example 3–7. Square Root of a Floating-Point Number (Continued)

```
*
*   GENERATE v/2.
*
       MPYF   0.5,R0    ;  v/2 and take rounding bit out
*
*   NOW THE ITERATIONS BEGIN.
*
       MPYF   R1,R1,R2  ;  R2 = x[0] * x[0]
       MPYF   R0,R2     ;  R2 = (v/2) * x[0] * x[0]
       SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[0] * x[0]
       MPYF   R2,R1     ;  R1 = x[1] = x[0] *
*                       ;      (1.5 ± (v/2)*x[0]*x[0])
       RND    R1
       MPYF   R1,R1,R2  ;  R2 = x[1] * x[1]
       MPYF   R0,R2     ;  R2 = (v/2) * x[1] * x[1]
       SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[1] * x[1]
       MPYF   R2,R1     ;  R1 = x[2] = x[1] *
*                       ;      (1.5 ± (v/2)*x[1]*x[1])
       RND    R1
       MPYF   R1,R1,R2  ;  R2 = x[2] * x[2]
       MPYF   R0,R2     ;  R2 = (v/2) * x[2] * x[2]
       SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[2] * x[2]
       MPYF   R2,R1     ;  R1 = x[3] = x[2]
*                       ;      *(1.5 ± (v/2)*x[2]*x[2])
       RND    R1
*
       MPYF   R1,R1,R2  ;  R2 = x[3] * x[3]
       MPYF   R0,R2     ;  R2 = (v/2) * x[3] * x[3]
       SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[3] * x[3]
       MPYF   R2,R1     ;  R1 = x[4] = x[3]
*                       ;      * (1.5 ± (v/2) * x[3] * x[3])
       RND    R1
*
       MPYF   R1,R1,R2  ;  R2 = x[4] * x[4]
       MPYF   R0,R2     ;  R2 = (v/2) * x[4] * x[4]
       SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[4] * x[4]
       MPYF   R2,R1     ;  R1 = x[5] = x[4]
*                       ;      * (1.5 ± (v/2) * x[4] * x[4])
*
*
       RND    R1,R0     ;  Round
*
       MPYF   R3,R0     ;  Sqrt(v) from sqrt(v**(±1))
*
       RETS
*
*   end
*
       .end
```

3.6 Extended-Precision Arithmetic

The ’C3x offers 32 bits of precision for integer arithmetic and 24 bits of precision in the mantissa for floating-point arithmetic. For higher precision in floating-point operations, the eight extended-precision registers R7 to R0 contain eight additional bits of accuracy. Since no comparable extension is available for fixed-point arithmetic, this section shows how you can achieve fixed-point double precision by using the processor. The technique consists of performing the arithmetic by parts (which is similar to performing longhand arithmetic).

In the instruction set, operations 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 arithmetic logic unit (ALU) and by the rotate and shift instructions. It can also be manipulated 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 3–8 and Example 3–9 show 64-bit addition and 64-bit subtraction. The first operand is stored in registers R0 (low word) and R1 (high word). The second operand is stored in R2 and R3. The result is stored in R0 and R1.

Example 3–8. 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 3–9. 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. The procedure for multiplication is to split the 32-bit magnitude values of the multiplicand X and the multiplier Y into two parts (X1, X0) and (X3, X2), respectively, with 16 bits each. The operation is done on unsigned numbers, and the product is adjusted for the sign bit. Example 3–10 shows the implementation of a 32-bit by 32-bit multiplication.

Example 3–10. 32-Bit-by-32-Bit Multiplication

```
*
*   TITLE 32 BIT X 32 BIT MULTIPLICATION
*
*
*   SUBROUTINE EXTMPY
*
*   FUNCTION: TWO 32±BIT NUMBERS ARE MULTIPLIED, PRODUCING A 64±BIT
*   RESULT. THE TWO NUMBERS (X and Y) ARE EACH SEPARATED INTO TWO
*   PARTS (X1 X0) AND (Y1 Y0), WHERE X0, X1, Y0, AND Y1 ARE 16 BITS.
*   THE TOP BIT IN X1 AND Y1 IS THE SIGN BIT. THE PRODUCT IS
*   IN TWO WORDS (W0 AND W1). THE MULTIPLICATION IS PERFORMED ON
*   POSITIVE NUMBERS, AND THE SIGN IS DETERMINED AT THE END.
*
*
*            X1  X0           BITS OF PRODUCTS
*         X  Y1  Y0           (NOT COUNTING SIGN)        PRODUCT
*         ---------
*            X0*Y0                        16+16          P1
*         X0*Y1                        16+16             P2
*         X1*Y0                        16+16             P3
*      X1*Y1                        16+16                P4
*      --------------
*      W1     W0
*
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*          R0 |   MULTIPLIER AND LOW WORD OF THE PRODUCT
*          R1 |   MULTIPLICAND AND UPPER WORD OF THE PRODUCT
*
*
*   REGISTERS USED AS INPUT: R0, R1
*   REGISTERS MODIFIED: R0, R1, R2, R3, R4, AR0, AR1
*   REGISTER CONTAINING RESULT: R0,R1
*
*
```

Example 3–10. 32-Bit-by-32-Bit Multiplication (Continued)

```
*   CYCLES: 28 (WORST CASE) WORDS: 25
*
    .global EXTMPY
*
EXTMPY    XOR3   R0,R1,AR0 ;  Store sign
          ABSI   R0        ;  Absolute values of X
          ABSI   R1        ;      and Y
*
*   SEPARATE MULTIPLIER AND MULTIPLICAND INTO TWO PARTS
*
       LDI    ±16,AR1
       LSH3   AR1,R0,R2    ;  R2 = X1 = upper 16 bits of X
       AND    0FFFFH,R0    ;  R0 = X0 = lower 16 bits of X
       LSH3   AR1,R1,R3    ;  R3 = Y1 = upper 16 bits of Y
       AND    0FFFFH,R1    ;  R1 = Y0 = lower 16 bits of Y
*
*   CARRY OUT THE MULTIPLICATION
*
       MPYI3  R0,R1,R4     ;  X0*Y0 = P1

       MPYI   R3,R0        ;  X0*Y1 = P2
       MPYI   R2,R1        ;  X1*Y0 = P3
       ADDI   R0,R1        ;  P2+P3
       MPYI   R2,R3        ;  X1*Y1 = P4
*
       LDI    R1,R2
       LSH    16,R2        ;  Lower 16 bits of P2+P3
       CMPI   0,AR0        ;  Check the sign of the product
       BGED   DONE         ;  If >0, multiplication complete
                           ;     (delayed)
       LSH    –16,R1       ;  Upper 16 bits of P2+P3
       ADDI3  R4,R2,R0     ;  W0 = R0 = lower word of the product
       ADDC3  R1,R3,R1     ;  W1 = R1 = upper word of the product
*
*   NEGATE THE PRODUCT IF THE NUMBERS ARE OF OPPOSITE SIGNS
*
       NOT    R0
       ADDI   1,R0
       NOT    R1
       ADDC   0,R1
*
DONE   RETS
       .end
```

3.7 IEEE/TMS320C3x Floating-Point Format Conversion

The fast version of the IEEE-to-'C3x conversion routine was originally developed by Apollo Computer, Inc. Other routines are based on this algorithm.

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. For example, if a 32-bit number has the binary point after the most significant bit (MSB), which is also the sign bit, only fractional numbers (numbers with absolute values less than 1) can be represented. A number having 31 fractional bits is called a Q31 number. 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.

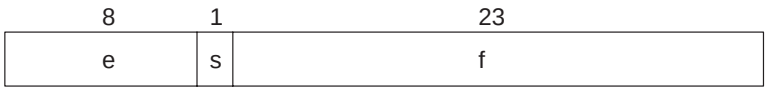
In a floating-point system, each integer or fraction is represented by three fixed-point numbers that constitute a floating-point number. Therefore, a floating-point number consists of a mantissa, m, multiplied by base b raised to an exponent e:

$$m \times b^e$$

To provide the greatest resolution, the mantissa is typically a normalized number with an absolute value between 1 and 2. Although the mantissa is represented as a fixed-point number, the position of the actual value is determined by the exponent e.

To achieve greater efficiency in hardware implementation, the 'C3x uses a floating-point format that differs from the IEEE standard. This section briefly describes the two formats and presents software routines that show how to make conversions between the two formats.

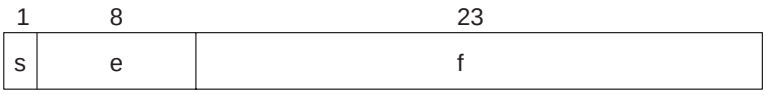
'C3x floating-point format:



In a 32-bit word representing a floating-point number in the ’C3x, the first eight bits correspond to the exponent, expressed in twos-complement format. There is one bit for sign and 23 bits for the mantissa. The mantissa is expressed in twos-complement form, with the binary point after the most significant nonsign bit. Since this bit is the complement of the sign bit s, it is suppressed; the mantissa actually has 24 bits. A special case occurs when e = –128. In this case, the number is interpreted as 0, independently of the values of s and f (which are set to 0 by default). The values of the represented numbers in the ’C3x floating-point format are as follows:

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

IEEE floating-point format:



The IEEE floating-point format uses sign-magnitude notation for the mantissa, and the exponent is biased by 127. In a 32-bit word representing a floating-point number, the first bit is the sign bit. The next eight bits correspond to the exponent, which is 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 after this most significant 1. The mantissa actually has 24 bits. Several special cases are summarized below.

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

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

Special cases:

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

Based on these definitions of the formats, two versions of the conversion routines were developed. One version handles the complete definition of the formats. The other ignores some of the special cases (typically the ones that are rarely used), but has the benefit of executing faster than the complete conversion. For this discussion, the two versions are referred to as the complete version and the fast version, respectively.

3.7.1 IEEE-to-TMS320C3x Floating-Point Format Conversion

Example 3–11 shows the fast conversion from IEEE to ’C3x floating-point format. It properly handles the general case when 0 < e < 255 and also handles 0s (that is, e = 0 and f = 0). The other special cases (denormalized, infinity, and NaN) are not treated and, if present, give erroneous results.

Example 3–11. IEEE-to-TMS320C3x Conversion (Fast Version)

```
*   TITLE IEEE TO TMS320C3x CONVERSION (FAST VERSION)
*
*
*   SUBROUTINE FMIEEE
*
*   FUNCTION: CONVERSION BETWEEN THE IEEE FORMAT AND THE
*   TMS320C3x FLOATING-POINT FORMAT. THE NUMBER TO
*   BE CONVERTED IS IN THE LOWER 32 BITS OF R0.
*   THE RESULT IS STORED IN THE UPPER 32 BITS OF R0.
*   UPON ENTERING THE ROUTINE, AR1 POINTS TO THE
*   FOLLOWING TABLE:
*
*   (0) 0xFF800000 <-- AR1
*   (1) 0xFF000000
*   (2) 0x7F000000
*   (3) 0x80000000
*   (4) 0x81000000
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |  FUNCTION
*   ----------+-------------------------------------
*   R0        |  NUMBER TO BE CONVERTED
*   AR1       |  POINTER TO TABLE WITH CONSTANTS
*
*   REGISTERS USED AS INPUT: R0, AR1
*   REGISTERS MODIFIED: R0, R1
*   REGISTER CONTAINING RESULT: R0
*
*   NOTE:  SINCE THE STACK POINTER SP IS USED, MAKE SURE TO
*      INITIALIZE IT IN THE CALLING PROGRAM.
*
*
*   CYCLES: 12 (WORST CASE) WORDS: 12
*
       .global FMIEEE
*
```

Example 3–11. IEEE-to-TMS320C3x Conversion (Fast Version) (Continued)

```
FMIEEE    AND3      R0,*AR1,R1   ;  Replace fraction with 0
          BND       NEG          ;  Test sign
          ADDI      R0,R1        ;  Shift sign
                                 ;     and exponent inserting 0
          LDIZ      *+AR1(1),R1  ;  If all 0, generate C30 0
          SUBI      *+AR1(2),R1  ;  Unbias exponent
          PUSH      R1
          POPF      R0           ;  Load this as a flt. pt. number
          RETS
*
NEG       PUSH      R1
          POPF      R0           ;  Load this as a flt. pt. number
          NEGF      R0,R0        ;  Negate if orig. sign is negative
          RETS
```

Example 3–12 shows the complete conversion between the IEEE and ’C3x formats. In addition to the general case and the 0s, it handles the special cases as follows:

- If NaN (e = 255, f< >0), the number is returned intact.
- If infinity (e = 255, f = 0), the output is saturated to the most positive or negative number, respectively.
- If denormalized (e = 0, f< >0), two cases are considered. If the MSB of f is 1, the number is converted to ’C3x format. Otherwise, an underflow occurs, and the number is set to 0.

Example 3–12. IEEE-to-TMS320C3x Conversion (Complete Version)

```
*   TITLE IEEE TO TMS320C3x CONVERSION (COMPLETE VERSION)
*
*
*   SUBROUTINE FMIEEE1
*
*   FUNCTION: CONVERSION BETWEEN THE IEEE FORMAT AND THE TMS320C3x
*   FLOATING-POINT FORMAT. THE NUMBER TO BE CONVERTED
*   IS IN THE LOWER 32 BITS OF R0. THE RESULT IS STORED
*   IN THE UPPER 32 BITS OF R0.
*
*
*   UPON ENTERING THE ROUTINE, AR1 POINTS TO THE FOLLOWING TABLE:
*
*   (0) 0xFF800000 <-- AR1
*   (1) 0xFF000000
*   (2) 0x7F000000
*   (3) 0x80000000
*   (4) 0x81000000
*   (5) 0x7F800000
*   (6) 0x00400000
*   (7) 0x007FFFFF
*   (8) 0x7F7FFFFF
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*   AR1       |   POINTER TO TABLE WITH CONSTANTS
*
*   REGISTERS USED AS INPUT: R0, AR1
*   REGISTERS MODIFIED: R0, R1
*   REGISTER CONTAINING RESULT: R0
*
*   NOTE:  SINCE THE STACK POINTER SP IS USED, MAKE SURE TO
*   INITIALIZE IT IN THE CALLING PROGRAM.
*
*
*   CYCLES: 23 (WORST CASE)     WORDS: 34
*
       .global   FMIEEE1
*
FMIEEE1   LDI    R0,R1
          AND    *+AR1(5),R1
          BZ     UNNORM              ;  If e = 0, number is either 0 or
*                                    ;     denormalized
          XOR    *+AR1(5),R1
          BNZ    NORMAL              ;  If e < 255, use regular routine
```

Example 3–12. IEEE-to-TMS320C3x Conversion (Complete Version) (Continued)

```
*   HANDLE NaN AND INFINITY

          TSTB   *+AR1(7),R0
          RETSNZ              ;  Return if NaN
          LDI    R0,R0
          LDFGT  *+AR1(8),R0  ;  If positive, infinity =
                              ;     most positive number
          LDFN   *+AR1(5),R0  ;  If negative, infinity =
          RETS                ;     most negative number RETS

*   HANDLE 0s AND UNNORMALIZED NUMBERS

UNNORM    TSTB   *+AR1(6),R0  ;  Is the MSB of f equal to 1?
          LDFZ   *+AR1(3),R0  ;  If not, force the number to 0
          RETSZ               ;     and return
          XOR    *+AR1(6),R0  ;  If MSB of f = 1, make it 0
          BND    NEG1
          LSH    1,R0         ;  Eliminate sign bit
                              ;     & line up mantissa
          SUBI   *+AR1(2),R0  ;  Make e = ±127
          PUSH   R0
          POPF   R0           ;  Put number in floating point format
          RETS
NEG1      POPF   R0
          NEGF   R0,R0        ;  If negative, negate R0
          RETS

*   HANDLE THE REGULAR CASES
*
NORMAL    AND3   R0,*AR1,R1   ;  Replace fraction with 0
          BND    NEG          ;  Test sign
          ADDI   R0,R1        ;  Shift sign and exponent inserting 0
          SUBI   *+AR1(2),R1  ;  Unbias exponent
          PUSH   R1
          POPF   R0           ;  Load this as a flt. pt. number
          RETS


NEG       POPF   R0           ;  Load this as a flt. pt. number
          NEGF   R0,R0        ;  Negate if original sign negative
          RETS
```

3.7.2 TMS320C3x-to-IEEE Floating-Point Format Conversion

The majority of the numbers represented by the ’C3x floating-point format are covered by the general IEEE format and the representation of 0s. The only special case is e = –127 in the ’C3x format; this corresponds to a denormalized number in IEEE format. It is ignored in the fast version but treated properly in the complete version. Example 3–13 shows the fast version, and Example 3–14 shows the complete version of the ’C3x-to-IEEE conversion.

Example 3–13. TMS320C3x-to-IEEE Conversion (Fast Version)

```
*
*   TITLE TMS320C3x TO IEEE CONVERSION (FAST VERSION)
*
*
*   SUBROUTINE TOIEEE
*
*   FUNCTION: CONVERSION BETWEEN THE TMS320C3x FORMAT AND THE IEEE
*   FLOATING-POINT FORMAT. THE NUMBER TO BE CONVERTED
*   IS IN THE UPPER 32 BITS OF R0. THE RESULT WILL BE IN
*   THE LOWER 32 BITS OF R0.
*
*   UPON ENTERING THE ROUTINE, AR1 POINTS TO THE FOLLOWING TABLE:
*
*   (0) 0xFF800000 <-- AR1
*   (1) 0xFF000000
*   (2) 0x7F000000
*   (3) 0x80000000
*   (4) 0x81000000
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-----------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*   AR1       |   POINTER TO TABLE WITH CONSTANTS
*
*   REGISTERS USED AS INPUT: R0, AR1
*   REGISTERS MODIFIED: R0
*   REGISTER CONTAINING RESULT: R0
*
*   NOTE:  SINCE THE STACK POINTER ‘SP’ IS USED, MAKE SURE TO
*          INITIALIZE IT IN THE CALLING PROGRAM.
*
*
```

Example 3–13. TMS320C3x-to-IEEE Conversion (Fast Version) (Continued)

```
*   CYCLES: 14 (WORST CASE)    WORDS: 15
*
       .global   TOIEEE
*
TOIEEE    LDF    R0,R0            ;  Determine the sign of the number
          LDFZ   *+AR1(4),R0      ;  If 0, load appropriate number
          BND    NEG              ;  Branch to NEG if negative (delayed)
          ABSF   R0               ;  Take the absolute value of the number
          LSH    1,R0             ;  Eliminate the sign bit in R0
          PUSHF  R0
          POP    R0               ;  Place number in lower 32 bits of R0
          ADDI   *+AR1(2),R0      ;  Add exponent bias (127)
          LSH    ±1,R0            ;  Add the positive sign
          RETS


NEG       POP    R0               ;  Place number in lower 32 bits
                                  ;  of R0
          ADDI   *+AR1(2),R0      ;  Add exponent bias (127)
          LSH    ±1,R0            ;  Make space for the sign
          ADDI   *+AR1(3),R0      ;  Add the negative sign
          RETS
```

Example 3–14. TMS320C3x-to-IEEE Conversion (Complete Version)

```
*
*   TITLE TMS320C3x TO IEEE CONVERSION (COMPLETE VERSION)
*
*
*   SUBROUTINE TOIEEE1
*
*
*   FUNCTION: CONVERSION BETWEEN THE TMS320C3x FORMAT AND THE IEEE
*   FLOATING-POINT FORMAT. THE NUMBER TO BE CONVERTED
*   IS IN THE UPPER 32 BITS OF R0. THE RESULT WILL BE
*   IN THE LOWER 32 BITS OF R0.
*
*

*   UPON ENTERING THE ROUTINE, AR1 POINTS TO THE FOLLOWING TABLE:
*

*   (0) 0xFF800000 <-- AR1
*   (1) 0xFF000000
*   (2) 0x7F000000
*   (3) 0x80000000
*   (4) 0x81000000
*   (5) 0x7F800000
*   (6) 0x00400000
*   (7) 0x007FFFFF
*   (8) 0x7F7FFFFF
*

*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*   AR1       |   POINTER TO TABLE WITH CONSTANTS
*

*   REGISTERS USED AS INPUT: R0, AR1
*   REGISTERS MODIFIED: R0
*   REGISTER CONTAINING RESULT: R0
*
*   NOTE:  SINCE THE STACK POINTER 'SP' IS USED, MAKE SURE TO
*          INITIALIZE IT IN THE CALLING PROGRAM.
*
*

*   CYCLES: 31 (WORST CASE)    WORDS: 25
*
    .global   TOIEEE1
```

Example 3–14. TMS320C3x-to-IEEE Conversion (Complete Version) (Continued)

```
*
TOIEEE1   LDF    R0,R0          ;  Determine the sign of the number
          LDFZ   *+AR1(4),R0    ;  If 0, load appropriate number
          BND    NEG            ;  Branch to NEG if negative (delayed)
          ABSF   R0             ;  Take the absolute value
                                ;     of the number
          LSH    1,R0           ;  Eliminate the sign bit in R0
          PUSHF  R0
          POP    R0             ;  Place number in lower 32 bits of R0
          ADDI   *+AR1(2),R0    ;  Add exponent bias (127)
          LSH    ±1,R0          ;  Add the positive sign

CONT      TSTB   *+AR1(5),R0
          RETSNZ                ;  If e > 0, return
          TSTB   *+AR1(7),R0
          RETSZ                 ;  If e = 0 & f = 0, return
          PUSH   R0
          POPF   R0
          LSH    ±1,R0          ;  Shift f right by one bit
          PUSHF  R0
          POP    R0
          ADDI   *+AR1(6),R0    ;  Add 1 to the MSB of f
          RETS

NEG       POP    R0             ;  Place number in lower 32 bits of R0
          BRD    CONT
          ADDI   *+ARI(2),R0    ;  Add exponent bias (127)
          LSH    ±1,R0          ;  Make space for the sign
          ADDI   *+AR1(3),R0    ;  Add the negative sign
          RETS
```

Chapter 4

Memory Interfacing

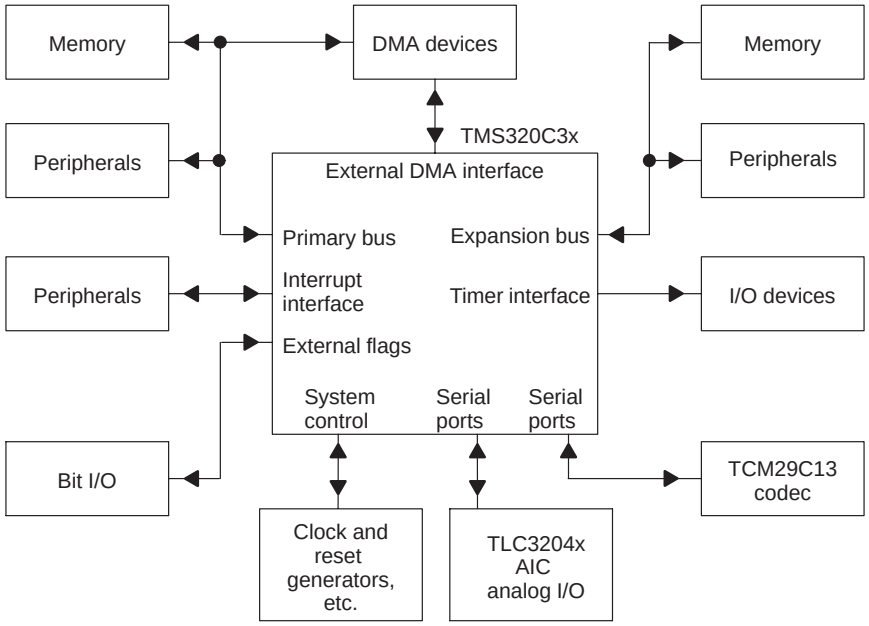
The ’C3x interfaces connect to many device types. Each of these interfaces is tailored to a particular family of devices.

Topic **Page**

4.1 System Configuration

The devices that can be interfaced to the ’C3x include memory, DMA devices, parallel and serial peripherals, and I/O devices. Figure 4–1 illustrates a typical configuration of a ’C3x system with various external devices and the interfaces to which they are connected.

Figure 4–1. Possible System Configurations

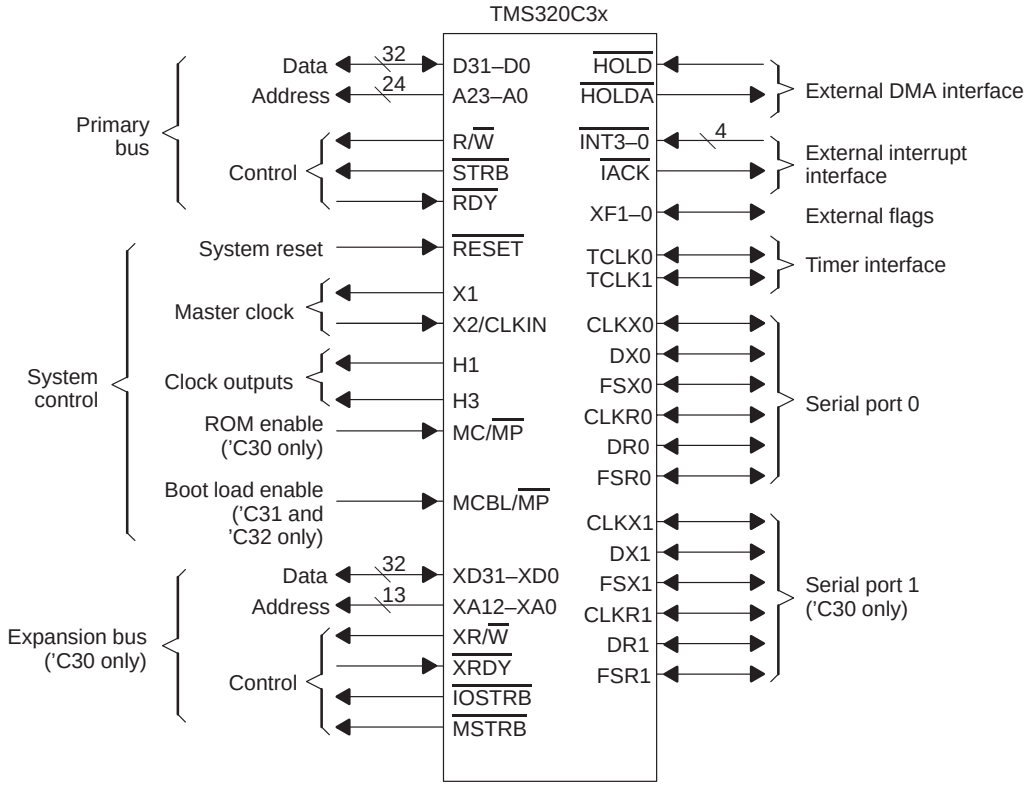


This block diagram represents a fully expanded system. In an actual design, you can use any subset of the illustrated configuration that is appropriate.

4.2 External Interfaces

The ’C3x interface type depends on the device to which it is to be connected. Each interface comprises one or more signal lines that transfer information and control its operation. Figure 4–2 shows the signal line groupings for each of these interfaces.

Figure 4–2. External Interfaces on the TMS320C3x



All of the interfaces are independent of one another, and you can perform different operations simultaneously on each interface.

The primary and expansion buses implement the memory-mapped interface to the device. The external direct memory access (DMA) interface allows external devices to cause the processor to relinquish the primary bus and allow direct memory access.

4.3 Primary Bus Interface

The ’C3x uses the primary bus to access the majority of its memory-mapped locations. When a large amount of external memory is required in a system, it is interfaced to the primary bus. The ’C30 expansion bus (discussed in the *External Memory Interface* chapter of the *TMS320C3x User’s Guide*) actually comprises two mutually exclusive interfaces, controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals. Cycles on the expansion bus that are controlled by the $\overline{\text{MSTRB}}$ signal are equivalent to cycles on the primary bus, except that bank switching is not implemented on the expansion bus. Accordingly, the discussion of primary bus cycles in this section applies equally to $\overline{\text{MSTRB}}$ cycles on the expansion bus.

Although you can use both the primary bus and the expansion bus to interface to a wide variety of devices, those most commonly interfaced to these buses are memory devices. This section presents detailed examples of memory interface.

4.4 Zero-Wait-State Interface to Static RAMs

Zero-wait-state read access time for the ’C3x is determined by the difference between the cycle time and the sum of the delay time for the interface signal H1 low to address valid and the data setup time before the next H1 low. (For more information, see the appropriate *TMS320C3x Digital Signal Processor* data sheet.)

$$t_{c(H)} - \left[t_{d(H1L - A)} + t_{su(D)R}\right]$$

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 example, for full-speed, zero-wait-state interface to any device, the 60-ns ’C3x requires a read access time of 30 ns from address valid to data valid. For most memories, access time from a chip-select pin is the same as access time from address valid; therefore, it is possible to use 30-ns memories at full speed with the ’C3x-33. This requires that there are no delays between the processor and the memories. However, because of interconnection delays and because some gating is normally required for chip-select generation, this is usually not the case. Slightly faster memories are required in most systems.

There are two distinct categories among currently available RAMs:

- RAMs without output enable ($\overline{OE}$) control lines, which include the 1-bit-wide organized RAMs and most of the 4-bit-wide RAMs
- RAMs with $\overline{OE}$ controls, which include the byte-wide RAMs and a few of the 4-bit-wide RAMs

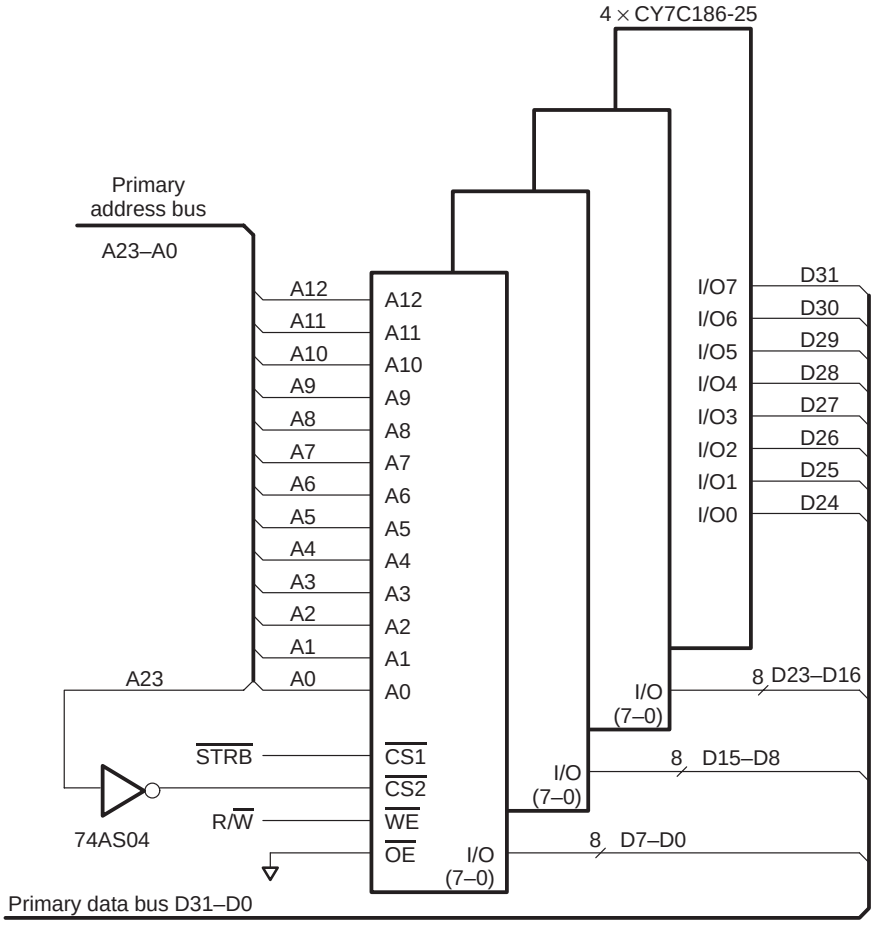
Many of the fastest RAMs do not provide $\overline{OE}$ control; they use chip-select ($\overline{CS}$)-controlled write cycles to ensure that data outputs do not turn on for write operations. In $\overline{CS}$-controlled write cycles, the write control line ($\overline{WE}$) goes low before $\overline{CS}$ goes low, and internal logic holds the outputs disabled until the cycle is completed. Using $\overline{CS}$-controlled write cycles is an efficient way to interface fast RAMs without $\overline{OE}$ controls to the ’C30 at full speed.

In the case of RAMs with $\overline{OE}$ controls, using this signal can add flexibility to many systems. Additionally, many of these devices can be interfaced by using $\overline{CS}$-controlled write cycles with $\overline{OE}$ tied low, in the same manner as with RAMs without $\overline{OE}$ controls. There are, however, two requirements for interfacing to $\overline{OE}$ RAMs in this manner:

- The RAM's $\overline{OE}$ input must be gated internally with the chip-select pin and $\overline{WE}$ so that the device's outputs do not turn on unless a read is being performed.
- The RAM must allow its address inputs to change while $\overline{WE}$ is low; some RAMs specifically prohibit this.

Figure 4–3 shows the 'C3x interface to Cypress Semiconductor's CY7C186 25-ns 8K $\times$ 8-bit CMOS static RAM with the $\overline{OE}$ control input tied low and a $\overline{CS}$-controlled write cycle.

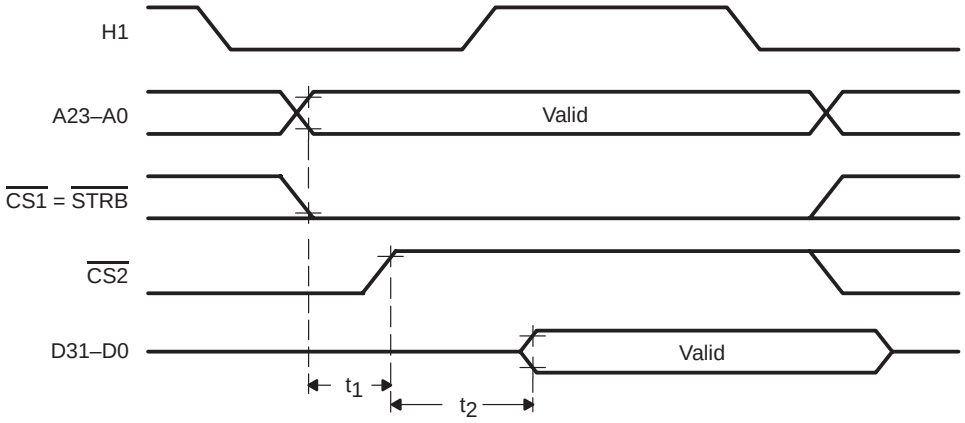
Figure 4–3. TMS320C3x Interface to Cypress Semiconductor's CY7C186 CMOS SRAM



In this circuit, the two chip-select pins on the RAM are driven by the $\overline{\text{STRB}}$ and $\overline{\text{A23}}$ pins, which are ANDed together internally. $\overline{\text{A23}}$ locates the RAM at addresses 00000h through 03FFFh in external memory, and $\overline{\text{STRB}}$ establishes the $\overline{\text{CS}}$-controlled write cycle. The $\overline{\text{WE}}$ control input is then driven by the 'C3x R/$\overline{\text{W}}$ signal. The $\overline{\text{OE}}$ input is not used and is connected to ground.

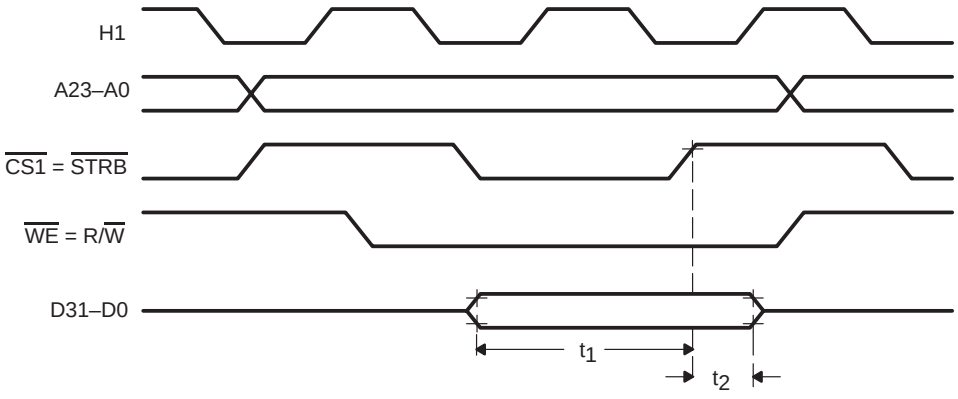
The timing of read operations, shown in Figure 4–4, is very straightforward because the two chip-select inputs are driven directly. The read access time of the circuit is the inverter propagation delay added to the RAM's chip-select access time ($t_1 + t_2 = 5 + 25 = 30$ ns). This access time meets the 'C3x-33's specified 30-ns read access time requirement.

Figure 4–4. Read Operations Timing



During write operations, shown in Figure 4–5, the RAM's outputs do not turn on at all, because of the chip-select controlled write cycles. The chip-select controlled write cycles are generated because R/$\overline{W}$ goes active (low) before the $\overline{STRB}$ term of the chip-select input. Because the RAM's output drivers are disabled whenever the $\overline{WE}$ input is low (regardless of the state of the $\overline{OE}$ input), bus conflicts with the 'C3x are automatically avoided with this interface. The circuit's data setup and hold times (t_1 and t_2 in Figure 4–5) of approximately 50 ns and 20 ns easily meet the RAM's minimum timing requirements of 10 ns and 0 ns.

Figure 4–5. Write Operations Timing



If you require more complex chip-select decode than can be accomplished in time to meet zero-wait-state timing, you can use wait states (see section 4.5, *Wait States and Ready Signal Generation*) or bank-switching techniques (see section 4.5.6).

The CY7C186 SRAM's $\overline{OE}$ control is gated internally with a $\overline{CS}$ pin; the RAM's outputs are not enabled unless the device is selected. This is critical if there are any other devices connected to the same bus. If there are no other devices connected to the bus, $\overline{OE}$ does not need not to be gated internally with a chip-select pin.

To interface RAM without $\overline{OE}$ controls to the 'C3x with a single memory bank and no other devices present on the bus, connect the memory's $\overline{CS}$ input to $\overline{STRB}$ directly. If several devices must be selected, an additional gate is required to AND the device select and $\overline{STRB}$ pins in order to drive the $\overline{CS}$ input that generates the chip-select controlled write cycles. In either case, the $\overline{WE}$ input is driven by the 'C3x R/$\overline{W}$ signal. If sufficient fast gating is used, 25-ns RAMs can be used.

As with RAM with $\overline{OE}$ control lines, this approach works well only if a few banks of memory are implemented and if the chip-select decode can be accomplished with only one level of gating. If many banks are required to implement very large memory spaces, bank switching can be used to provide for multiple bank select generation and still maintain full-speed accesses within each bank. Bank switching is discussed in detail in section 4.5.6 on page 4-15.

4.5 Wait States and Ready Signal Generation

Wait states can greatly increase system flexibility and reduce hardware requirements. The ’C3x can generate wait states on either the primary bus or the expansion bus; both buses have independent sets of ready control logic. This section discusses ready signal generation from the perspective of the primary bus interface. However, since wait-state operation on the expansion bus is similar to that on the primary bus, these discussions also pertain to expansion bus operation. Ready signal generation is not included in discussions of the expansion bus interface. See the *TMS320C3x User’s Guide* for more information.

Wait states are generated on the basis of the:

- Internal wait-state generator
- External ready input ($\overline{\text{RDY}}$)
- Logical AND or OR of the two

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 $\overline{\text{RDY}}$ input may be used for wait-state generation to specific system requirements.

If the logical AND (electrical OR) of the wait count and external ready signals is selected, the latter of the two signals controls the internal ready signal. Both signals must occur. Accordingly, external ready control must be implemented for each wait-state device, and the wait count ready signal must be enabled.

If the logical OR (or electrical AND, since the signals are low true) of the external and internal wait-count ready signals is selected, the earlier of the two signals generates a ready condition and allows the cycle to be completed. Both signals do not need to be present.

4.5.1 ORing the Ready Signals

Performing an OR of the two ready signals can implement wait states for devices that require a greater number of wait states than are implemented with external logic (up to seven). This is useful, for example, if a system contains both fast and slow devices. In this case, fast devices can externally generate a ready signal with a minimum of logic, and slow devices can use the internal wait counter for larger numbers of wait states. When fast devices are accessed, the external hardware responds promptly with a ready signal that terminates the cycle. When slow devices are accessed, the external hardware does not respond and the cycle is terminated after the internal wait count.

You can perform an OR of the two ready signals if conditions require the termination of bus cycles before the number of wait states implemented when external logic takes place. In this case, the wait count that is specified internally is shorter than the number of wait states implemented with the external ready logic, and the bus cycle is terminated after the wait count. This technique can also safeguard against inadvertent accesses to nonexistent memory that would never respond with a ready signal and would lock up the ’C3x.

If an OR of the two ready signals is used and the internal wait-state count is less than the number of wait states implemented externally, the external ready generation logic resets its sequencing to allow a new cycle to begin immediately following the end of the internal wait count. This requires that consecutive cycles come from independently decoded areas of memory and that the external ready generation logic restarts its sequence as soon as a new cycle begins. Otherwise, the external ready generation logic can lose synchronization with bus cycles and generate improperly timed wait states.

4.5.2 ANDing the Ready Signals

Performing an AND of the two ready signals can implement wait states for devices that are equipped to provide a ready signal but cannot respond quickly enough to meet the ’C3x’s timing requirements. Specifically, if these devices normally indicate a ready condition and respond, when accessed, with a wait state until they are ready, using the logical AND of the two ready signals lowers the chip count in the system. In this case, the internal wait counter provides wait states initially and becomes ready after the external device has had time to send a not ready indication. The internal wait counter then remains ready until the external device also becomes ready, which terminates the cycle.

In addition, performing an AND of the two ready signals can extend the number of wait states for devices that already have external ready logic implemented but require additional wait states under certain circumstances.

4.5.3 External Ready Signal Generation

The technique for implementing external ready generation hardware depends on the characteristics of the system. The optimum approach to ready signal generation varies, depending on the relative number of wait-state and non-wait-state devices in the system and on the maximum number of wait states required for any one device. The approach discussed here is general enough for most applications and can easily be modified and applied to many different system configurations.

Ready signal generation involves the following steps:

1) Segmenting the address space to distinguish fast and slow devices

2) Generating properly timed ready indications

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

Segmenting the address space, which is commonly performed by chip-select generation, is required to obtain a unique indication of each area within the address space that requires wait states. You can use chip-select signals to initiate wait states; however, chip-select decoding considerations may occasionally provide signals that do not meet ready input timing requirements. In this case, you can use a small number of address lines to segment coarse address space. The simpler gating allows signals to be generated more quickly. In either case, the signal that indicates a particular area of memory is being addressed normally initiates a ready or wait-state indication.

Once the region of address space being accessed has been established, a timing circuit provides a ready indication to the processor at the appropriate point in the cycle.

Finally, since indications of ready status from multiple devices are typically present, the signals are logically ORed 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:

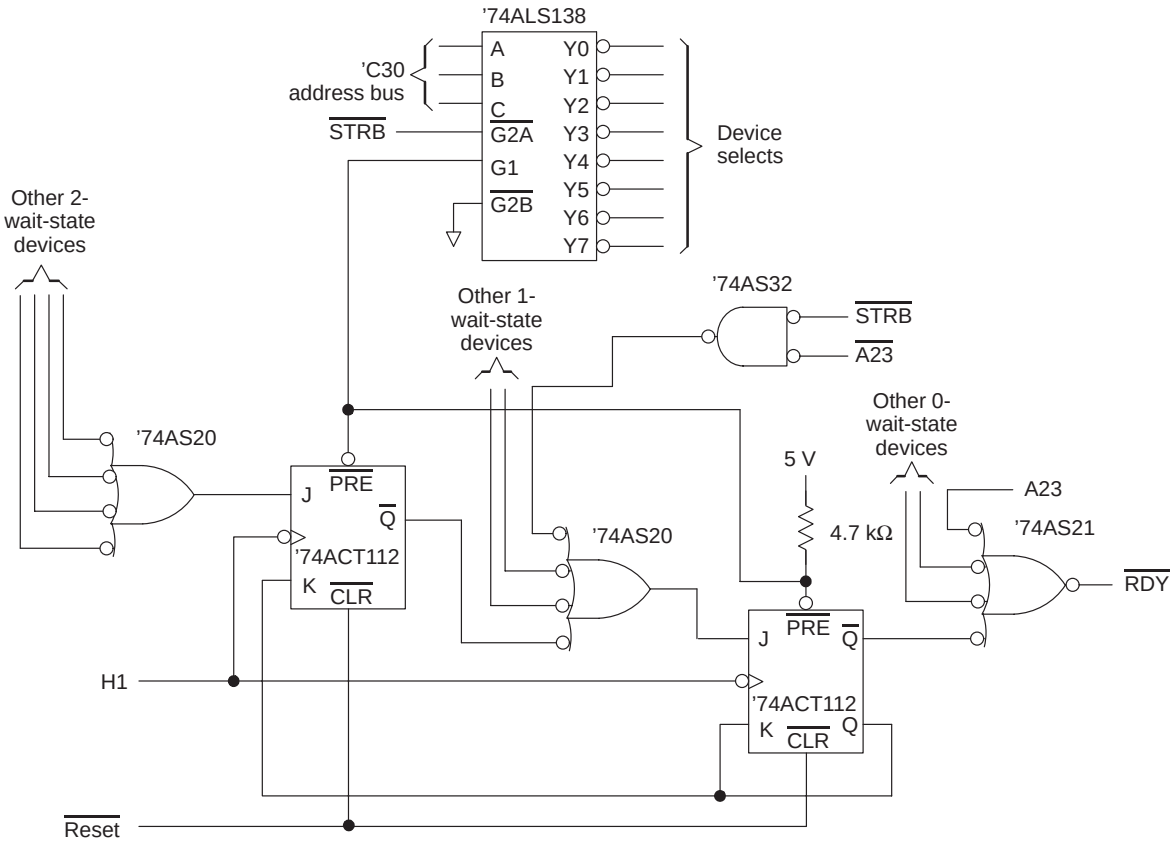
- If $\overline{\text{RDY}}$ is low between accesses, the processor is always ready unless a wait state is required.

  Control of full-speed devices is straightforward; no action is necessary because the ready signal is always active unless otherwise programmed. 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, the processor enters a wait state unless a ready indication is generated.

  Zero-wait-state devices, which tend to be inherently fast, can usually respond immediately with a ready indication. Wait-state devices can delay their select signals to generate a ready indication. Typically, this approach results in the most efficient implementation of ready control logic. Figure 4–6 shows a circuit of this type, which can be used to generate zero, one, or two wait states for multiple devices in a system.

Figure 4–6. Circuit for Generation of Zero, One, or Two Wait States for Multiple Devices



4.5.5 Example Circuit

In the circuit in Figure 4–6, full-speed devices drive ready signals directly through the '74AS21 NOR gate, and the two flip-flops delay wait-state devices' select signals one or two H1 cycles to provide one or two wait states.

Considering the 'C3x-33's ready signal delay time of 8 ns following the address, zero-wait-state devices must use ungated address lines directly to drive the input of the '74AS21, since this gate contributes a maximum propagation delay of 6 ns to the $\overline{\text{RDY}}$ signal. Zero-wait-state devices must be grouped together within a memory address range if other devices in the system require wait states.

With this circuit, devices requiring wait states might take up to 36 ns to provide inputs to the '74AS20 OR gate's inputs from a valid address on the 'C3x. This usually allows sufficient time for any decoding required in generating select signals for slower devices in the system. For example, the 74ALS138 multi-

plexer, driven by the address bus and $\overline{\text{STRB}}$ pin, can generate select decodes in 22 ns, which easily meets the ’C3x-33’s timing requirements.

With this circuit, unused inputs to either the ’74AS20 OR gates or the ’74AS21 NOR gate must be tied to a logic high level to prevent noise from generating spurious wait states.

If more than two wait states are required by devices within a system, other approaches can be used for ready signal generation. If between three and seven wait states are required, additional flip-flops can be included in the same manner shown in Figure 4–6, or internally generated wait states can be used in conjunction with external hardware. If more than seven wait states are required, an external circuit using a counter can be used to supplement the capabilities of the internal wait-state generators.

4.5.6 Bank-Switching Techniques

The ’C3x’s programmable bank-switching feature can greatly ease conflicts on system design circuits when large amounts of memory are required. Normally, devices take longer to release the bus than they take to drive the bus; bank switching provides a period of time for disabling all device selects that are not present otherwise. During this interval, slow devices are allowed time to turn off before other devices have the opportunity to drive the data bus, thus avoiding bus contention. (See the *TMS320C3x User’s Guide* for further information on bank switching.)

When a portion of the high order address lines changes (as defined by the contents of the BNKCMPR register) and bank switching is enabled, $\overline{\text{STRB}}$ goes high for one full H1 cycle. If $\overline{\text{STRB}}$ is included in chip-select decodes, this causes all devices to be disabled during this period. The next bank of devices is not enabled until $\overline{\text{STRB}}$ goes low again.

In general, bank switching is not required during writes because write cycles always exhibit an inherent one-half H1 cycle setup of address information before $\overline{\text{STRB}}$ goes low. When you use bank switching for read/write devices, a minimum of one-half H1 cycle of address setup is provided for all accesses. Therefore, large amounts of memory can be accessed without requiring wait states or extra hardware for isolation between banks. Access time for cycles with bank switching is the same as that for cycles without bank switching. Accordingly, full-speed accesses can still be accomplished within each bank.

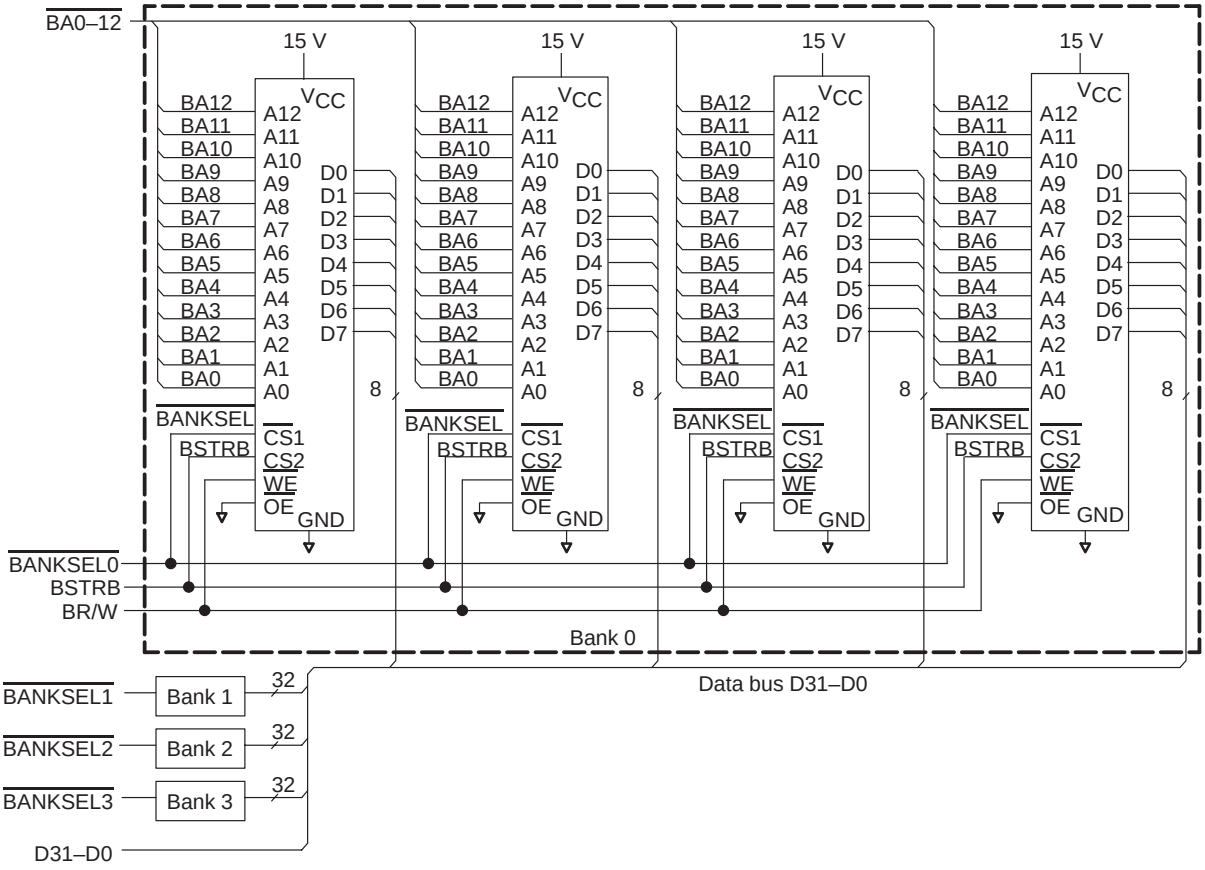
When you use bank switching to implement large multiple-bank memory systems, you must consider address line fanout/loading. Besides parametric specifications which must be accounted for, ac characteristics are crucial in memory system design. With large memory arrays, which commonly require large numbers of address line inputs to be driven in parallel, capacitive loading of address outputs is often quite large. Because all ’C3x timing specifications are guaranteed up to a capacitive load of 80 pF, using greater loads invalidates guaranteed ac characteristics. It is often necessary to provide buffering for address lines when using large memory arrays. The ac timing specifications for buffer performance can then be derated according to manufacturer specifications to accommodate a wide variety of memory array sizes.

The circuit shown in Figure 4–7 illustrates the use of bank switching with Cypress Semiconductor’s CY7C185 25-ns 8K $\times$ 8-bit CMOS static RAM. This circuit implements 32K 32-bit words of memory with one-wait-state accesses for each bank.

The bank memory requires a wait state with this implementation because of the added propagation delay presented by the address bus buffers used in the circuit. The wait state is not a function of the memory organization of multiple banks or the use of bank switching. Memory access speeds are the same with and without bank switching, once bank boundaries are crossed. No speed penalty is incurred by using bank switching, except for the occasional extra cycle inserted when bank boundaries are crossed. If this extra cycle impacts software performance significantly, you can often restructure code to minimize bank boundary crossings and reduce the effect of these boundary crossings on software performance.

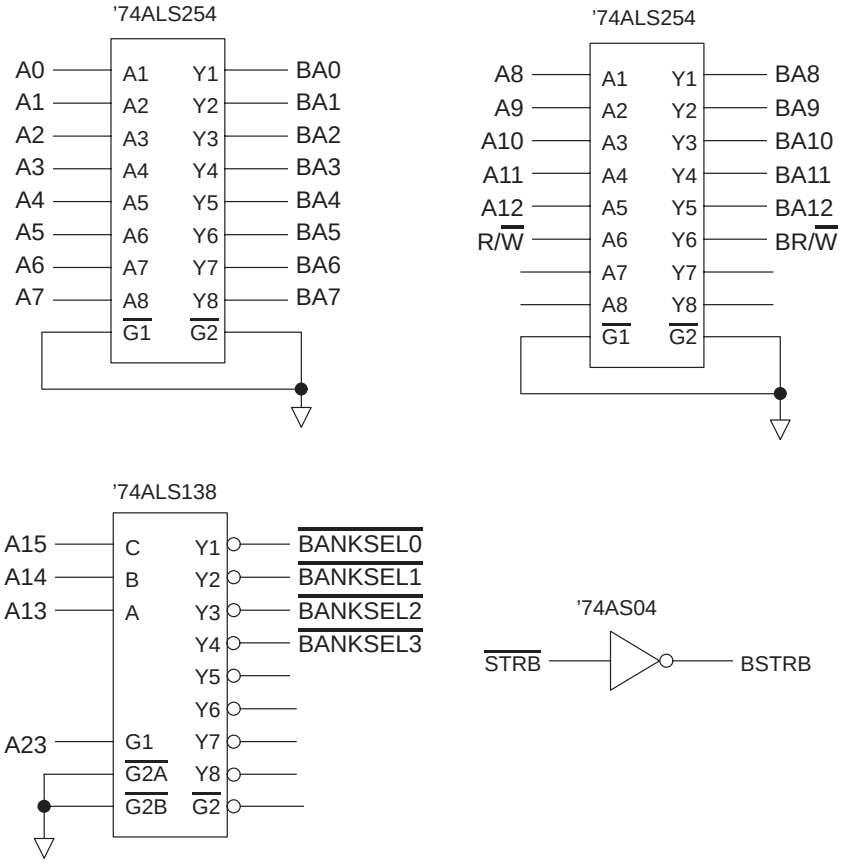
The wait state for this bank memory is generated by using the wait-state generator circuit described in section 4.5.5 on page 4-14. Because the $\overline{\text{A23}}$ signal enables the entire bank memory system, the inverted version of this signal is ANDed with $\overline{\text{STRB}}$ to derive a one-wait-state device select. This signal is then connected in the circuit along with the other one-wait-state device selects. Any time a bank memory access occurs, one wait state is generated.

Figure 4–7. Bank Switching for Cypress Semiconductor's CY7C185 SRAM



Each of the four banks in this circuit is selected by decoding signals A15–A13 generated by the '74ALS138 multiplexer (see Figure 4–8). With the BNKCMPR register set to 0Bh, the banks are selected on even 8K-word boundaries, starting at location 080A000h in external memory space.

Figure 4–8. Bank-Memory Control Logic



The 'C3x rated capacitive loading is 80 pF. The '74ALS254 buffers used on the address lines are necessary in this design because the total capacitive load presented to each address line is a maximum of 16 × 10 pF or 160 pF (bank memory plus zero-wait-state static RAM). Using the manufacturer's derating curves for these devices at a load of 80 pF (the load presented by the bank memory) predicts propagation delays at the output of the buffers to a maximum of 16 ns. The access time of a read cycle within a bank of the memory is the sum of the memory access time and the maximum buffer propagation delay (25 + 16 = 41 ns). Since this propagation delay falls between 30 and 90 ns, it requires only one wait state on the 'C3x-33.

The '74ALS254 buffers offer an additional system-performance enhancement—they include 25-Ω resistors in series with each buffer output. These resistors greatly improve the transient response characteristics of the buffers, especially when driving CMOS loads, such as the memories used here. The effect of these resistors is to reduce overshoot and ringing, which are common

when driving predominantly capacitive loads, such as for CMOS devices. The result is reduced noise and increased immunity in the circuit, which, in turn, results in a more reliable memory system. Having these resistors included in the buffers eliminates the need to put discrete resistors in the system, which is often required in high-speed memory systems.

This circuit cannot be implemented without bank switching because the data output's turn-on and turn-off delays cause bus conflicts. The propagation delay of the '74ALS138 multiplexer is involved only during bank switches, when there is sufficient time between cycles to allow new chip-selects to be decoded.

Figure 4–9 shows the timing of this circuit for read operations using bank switching. With the BNKCMPR register set to 0Bh, when a bank switch occurs, the bank address on address lines A23–A13 is updated during the extra H1 cycle while $\overline{\text{STRB}}$ is high. Then, after chip-select decodes have stabilized and the previously selected bank has disabled its outputs, $\overline{\text{STRB}}$ goes low for the next read cycle. Further accesses occur at normal bus timings with one wait state, as long as another bank switch is not necessary. Write cycles do not require bank switching because of the inherent address setup provided in their timings. This timing is summarized in Table 4–1.

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

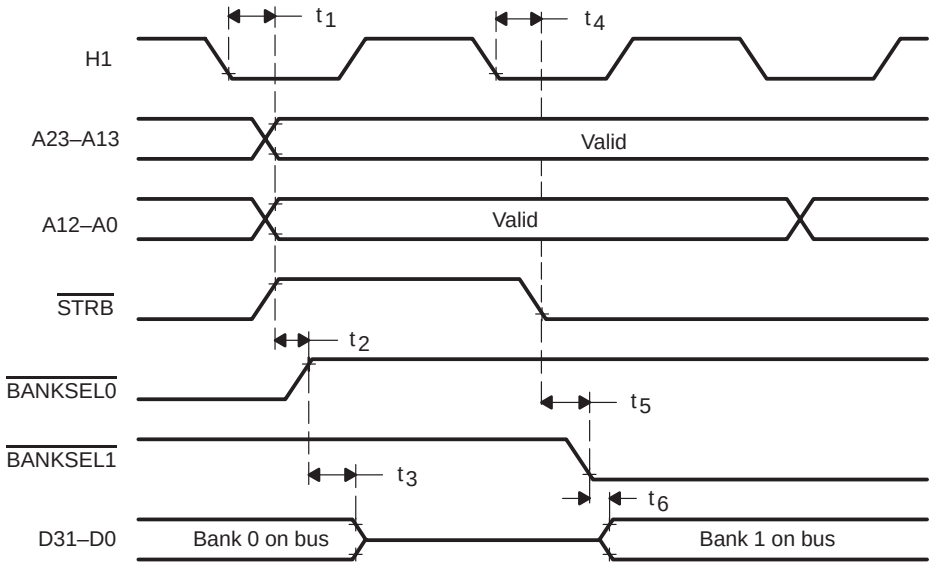


Table 4–1. Bank-Switching Interface Timing for the TMS320C3x-33

Timer Interval	Event	Time Period
t1	H1 falling to address valid/$\overline{\text{STRB}}$ rising	14 ns
t2	Address valid to select delay	10 ns
t3	Memory disable from $\overline{\text{STRB}}$	10 ns
t4	H1 falling to $\overline{\text{STRB}}$	10 ns
t5	$\overline{\text{STRB}}$ to select delay	4.5 ns
t6	Memory output enable delay	3 ns

4.6 Interfacing Memory to the TMS320C32 DSP

The ’C32 accesses external memory with one 24-bit address bus, one 32-bit data bus, and three strobes: $\overline{\text{IOSTRB}}$, $\overline{\text{STRB0}}$, and $\overline{\text{STRB1}}$. The strobes are mapped to selected portions of the memory map as shown in Figure 4–10 on page 4-23. For example, if the CPU is reading data from location 881234h, the active strobe during the read bus cycle is $\overline{\text{STRB0}}$. Unlike the other two strobes, $\overline{\text{STRB0}}$ is assigned to two noncontiguous address spaces within the memory map to provide extra flexibility in address decoding for glueless memory interfaces.

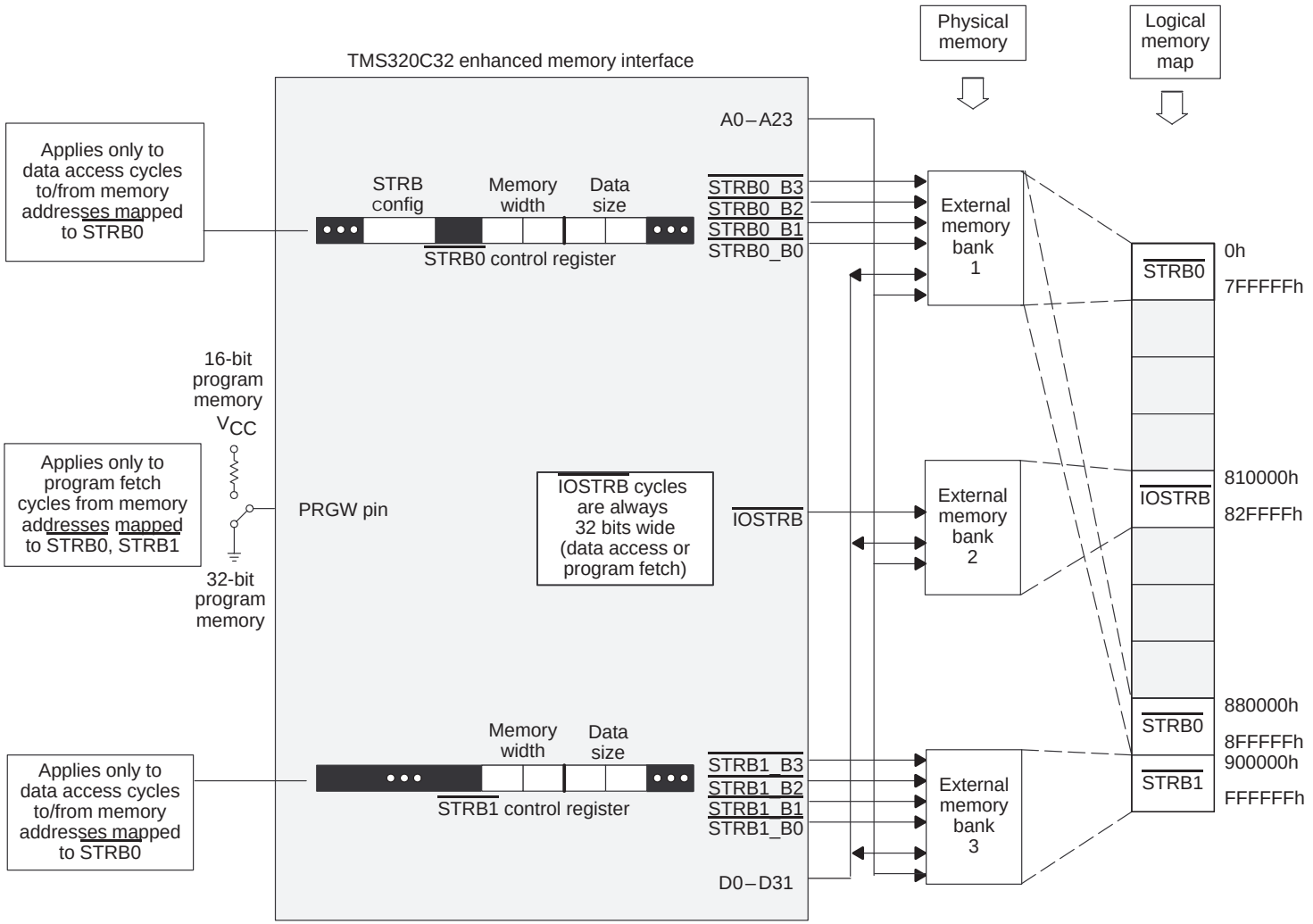
The behavior of $\overline{\text{IOSTRB}}$ is similar to that of its counterpart in the ’C30. Its timing characteristics are slightly relaxed in comparison with $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ cycles to better accommodate slower I/O peripherals. In contrast to $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$, $\overline{\text{IOSTRB}}$ uses a single signal line and accesses the external data one full 32-bit word at a time. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are composed of four signal lines each. The multiple signal lines per strobe enable the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ cycles to access external memory one byte, one half-word, or one full word at a time. For example, to read a single byte from a 32-bit-wide external memory location mapped to $\overline{\text{STRB0}}$, the address on the address bus points to the selected 32-bit word and only one $\overline{\text{STRB0}}$ signal is activated (driven low) to select the desired byte. To access two bytes of data at the memory location mapped to $\overline{\text{STRB1}}$, two $\overline{\text{STRB1}}$ signal lines are asserted during the bus cycle. Full 32-bit bus cycles involving $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ memory space result in four strobe signals simultaneously accessing four bytes of data. The 32-bit $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ bus cycles are no different functionally from the $\overline{\text{IOSTRB}}$ cycles but simply have tighter timing parameters.

The $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ cycles are not limited to just selecting bytes out of 32-bit memory locations. There are two strobe control registers that configure the data size and memory width for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ bus cycles (one control register per strobe). With proper initialization of the strobe control registers, the bus cycles can be configured to encompass any combination of data size and physical memory width. For example, a byte can be read from a 16-bit-wide memory or a 32-bit word can be written to an 8-bit-wide memory by configuring the memory width and data size fields of the corresponding strobe control registers (see Figure 4–10).

Like other members of the ’C3x generation, the ’C32 program, as well as the data, can reside in any portion of the memory map. The ’C32 program fetches from address space mapped to $\overline{\text{IOSTRB}}$ are indistinguishable from $\overline{\text{IOSTRB}}$ data reads or writes. However, the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ cycles are configured slightly differently for program fetches than for data accesses. Program and data can still share the same portions of the memory map, but instead of set-

ting the memory width and data size fields in $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers, the program fetch cycles from the memory spaces mapped to $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are configured by hardwiring the PRGW (program memory width select) pin. There is no need to use the data size fields, because all program fetches apply only to instruction words that are 32 bits wide. The memory width field of the strobe control register is useless at reset, when the processor is fetching the reset vector from memory. At that point the strobe control register is always configured in the same way, but different systems can have different memory widths. The PRGW pin indicates to the memory interface whether the program memory is 16 or 32 bits wide. Program memory that is 8 bits wide is not supported, because four cycles per instruction degrade the performance too much for it to be useful for most applications.

Figure 4–10. $\overline{STRB0}$ and $\overline{STRB1}$ Control Registers and the PRGW Pin



Note: Heavy lines indicate multiple signals.

4.6.1 Functional Description of the Enhanced Memory Interface

The enhanced memory interface controls all data and program traffic between data buses inside the chip and the 32-bit external memory bus as shown in Figure 4–10 through Figure 4–13. For any bus cycle involving a logical memory address range mapped to $\overline{\text{IOSTRB}}$, the memory interface simply connects the external data bus with an appropriate internal data bus without further data manipulation.

The memory interface is much busier when the ’C32 is accessing logical memory addresses mapped to $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$. Depending on the data size and external memory width (as defined by corresponding strobe control registers), data can be packed, unpacked, truncated, or shifted on its way to and from the chip.

Section 4.6.1.1 through section 4.6.1.4 illustrate how the data is manipulated when the interface has to match variable-size data with 8-, 16-, and 32-bit-wide physical memories. In these sections, five lines of code are included in the program space in each figure:

```
   LDI   4,RC
   RPTB  L1
   LDI   *AR0++, R0
   FLOAT R0,R1
L1 STF   R1, *AR1++
```

These lines of code read five integers from one data space, convert them to floating-point format, and write them to another memory space that is assigned to a different strobe. Each example has a different combination of data sizes and external memory widths to illustrate the range of possible combinations.

For data access and program fetch cycles in which the data size exceeds the physical memory width, the least significant bytes/half-words are always transferred first.

4.6.1.1 $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size = Memory Width

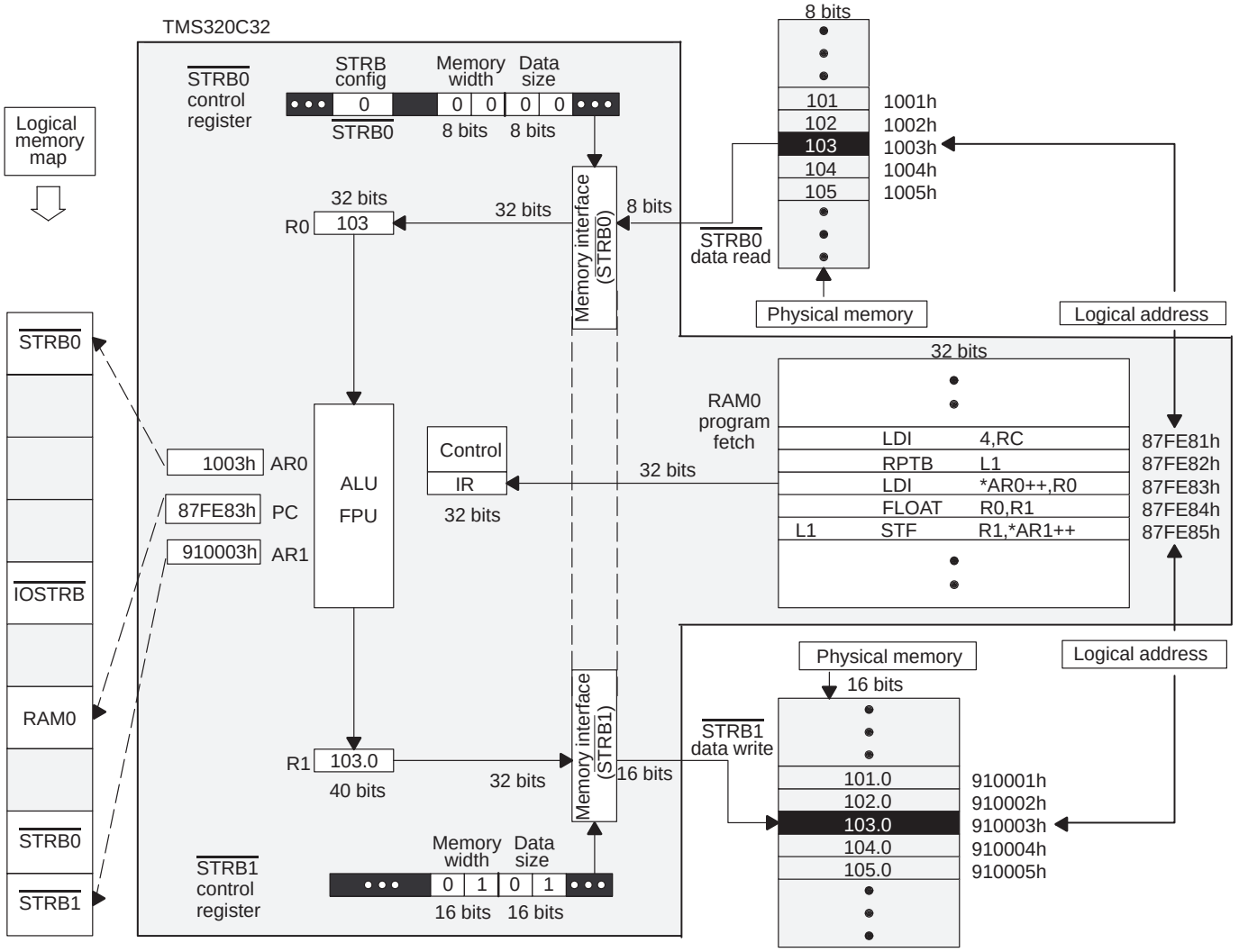
In the case of $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ data access, where data size equals memory width, the data size and memory width for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ data access cycles are configured in the corresponding strobe control registers (see Table 4–2).

The short program stored in the internal RAM0 memory begins with the load integer (LDI) instruction reading an 8-bit integer from 8-bit-wide $\overline{\text{STRB0}}$ memory (see Figure 4–11). As the integer data passes through the memory interface, it is sign extended to 32 bits and loaded to R0 as a 32-bit integer. Next, the integer-to-floating-point conversion (FLOAT) instruction converts the integer in R0 to a 40-bit floating-point number and loads it into R1. Finally, the store floating-point value (STF) instruction truncates the 40-bit contents of R1 to 32 bits and stores it in the 16-bit-wide $\overline{\text{STRB1}}$ memory. As the data passes through the memory interface, the 24-bit mantissa is truncated to eight bits (the 8-bit exponent remains unmodified).

Table 4–2. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size = Memory Width

Data Access	Strobe	Data Size	Memory Width
Input data	$\overline{\text{STRB0}}$	8	8
Output data	$\overline{\text{STRB1}}$	16	16
Program	RAM0	32	32

Figure 4–11. $\overline{STRB0}$ and $\overline{STRB1}$ Data Access: Data Size = Memory Width



4.6.1.2 $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size ≠ Memory Width

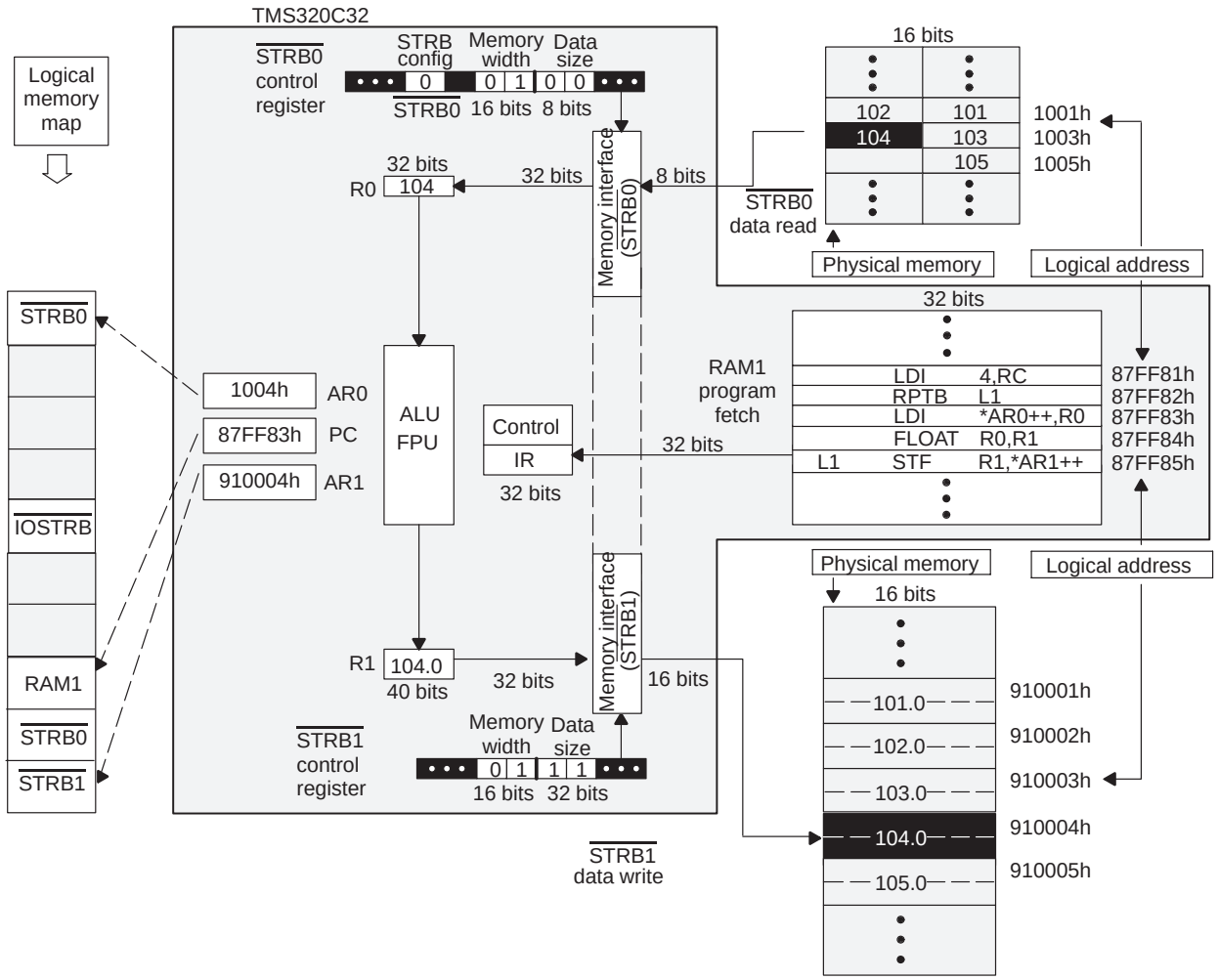
The input and/or output data does not have to be the same size as the memory it is being read to or written from (see Table 4–3). The data size and memory width for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ data access cycles are configured in the corresponding strobe control registers.

The short program stored in the RAM1 memory begins with the LDI instruction reading an 8-bit integer from 16-bit-wide $\overline{\text{STRB0}}$ memory (see Figure 4–12). Since each address contains two data bytes, the memory interface uses different $\overline{\text{STRB0}}$ lines to differentiate between the high byte and the low byte. (Both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ comprise four signals each, one for each byte of the 32 bits.) Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it to R1. Finally, the STF instruction stores the contents of R1 to 16-bit-wide memory as a 32-bit number. Before the data arrives at the memory interface, the 32-bit mantissa is truncated to 24 bits (the 8-bit exponent remains unmodified). The memory interface then stores the 24-bit mantissa and the 8-bit exponent in 16-bit-wide memory, two bytes at a time, using two cycles and two physical memory addresses.

Table 4–3. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ Data Access: Data Size ≠ Memory Width

Data Access	Strobe	Data Size	Memory Width
Input data	$\overline{\text{STRB0}}$	8	16
Output data	$\overline{\text{STRB1}}$	32	16
Program	RAM1	32	32

Figure 4–12. $\overline{STRB0}$ and $\overline{STRB1}$ Data Access: Data Size ≠ Memory Width



4.6.1.3 Program Fetch From 16-Bit $\overline{\text{STRB0}}$ Memory

Table 4–4 shows program memory mapped to 16-bit-wide $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ memory. By hardwiring the PRGW pin to a high state, 32-bit data transfers to and from the 32-bit-wide external memory do not involve any data operations in the memory interface.

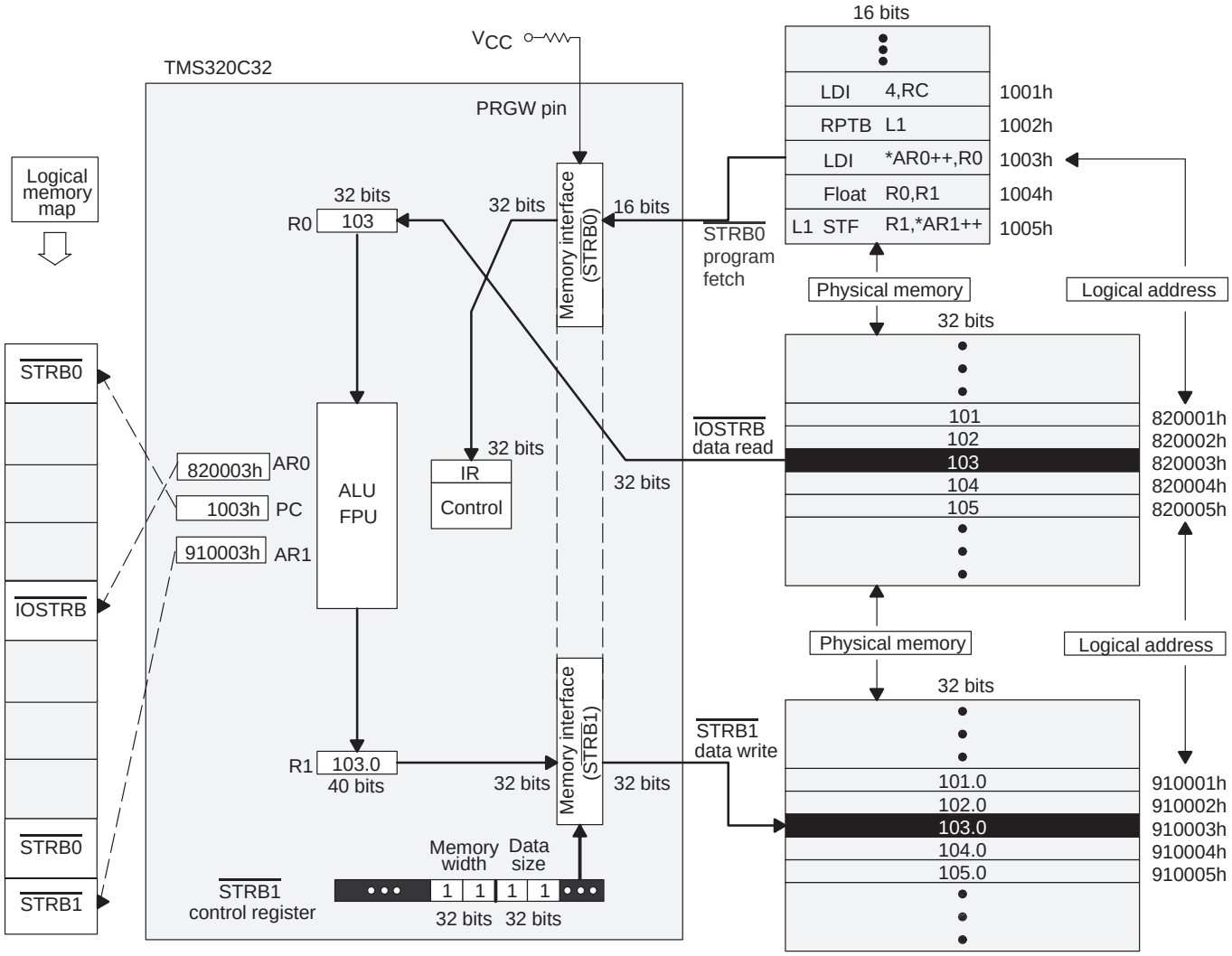
The short program stored in $\overline{\text{STRB0}}$ memory begins with the LDI instruction reading a 32-bit integer from 32-bit-wide $\overline{\text{IOSTRB}}$ memory and loading it to R0 (see Figure 4–13). Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it into R1. Finally, the STF instruction truncates the 40-bit contents of R1 to 32 bits and stores it in the 32-bit-wide $\overline{\text{STRB1}}$ memory. The data is not modified as it passes through the memory interface.

The program controlling the data conversion in this example is stored in the 32-bit-wide memory bank mapped to $\overline{\text{STRB0}}$. As discussed earlier, program fetch cycles do not reference the strobe control register to determine the width of the program memory. Instead, the memory interface checks the state of the PRGW pin to determine the memory width. Because the program memory is 16 bits wide, the PRGW pin should be pulled up to V_{CC}, effectively directing the memory interface to fetch instructions in two bus cycles per instruction (16 bits at a time).

Table 4–4. Program Fetch From 16-Bit $\overline{\text{STRB0}}$ Memory

Data Access	Strobe	Data Size	Memory Width
Input data	$\overline{\text{STRB0}}$	32	32
Output data	$\overline{\text{STRB1}}$	32	32
Program	$\overline{\text{IOSTRB}}$	32	16

Figure 4–13. Program Fetch From 16-Bit $\overline{\text{STRB0}}$ Memory



4.6.1.4 Program Fetch From 32-Bit $\overline{\text{STRB1}}$ Memory

Table 4–5 shows program memory mapped to 32-bit-wide $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ memory. By hardwiring the PRGW pin to a low state, 32-bit data transfers to and from the 32-bit-wide external memory do not involve any data operations in the memory interface.

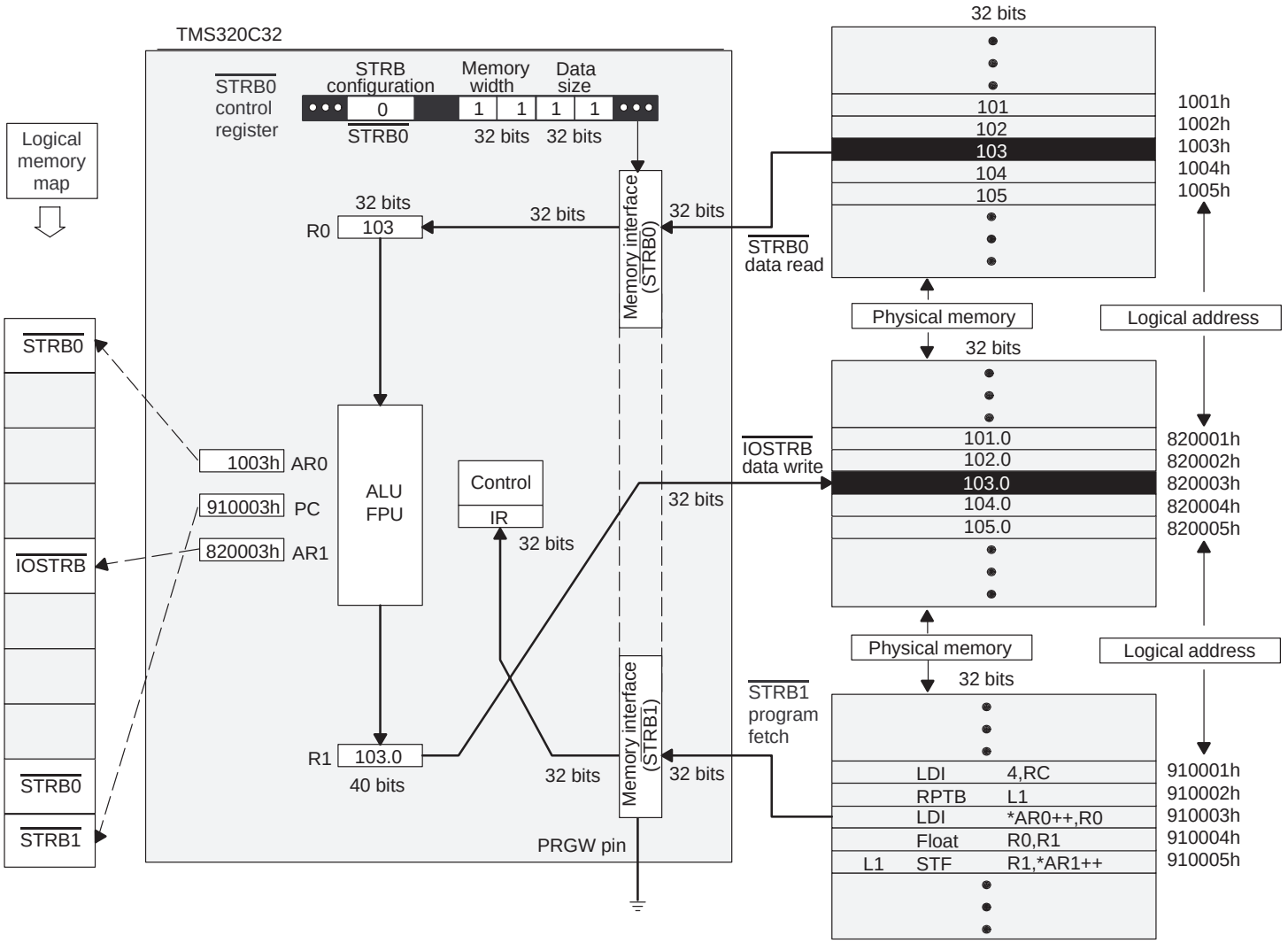
The small program stored in $\overline{\text{STRB1}}$ memory begins with the LDI instruction reading a 32-bit integer from 32-bit-wide $\overline{\text{STRB0}}$ memory and loading it into R0 (see Figure 4–14). Next, the FLOAT instruction converts the integer in R0 to a 40-bit floating-point number and loads it into R1. Finally, the STF instruction truncates the 40-bit contents of R1 to 32 bits and stores it in the 32-bit-wide $\overline{\text{IOSTRB}}$ memory. The data is not modified as it passes through the memory interface.

The program controlling the data conversion in this example is stored in the 32-bit-wide memory bank mapped to $\overline{\text{STRB1}}$. Program fetch cycles do not reference the strobe control register to determine the width of the program memory. Instead, the memory interface checks the state of the PRGW pin to determine the memory width. Because the program memory is 32 bits wide, the PRGW pin should be grounded, effectively directing the memory interface to fetch instructions in one bus cycle per instruction (32 bits at a time).

Table 4–5. Program Fetch From 32-Bit $\overline{\text{STRB1}}$ Memory

Data Access	Strobe	Data Size	Memory Width
Input Data	$\overline{\text{STRB0}}$	32	32
Output Data	$\overline{\text{STRB1}}$	32	32
Program	$\overline{\text{IOSTRB}}$	32	32

Figure 4–14. Program Fetch From 32-Bit $\overline{\text{STRB1}}$ Memory



4.6.2 Logical Versus Physical Address

The ’C32 is a 32-bit processor. Its instruction set operates on 32-bit registers; the CPU alone does not read 8- or 16-bit data or data transfers. When a ’C32 instruction writes to a physical address, it sends all 32 bits of data to the memory interface unit through an internal bus. It is only in the memory interface that the internal 32-bit data can assume 8-bit or 16-bit form, provided that the address is in the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ range of the memory map. The data size field of the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ control register determines the actual size of the data portion that is placed on the external memory bus of the ’C32. Likewise, when a ’C32 instruction reads a portion of data from external memory, the memory interface always converts it to 32 bits as it enters the chip. What happens to the external data as it goes through the memory interface on the way to the CPU depends on the contents of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers. Again, only the data whose address falls within the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ range of the memory map can be manipulated inside the memory interface unit.

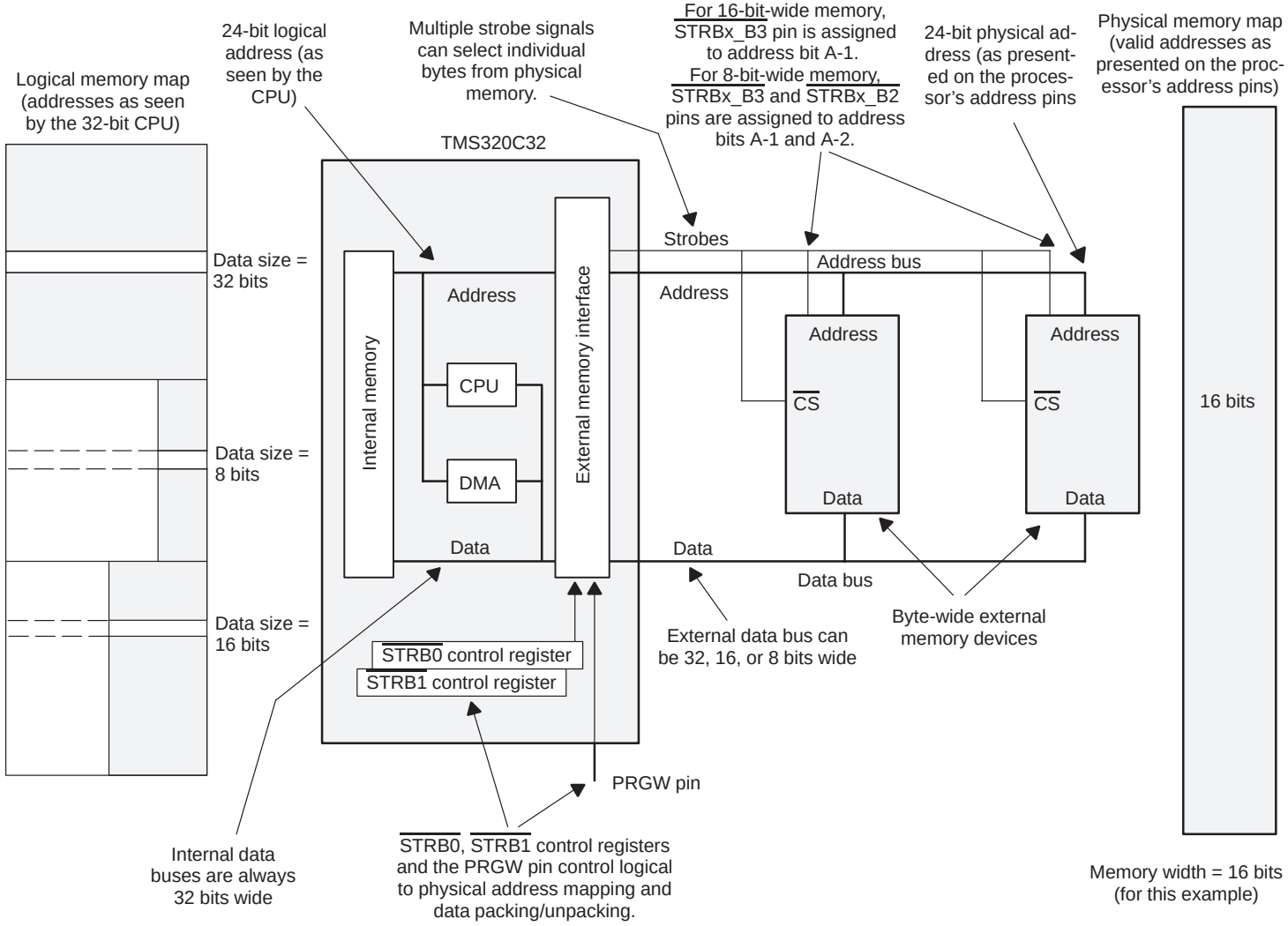
Throughout this document, the term *logical* address applies to a memory location that is referenced by ’C32 instructions; the logical address is a part of the processor’s logical memory map. The *physical* address refers to the address that appears at the ’C32 address pins. The valid ranges of the logical memory map that the program instructions can reference are determined by:

- The external memory available in the system
- The manner in which the external memory address pins are matched with the ’C32 address pins (which depends on physical memory width)
- The contents of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ registers (which define physical memory width and the data size)

The logical memory map shown in Figure 4–15 always contains 32-bit data as far as the CPU is concerned. It is only when the data passes through the memory-interface block that the data size can actually change to 8 or 16 bits, as directed by the appropriate strobe control register. For example, when the processor reads a byte (eight bits) from external memory, the 8-bit data is sign-extended or padded with 0s as it passes through the memory interface so that it becomes 32-bit data inside the ’C32. Likewise, when the processor writes the contents of a 32-bit register to 16-bit-wide external memory, the internal 32-bit data is truncated to 16 bits as it passes through the memory interface. The dashed lines inside the logical memory map in Figure 4–15 show the internal 32-bit representation of the external data that has a physical size of 8 or 16 bits.

Figure 4–15 explains logical/physical addresses and other terms related to the ’C32 memory interface.

Figure 4–15. Description of Terms Involved In TMS320C32 Memory Interface



4.6.3 32-Bit Memory Configuration Design Examples

The following sections describe examples of interfacing the ’C32 to 32-bit-wide external memory from both the hardware and software-addressing viewpoints.

4.6.3.1 32-Bit Memory Address Translation for Data Size = Memory Width

When both data size and memory width are 32 bits, the $\overline{\text{STRB0}}$ memory interface behaves like the $\overline{\text{IOSTRB}}$ memory interface. The only difference between the two is the number of strobe lines connected to the respective memory banks: four for $\overline{\text{STRB0}}$ and one for $\overline{\text{IOSTRB}}$.

Figure 4–16 is a schematic diagram of a 32-bit interface consisting of two memory banks, each controlled by a separate strobe. The four signal lines of $\overline{\text{STRB0}}$ are assigned to the chip-select pins of four 32K $\times$ 8 15-ns SRAMs. The single $\overline{\text{IOSTRB}}$ signal line is connected to the chip-enable pins of four 32K $\times$ 8 30-ns EPROMs. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate with zero wait states and the 30-ns EPROMs require one wait state. (Software wait states can be programmed in the strobe control registers.)

The hardware memory configuration is depicted in Figure 4–16. Figure 4–17 illustrates the programmer’s view of the hardware memory configuration. The logical addresses (appearing in program instructions) are represented in the context of the entire memory map to identify the respective strobes. The physical addresses are the values that actually appear at the pins of the processor. Since $\overline{\text{IOSTRB}}$ operates exclusively on 32-bit data types, the memory interface does not modify the address going in and out of the CPU; the logical and physical addresses are identical. In this example, $\overline{\text{STRB0}}$ also operates on 32-bit data since the memory width field of the $\overline{\text{STRB0}}$ control register contains a binary value of 11. Since the $\overline{\text{STRB0}}$ physical memory width is also 32 bits (see data size field in Figure 4–17), there is no need for address translation from the logical address to its physical representation.

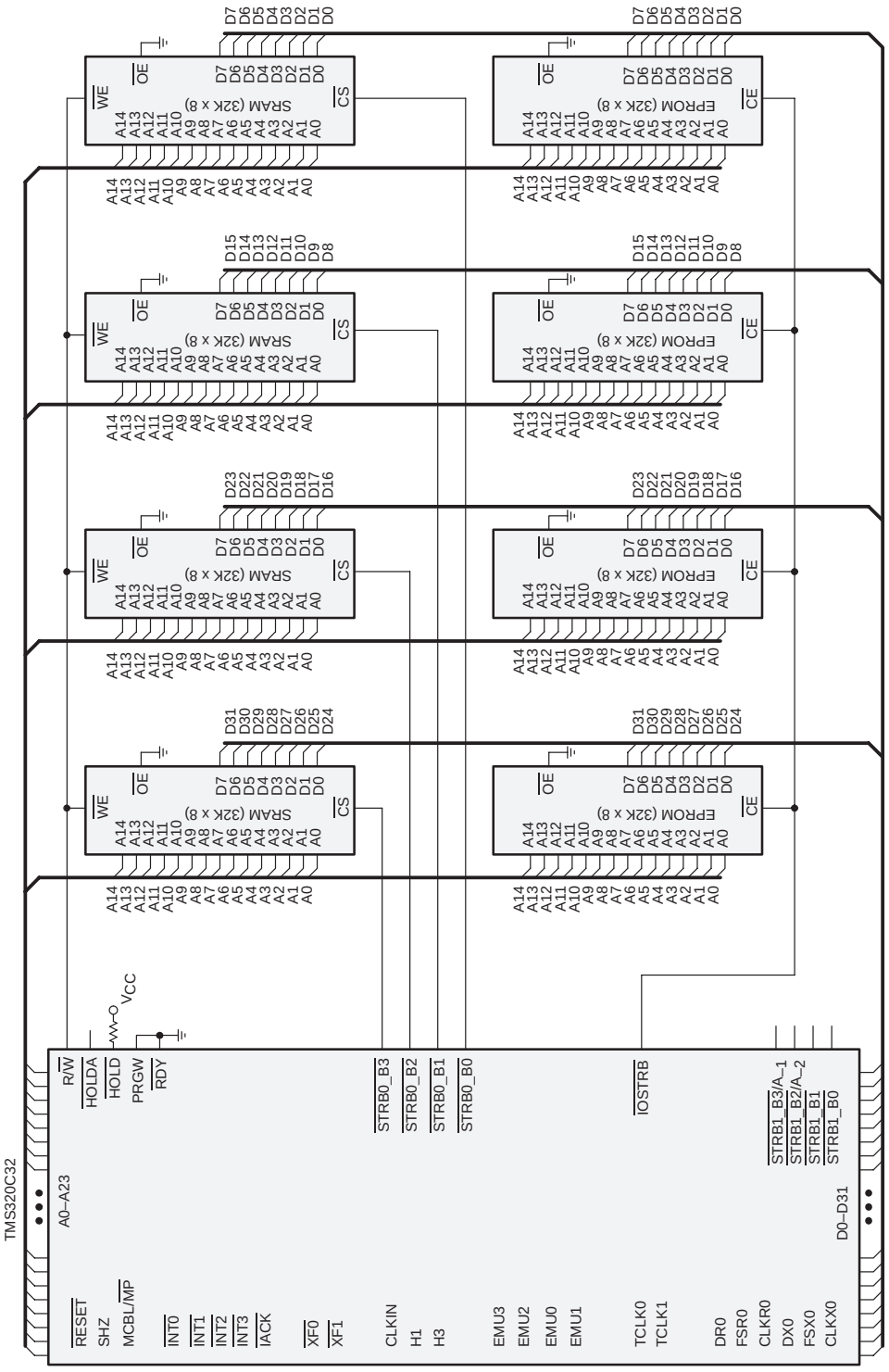
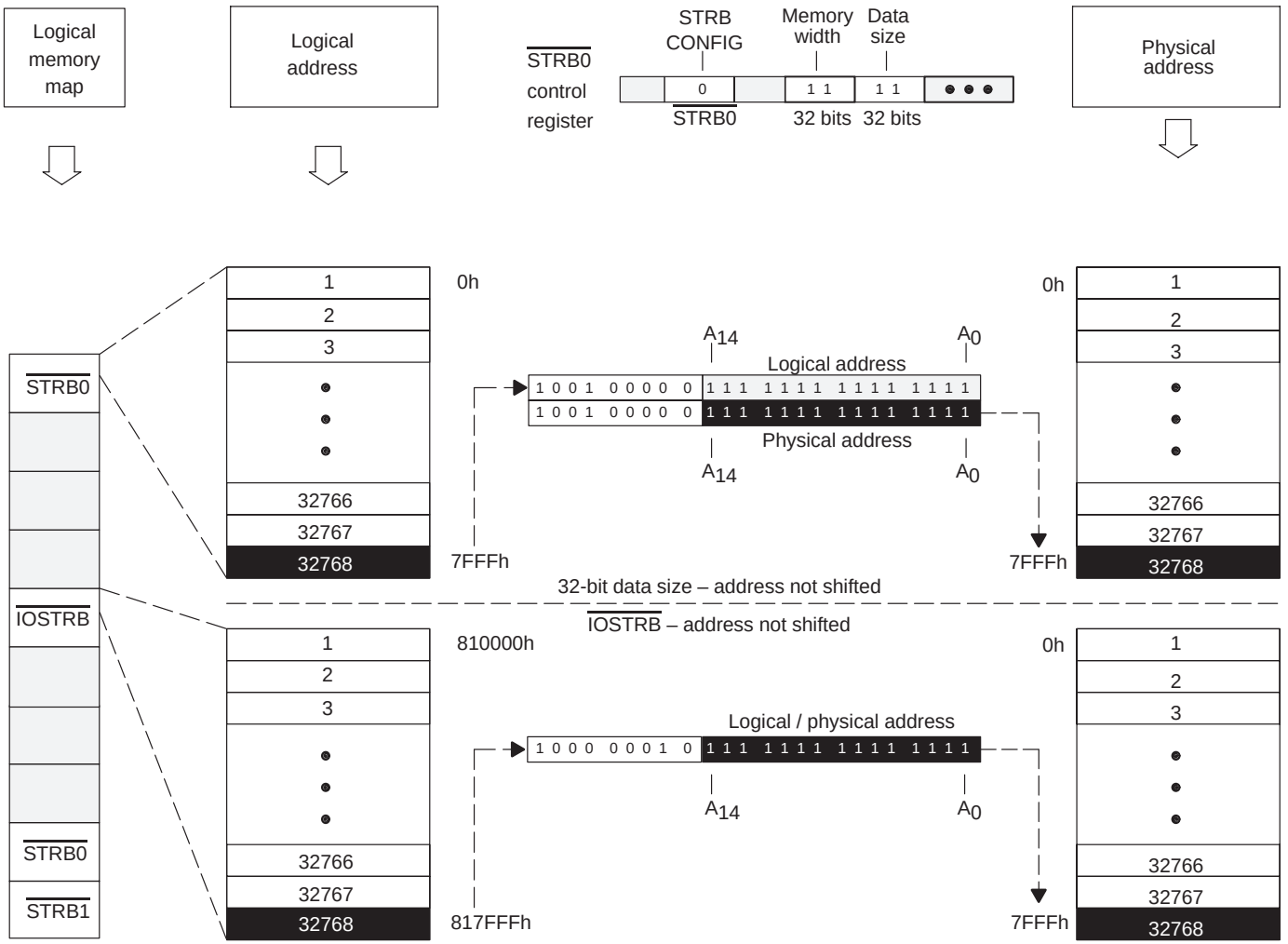


Figure 4–16. 32-Bit Memory Configuration ($\overline{STRB0}$ and $\overline{IOSTRB}$)

Figure 4–17. 32-Bit Memory Configuration ($\overline{STRB0}$ and $\overline{IOSTRB}$)



Note: 32-bit memory address translation: data size = memory width

4.6.3.2 32-Bit Memory Address Translation for Data Size < Memory Width

One memory location can store 2 or 4 data values. Therefore, if the data requires 16 or 8 bits of precision, the effective addressing range of the same physical 32-bit memory is doubled or quadrupled by simply changing the data size field of the appropriate strobe control register before the transfers begin. The logical-to-physical address translation involves a 2-bit address shift if the data size is 8 bits and a 1-bit shift if the data size is 16 bits. The memory interface automatically performs address shifts and the activation of selected external memory bytes with appropriate strobe control lines (as directed by the strobe control registers).

Figure 4–18 is the schematic diagram of a 32-bit interface consisting of two memory banks, each controlled by a separate strobe. The four signal lines of $\overline{\text{STRB0}}$ are assigned to the chip-select pins of four 32K × 8 15-ns SRAMs, and the four signal lines of $\overline{\text{STRB1}}$ are connected to the chip-enable pins of four 32K × 8 30-ns EPROMs. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate at zero wait states and the 30-ns EPROMs require one wait state. (Software wait states can be programmed in strobe control registers.)

Figure 4–19 illustrates the programmer’s view of the hardware memory configuration depicted in Figure 4–18. The logical addresses (appearing in program instructions) are represented in the context of the entire memory map to identify the respective strobes. In this case, the $\overline{\text{STRB0}}$ memory transfers operate on 16-bit data to and from 32-bit-wide memory, as defined in the $\overline{\text{STRB0}}$ control register. $\overline{\text{STRB1}}$ accesses 8-bit data to and from 32-bit-wide memory, as defined by the $\overline{\text{STRB1}}$ control register. Since two 16-bit data types can fit in a single 32-bit-wide memory location referenced by a single physical address, a mechanism is needed to distinguish between the 16-bit data portions. This is accomplished by using the least significant bit (LSB) of the logical address to activate a different pair of the four $\overline{\text{STRB0}}$ signal lines for each access, leaving the second LSB of the logical address to become the LSB of the physical address and effectively shifting the logical address by one bit. Similarly, $\overline{\text{STRB1}}$ 8-bit data transfers to the 32-bit-wide external memory cause the address to be shifted by two bits, because the two LSBs of the logical address are used to select one out of four bytes sharing the same physical 32-bit memory location.

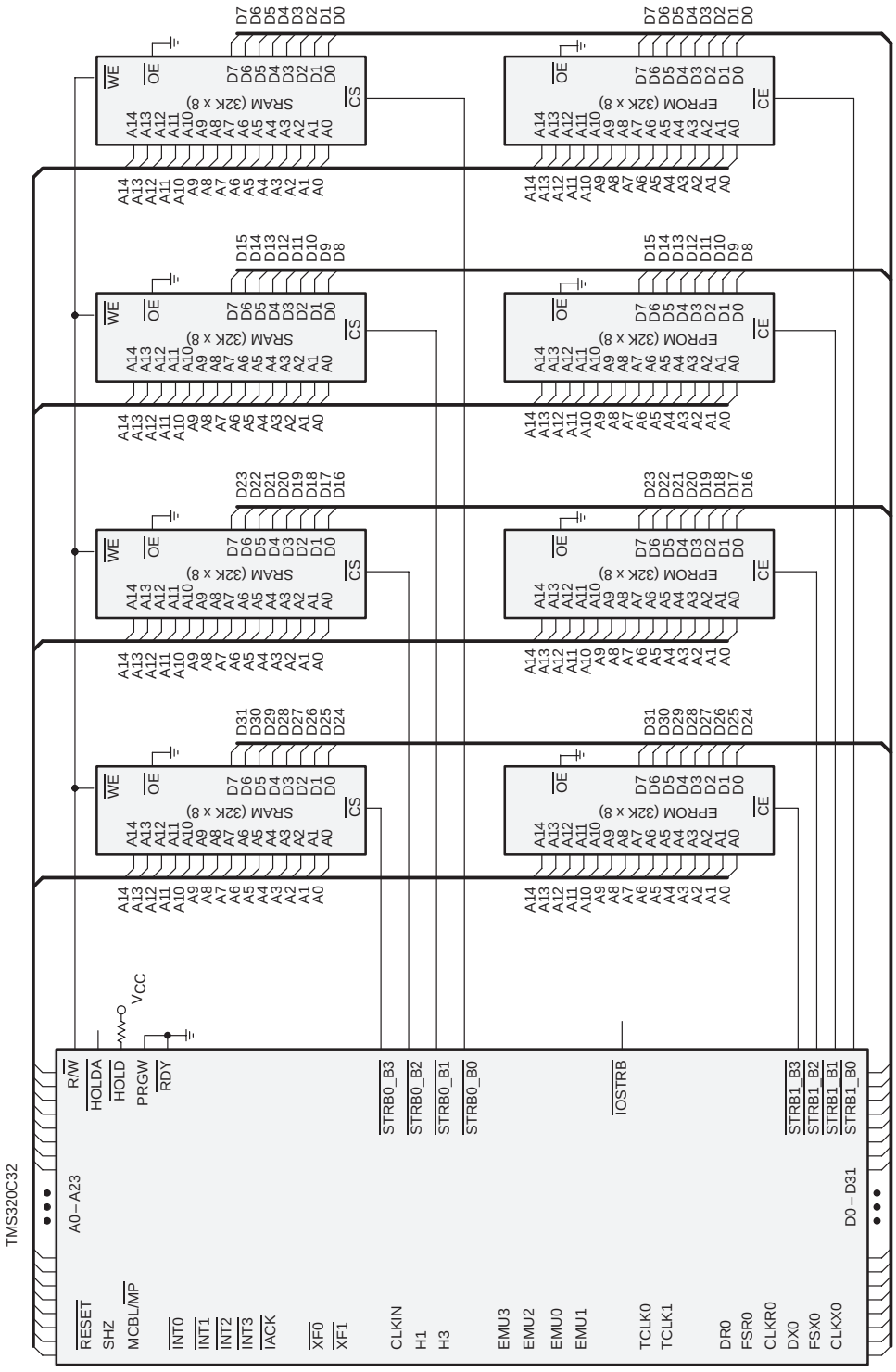
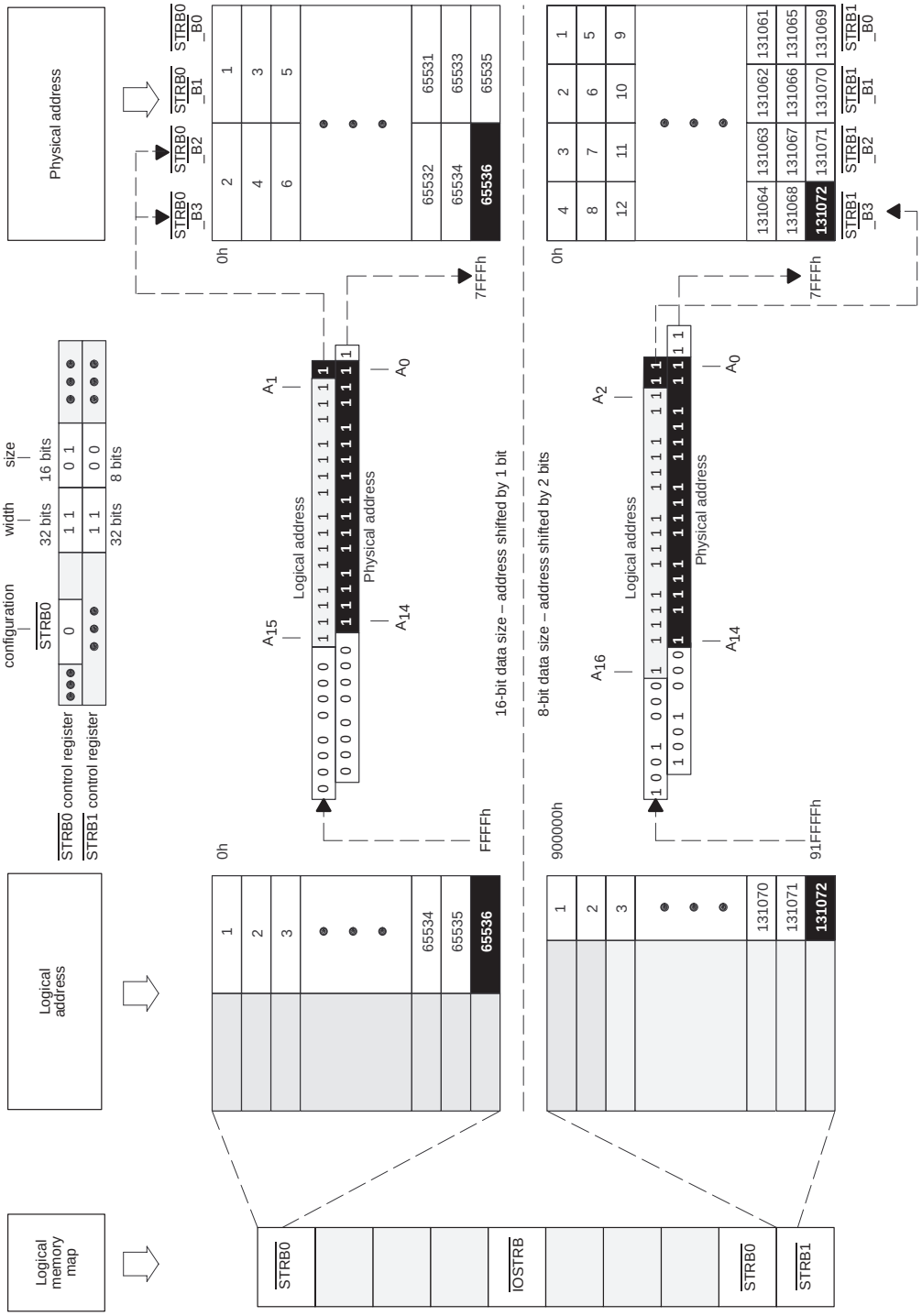


Figure 4–18. 32-Bit Memory Configuration ($\overline{STRB0}$ and $\overline{STRB1}$)

Figure 4–19. 32-Bit Memory Address Translation: Data Size < Memory Width



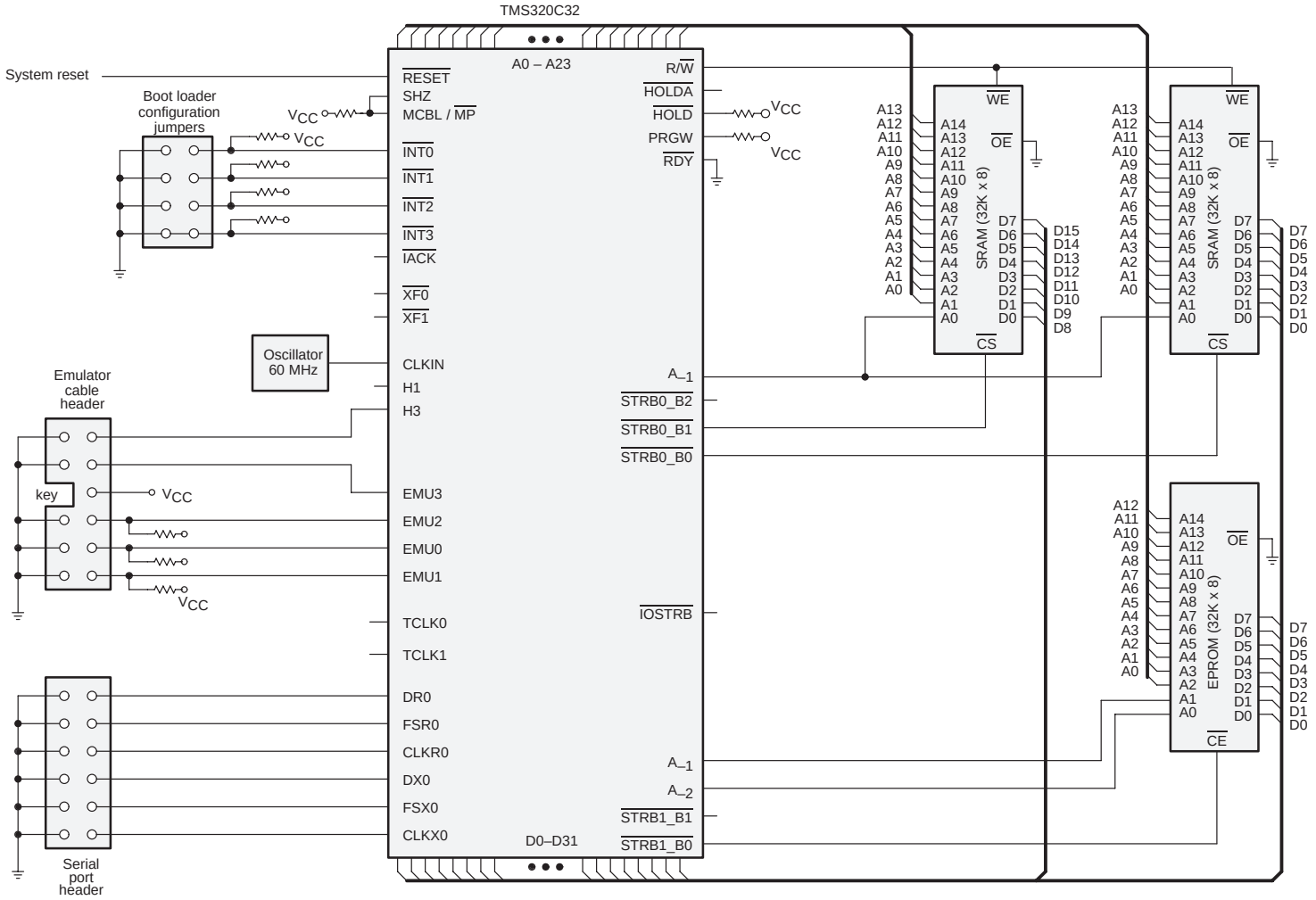
4.6.4 16-Bit and 8-Bit Memory Configuration Design Examples

This section describes how to interface the ’C32 to both 8- and 16-bit-wide external memories in the same design from both the hardware and software-addressing perspectives.

Figure 4–20 contains a schematic diagram of the external memory interface consisting of two banks, each controlled by a separate strobe. Two of four $\overline{\text{STRB0}}$ signal lines are assigned to the chip-select pins of two 32K $\times$ 8 15-ns SRAMs; one of four $\overline{\text{STRB1}}$ signals is connected to a chip-enable pin of one 32K $\times$ 8 30-ns EPROM. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate at zero wait states and the 30-ns EPROMs require one wait state. (Software wait states can be programmed in strobe control registers.) Any time the external memory is less than 32 bits wide, some of the strobe pins switch functions and become additional address pins. For 16-bit-wide memory, $\overline{\text{STRB0\_B3}}$ becomes A_{-1}; for 8-bit-wide memory, $\overline{\text{STRB1\_B3}}$ and $\overline{\text{STRB1\_B2}}$ become A_{-1} and A_{-2}, respectively. This is the only external change that differentiates the 32-bit-wide memory interface from the 16- and 8-bit-wide memory interfaces. This feature can be considered transparent to the software programmer, except that the programmer must configure the strobe control registers appropriately. The memory interface automatically drives the additional address lines with correct values, depending on the size of the data being transferred.

The following three sections illustrate how the physical addresses are derived from the logical addresses when the data size is equal to, greater than, and less than the width of the physical memory. Though address translation is completely automatic, these cases provide insight into the range of physical addresses actually affected during transfer of 32-, 16-, and 8-bit data.

Figure 4–20. 16-Bit and 8-Bit Memory Configuration: A Complete Minimum Design

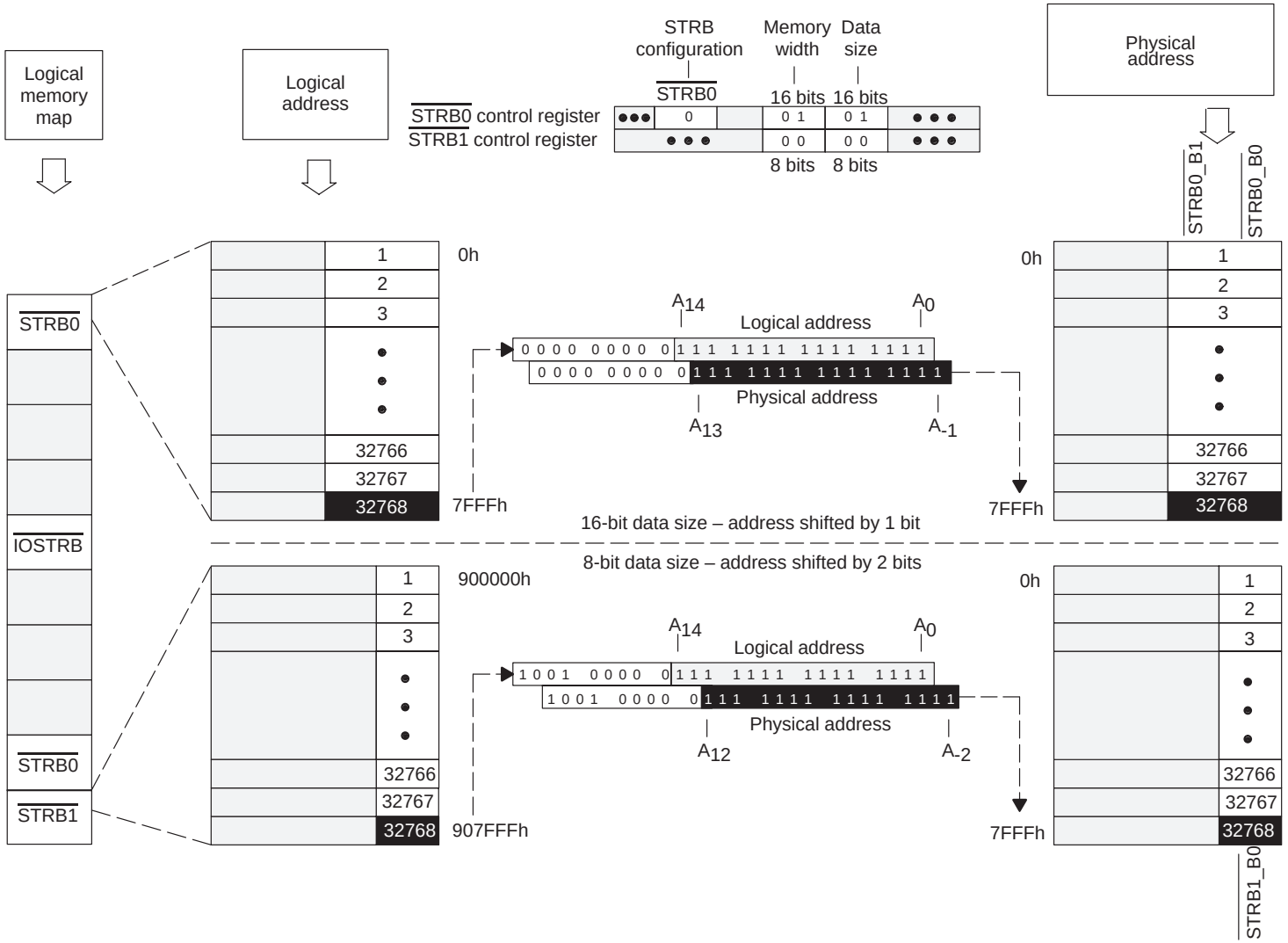


Note: The EPROM is connected for data access (shifted address) and not for boot table access. This system is booted from the serial port (see INT3 signal).

4.6.4.1 16-Bit and 8-Bit Memory Address Translation for Data Size = Memory Width

As shown in Figure 4–21, when the external memory width matches the size of data being transferred, the physical address also matches the logical address with one exception: the physical address is shifted relative to the logical address by one bit for 16-bit transfers and by two bits for 8-bit transfers. This means that the address bit that would normally be expected on pin A0 actually appears on pin A_{-1} or A_{-2}. As Figure 4–21 shows, there is one-to-one correspondence between logical data and its counterpart in physical memory.

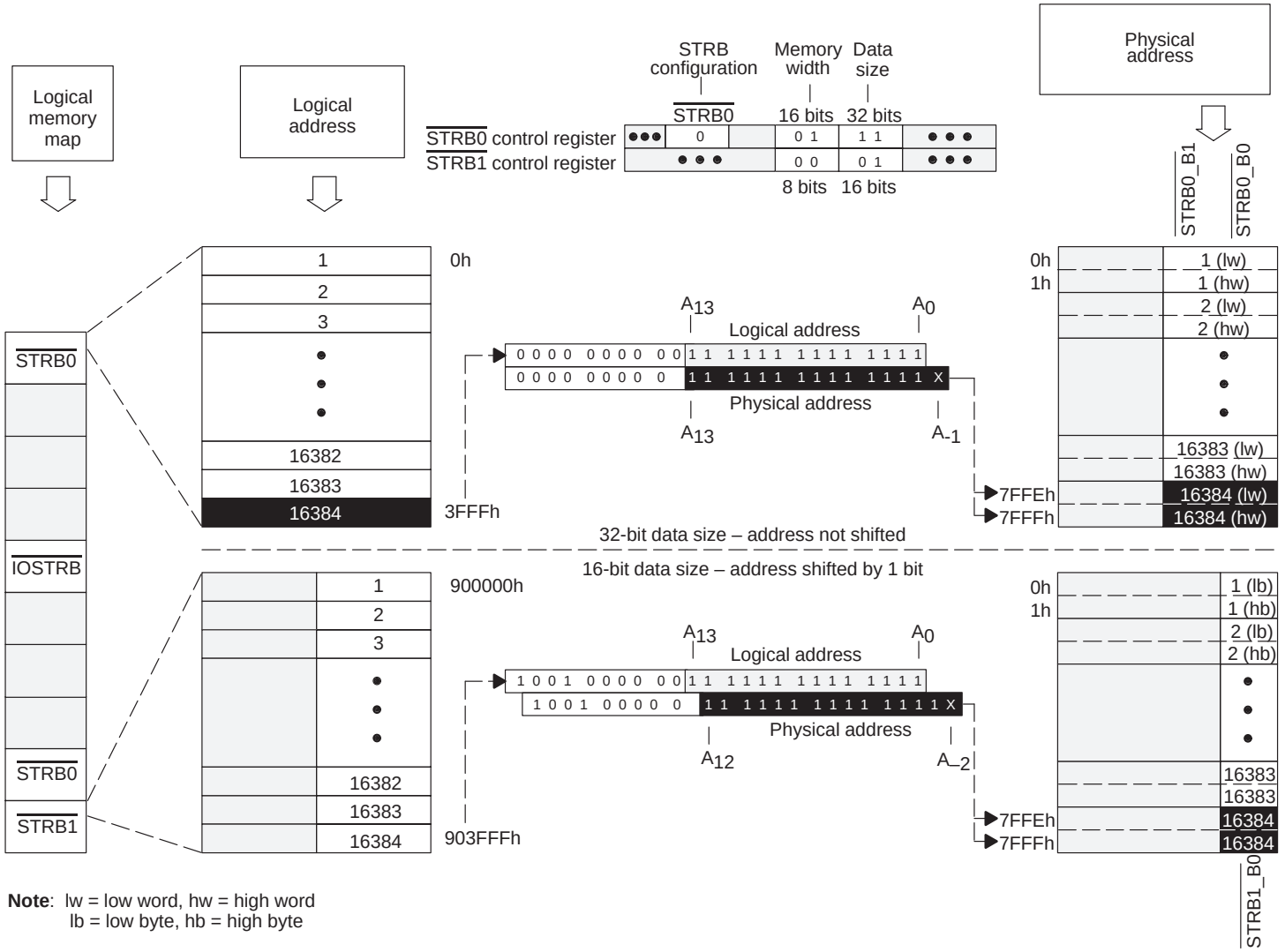
Figure 4–21. 16-Bit and 8-Bit Memory Address Translation: Data Size = Memory Width



4.6.4.2 16-Bit and 8-Bit Memory Address Translation for Data Size > Memory Width

Figure 4–22 depicts what happens when data is transferred that is larger than the physical memory in which it is to reside. As shown by the contents of the strobe control registers, $\overline{\text{STRB0}}$ controls transfers of 32-bit data to and from 16-bit-wide physical memory and $\overline{\text{STRB1}}$ controls transfers of 16-bit data to and from byte-wide memory. When an instruction stores 32-bit data to logical address 0h, the memory interface must perform two write cycles to 16-bit-wide external memory. These two write cycles involve two consecutive addresses, 0h and 1h. A 16-bit portion of data logically referenced with a single address actually requires two physical addresses to be stored in 8-bit-wide physical memory (as is the case with the $\overline{\text{STRB1}}$ transfer shown at the bottom of Figure 4–22). To implement these extra bus cycles, the memory interface appends an extra address bit to the least significant end of both addresses. As in section 4.6.4.1, the LSBs of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ addresses appear at pins A_{-1} and A_{-2}, respectively, because they represent 16- and 8-bit-wide memories.

Figure 4–22. 16-Bit and 8-Bit Memory Address Translation: Data Size > Memory Width



Note: lw = low word, hw = high word
lb = low byte, hb = high byte

4.6.4.3 16-Bit and 8-Bit Memory Address Translation for Data Size < Memory Width

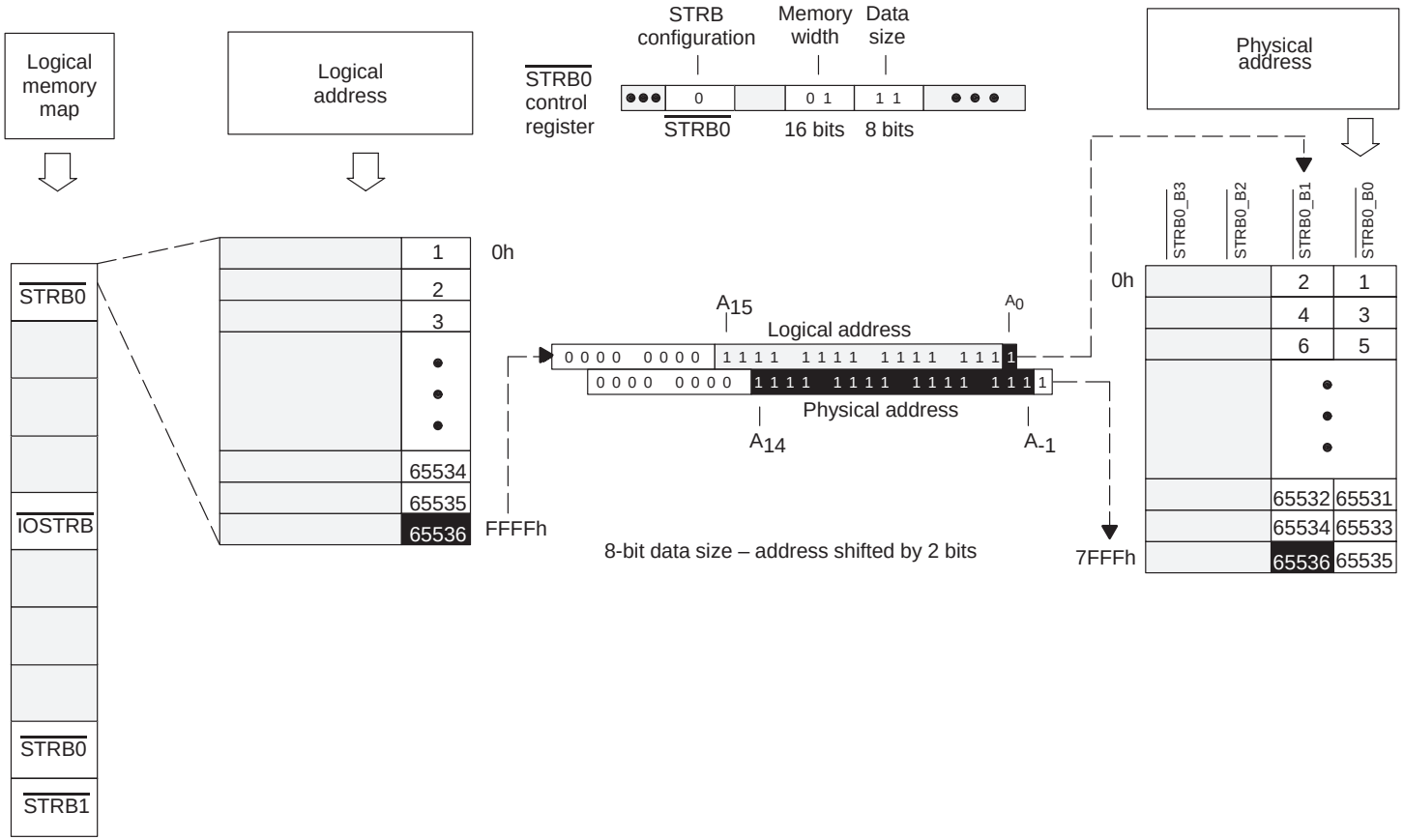
The example in Figure 4–23 is, in a way, an inverse of that in Figure 4–22. The 8-bit data is transferred to and from 16-bit-wide external memory. To put this example in perspective, assume that the data transfer is triggered by the following ’C32 instruction: STI R0,@7FFFh. While in R0, the data is sized at 32 bits, but when it arrives at the memory interface, the $\overline{\text{STRB0}}$ control register data size field indicates 8-bit-wide data. So, the 32-bit data is truncated to 8 bits. The now byte-sized data is transferred to address 7FFFh of the 16-bit-wide external memory. In this case, the LSB of the logical address (as referenced by the instruction) is actually rerouted to control one of the two $\overline{\text{STRB0}}$ lines assigned to the 16-bit physical memory. If the LSB is 1 (as in this case), $\overline{\text{STRB0\_B1}}$ is asserted during the write cycle. If the LSB is 0, $\overline{\text{STRB0\_B0}}$ is asserted during the write cycle. The remaining bits of the original logical address are placed on the external address bus starting at pin A_{-1} (because the memory width is 16 bits).

4.6.4.4 Design Considerations

While designing the external memory interface to the ’C32, a hardware engineer must remember to match address pin A_{-1} with the A0 pin of a 16-bit-wide memory, or to match the A_{-2} address pin with the A0 pin of a byte-wide memory. If the external memory is 32 bits wide, the pins are not shifted relative to each other and, therefore, match perfectly at A0.

When writing code for the ’C32, the programmer does not have to be concerned about the structure of the physical memory. The programmer must simply be aware of the logical memory map and the configuration of the two strobe control registers. The ’C32 memory interface automatically performs all of the address translation tasks and byte packing/unpacking necessary to match variable-size data with physical memories of different widths; they are controlled by the data size and memory width fields of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers.

Figure 4–23. 16-Bit and 8-Bit Memory Address Translation: Data Size < Memory Width



4.6.5 One Bank /Two Strobes (32-Bit-Wide Memory) Design Examples

This section describes how to use two strobes in interfacing the ’C32 to a single physical bank of memory. Such configuration enables the access to 32-bit programs and to two differently sized portions of data out of the same bank of memory with no speed penalty. This feature is implemented by internally ANDing $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ and outputting the combined strobes on $\overline{\text{STRB0}}$ (a total of four lines). The one bank/two strobes memory configuration is useful in systems in which, for example, the program requiring 32-bit instruction words for maximum execution speed operates on data that needs only 16 bits of precision (see Figure 4–27 on page 4-56).

Figure 4–24 is the schematic diagram of a 32-bit-wide external memory configuration arranged as one bank with two separate logical control strobes sharing the same $\overline{\text{STRB0}}$ physical signal lines. The four $\overline{\text{STRB0}}$ signals are assigned to the chip-select pins of four 32K $\times$ 8 15-ns SRAMs, one signal per chip. For the 60-MHz version of the ’C32, the 15-ns SRAMs operate at zero wait states. (For slower devices, additional software wait states can be programmed in the appropriate fields of the strobe control registers.) Because the total memory width is 32 bits, there is no mismatch between the processor’s and the memory’s address pins. Therefore, the ’C32 pin A0 is matched with memory pin A0, A1 is matched with A1, and so on. As mentioned earlier, both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ signals appear together on the four $\overline{\text{STRB0}}$ control pins. This behavior is selected by setting the strobe configuration bit of the $\overline{\text{STRB0}}$ control register to 1 (see Figure 4–24). Since both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to different ranges of the logical memory map, the strobe that actually appears on the physical $\overline{\text{STRB0}}$ pins depends on the internal address of the data/program being accessed. The two strobes effectively split the physical memory into two, with the high memory address bit selecting either the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ address space. For example, if all program instructions are fetched from logical addresses 880000h–881000h and all data reads/writes are confined between 980000h and 981000h, the program fetches are associated with $\overline{\text{STRB0}}$ and all data accesses are driven by $\overline{\text{STRB1}}$ (see Figure 4–10 on page 4-23 for strobe/memory mapping). Since the behavior of each strobe is determined by a different control register, the program fetches and data reads/writes, in each case, can vary in the number of $\overline{\text{STRB0}}$ lines that are simultaneously driven and in the number of bus cycles required per access. This is shown on the following pages.

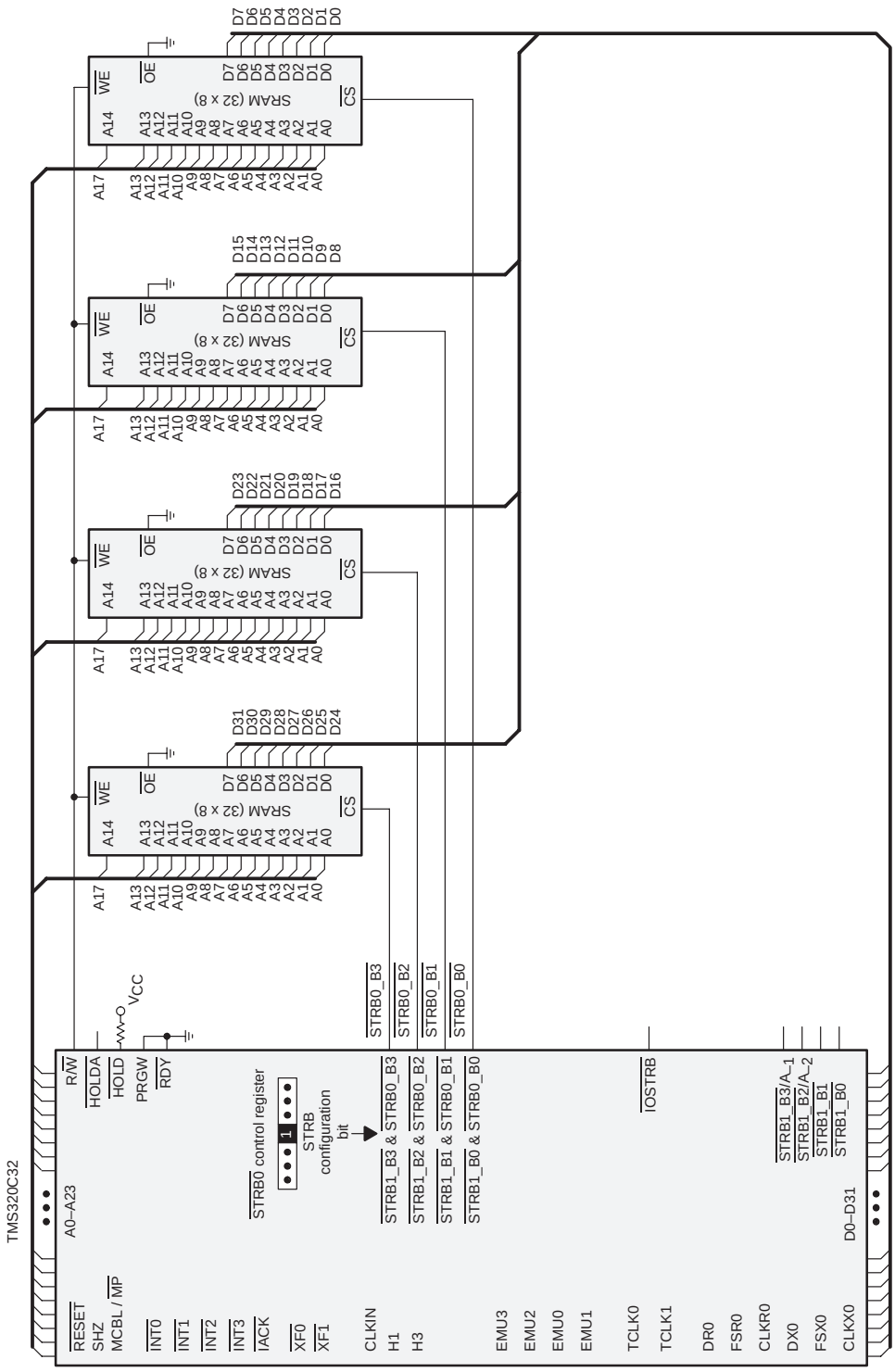


Figure 4–24. One Bank/Two Strobes Memory Configuration: Memory Width = 32 Bits

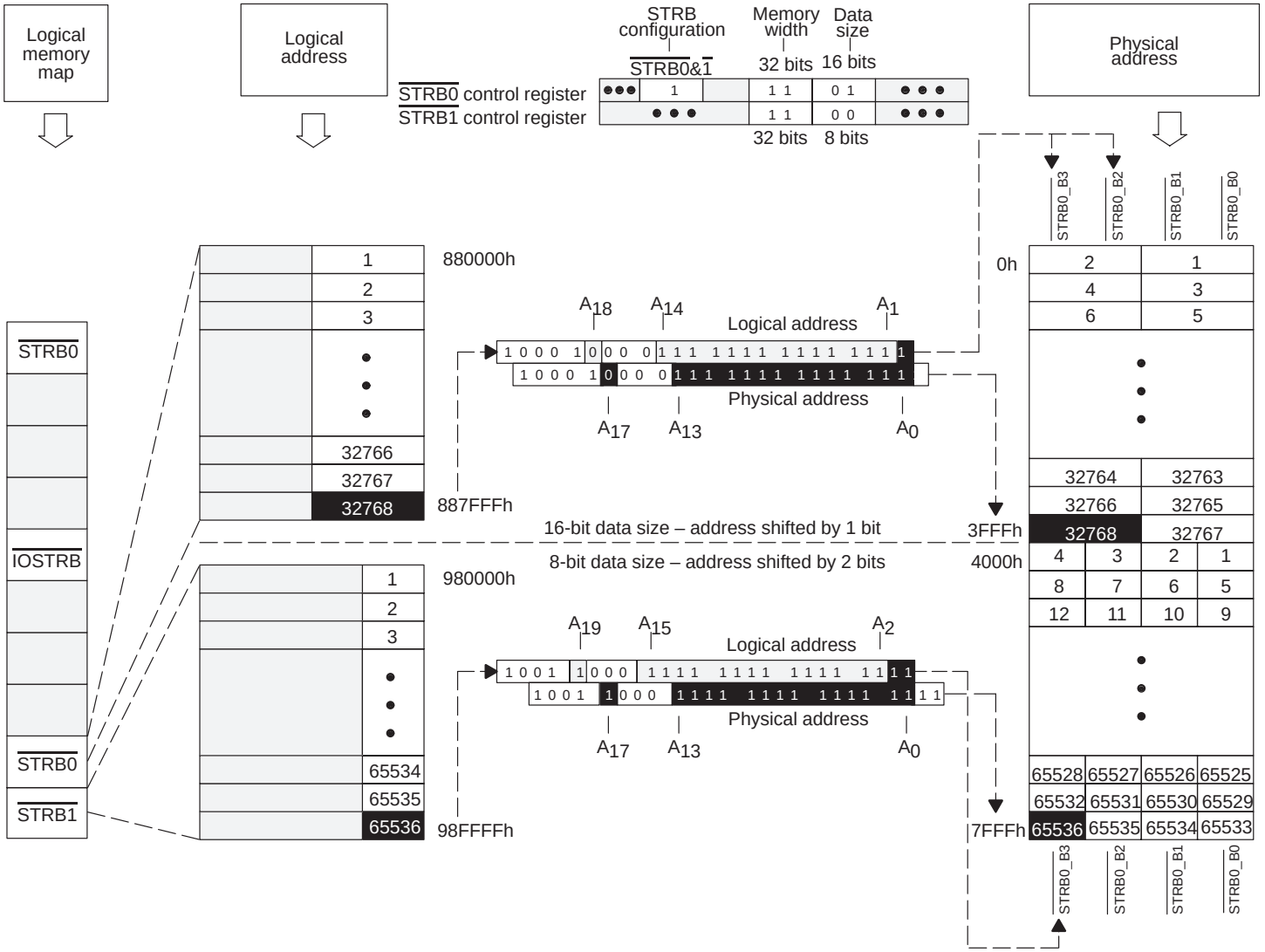
4.6.5.1 One Bank/Two Strobes Address Translation for Data Size = 16 and 8 Bits

Figure 4–25 illustrates how a single physical block of memory can be split into two separate logical halves, one with 16-bit data and the other with 8-bit data. The access to each half is controlled by a separate strobe control register with corresponding memory width and data size fields. Another $\overline{\text{STRB0}}$ control register field, STRB CONFIG (strobe configuration), is set to 1 to indicate that both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to the same set of four $\overline{\text{STRB0}}$ pins. The high memory address pin (in this case, A14) selects between the two halves of the memory. For this example, the ’C32 address pin A17 drives the memory pin A14.

The state of the A17 bit of the physical address is derived from the logical address (logical as seen by the instruction). The state of the A17 bit also depends on the logical/physical address shift as determined by the size of the program/data that is being accessed. In this case, the logical $\overline{\text{STRB0}}$ address range drives the physical address bit A17 to 0 (after accounting for a 1-bit address shift due to the 16-bit width of the data). Similarly, the logical $\overline{\text{STRB1}}$ range drives the physical address bit A17 to 1 (after accounting for a 2-bit address shift due to the 8-bit width of the data). The logical $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges selected to drive the physical address pin A17 to 0 and 1, respectively, must still conform to the logical memory map that assigns fixed blocks of addresses to different strobe spaces.

An STI R0,*AR0 instruction (with AR0 = 887FFFh) results in a $\overline{\text{STRB0}}$ data access (data size = 16 bits) driving the $\overline{\text{STRB0\_B2}}$ and $\overline{\text{STRB0\_B3}}$ control pins to write the contents of the 32-bit register R0 into a 16-bit data location in the lower half of the external memory addressed by 3FFFh. Similarly, an LDI *AR1,R1 instruction (with AR1 = 98FFFFh) results in a $\overline{\text{STRB1}}$ data access (data size = 8 bits) driving the $\overline{\text{STRB0\_B3}}$ control pin (STRB CONFIG = 1) to read the contents of an 8-bit data location in the upper half of the external memory addressed by 7FFFh to the 32-bit R1 register. The ’C32 automatically performs all address translation; the programmer merely monitors the logical memory map and the two strobe control registers.

Figure 4–25. One Bank/Two Strobes Address Translation: Data Size = 16 and 8 Bits



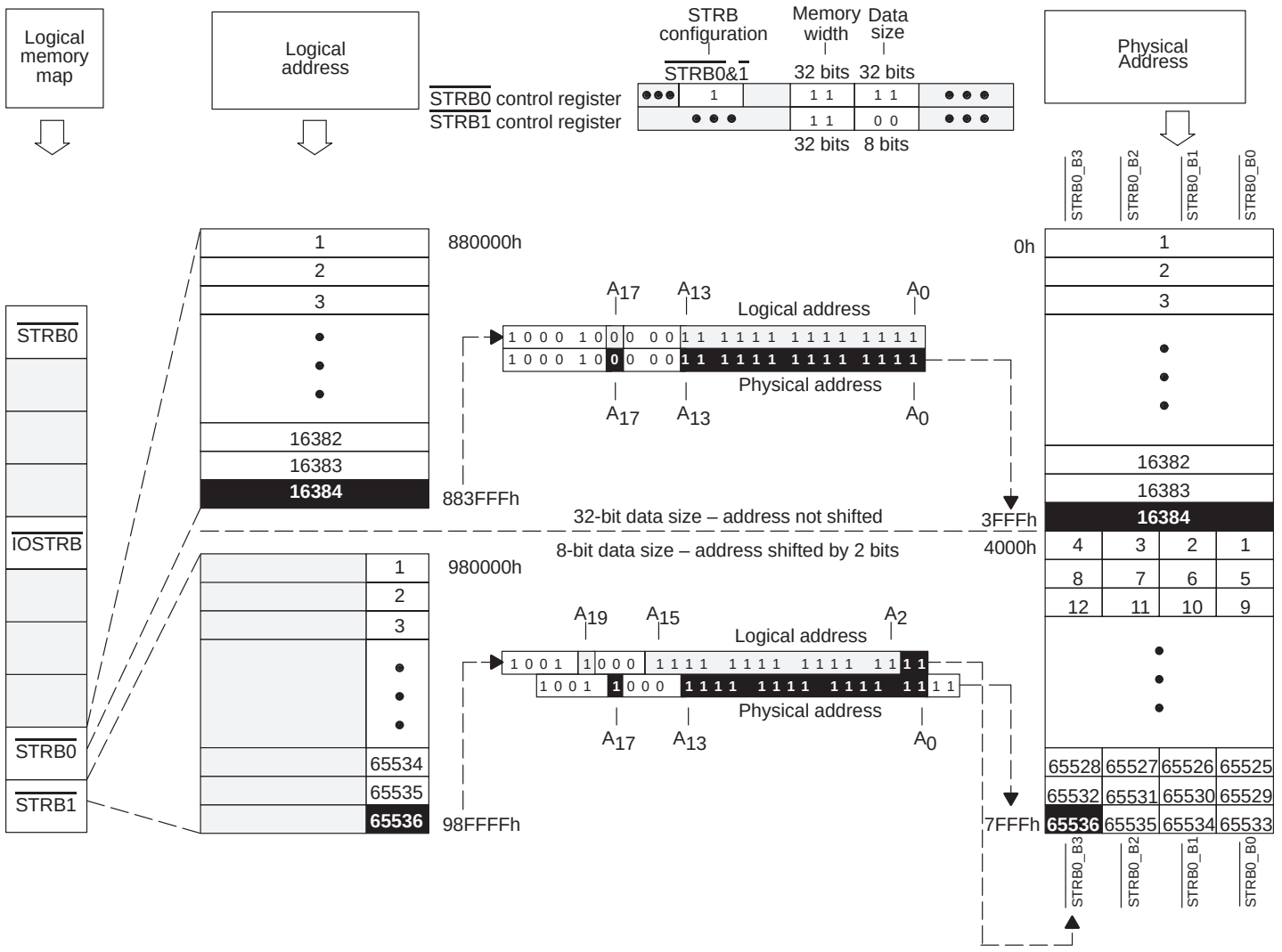
4.6.5.2 One Bank/Two Strobes Address Translation for Data Size = 32 and 8 Bits

Figure 4–26 illustrates how a single physical block of memory can be split into two separate logical halves, one with 32-bit data and the other with 8-bit data. The access to each half is controlled by a separate strobe control register with corresponding memory width and data size fields. Another $\overline{\text{STRB0}}$ control register field, STRB CONFIG, is set to 1 to indicate that both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to the same set of four $\overline{\text{STRB0}}$ pins. The high memory address pin (in this case, A14) selects between the two halves of the memory. For this example, the ’C32 address pin A17 drives the memory pin A14.

The state of the A17 bit of the physical address is derived from the logical address (logical as seen by the instruction). The state of the A17 bit also depends on the logical/physical address shift as determined by the size of the program/data that is being accessed. In this case, the logical $\overline{\text{STRB0}}$ address range drives the physical address bit A17 to 0. Similarly, the logical $\overline{\text{STRB1}}$ range drives the physical address bit A17 to 1 (after accounting for a 2-bit address shift due to the 8-bit width of the data). Additionally, the logical $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges that drive the physical address pin A17 to 0 and 1, respectively, must still conform to the logical memory map that assigns fixed blocks of addresses to different strobe spaces.

An STI R0,*AR0 instruction (with AR0 = 883FFFh) results in a $\overline{\text{STRB0}}$ data access (data size = 32 bits) driving the $\overline{\text{STRB0\_B0}}$, $\overline{\text{STRB0\_B1}}$, $\overline{\text{STRB0\_B2}}$, and $\overline{\text{STRB0\_B3}}$ control pins to write the contents of the 32-bit register R0 into a 32-bit data location in the lower half of the external memory addressed by 3FFFh. Similarly, an LDI *AR1,R1 instruction (with AR1 = 98FFFFh) results in a $\overline{\text{STRB1}}$ data access (data size = 8 bits) driving the $\overline{\text{STRB0\_B3}}$ control pin (because STRB CONFIG = 1) to read the contents of an 8-bit data location in the upper half of the external memory addressed by 7FFFh to the 32-bit R1 register. The ’C32 automatically performs all address translation; the programmer merely monitors the logical memory map and the two strobe control registers.

Figure 4–26. One Bank/Two Strobes Address Translation: Data Size = 32 and 8 Bits



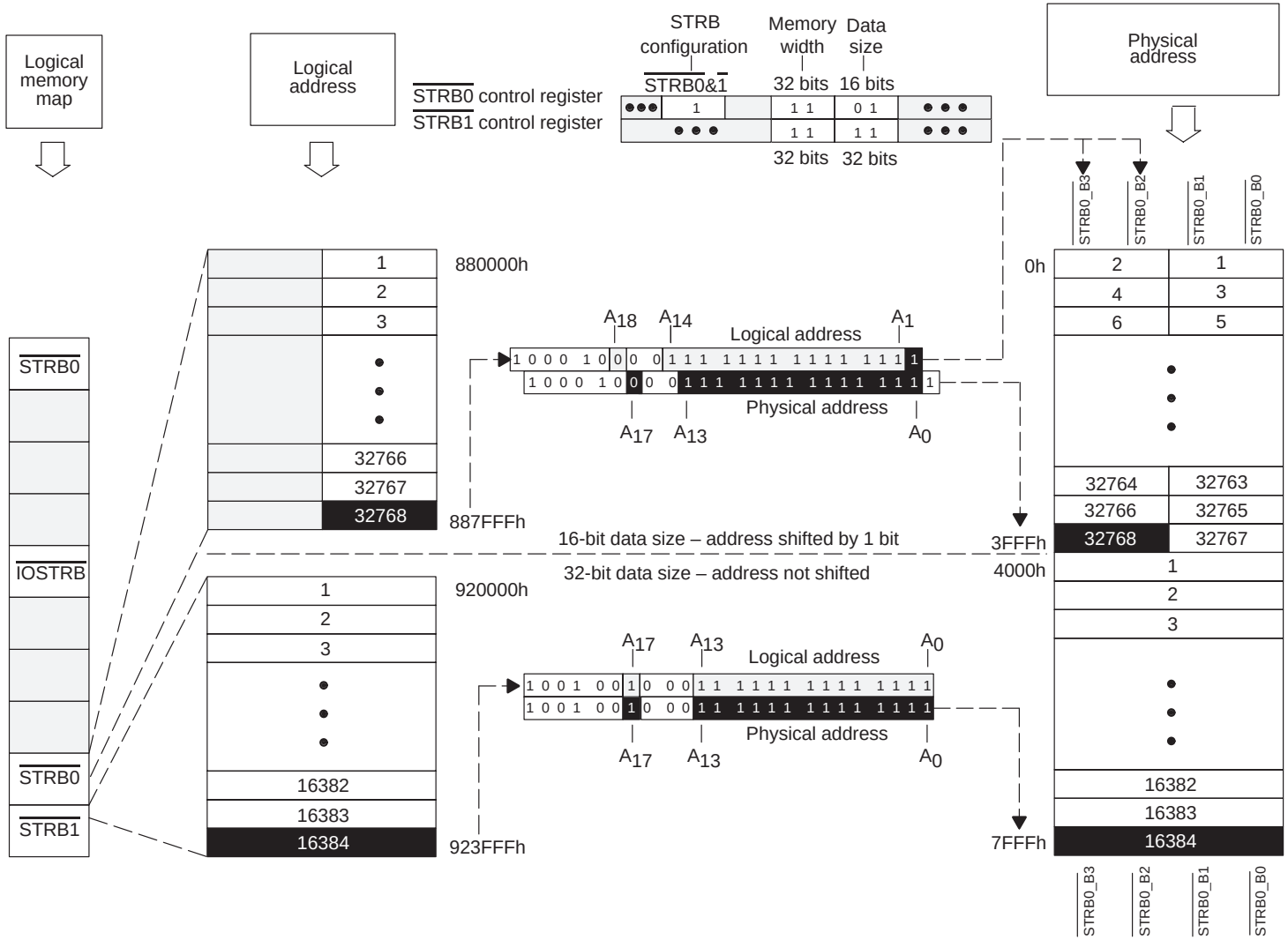
4.6.5.3 One Bank/Two Strobes Address Translation for Data Size = 16 and 32 Bits

Figure 4–27 illustrates how a single physical block of memory can be split into two separate logical halves, one with 16-bit data and the other with 32-bit data. The access to each half is controlled by a separate strobe control register with corresponding memory width and data size fields. Another $\overline{\text{STRB0}}$ control register field, STRB CONFIG, is set to 1 to indicate that both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ are mapped to the same set of four $\overline{\text{STRB0}}$ pins. The high memory address pin (in this case, A14) selects between the two halves of the memory. For this example, the ’C32 address pin A17 drives the memory pin A14.

The state of the A17 bit of the physical address is derived from the logical address (logical as seen by the instruction). The state of the A17 bit also depends on the logical/physical address shift as determined by the size of the program/data that is being accessed. In this case, the logical $\overline{\text{STRB0}}$ address range drives the physical address bit A17 to 0 (after accounting for a 1-bit address shift due to the 16-bit width of the data). Similarly, the logical $\overline{\text{STRB1}}$ range drives the physical address bit A17 to 1. The logical $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges that drive the physical address pin A17 to 0 and 1, respectively, must still conform to the logical memory map that assigns fixed blocks of addresses to different strobe spaces.

An STI R0,*AR0 instruction (with AR0 = 887FFFh) results in a $\overline{\text{STRB0}}$ data access (data size = 16 bits) driving the $\overline{\text{STRB0\_B2}}$ and $\overline{\text{STRB0\_B3}}$ control pins to write the contents of the 32-bit register R0 into a 16-bit data location in the lower half of the external memory addressed by 3FFFh. Similarly, an LDI *AR1,R1 instruction (with AR1 = 923FFFh) results in a $\overline{\text{STRB1}}$ data access (data size = 32 bits) driving the $\overline{\text{STRB0\_B0}}$, $\overline{\text{STRB0\_B1}}$, $\overline{\text{STRB0\_B2}}$, and $\overline{\text{STRB0\_B3}}$ control pins (because STRB CONFIG = 1) to read the contents of a 32-bit data location in the upper half of the external memory addressed by 7FFFh to the 32-bit R1 register. The ’C32 automatically performs all address translation; the programmer merely monitors the logical memory map and the two strobe control registers.

Figure 4–27. One Bank/Two Strobes Address Translation: Data Size = 16 and 32 Bits



4.6.5.4 Example Summary

The one bank/two strobes memory interface to the ’C32 supports any combination of data size pairs (16/8, 32/8, and 16/32 bits) with no speed penalty. (The strobe control registers do not have to be reconfigured each time the data size changes.) Likewise, 16-bit external memory can be divided into two halves, each containing data of a different size (8, 16, or 32 bits). The same holds true for 8-bit external memory. All address translation information given in section 4.6.1 through section 4.6.4 also applies to the one bank/two strobes examples.

To configure the external memory for one bank/two strobes access mode, use the following steps:

1) Set the strobe configuration field in the $\overline{\text{STRB0}}$ control register to 1.

2) Set the memory width field in both the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers to reflect the width of the physical memory.

3) Set the data size field in both the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers to reflect the size of the data portions chosen for each strobe.

4) Choose one of the high physical address bits to split the physical memory into two halves.

5) For the two memory halves, choose the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ logical address ranges to drive the chosen bit to 0 and 1, respectively. The chosen $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ address ranges must fit inside the legal $\overline{\text{STRB0}}$/$\overline{\text{STRB1}}$ address spaces, as defined by the memory map.

4.6.6 $\overline{\text{RDY}}$ Signal Generation

The ’C32 uses the $\overline{\text{RDY}}$ pin to determine whether the current bus cycle finishes at the end of the current clock cycle or requires additional clock cycles to complete. Even though the ’C32 can fetch instructions and access data in one clock cycle, a slow memory may need additional clock cycles (wait states) to complete the bus cycle. The $\overline{\text{RDY}}$ signal can be handled in one of three ways:

- The $\overline{\text{RDY}}$ pin can be permanently grounded, indicating to the CPU that the external memory is always ready for the next cycle. This is used where all external memory is fast enough to preclude wait states.
- The wait states can be programmed in software by setting bits in corresponding strobe control registers, if there is only one device per strobe. This method can be used even if there are external devices that require wait states. The $\overline{\text{RDY}}$ pin must be permanently grounded.

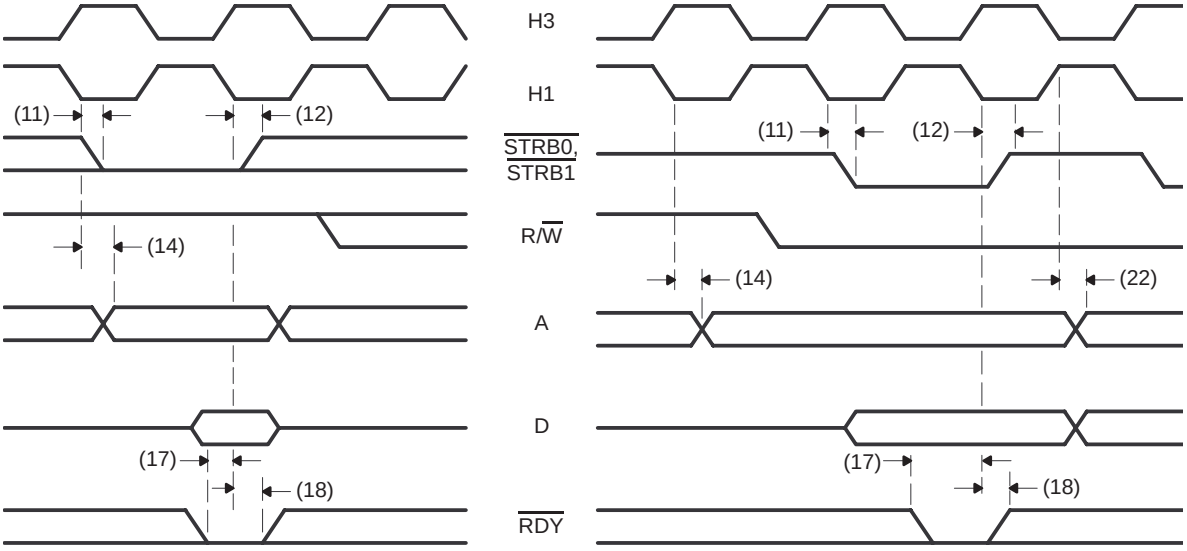
- The active generation of the $\overline{\text{RDY}}$ signal is required only if a single strobe controls two or more external memory banks or peripherals requiring different numbers of wait states.

The remainder of this section describes the active generation of the $\overline{\text{RDY}}$ signal. The example involves three memory banks controlled by $\overline{\text{STRB0}}$, each requiring a different number of wait states. This example directly applies to $\overline{\text{RDY}}$ signal generation involving $\overline{\text{STRB1}}$ and is similar to the case of $\overline{\text{IOSTRB}}$, which involves a more relaxed set of timing parameters.

4.6.6.1 $\overline{\text{RDY}}$ Signal Timing Parameters for $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$

Figure 4–28 and Table 4–6 contain $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ timing parameters that are typically used to generate the $\overline{\text{RDY}}$ signal. As evident in the read and write timing waveforms, the $\overline{\text{RDY}}$ signal generated by the external logic is clocked into the ’C32 on the falling edge of the H1 clock. The associated setup time is represented by parameter 17 and the hold time by parameter 18. Thus, for the 60-MHz ’C32, the $\overline{\text{RDY}}$ signal must arrive at the $\overline{\text{RDY}}$ pin at least 17 ns before the falling edge of H1 and remain valid at least until H1 goes low. Timing parameters 11 and 12 represent the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ low and high delays from the falling edge of H1. Timing parameter 14 represents the address valid delay from the falling edge of H1. For back-to-back write cycles, timing parameter 22 represents the address valid delay from the rising edge of H1. Parameters 11, 12, 14, and 22 do not directly apply to $\overline{\text{RDY}}$ setup and hold, but are nevertheless involved in the generation of the $\overline{\text{RDY}}$ signal.

Figure 4–28. $\overline{RDY}$ Signal Timing for $\overline{STRB0}$ and $\overline{STRB1}$ Cycles



STRB0, STRB1, read cycle

STRB0, STRB1, write cycle

Table 4–6. $\overline{RDY}$ Signal Generation

Parameter number		Description	'32-40† (50 ns) Min	'32-40† (50 ns) Max	'32-50† (40 ns) Min	'32-50† (40 ns) Max	'32-60† (33 ns) Min	'32-60† (33 ns) Max	Unit
11	$t_{d(H1L-SL)}$	Delay time, H1 low to $\overline{STRBx}$ low	0	11	0	9	0	8	ns
12	$t_{d(H1L-SH)}$	Delay time, H1 low to $\overline{SRBx}$ high	0	11	0	9	0	8	ns
14	$t_{d(H1L-A)}$	Delay time, H1 low to A valid	0	11	0	9	0	8	ns
17	$t_{su(RDY)}$	Setup time, $\overline{RDY}$ before H1 low	21		19		17		ns
18	$t_{h(RDY)}$	Hold time, $\overline{RDY}$ after H1 low	0		0		0		ns
22	$t_{d(H1H-A)}$	Delay time, H1 high to A valid on back-to-back write cycles (write)		11		9		8	ns

† These timing specifications are subject to change without notice. See the *TMS320C32 Digital Signal Processor* data sheet for current timing information.

4.6.6.2 $\overline{\text{RDY}}$ Signal Generation for $\overline{\text{STRB0}}$ Signals

Figure 4–29 shows three memory banks controlled by a single strobe ($\overline{\text{STRB0}}$). The first bank is composed of four 8-bit-wide SRAMs requiring zero wait states to operate at 60 MHz (15-ns devices). Bank 2 is composed of two 1-wait-state SRAMs, and bank 3 contains one 3-wait-state EPROM (which is 8 bits wide). The $\overline{\text{RDY}}$ pin is normally high, indicating a not-ready state. It goes low if either $\overline{\text{RDY\_BANK1}}$ or $\overline{\text{RDY\_BANK23}}$ goes low.

The $\overline{\text{RDY\_BANK1}}$ signal is asserted only if two conditions are satisfied:

- At least one of the four $\overline{\text{STRB0}}$ signal lines must be active.
- The three address decode bits must match the bank 1 space.

Since no wait states are involved, the $\overline{\text{RDY\_BANK1}}$ signal does not have to be synchronized with the H1/H3 clocks, and, therefore, it can directly drive the $\overline{\text{RDY}}$ pin after being gated with its bank 2/bank 3 counterpart.

The $\overline{\text{STRB0\_BANK23}}$ signal becomes active (high) if the three address decode bits match bank 2 or bank 3 address spaces while $\overline{\text{STRB0\_B0}}$ and/or $\overline{\text{STRB0\_B1}}$ are active (low). The $\overline{\text{STRB0\_BANK23}}$ signal, when high, sets a high data state in a synchronous progression through a chain of four registers. Depending on which point in the chain is tapped, a $\overline{\text{RDY}}$ signal delay ranging from zero to three wait states can be achieved. In this case, both 1-wait-state and 3-wait-state taps assert the $\overline{\text{RDY\_B23YES}}$ signal to reflect bank 2 or bank 3 access. Finally, a 2-register circuit removes the trailing active low edge of the $\overline{\text{RDY\_B23YES}}$ signal by ORing it with RDY_23NOT (see Figure 4–30). The resulting $\overline{\text{RDY\_BANK23}}$ is ANDed with its bank 1 counterpart to drive the $\overline{\text{RDY}}$ pin.

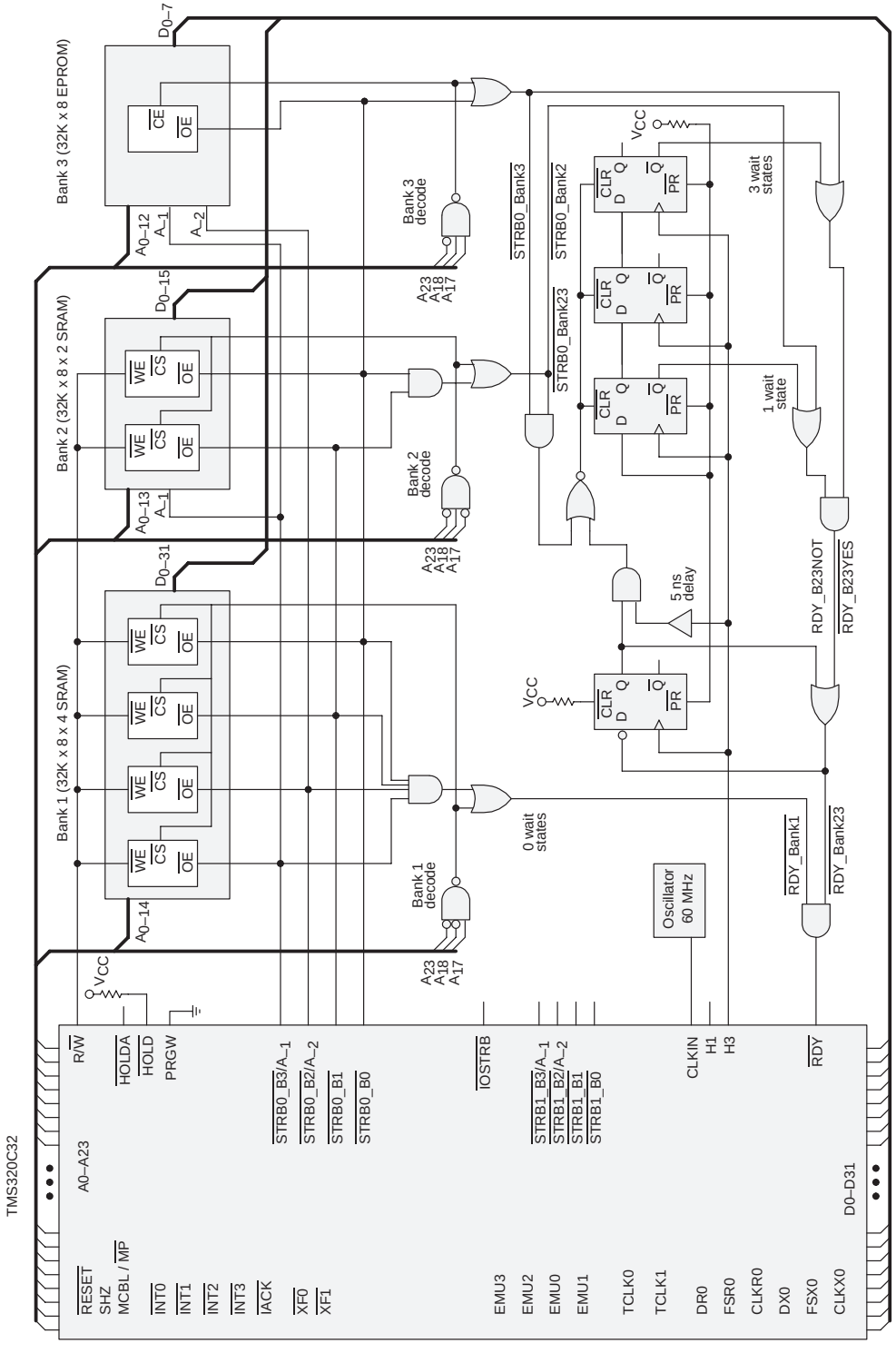


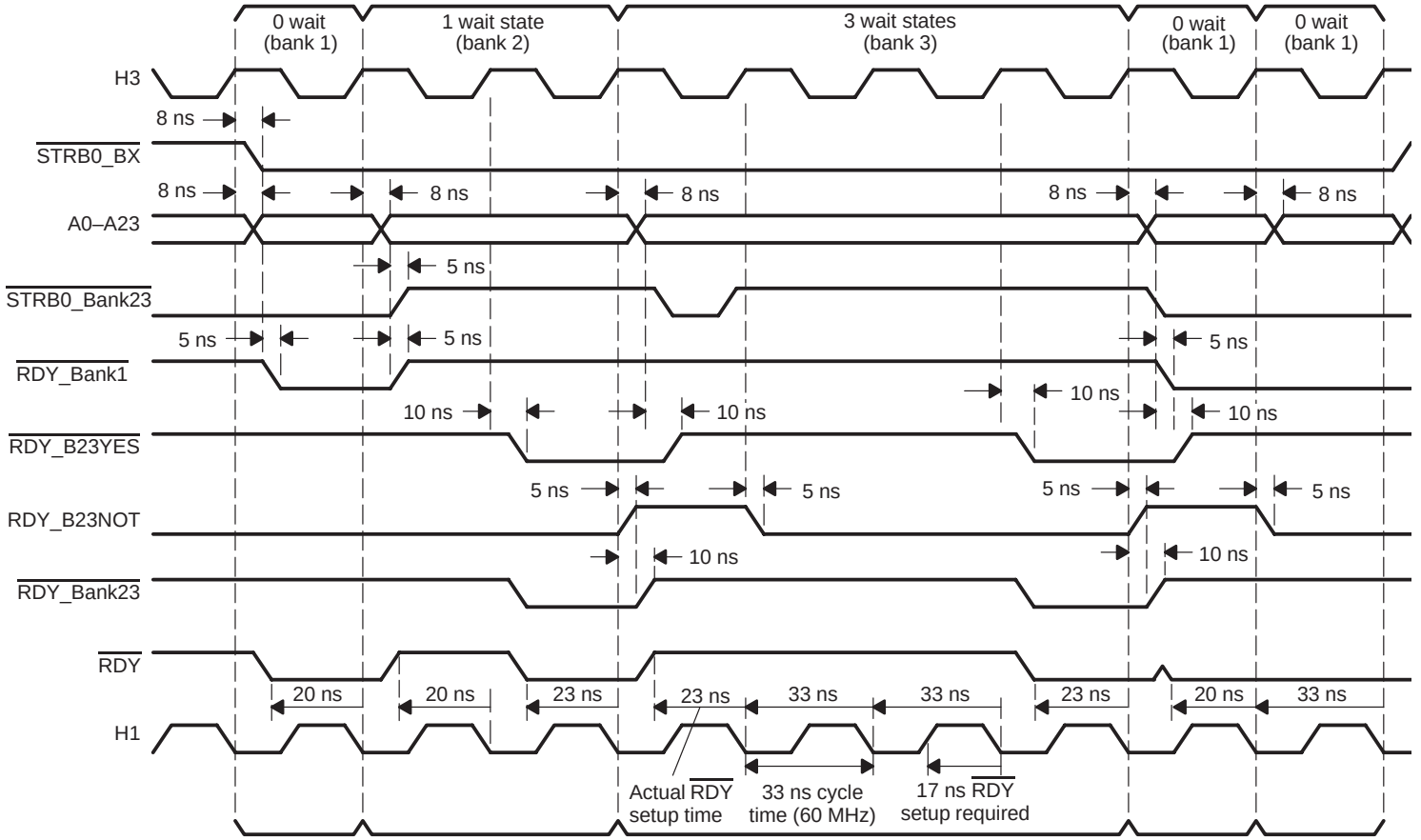
Figure 4–29. $\overline{RDY}$ Signal Generation for $\overline{STRB0}$ Cycles

Figure 4–30 contains timing waveforms for $\overline{\text{RDY}}$ signal generation. It illustrates how the $\overline{\text{RDY}}$ signal is generated for a series of external back-to-back memory read cycles in which the first cycle accesses bank 1 (zero wait states), the second cycle accesses bank 2 (one wait state), the third cycle accesses bank 3 (three wait states), and the fourth and fifth cycles access bank 1 (zero wait states). For each read cycle, the $\overline{\text{RDY}}$ waveform is marked with a resulting setup time. For the 60-MHz device, the $\overline{\text{RDY}}$ signal must become valid at least 17 ns before every falling edge of the H1 clock.

In the 0-wait-state cycle, the address and strobe signals become valid 8 ns from the falling edge of H1. An additional 5 ns are needed for a single pass through a fast combinational logic device for a total setup time of the resulting $\overline{\text{RDY}}$ signal equal to 20 ns. This leaves 3 ns for board delays and a modest safety factor.

For the 1- and 3-wait-state cycles, the bank decode and strobe signals do not directly drive the $\overline{\text{RDY}}$ signal. They are instead combined into the $\overline{\text{STRB0\_BANK23}}$ signal that, when active, releases the clear condition on the 3-register delay chain driven by the H3 clock. The register chain is then free to propagate a high state at the rate of one register per clock cycle. The two taps in the register chain (at the first and third registers, representing one wait state and three wait states, respectively) are ORed with their corresponding bank select signals to result in the $\overline{\text{RDY\_B23YES}}$ signal synchronous to H1/H3 clocks. The $\overline{\text{RDY\_B23YES}}$ leading-edge 10-ns delay is caused by two passes through a fast PAL® device (such as a 22V10). The trailing edge of this signal is caused by bank 2 or bank 3 decode circuits going inactive after the $\overline{\text{RDY}}$ signal is recognized by the processor. The address decode (8 ns) plus two passes through the PAL (5 + 5 ns) combine for a total delay of 18 ns that can cut into the next cycle's $\overline{\text{RDY}}$ setup requirement (33 – 18 = 15 ns) if not modified. To deactivate the $\overline{\text{RDY}}$ signal sooner, a single-register circuit is added to generate the RDY_B23NOT, which, when ORed with the $\overline{\text{RDY\_B23YES}}$, yields the $\overline{\text{RDY\_BANK23}}$ signal that satisfies the $\overline{\text{RDY}}$ setup time for the next cycle. Finally, $\overline{\text{RDY\_BANK1}}$ and $\overline{\text{RDY\_BANK23}}$ are ANDed together to produce the final $\overline{\text{RDY}}$ signal that is wired to the processor's $\overline{\text{RDY}}$ pin.

Figure 4–30. $\overline{RDY}$ Signal Generation Timing Waveforms



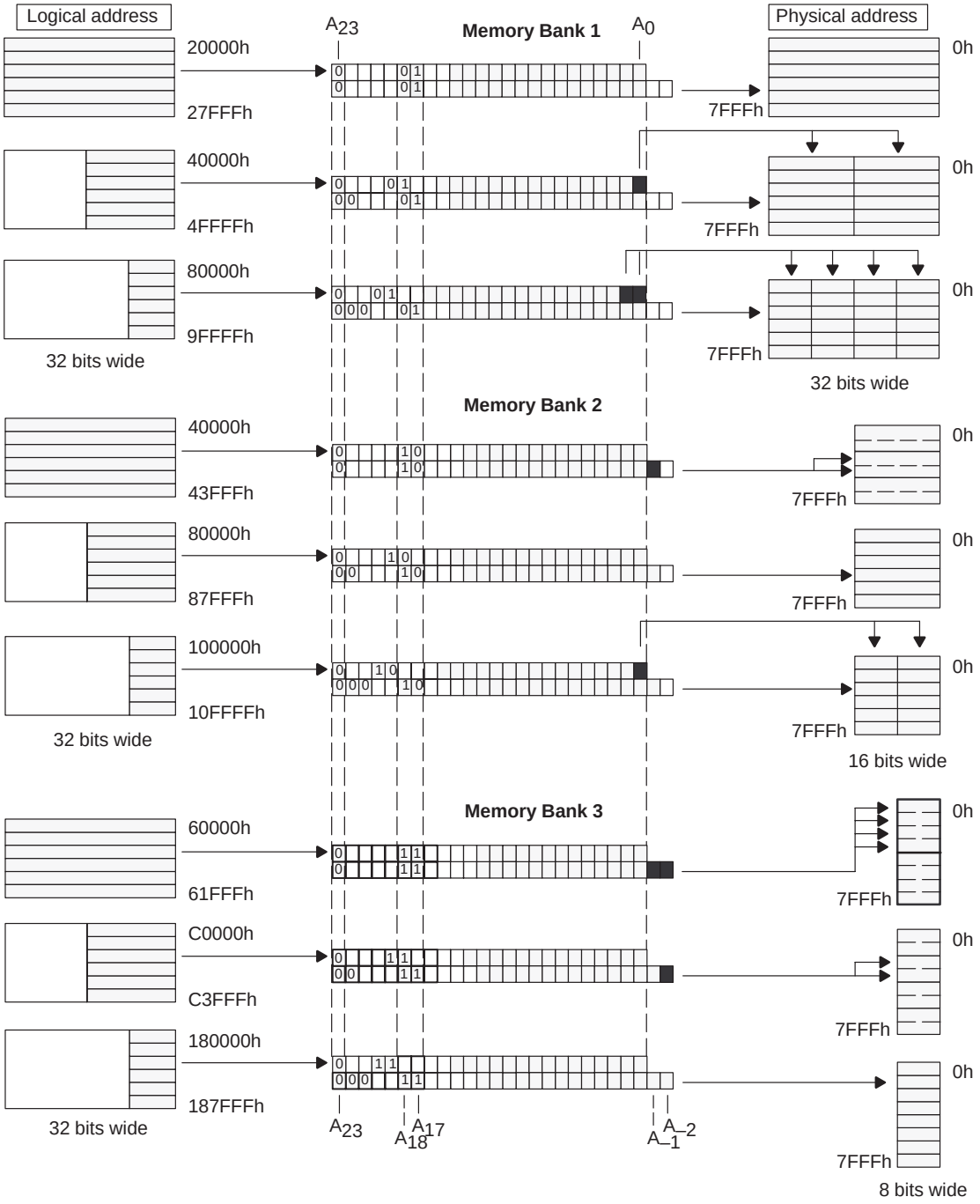
4.6.7 Address Decode for Multiple Banks

Figure 4–31 illustrates the logical-to-physical address translation for the three memory banks used in the $\overline{\text{RDY}}$ signal generation example in section 4.6.6. Each memory bank is a different physical width, as shown by the physical address column on the right side of the figure. The left side of the figure represents the internal (logical) address ranges for each of the three memory banks. Logical-to-physical address translation is controlled by strobe control registers and by their data size and memory width fields. The middle column of Figure 4–31 shows the logical address field (top row) over the physical address (bottom row) for each address translation case. The active address fields are shaded gray, and the inactive address bits are white. The black fields are special address bits that can selectively control multiple strobe lines or choose between individual portions of a data word that is larger than the physical memory it is accessing.

For example, in bank 2, the right side of the figure indicates that the physical memory width for this bank is 16 bits. The left side indicates that, regardless of the physical memory width, 32-, 16-, and 8-bit data can be moved by programming the $\overline{\text{STRB0}}$ control register. The low-order (shaded) bits of logical/physical address rows show how many bits are actually used for addresses so that the correct high-order address bits can be assigned to bank decode. Physical address bits A17 and A18 are chosen for bank decode because they lie outside the used address bits. A17 and A18 decode between banks 1, 2, and 3, with A18–A17 = (0,1) assigned to bank 1, (1,0) assigned to bank 2, and (1,1) assigned to bank 3. Address bit A23 is set to 0 to isolate the $\overline{\text{STRB0}}$ address space from the $\overline{\text{STRB1}}$ and $\overline{\text{IOSTRB}}$ memory maps.

The dotted lines bounding the bank decode bits allow you to see that the external address bits, A18–A17, line up perfectly, but their logical address counterparts do not. The amount of reverse shift between the logical and physical addresses depends on the size of the data being accessed and the width of the physical memory. Each of the three address translation cases for each of the three banks translates physical address bits A18–A17 into two contiguous logical address bits that can lie anywhere between A20 and A17. Once the logical images of the external bank decode bits are identified along with low-order address bits and the A23 strobe decode bit, they define the final logical memory map for the three $\overline{\text{STRB0}}$ banks together.

Figure 4–31. Address Decode for Multiple Memory Banks



Note: Active address fields are shaded gray; inactive address bits are white. The black fields are special address bits that control multiple strobe lines or choose between portions of a data word that is larger than the physical memory it is accessing.

Each memory bank actually has three logical memory maps, depending on the size of the data being accessed and the setting of the corresponding bits in the $\overline{\text{STRB0}}$ control register.

The address ranges in these logical memory maps are all different, yet all three maps translate perfectly into a single physical address map that identifies the bank. In using the three logical memory maps, the programmer must exercise caution to prevent overwriting 8-bit data with 16-bit data (or 16-bit data with 32-bit data) that may have a different logical address but still occupy the same place in physical memory. To be certain that the logical address maps associated with 8-, 16-, and 32-bit data sizes do not overlap within a single physical memory bank, the three logical maps must be further divided into mutually exclusive areas before they are used by the programmer. Furthermore, when a program jumps from one physical memory bank to another of a different width, the memory width configuration bits in the appropriate strobe register must be changed.

4.7 How TMS320 Tools Interact With the TMS320C32's Enhanced Memory Interface

The 'C32's memory interface accesses external memory through one 24-bit address bus and one 32-bit data bus. The data bus is shared by three mutually-exclusive strobes: $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$. Depending upon the address accessed, the 'C32 activates one of these strobes. (See the *TMS320C3x User's Guide* for more information about memory maps.)

$\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ can access 8-, 16-, or 32-bit data quantities from 8-, 16-, or 32-bit-wide memory. Access is achieved by four signals within each strobe. These signals are:

- $\overline{\text{STRBx\_B3}}$/A$_{-1}$
- $\overline{\text{STRBx\_B2}}$/A$_{-2}$
- $\overline{\text{STRBx\_B1}}$
- $\overline{\text{STRBx\_B0}}$

The listed signals serve as byte-enable pins for accessing a byte, half-word, or full-word from external memory. The first two signals also serve as additional address pins when performing two or four consecutive accesses in 8- or 16-bit-wide external memory. The data accessed is truncated, packed, or unpacked accordingly, with no additional overhead. The following list shows the behavior of these pins, as dictated by the data size and memory-width bit fields.

The default value of a strobe control register depends on the program memory width select (PRGW) pin level.

- 8-bit-wide memory
  - $\overline{\text{STRBx\_B3}}$/A$_{-1}$ and $\overline{\text{STRBx\_B2}}$/A$_{-2}$ are address pins.
  - $\overline{\text{STRBx\_B0}}$ is a byte-enable/chip-select signal.
  - $\overline{\text{STRBx\_B1}}$ is not used.

- 16-bit-wide memory
  - $\overline{\text{STRBx\_B3}}$/A$_{-1}$ are address pins.
  - $\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$ are byte-enable signals.
  - $\overline{\text{STRBx\_B2}}$/A$_{-2}$ are not used.

- 32-bit-wide memory
  - $\overline{\text{STRBx\_B3}}$/A$_{-1}$, $\overline{\text{STRBx\_B2}}$/A$_{-2}$, $\overline{\text{STRBx\_B1}}$, and $\overline{\text{STRBx\_B0}}$ are byte-enable signals.

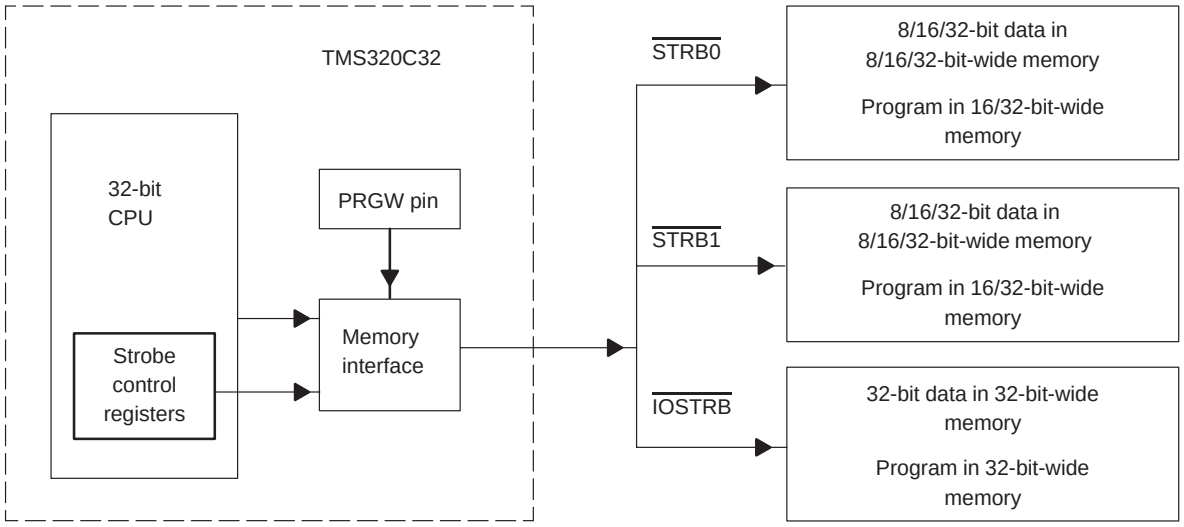
- Data size:
  - 8-bit data: The physical address is the logical address shifted right by 2.
  - 16-bit data: The physical address is the logical address shifted right by 1.
  - 32-bit data: The physical address is the logical address.

$\overline{\text{IOSTRB}}$ can access 32-bit data from 32-bit-wide memory. However, $\overline{\text{IOSTRB}}$ does not have the flexibility of $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ because it is composed of a single signal. $\overline{\text{IOSTRB}}$ bus cycles differ from $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ bus cycles. (See the *Interlocked Operations* section in the *Program Flow Control* chapter of the *TMS320C3x User's Guide* for more information.) This timing difference accommodates slower I/O peripherals.

The 'C32 also supports program execution from 16- and 32-bit external memory widths. Execution is controlled through the status of the PRGW pin. When this pin is pulled high, the 'C32 executes from 16-bit-wide memory. When the PRGW pin is pulled low, the 'C32 executes from 32-bit-wide memory. For 16-bit-wide zero-wait-state memory, the 'C32 takes two instruction cycles to fetch a single 32-bit instruction. The lower 16 bits of the instruction are obtained during the first cycle; the upper 16 bits are retrieved and concatenated with the lower 16 bits during the second cycle. The 'C32's 32-bit memory fetches are identical to those of the 'C30 and 'C31.

In summary, the 'C32 memory interface parallel bus implements three mutually exclusive address spaces that are distinguished through the use of three separate control signals (see Figure 4–32). $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ support 8-, 16-, and 32-bit data access in 8-, 16-, and 32-bit-wide external memory and 32-bit program access in 16/32-bit-wide external memory. $\overline{\text{IOSTRB}}$ address space supports 32-bit data/program access in 32-bit-wide external memory. Internally, the 'C32 has a 32-bit architecture; accordingly, the memory interface packs and unpacks the data accessed. Three strobe control registers manipulate the variable-width memory interface of the 'C32. (See the *TMS320C3x User's Guide* for a detailed description of the 'C32 memory interface.)

Figure 4–32. TMS320C32 Memory Address Spaces



4.7.1 C Compiler Interaction With the TMS320C32 Memory Interface

The 'C32's internal 32-bit architecture allows the C compiler's data types to remain 32 bits wide. However, the C compiler's runtime-support library includes pragma directives and new dynamic-allocation routines (malloc, realloc, calloc, bmalloc, free, etc.) that support the creation of data sections. These data sections serve as memory pools for storing 8- and 16-bit data. These sections can reside in 8-, 16-, and 32-bit-wide memory. The programmer must ensure that the appropriate strobe control register is loaded with the correct data size and memory width. The 'C32's memory interface truncates, packs, or unpacks the data in the manner specified by the settings of the strobe control register. Table 4–7 lists the data sizes supported by the sections created by the C compiler.

Table 4–7. Data Sizes Supported by Sections Created by the C Compiler

Section Type	32 Bits	16 Bits	8 Bits
Initialized	.text .cinit .const *.user_section*	*.user_section*	*.user_section*
Uninitialized	.bss .stack .sysmem *.user_section*	.sysm16 *.user_section*	.sysm8 *.user_section*

The contents of the named sections are as follows:

- **.text**: executable code and/or string literals
- **.cinit**: tables for variable and constant initialization
- **.const**: string literals and switch tables
- **.bss**: global variables and statically allocated variables
- **.stack**: system stack used to pass function arguments and to allocate local function variables
- **.sysmem**: memory pool for dynamic allocation of 32-bit data
- **.sysm16**: memory pool for dynamic allocation of 16-bit data
- **.sysm8**: memory pool for dynamic allocation of 8-bit data
- ***.user_section***: section created using the #pragma DATA_SECTION directive

The following sections describe the C compiler's preprocessor pragma and modules in the runtime-support library that support 8- and 16-bit memory pools. The 32-bit memory pools are handled through the standard minit(), malloc(), smalloc(), calloc(), realloc(), and free() routines, which operate on the .sysmem section.

4.7.1.1 DATA_SECTION Pragma Directive

To support additional memory pools, the C compiler uses a data section pragma directive. This directive instructs the C compiler to allocate space for *symbol_name* in the section specified by *section_name* of size *symbol_size*. (See the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* for additional information.) The syntax for DATA_SECTION is as follows:

```
#pragma DATA_SECTION(symbol_name, "section_name")
type symbol_name;
```

For example, define a new section called .mydata as an array of 1K integer values in the following manner:

```
#pragma DATA_SECTION(dataBuf, ".mydata")
int dataBuf[1024];
```

4.7.1.2 MEMORY8.C Module

The MEMORY8.C module contains functions that implement dynamic memory management routines for using 8-bit data with the 'C32. (See the *TMS320C3x/C4x Optimizing C Compiler User's Guide* for more information on 8-bit runtime-support functions.)

The pragma directive in the MEMORY8.C module defines a .sysm8 section. The size of this memory pool in words (system memory or heap) is set at link time by using the -heap8 option. If the -heap8 option is not used, the compiler does not allocate an 8-bit system memory area. If arguments are not used in conjunction with this switch, the size of the 8-bit system memory area defaults to 1K 8-bit words. The following functions operate in the 8-bit .sysm8 section:

- **minit8()**: initializes and resets the 8-bit dynamic memory management system
- **malloc8()**: allocates 8-bit words from the 8-bit memory pool and returns a pointer to the allocated space
- **calloc8()**: allocates 8-bit words from the 8-bit memory pool, clears allocated memory locations, and returns a pointer to the allocated space
- **realloc8()**: reallocates 8-bit words from previously unallocated areas in the 8-bit memory pool; a pointer to the allocated space is returned
- **free8()**: frees previously allocated space from the 8-bit memory pool
- **bmalloc8()**: allocates 8-bit words from the 8-bit memory pool. The allocated words are aligned to a boundary that is suitable for the 'C32's circular and bit-reversed buffers; a pointer to the allocated space is returned.
- **_SYSMEM8_SIZE**: an external label that contains the size, in words, of the 8-bit system memory pool

4.7.1.3 MEMORY16.C Module

The MEMORY16.C module contains functions that implement dynamic memory management routines for the 'C32's 16-bit data. (See the *TMS320C3x/C4x Optimizing C Compiler User's Guide* for more information on 16-bit runtime-support functions.)

The pragma directive in the MEMORY16.C module defines a .sysm16 section. The size of this memory pool in words (system memory or heap) is set at link time by using the -heap16 option. If the -heap16 option is not used, the compiler does not allocate a 16-bit system memory area. If arguments are not used in conjunction with this switch, the size of the 16-bit system memory area

defaults to 1K 16-bit words. The following functions operate in the 16-bit .sysm16 section.

- **minit16()**: initializes and resets the 16-bit dynamic memory management system
- **malloc16()**: allocates 16-bit words from the 16-bit memory pool and returns a pointer to the allocated space
- **calloc16()**: allocates 16-bit words from the 16-bit memory pool, clears allocated memory locations, and returns a pointer to the allocated space
- **realloc16()**: reallocates 16-bit words from previously unallocated areas in the 16-bit memory pool; a pointer to the allocated space is also returned
- **free16()**: frees previously allocated space from the 16-bit memory pool
- **bmalloc16()**: allocates 16-bit words from the 16-bit memory pool. The allocated words are aligned to a boundary that is suitable for the 'C32's circular- and bit-reversed buffers; a pointer to the allocated space is also returned.
- **_SYSMEM16_ SIZE**: an external label that contains the size, in words, of the 16-bit system memory pool

4.7.1.4 Memory Pool Limitations

The 'C32 has only three strobes: $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$. This means a programmer cannot have more than three memory pools; one memory pool assigned to each strobe. $\overline{\text{IOSTRB}}$ can hold only 32-bit data and can only accommodate the 32-bit memory pool .sysmem. Conversely, $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ can hold 8-, 16-, and 32-bit data and can accommodate the 8-, 16-, and 32-bit memory pools .sysm8, .sysm16, and .sysmem.

All pointers and constants must be stored in memory configured to hold 32-bit data. Hence, the .bss, .stack, .cinit, and .const sections must reside in memory with data size configured to 32 bits.

4.7.2 C Compiler and Assembler Switch

To create code for the 'C32, the assembler and C compiler use the -v32 version specification switch. The following example demonstrates the use of this switch with the assembler and C compiler, respectively:

```
asm30  -v32  myfile.asm
cl30   -v32  myfile.c
```

4.7.3 Linker Switches

To support the 'C32's 8- and 16-bit memory pools, the linker uses the following switches: -heap8, -heap16, and -heap. These switches set the size, in words, of the respective 8-, 16-, and 32-bit memory system areas .sysm8, .sysm16, and .sysmem. The user must link these sections into the appropriate addresses, thereby activating strobes that are configured to access 8-, 16-, or 32-bit data.

The following example demonstrates the link-time sizing of an 8-bit memory pool to 256K words:

```
lnk30 -heap8 0x4000
```

The linker creates these memory system areas using an input file that contains the .sysmem, .sysm8, and .sysm16 data-section definitions. If the input file does not exist, the linker is unable to perform memory area processing.

The linker also creates the global symbols _SYSMEM_SIZE, _SYSMEM8_SIZE, and _SYSMEM16_SIZE and subsequently assigns each a value equal to the respective -heap, -heap8, and -heap16 size. The default size for each memory system area is 1K words (word size depends on system memory width).

4.7.4 Debugger Configuration

For the debugger to properly disassemble and read/write external memory, the user must configure the strobe control registers before loading and executing code. Because the 'C32 supports code execution from 16- or 32-bit memory, the debugger may need to temporarily set the strobe control register to a 32-bit data size in order to write an instruction (either by loading code or patching code) or to read an instruction with the objective of disassembling a range of program memory.

To support code execution from 16- and 32-bit memory, the memory map add (ma) command includes a new *type* parameter that directs the debugger to treat .text sections as 32-bit data. While reading or writing .text sections, the debugger does the following:

- ❏ Temporarily stores the configuration of the appropriate strobe control register
- ❏ Temporarily sets the data size to 32 bits
- ❏ Reads or writes the targeted portion of the .text section
- ❏ Restores the strobe control register to its previous value

The syntax for the memory map add command is:

ma *address, length, type*

where:

address defines the starting address of a range of memory

length defines the length of the memory range

type identifies the read/write characteristic of the memory range depending upon one or more of the following keywords:

- **R**: read only
- **W**: write only
- **WR** or **RAM**: read/write
- **PROTECT**: no-access memory
- **TX**: memory that stores *.text* (code) section

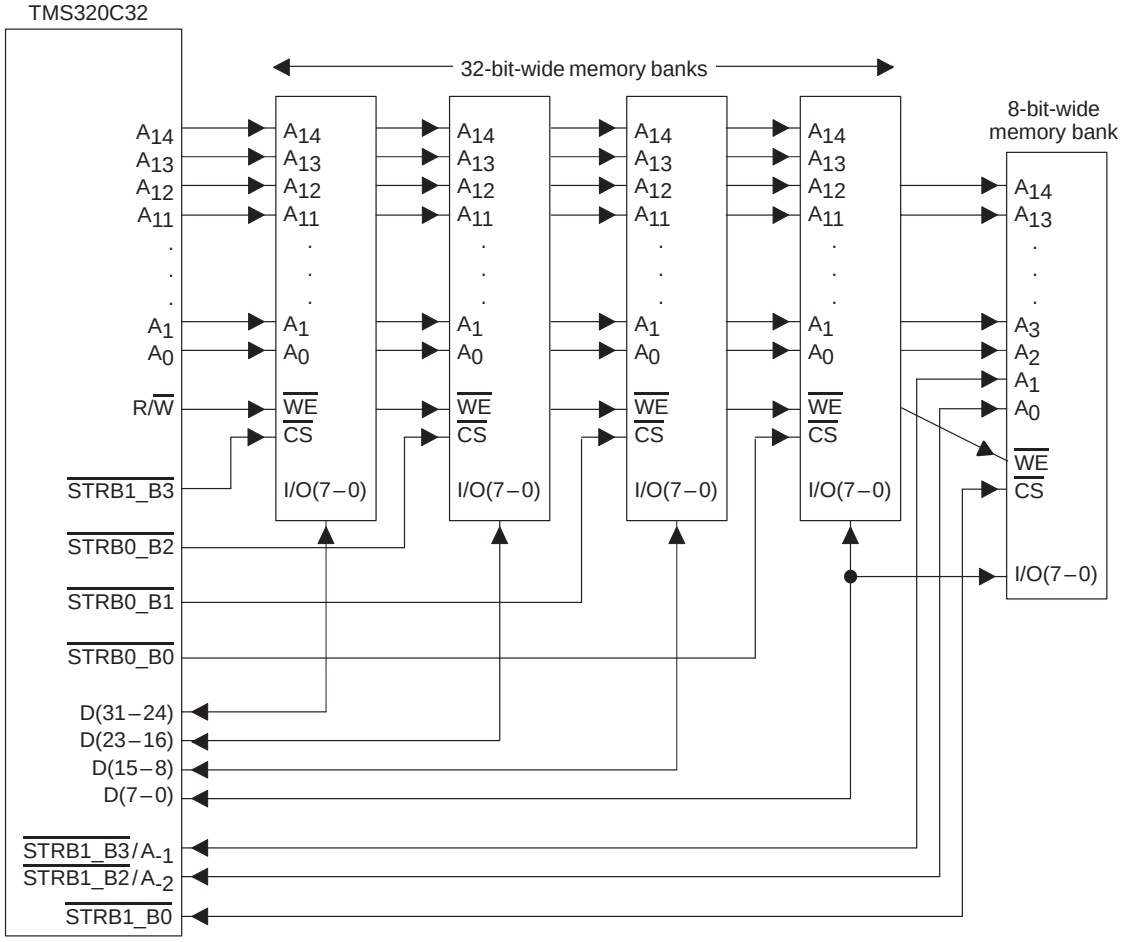
4.7.5 TMS320C32 Configuration Examples

Ths section describes the possible 'C32 memory interface configurations, including instructions on how to allocate buffers, build link files, and configure the debugger for each memory configuration.

4.7.5.1 Two External Memory Banks

The 'C32's external memory interface allows the use of two zero-wait-state external memory banks with different widths without requiring additional logic or incurring access penalty costs. These external memory banks provide flexibility in balancing performance and system cost (performance and system cost increase with wider memory chips). For example, the programmer can execute code from 32-bit wide memory while storing data in 8-bit memory (see Figure 4–33). This approach is advantageous for applications with large amounts of 8-bit data that require execution at the fastest speed of the device.

Figure 4–33. Zero-Wait-State Interface for 32-Bit and 8-Bit SRAM Banks



In Figure 4–33, a bank of 32K × 32 bits is mapped to $\overline{\text{STRB0}}$, and a bank of 32K × 8 bits is mapped to $\overline{\text{STRB1}}$. For this configuration, the programmer must set the following:

- $\overline{\text{STRB0}}$ control register physical memory width to 32 bits and the data type size to 32 bits
- STRB config bit field to 0, that is, $\overline{\text{STRB0}}$ control register = 000F0000h (banks are separate)
- $\overline{\text{STRB1}}$ control register physical memory width to 8 bits and the data type size to 8 bits, that is, $\overline{\text{STRB1}}$ control register = 00000000h

Additionally, the PRGW pin must be pulled low to indicate 32-bit program memory width.

Figure 4–33 also maps the 32-bit-wide bank's external memory address pins, $A_{14}A_{13}...A_1A_0$, to the 'C32's $A_{14}A_{13}A_{12}...A_1A_0$ pins. Conversely, the 8-bit-wide bank's memory address pins, $A_{14}A_{13}...A_1A_0$, are mapped to the 'C32's $A_{12}...A_1A_0A_{-1}$ pins. Because $\overline{STRB1}$ is configured for 8-bit memory width, the external address presented on 'C32 pins is shifted right by two bits. As a result of this mapping, external memory accesses in the range 0h through 7FFFh read or write 32-bit data to the 32-bit-wide bank ($\overline{STRB0}$). Memory accesses in the range 900000h through 907FFFh read or write 8-bit data to the 8-bit-wide bank ($\overline{STRB1}$).

Two banks of different memory widths must not be connected to the same STRB without external decode logic. Different memory widths require $\overline{STRBx\_Bx}$ signals to be configured as address pins. These address pins are active for any external memory access, that is, $\overline{STRB0}$, $\overline{STRB1}$, $\overline{IOSTRB}$, and program fetches.

8-bit Dynamic Memory Allocation

This section contains C code examples of 8-bit dynamic buffer allocation, linker configuration, and a debugger batch file.

Example 4–1 demonstrates the allocation of two buffers (1K and 4K 8-bit words) using the 8-bit dynamic memory allocation routines.

Example 4–1. 8-Bit Dynamic Buffer Allocation

```
void main()
{
  int    *buffer1;
  float  *buffer2;          /* Configure the STRB0 control register for 32-bit wide
                            memory, 32-bit data size. */
  *0x808064 = 0xF0000;      /* Configure the STRB1 control register for 8-bit wide
                            memory, 8-bit data size. */
  *0x808068 = 0x00000;      /* Allocate 1K 8-bit words in the 8-bit memory pool. */
  buffer1 = malloc8(1024 * sizeof(int) ); /* Allocate 4K 8-bit floats in the 8-bit
                                          memory pool. */
  buffer2 = malloc8(4096 * sizeof(float) );    /* Process buffers. */
  callDSPoperation(buffer1, buffer2);
  /* Free buffers. */
  free8(buffer2);
  free8(buffer1);
}
```

Note:

The TMS320 floating-point C compiler *sizeof* function returns 1 for both integer and float data types.

Example 4–2 allocates sections of the preceding code into the desired memory configuration.

Example 4–2. Linker Command File

```
sample.obj                      /* Input filename                      */
–heap8 32768                    /* Set 8-bit memory pool size.         */
–stack 8704                     /* Set C system stack size.            */
–o sample.out                   /* Specify output file.                */
–m sample.map                   /* Specify map file.                   */
MEMORY
{
      PRGRAM       :     org = 0x0000,            len = 0x2000
      STRB0RAM     :     org = 0x2000,            len = 0x6000
      ONCHIRAM     :     org = 0x87Fe00,          len = 0x200
      STRB1RAM     :     org = 0x900000,          len = 0x8000
}
SECTIONS
{
      .text  > PRGRAM                 /* 32-bit data section          */
      .cinit > STRB0RAM         /* 32-bit data section               */
      .const > STRB0RAM         /* 32-bit data section               */
      .bss   > STRB0RAM         /* 32-bit data section               */
      .stack > STRB0RAM         /* 32-bit data section               */
      .sysm8 > STRB1RAM         /* 8-bit memory pool mapped to
       STRB1 */
}
```

The debugger batch file shown in Example 4–3 executes initialization commands that configure the C source debugger to handle a 'C32 with the memory configuration shown in Figure 4–33 on page 4-75.

Example 4–3. Debugger Batch File

```
mr
sconfig init.clr
;   Define memory configuration.
ma 0x0000, 0x2000, R|W|TX  ; Inform debugger that this section holds code
                             (.text).
ma 0x2000, 0x6000, RAM     ; No code here, STRB0
ma 0x87FE00, 0x200, RAM    ; On-chip
ma 0x808000, 0x10, RAM     ; Peripheral Bus Control – DMA
ma 0x808020, 0x20, RAM     ; Peripheral Bus Control – Timers
ma 0x808040, 0x10, RAM     ; Peripheral Bus Control – Serial Port 0
ma 0x808060, 0x10, RAM     ; Peripheral Bus Control – External Memory Interface
ma 0x900000, 0x8000, RAM   ; STRB1
;
reset
map on                     ; Make emulator aware of this memory configuration.
;
?*0x808064 = 0xF0000       ; Set STRB0 control register to 32-bit memory width,
                           ; 32-bit data size.
?*0x808068 = 0x00000       ; Set STRB1 control register to 8-bit memory width,
                           ; 8-bit data size.
;
load sample.out            ; Configure STRB0 and STRB1 control registers before
                           ; loading code.
```

8-Bit Static Memory Allocation

This section provides examples of 8-bit static buffer allocation and associated linker configuration. The debugger batch file is identical to the batch file in Example 4–3 and, therefore, is not shown.

The C code in Example 4–4 demonstrates the static allocation of two buffers (1K and 4K 8-bit words) by defining a user section called .mydata8. This section is used to hold a structure consisting of two arrays of data values.

Example 4–4. 8-Bit Static Buffer Allocation

```
#pragma DATA_SECTION(buffer8, ".mydata8")
struct bufferStruct {
       in[1024];
       out[4096];
} buffer8;
void main( )
{
       /* Configure the STRB0 control register for 32-bit wide memory, 32-bit
       data size. */
       *0x808064 = 0xF0000;
       /* Configure the STRB1 control register to 8-bit wide memory, 8-bit data
       size. */
       *0x808068 = 0x00000;
       /* Process buffers. */
       callDSPoperation(buffer8.in, buffer8.out);
}
```

The linker command file in Example 4–5 allocates sections of the above C code into the desired memory configuration.

Example 4–5. Linker Command File

```
sample.obj                      /* Input filename                        */
–stack 8704                     /* Set C system stack size.              */
–o sample.out                   /* Specify output file.                  */
–m sample.map                   /* Specify map file.                     */
MEMORY
{
       PRGRAM       :     org = 0x0000,       len = 0x2000
       STRB0RAM     :     org = 0x2000,       len = 0x6000
       ONCHIRAM     :     org = 0x87Fe00,     len = 0x200
       STRB1RAM     :     org = 0x900000,     len = 0x8000
}
SECTIONS
{
       .text  > PRGRAM           /* 32-bit data section                   */
       .cinit > STRB0RAM         /* 32-bit data section                   */
       .const > STRB0RAM         /* 32-bit data section                   */
       .bss   > STRB0RAM         /* 32-bit data section                   */
       .stack > STRB0RAM         /* 32-bit data section                   */
       .mydata8 > STRB1RAM       /* 8-bit memory pool mapped to STRB1     */
}
```

4.7.5.2 Single External Memory Bank

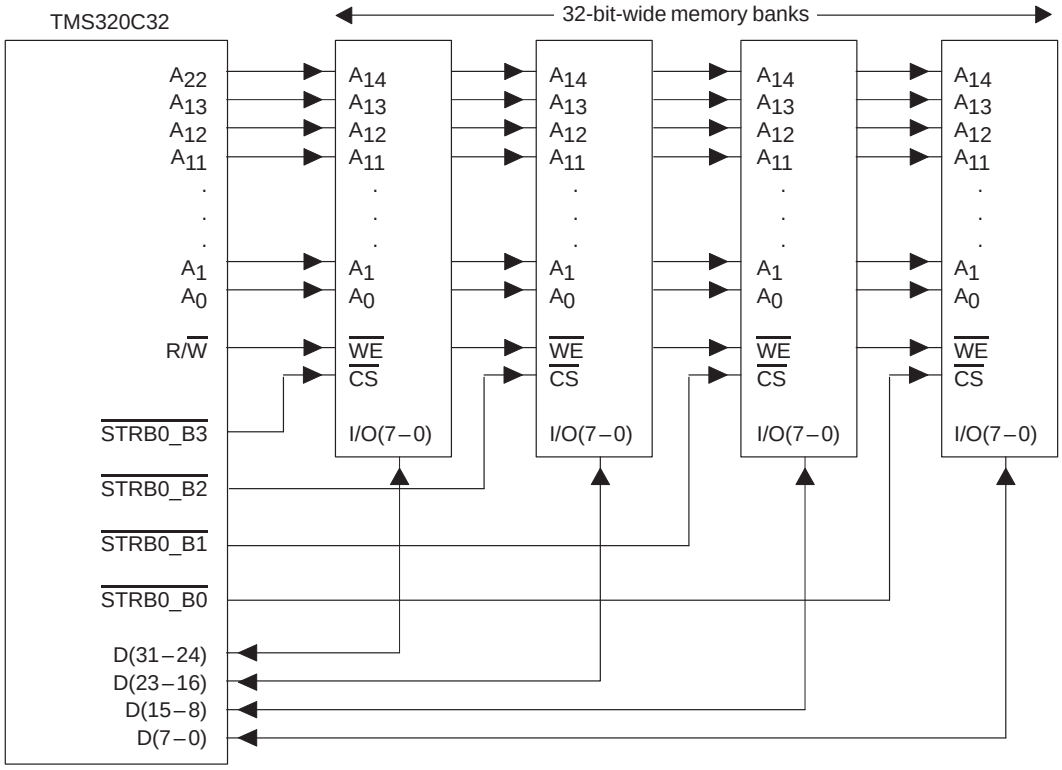
Consider the case of a typical audio compression application written in C that requires 32-bit data for the system stack and 16-bit data for the audio buffers. In this case, the programmer can interface the 'C32, as shown in Figure 4–34. This example assumes 32K 32-bit words of external memory. This memory is further defined as containing 8.5K 32-bit words of stack and 8K 32-bit words of program space; both areas are mapped to $\overline{\text{STRB0}}$ (program space includes constants and global/static variables). Also, external memory contains 32K 16-bit word data buffers that are mapped into $\overline{\text{STRB1}}$.

Due to this mapping, the programmer must set the following:

- $\overline{\text{STRB0}}$ control register physical memory width to 32 bits and the data type size to 32 bits
- STRB configuration bit field to 1 ($\overline{\text{STRB0}}$ control register = 002F0000h)
- $\overline{\text{STRB1}}$ control register physical memory width to 32 bits and the data type size to 16 bits, that is, $\overline{\text{STRB1}}$ control register = 000D0000h

Additionally, the PRGW pin must be pulled low to indicate 32-bit program memory width.

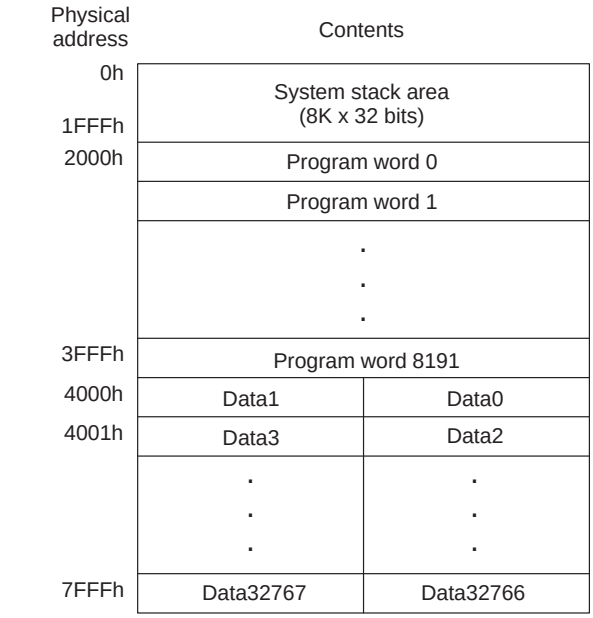
Figure 4–34. Zero-Wait-State Interface for 32-Bit SRAMs with 16- and 32-Bit Data Accesses



The external memory address pins $A_{14}A_{13}...A_1A_0$ are mapped to the 'C32's $A_{22}A_{13}A_{12}...A_1A_0$ pins. This mapping was selected to position the system stack immediately after the 'C32's internal RAM. Performance is improved because the top of the stack resides in internal RAM, and the stack is allowed to grow into external RAM. With this mapping, external memory accesses in the range 4000h through 7FFFh read or write 16-bit data; memory accesses in the range 0h through 3FFFh read or write 32-bit data. The PRGW pin controls the program fetches.

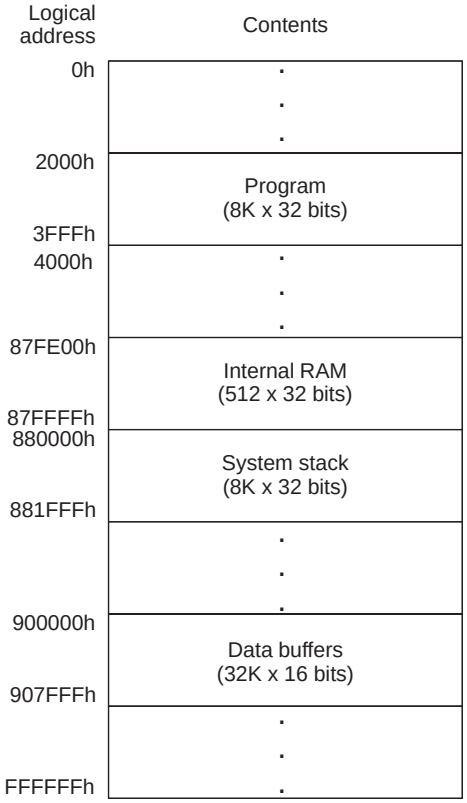
Figure 4–35 shows the contents of external memory. Because of the address shift of the 'C32's external memory interface, the memory map for the 'C32 CPU is slightly different (see Figure 4–36).

Figure 4–35. External Memory Map



Note: For 32-bit data, physical address = logical address.
For 16-bit data, physical address = logical address shifted left by 1.

Figure 4–36. TMS320C32 Memory Map



Note: For 32-bit data, physical address = logical address.
For 16-bit data, physical address = logical address shifted left by 1.

16-Bit Dynamic Memory Allocation

This section contains C code examples of 16-bit dynamic buffer allocation, linker configuration, and a debugger batch file.

The following C code demonstrates the allocation of two buffers (1K and 4K, 16-bit words) using the 16-bit dynamic memory allocation routines provided by the runtime-support library.

Example 4–6. 16-Bit Dynamic Buffer Allocation

```
# include <bus30.h>
void main( )
{
       int          *buffer1;
       float  *buffer2;
       /* Configure the STRB0 control register to STRB0 and STRB1 overlay. */
       /* 32-bit wide memory, 32-bit data size */
       /* If using the PRTS30 headers,
              BUS_ADDR–>STRB0_gcontrol = STRB0_1_CNFG | MEMW_32 | DATA_32; */
       *0x808064 = 0x2F0000;
       /* Configure STRB1 control register to 32-bit wide memory, 16-bit data
       size. */
       /* If using the PRTS30 headers,
              BUS_ADDR–>STRB1_gcontrol = MEMW_32 | DATA_16; */
       *0x808068 = 0xD0000;
       /* Allocate 1K 16-bit words in the 16-bit memory pool. */
       buffer1 = malloc16(1024 * sizeof(int) );
       /* Allocate 4K 16-bit floats in the 16-bit memory pool. */
       buffer2 = malloc16(4096 * sizeof(float));
       /* Process buffers. */
       callDSPoperation(buffer1, buffer2);
       /* Free buffers. */
       free16(buffer2);
       free16(buffer1);
}
```

The linker command file in Example 4–7 allocates sections of the preceding C code into the memory configuration depicted in Figure 4–35 on page 4-82.

Example 4–7. Linker Command File

```
sample.obj                      /* Input filename                      */
–heap16 32768                   /* Set 16-bit memory pool size.        */
–stack 8704                     /* Set C system stack size.            */
–o sample.out                   /* Specify output file.                */
–m sample.map                   /* Specify map file.                   */
MEMORY
{
      STRB0RAM     :     org = 0x2000,  len = 0x2000
      STACKRAM     :     org = 0x87Fe00, len = 0x2200
      STRB1RAM     :     org = 0x900000, len = 0x8000
}
SECTIONS
{
      .text  > STRB0RAM         /* 32-bit data section                 */
      .cinit > STRB0RAM         /* 32-bit data section                 */
      .const > STRB0RAM         /* 32-bit data section                 */
      .bss   > STRB0RAM         /* 32-bit data section                 */
      .stack > STACKRAM         /* 32-bit data section                 */
      .sysm16 > STRB1RAM        /* 16-bit memory pool mapped to STRB1  */
}
```

The debugger batch file in Example 4–8 executes initialization commands that configure the C source debugger to handle a 'C32 with the memory configuration shown in Figure 4–36 on page 4-83.

Example 4–8. Debugger Batch File

```
mr
sconfig init.clr
;   Define memory configuration.
ma 0x2000, 0x2000, R|W|TX  ; Inform debugger that this section holds code
(.text).
ma 0x87FE00, 0x2000, RAM
ma 0x900000, 0x8000, RAM
map on                     ; Make emulator aware of this memory configuration.
?*0x808064 = 0x2F0000      ; Set STRB0 control register to STRB0 and STRB1
                           ; overlay.
                           ; 32-bit memory width, 32-bit data size
                           ;
?*0x808068 = 0xD0000       ; Set STRB1 control register.
                           ; 32-bit memory width, 16-bit data size
                           ;
load sample.out            ; Configure STRB0/STRB1 control registers before
                             loading code.
```

4.8 Booting a TMS320C32 Target System in a C Environment

A DSP system uses a boot procedure following power-up or reset to initialize the system volatile memory (such as SRAM) with the application program/data and to start execution of the application code. The SRAM loads from a nonvolatile medium (EPROM) or from a PC development platform using a debugger/loader program. The loader uses an emulator cable to move the load file from the PC hard disk to the SRAM on the DSP target board. An EPROM boot causes the DSP to start program execution directly from 16- or 32-bit EPROM (microprocessor mode). A hard-wired on-chip boot loader program copies the boot table from the 8-bit EPROM to internal or external SRAM and then starts execution from the SRAM (microcomputer/boot loader mode).

TI supports four ways to boot a DSP system following power-up/reset. Each boot procedure uses a different combination of 'C32 silicon features, software, and hardware tools. Each combination forms an integrated development environment that includes features to support most system boot requirements.

A boot development flow includes two major tasks:

1) Use C source debugger and assembly level tools to compile, assemble and link the boot code/data to create a binary common object file format (COFF) executable object.

2) Load the COFF file into the DSP target system.

Generating the COFF file (linker output .out file) uses the same flow for all boot methods.

4.8.1 Generating a COFF File

Generating a COFF file requires compiling the source code with the C compiler, then assembling and linking the resulting assembly files, with the assembly level tools. A text editor creates additional assembly files or the files are extracted from the RTS30 library. The linking process resolves all external references between program files and generates the .out COFF file subject to specified options (such as –c or –cr boot options).

4.8.1.1 Compiler

Figure 4–37 on page 4-89 shows how one or more C files are compiled into multiple assembly files. Each assembly file is constructed from former C functions that were individually decomposed into standard logical sections:

- The program code is assigned to *.text*.
- The stack is assigned to *.stack*.
- Dynamically allocated memory is assigned to *.sysmem*.
- The switch tables are assigned to *.const*.
- Uninitialized variables are assigned to *.bss*.
- initialized variables are assigned to *.cinit*.

If, following system reset, the program executes directly out of EPROM (microprocessor mode), a separate assembly file holds the reset vector (and possibly other interrupt vectors). The reset vector points to the address contained in the c_int00 symbol that the linker resolves with the beginning of the BOOT.ASM routine (from the RTS30 library).

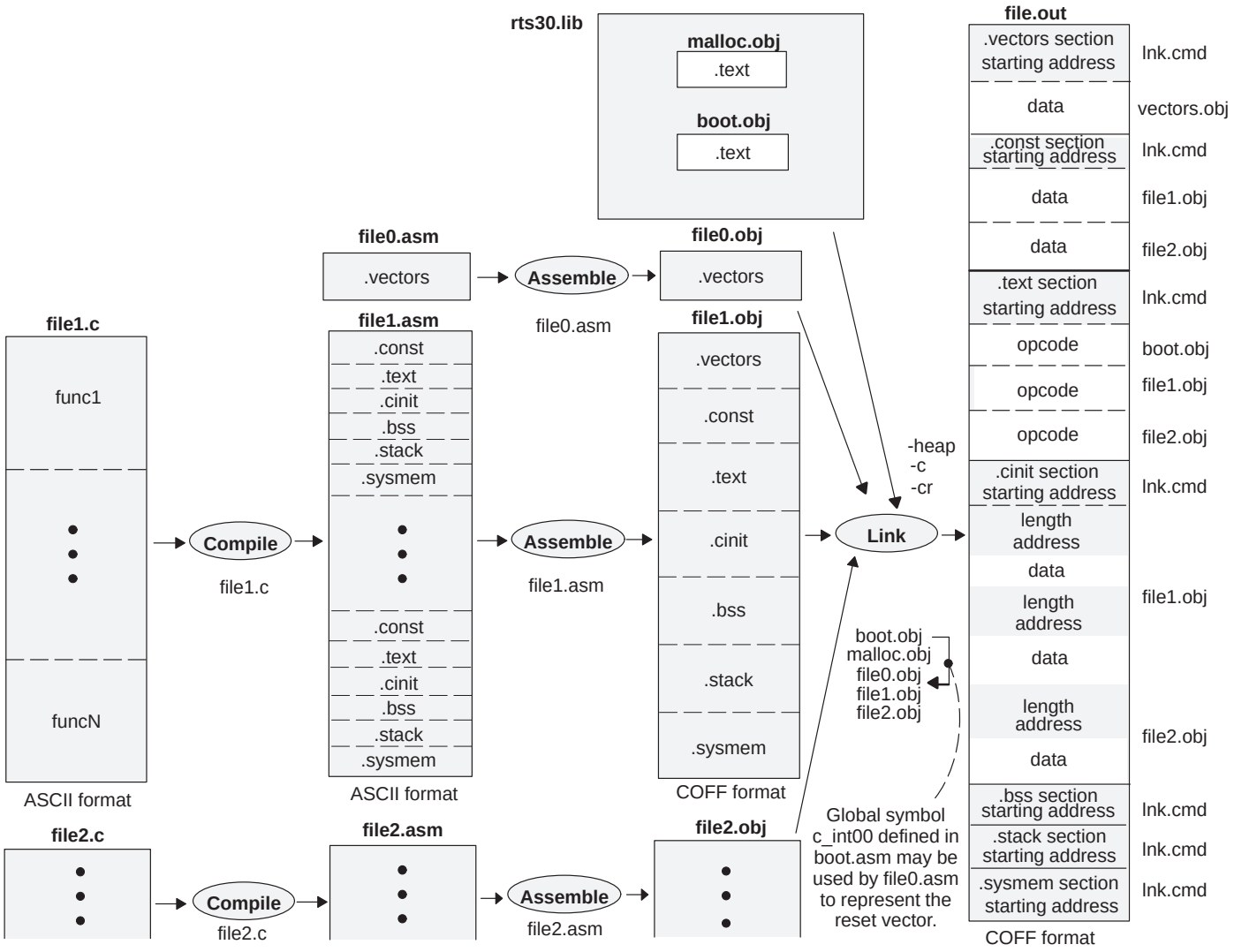
4.8.1.2 Assembler

The assembler assembles all .asm files into their respective .obj files. Since each .asm file may have a .text section fragment for each function in the file, its .obj counterpart groups all the fragments into a single .text section. This applies to all sections in that file. The results of the assembler process are multiple .obj files composed of single instances of all standard C sections. In addition to the object files generated by the user, the subsequent boot procedures require another .obj file. The boot.asm file can be extracted from the RTS30 library and assembled separately into boot.obj. The boot.obj is the first routine executed following reset. It initializes the C environment by setting up the system stack, processing initialized variables, setting up the page pointer, and calling the main function. While boot.asm file is required for a C program, other files may be extracted from the library, such as malloc.asm, which is used to allocate additional memory at run time.

4.8.1.3 Linker

The linker assigns physical addresses to logical program sections from .obj files. A linker command file defines the available physical memory segments using the MEMORY directive, assigns one or more sections to individual memory segments using the SECTIONS directive, and lists all object files containing sections to be processed. The order in which object files are listed is important and reflects the order in which individual sections are stacked in physical memory. For that reason, the boot.obj file must always be the first one listed, since it represents the execution entry point for every C program. The boot.obj global symbol c_int00 provides the entry address that can be resolved to other files that are linked with boot.obj (for example, the vector file that needs an address for the reset vector). Depending on the method, the linker can be invoked with the –c or –cr option. These two options control how a C program's initialized variables are handled during the later stages of the boot process. See the *TMS320C3x/C4x Assembly Language Tools User's Guide* for more information.

Figure 4–37. Compile, Assemble, and Link Flow



4.8.1.4 The .out (COFF) File

After resolving the external references among all program sections, the linker builds the .out file. The .out file is constructed in the binary COFF format, and it contains all the sections listed in the linker SECTIONS directive. It contains information about the program, information about how to load it into the target DSP system, and symbol information for the debugger that is later used to verify the code. All C and assembly symbols, such as subroutine labels, etc., can be made visible in the debugger window (by embedding them in the COFF file), provided that they are declared as global symbols and the appropriate options are used with the code generation tools.

Some .out sections contain only the starting addresses and no code or data. They include the .stack section for the system stack, the .sysmem section for dynamically allocated memory, and the .bss section for uninitialized data. The boot process also uses the .bss section as a destination for the initialized variables that are originally stored in the .cinit section of the .out file. Although they contain no data, the .stack and .sysmem sections are included in .out to allow the debugger tools to verify that the physical memory for those sections exists on the target board. Other sections in the COFF file, such as .vectors, .const, and .text, contain the starting addresses and the contents of the sections. When the debugger loads the .text section into the target system, for example, the opcodes for all assembly instructions for the entire program are copied, beginning at the section starting address.

The .cinit section is different because it contains initialized variables. Once the .out file is generated, it can be burned into a 16- or 32-bit-wide EPROM, and the program can start executing directly from that EPROM following reset (in the microprocessor mode). But if the initialized variables reside in the same EPROM, they are not really variables, since one cannot write to an EPROM device and actually change the values of those variables. For that reason, before user program execution begins, the boot.asm library routine copies the initialized variables from the EPROM .cinit section to the SRAM .bss section, one array of data at a time. Figure 4–37 on page 4-89 shows that the .cinit section is divided into individual array records; each array has a length, data content, and destination address in the SRAM .bss section. The .bss section is the final destination for initialized variables, while the .cinit EPROM section is a temporary holding place for use before power-up/reset. The .cinit section also stores the –c/–cr linker option selection for use in the later stages of the boot process.

4.8.2 Loading the COFF File to the Target System

When the COFF file is loaded into the DSP target system, program and data content, as well as control information, are extracted. Then the control information is used to place the program/data content in target memory. Some control information embedded in the COFF file may not apply directly to the program/data content. For example, the COFF file may include a symbol table for the debugger or a memory width control word for the on-chip boot loader.

Using the debugger to load the COFF file to target memory requires connecting the target board to the PC (on which the debugger is running) with an emulator cable and pod and then transferring the COFF file with the LOAD command. The linker –c/–cr options control processing of the .cinit section during the load operation.

The COFF file can also be loaded to a target system from an EPROM. The Hex30 utility converts the COFF file to an EPROM-programmer-compatible file that can be programmed to the EPROM. In the microprocessor mode, the program executes directly from the EPROM. In the microcontroller/boot loader mode, the on-chip boot loader first expands the EPROM contents into target SRAM and the program executes from SRAM. In either case, the C program begins execution at the start of the boot.asm library routine to initialize the C environment before the rest of the C program runs.

4.8.3 Debugger Boot

Figure 4–38 on page 4-93 and Figure 4–39 on page 4-94 show how to load the COFF file into the target system using the debugger load command.

The debugger is a standard TI software development tool that runs on a PC platform. The debugger accesses the target board through the PC emulator card and cable. The cable connects to the target board through a 12-pin connector that routes the signals to the DSP's emulation pins. The emulation pins control the operation of the modular port scan device (MPSD) scan chain in the processor. Depending on the command issued by the debugger, the emulation circuitry in the scan chain stops or resumes processor operation, examines/loads registers or memory, sets breakpoints, or executes code one instruction at a time (called single-step execution). The debugger LOAD command reads the COFF file from the PC hard drive, extracts program/data content, and transfers it through the emulator cable to the target board's memory.

4.8.3.1 RAM Model (Linker –cr Option)

When the COFF file is loaded into the target board's memory, most sections in the file are processed by copying the program/data to the address defined at the beginning of each section; however, the initialized variables in the .cinit section are processed differently. If the COFF file is generated by the linker using a –cr option, the .cinit section of the file is loaded using the RAM model (see Figure 4–38). The RAM model assumes that the target memory is composed exclusively of SRAM devices. Thus, the initialized variables can be directly copied to the SRAM .bss section, one array at a time, without first placing them in a temporary EPROM .cinit section. Once the initialized variables have been loaded into SRAM, they can be read or written to by the CPU without further initialization steps by boot.asm at the beginning of C program execution.

4.8.3.2 ROM Model (Linker –c Option)

If the COFF file is created with the linker –c option, the loader places the .cinit section in the target memory according to the ROM model. The ROM model copies the .cinit section as one block to the address specified at the beginning of the same .cinit section. Following the load operation, the ROM model expects the boot.asm routine (at the beginning of the C program) to further process the .cinit section by copying its contents to the SRAM .bss section, one array at a time. After the COFF load operation, the memory content is the same as that created by the RAM model with one exception: the target SRAM still contains the temporary .cinit section, which serves no purpose after it is processed by boot.asm. The ROM model can still be useful; for example, it is useful to simulate the microprocessor-mode EPROM boot (see Figure 4–39). During the development cycle, instead of burning a new EPROM each time the code is modified, the EPROM can be removed and replaced with an equivalent SRAM device (by reconfiguring jumpers). The ROM model allows use of the loader to quickly load and debug the modified code while preserving the bus activity at power up to simulate an EPROM boot.

Figure 4–38. Loading C Object File into TMS320C32 Memory (Linker –cr Option)

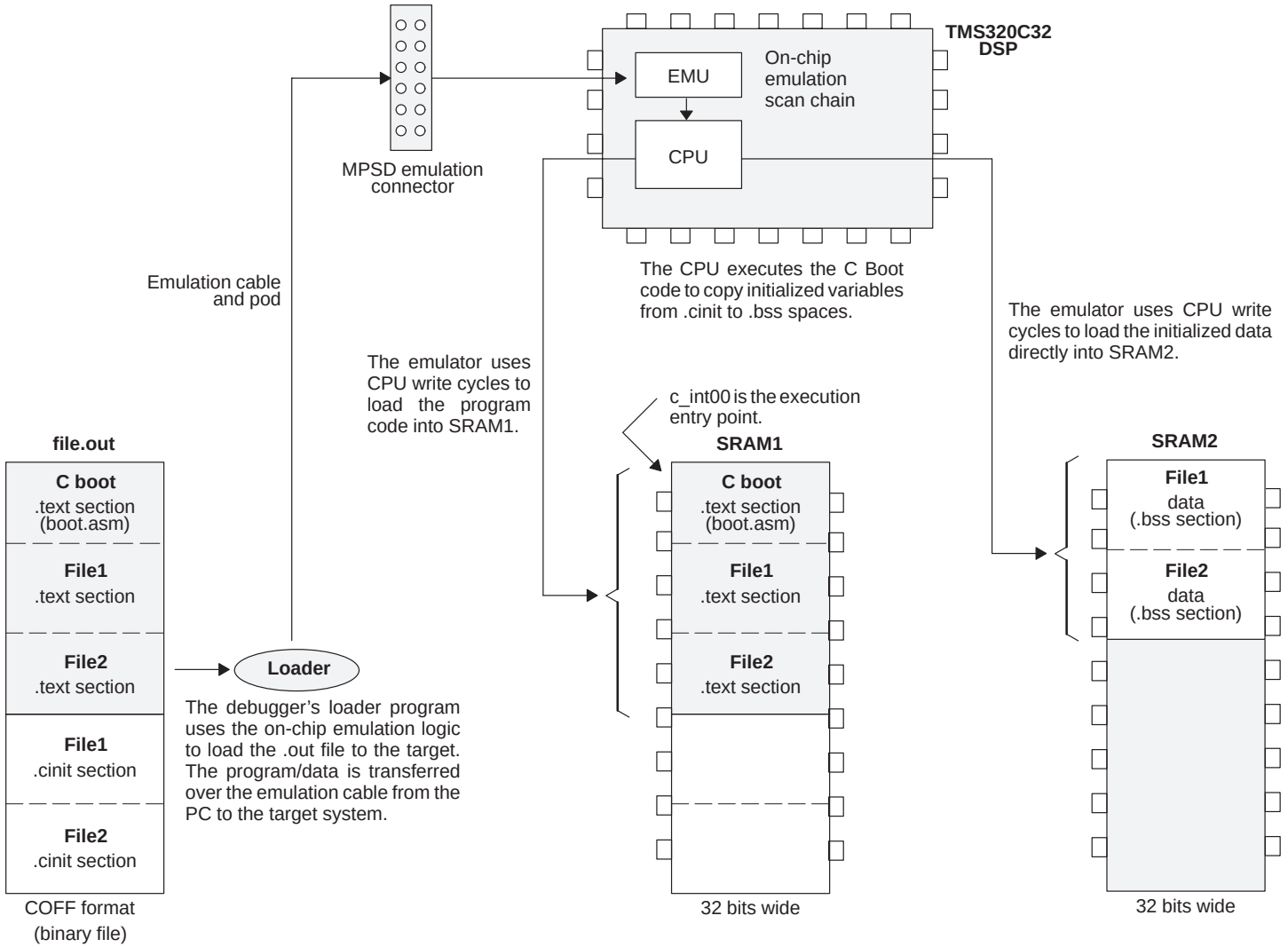
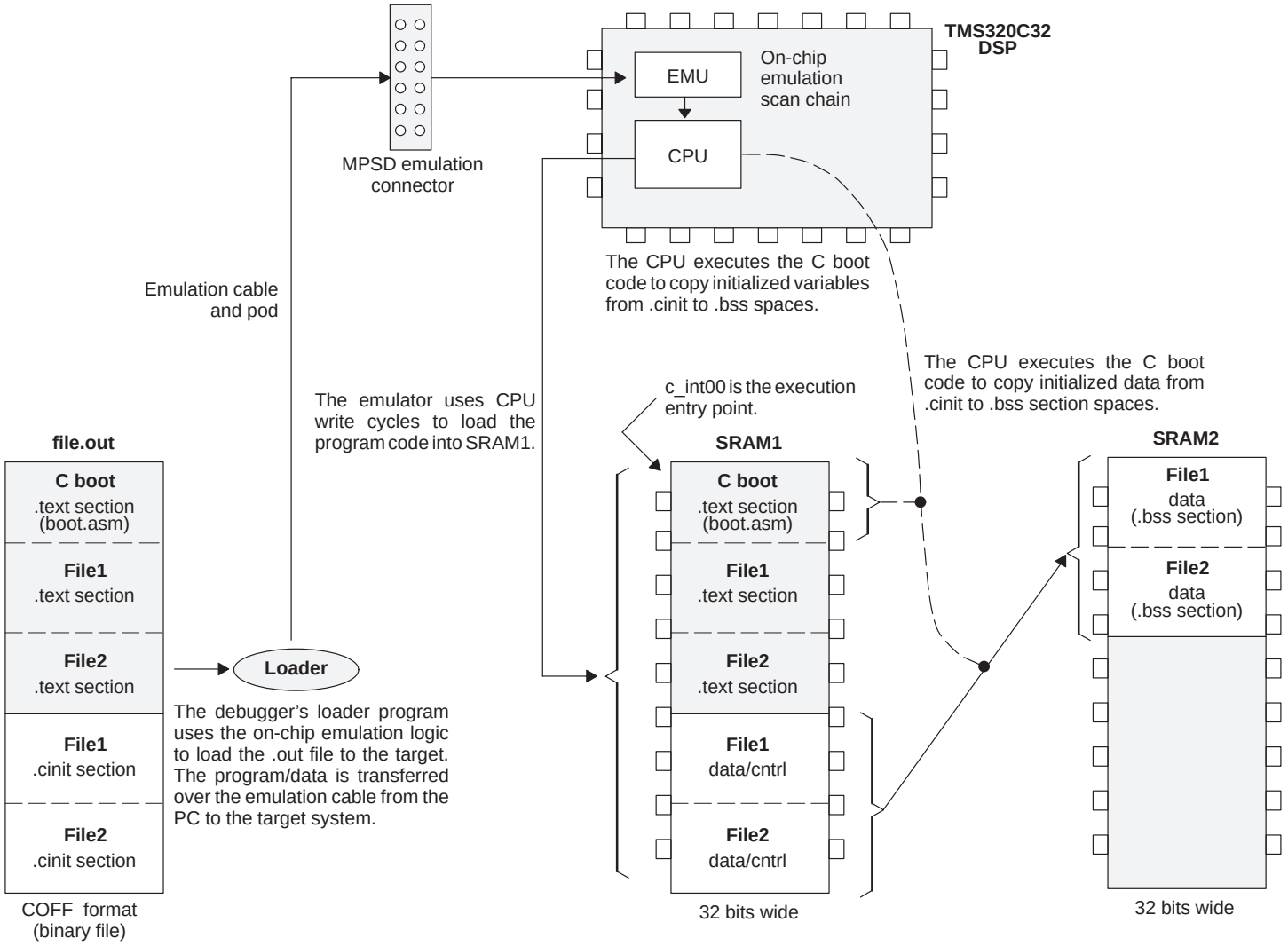


Figure 4–39. Loading C Object File into TMS320C32 Memory (Linker –c Option)



4.8.4 EPROM Boot

Booting a DSP target board from C code stored in nonvolatile memory and accessible to the DSP can be done in two ways. If the DSP is powered up in the microprocessor mode, the reset causes the program to start execution from 32- or 16-bit EPROM by fetching the reset vector from memory address 000000h and branching to the reset interrupt service routine (ISR) pointed to by that vector.

On the other hand, if the DSP is powered up in the microcomputer/boot loader mode, program execution starts with the on-chip boot loader program. The boot loader reads the COFF file from an 8-bit EPROM and expands it to the system SRAM from which it can be executed (16 or 32 bits wide). In either case, program entry occurs at the beginning of the boot.asm library routine to initialize the C environment prior to execution of the C code.

4.8.4.1 Microprocessor Mode (Linker –c Option)

Before the binary COFF file can be burned into an EPROM, it must be converted to an ASCII format that an EPROM programmer can recognize (see Figure 4–40 on page 4-97). The hex conversion utility converts COFF files to a programmer object file format such as Intel™ Hex. The EPROM programmer uses the converted files to program one or more EPROMs that can be inserted into the DSP target board.

If the linker –c option is used to create the COFF file (ROM model), the hex utility copies the .cinit section directly into the programmer object file without processing its content. In other words, the .cinit section in the programmed EPROM contains the initialized data as well as destination addresses and lengths in .bss for individual .cinit data arrays. To start program execution from EPROM at power up, the DSP must be configured in the microprocessor mode by pulling the MCBL/$\overline{\text{MP}}$ pin low. Triggered by the low-to-high transition of the $\overline{\text{RESET}}$ pin, the DSP executes the reset vector fetch read cycle. The reset vector points to the boot.asm routine, which is executed next. The linker –c option sets a control bit in the .cinit section of the COFF file.

When the boot.asm program executes the .cinit section, it checks the –c/–cr control bit. The –c option (ROM model) causes boot.asm to copy the contents of each array within the .cinit section to its destination in the .bss section mapped to SRAM. The initialized variables must be copied from EPROM to SRAM at the beginning of program execution, because they cannot be modified in EPROM (variable data must be changeable during program execution).

4.8.4.2 Microcomputer/Boot Loader Mode (Linker –cr Option)

The ’C32 features an on-chip hardwired boot loader program in the internal programmable logic array (PLA). The boot loader reduces the DSP target board cost by replacing multiple fast EPROMs with a single 8-bit slow (inexpensive) EPROM. Because the ’C32 cannot execute code from memory that is only 8 bits wide, the on-chip boot loader program reads the boot table from the byte-wide EPROM and reconstructs all sections of the original COFF file one byte at a time before placing the program/data in SRAM (see Figure 4–41 on page 4-98).

To power up the DSP in the boot loader mode, the MCBL/$\overline{\text{MP}}$ pin must be held high when the $\overline{\text{RESET}}$ signal is deasserted. At that stage, the DSP starts executing the boot loader code from internal address 000045h. Immediately after it starts execution, the boot loader checks the interrupt flag (IF) register. All interrupts are disabled and remain disabled until the application program enables them. Depending on which external interrupt is asserted, the boot loader looks for the boot table at one of three external memory locations or at the serial port. The interrupt pins carry a message to the boot loader telling it where to get the boot table after reset.

The boot table structure resembles the COFF file from which it was derived by the hex conversion utility. The main feature that distinguishes the boot table from a regular hex utility output (such as the microprocessor mode boot example) is that in addition to the contents of the COFF sections, the boot table includes special control words for the on-chip boot loader program to instruct it on how to assemble and load those sections. Each section is built into a block preceded by three control words: block size, destination address, and destination memory width/data size. Multiple blocks can be transferred to selected parts of the DSP memory map. To format the COFF file into the boot table, the program section to be booted must be identified to the hex conversion utility with the SECTIONS directive. The boot table is constructed of the COFF sections identified in the SECTIONS directive and marked with the boot option (see Figure 4–41).

If the linker uses the –cr option to create the COFF file, the hex utility processes the COFF .cinit section and assigns the addresses in the .bss section to the corresponding .cinit arrays in the boot table. Every C program starts execution with the boot.asm routine, but because one of the boot.asm control flags indicates that the COFF file was created with the linker –cr option, the code skips transfer of .cinit contents to .bss. The hex utility performs that task by placing all the initialized variables in .bss while creating the boot table without relying on boot.asm to make the transfer at run time (see Figure 4–41).

Figure 4–40. 32-Bit EPROM Boot in the Microprocessor Mode (Linker –c Option)

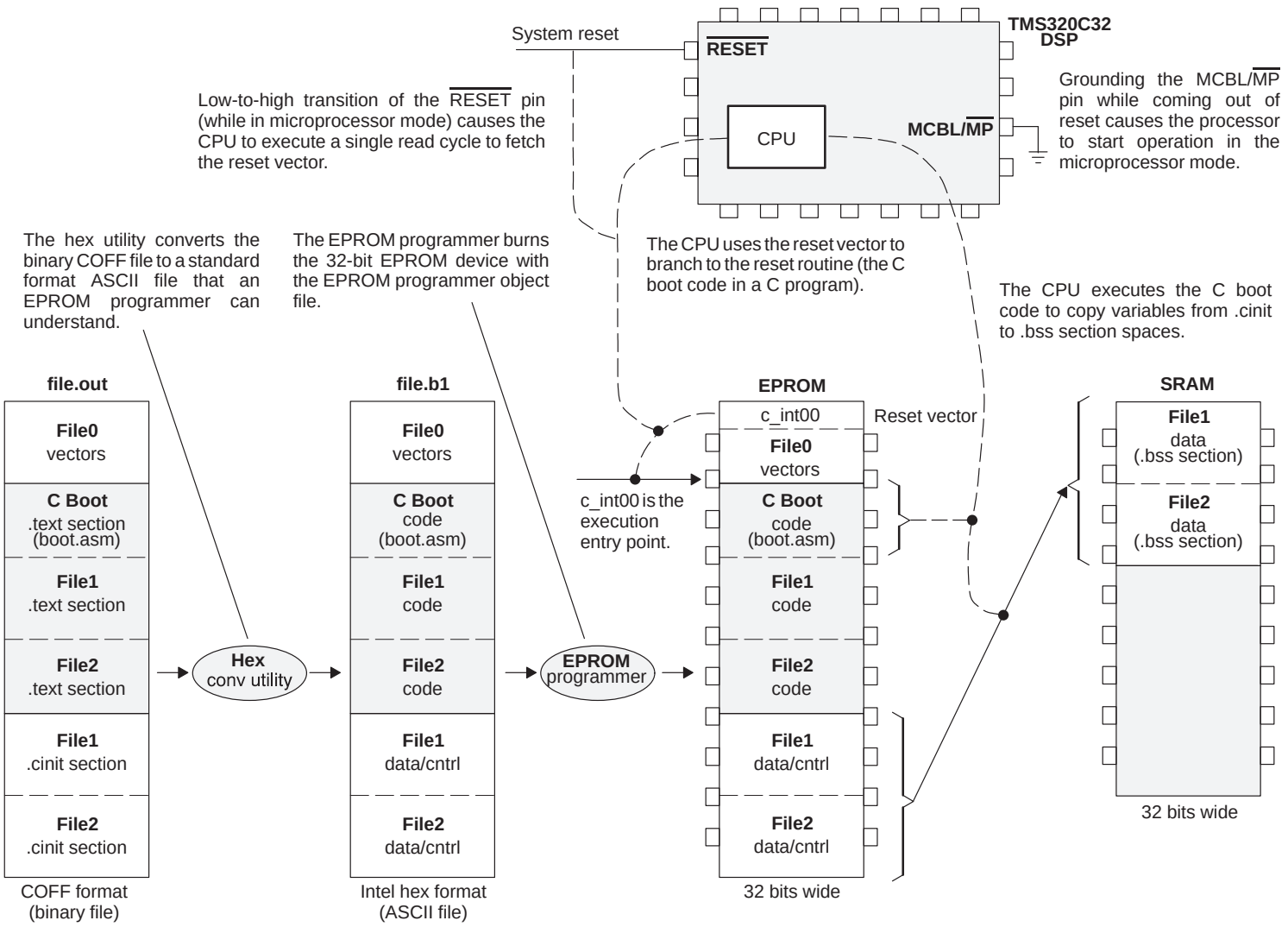
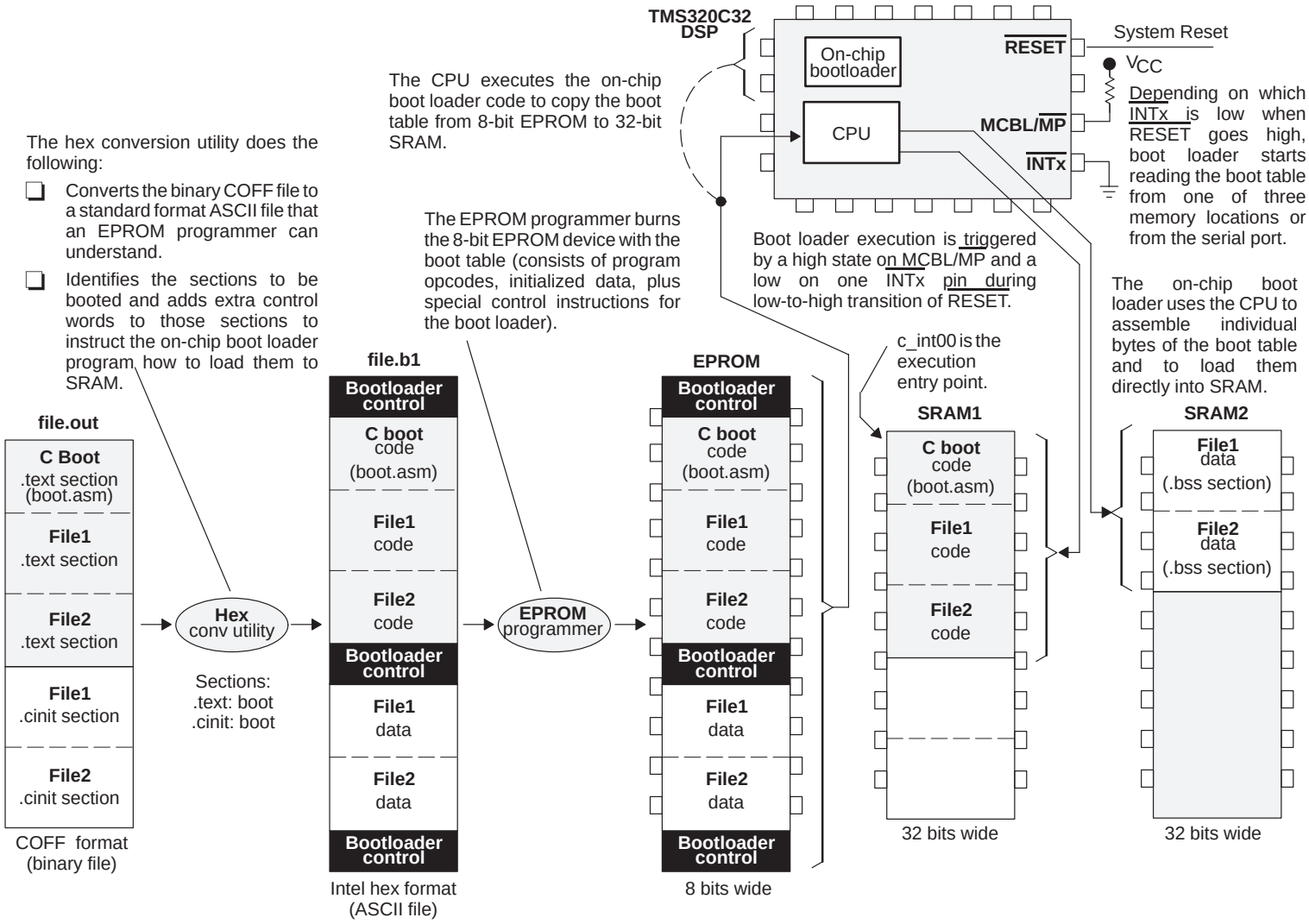


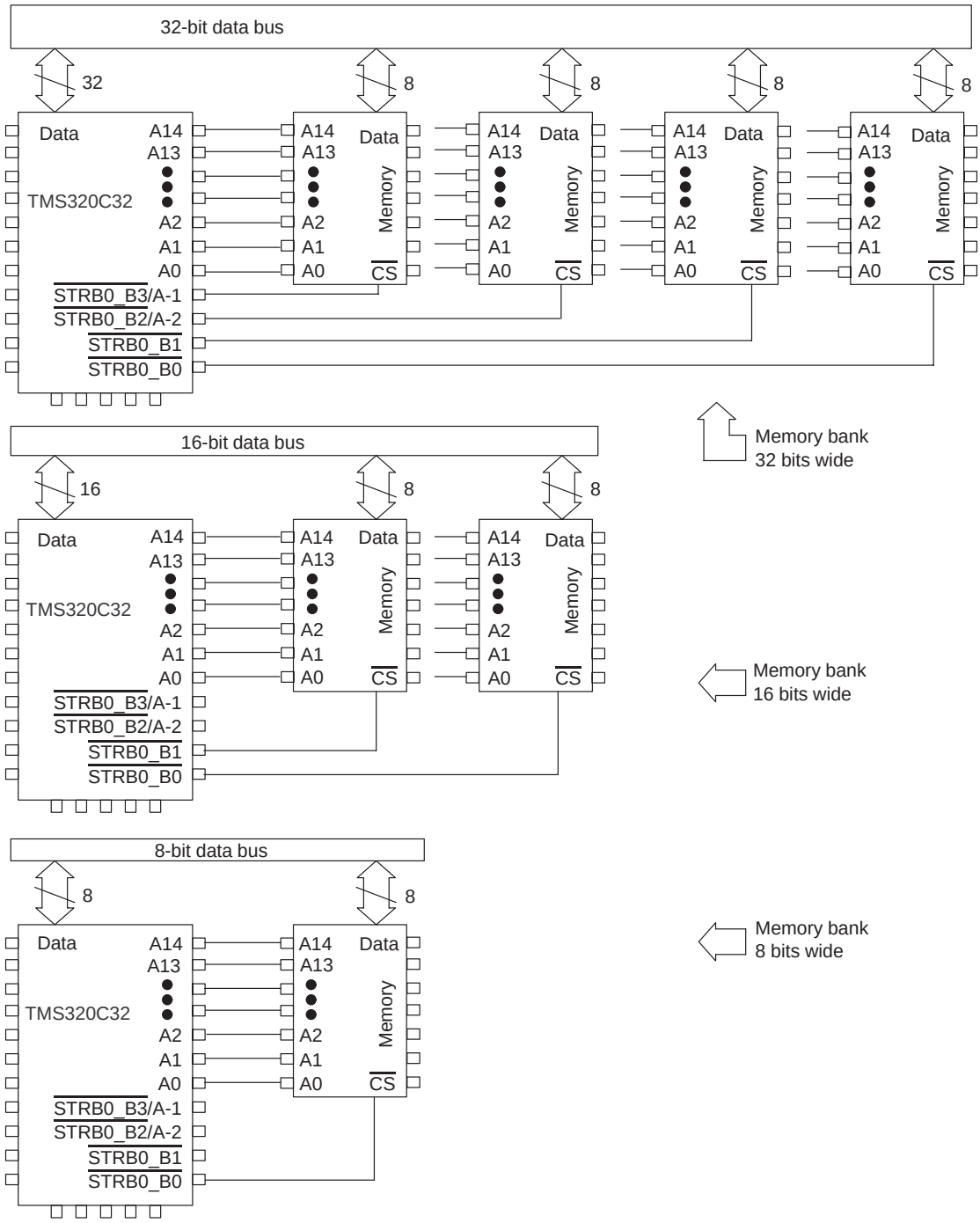
Figure 4–41. 8-Bit EPROM Boot Using the On-Chip Boot Loader (Linker –cr Option)



4.8.5 Boot Table Memory Considerations

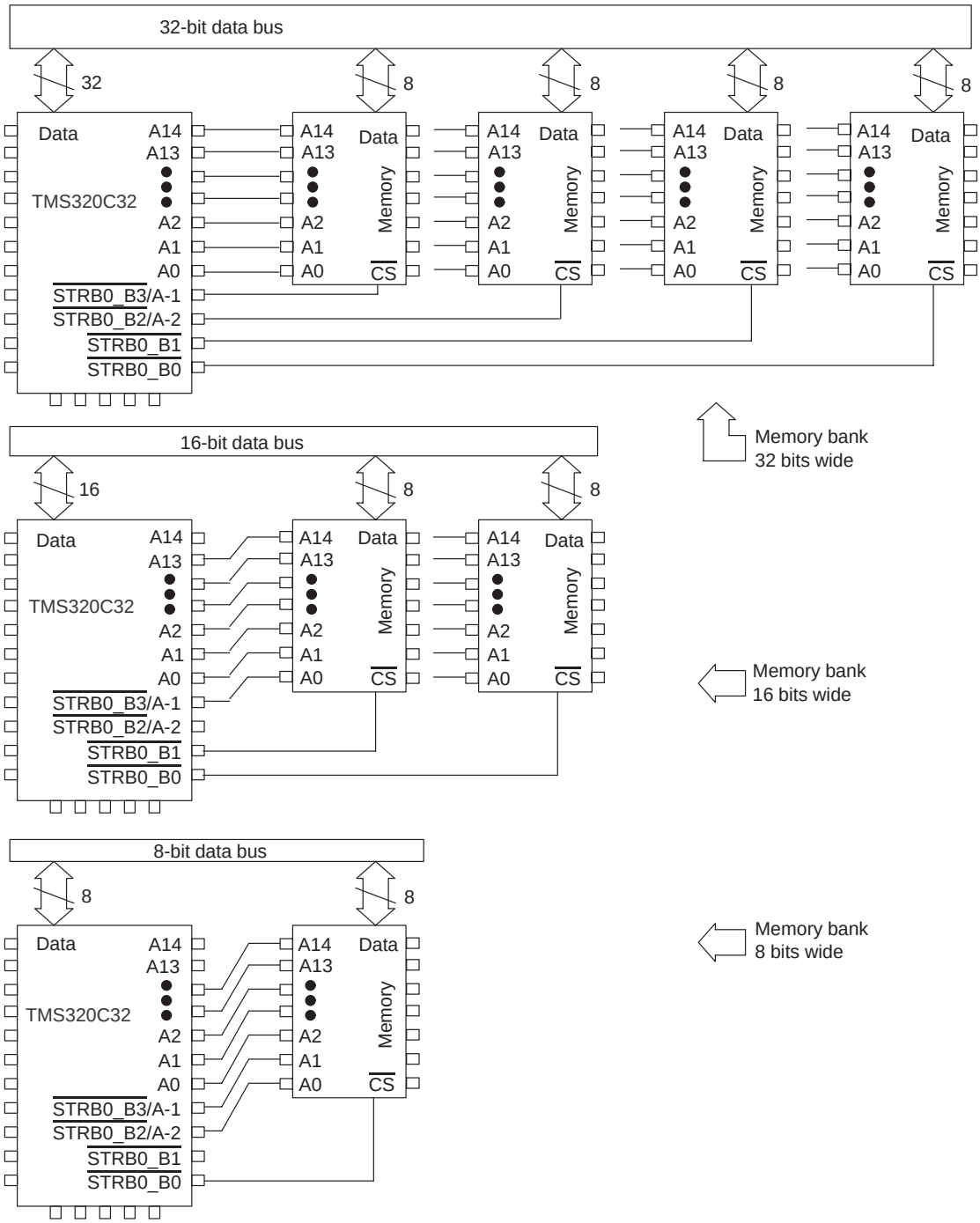
There is a significant difference in the methods of interfacing the external memory holding the boot table and the program/data memory used during normal code execution. The address presented on the ’C32’s pins may be shifted by one or two bits, depending on the size of the memory bank (see Figure 4–42), but the external memory holding the boot table must have *no* address shift relative to the ’C32 address pins, regardless of the width of the boot memory (see Figure 4–43). The boot loader program reads the boot table memory width from the first word of the boot table. It reads the boot table contents as 32-bit data, and, depending on the memory width, it reconstructs the program and data before sending them to the memory map. Because of this difference in the address shift, the byte-wide EPROM containing the boot table is not best suited to store normal data unless special hardware is added to handle the address shift.

Figure 4–42. Memory Configuration for Normal Program Execution



Note: The boot table memory used by the on-chip boot loader should be connected to the 'C32 with no address shift, regardless of the width of the memory bank.

Figure 4–43. Boot Table Memory Configuration



Note: For external memory used during normal program execution, the amount of external address shift depends only on the width of the memory bank.

4.8.6 Host Load

While some DSP systems stand alone, others may be embedded DSPs controlled by a host, such as a microcontroller or another DSP. During system power up, the DSP boot table may be transferred from the host to the DSP through a serial port or through a byte-wide latch. This eliminates the need for a dedicated boot EPROM on the DSP side of the system. On the host side, the DSP boot table may be temporarily stored in an EPROM, prior to the DSP boot. Following reset, the host transfers the boot table to the DSP to initialize it and start program execution.

4.8.6.1 Boot From Serial Port

If the DSP powers up in the microcomputer/boot loader mode (MCBL/$\overline{\text{MP}}$ high), the low on the $\overline{\text{INT3}}$ pin and high on all other $\overline{\text{INTx}}$ pins causes the on-chip boot loader program to read the boot table from the serial port. Most microcontrollers also feature a serial port, and in many cases the two ports can be connected directly without additional glue logic for an economical host/DSP interface. Following the boot, the serial channel can also be used by the host to send/receive data and to control the operation of the DSP (see Figure 4–44 on page 4-104). Generating the boot table requires linking the object files with the –cr option (RAM model) and then appending the hex utility's SECTIONS directive with the boot keyword to identify the COFF sections to be included in the boot table.

4.8.6.2 Boot From a Latch

If the host processor does not have a serial port, the DSP can be booted from the host using an 8-bit latch. During the boot operation, the host feeds the boot table bytes to the latch on one side, while the DSP reads the data from the other. Following reset, interrupts 0, 1, and 2 direct the DSP boot loader to the latch address. The same interrupts cause the boot loader to read from the parallel port, so some control/decode logic is required to make the DSP read from memory instead of from a latch. The same glue logic must also be connected to the host side of the latch to ensure proper data-transfer synchronization between two asynchronous systems (see Figure 4–45 on page 4-105). At power up, the DSP boot table most likely resides in the host's EPROM, and the host outputs the boot table to the latch one byte at a time following reset. Creating the boot table for this operation uses the same linker/COFF options as for the host/serial boot and the direct EPROM boot.

4.8.6.3 Asynchronous Boot From a Communications Port

If the host processor has an asynchronous communications capability, then the ’C32 can make a glueless connection to the host’s communication port (see Figure 4–46 on page 4-106). In addition to the data bus, three ’C32 pins are involved in the asynchronous boot: XF0, XF1, and $\overline{\text{IACK}}$. The XF1 pin serves as the data ready input to the ’C32, and XF0 is the data acknowledge.

The $\overline{\text{IACK}}$ pin pulses when there is no valid data present on the data lines (which are needed for the ’C4x comm-port interface). For boot loader mode, it is assumed that the host (such as a ’C4x) connects directly to the data ready and data acknowledge control lines. The host drives the data ready signal low to indicate to the DSP that the next byte of the boot table has been placed on the data lines. The DSP responds by pulling the data acknowledge signal low after reading the data. When the host sees the data acknowledge signal, it stops driving the data bus and brings the data ready line high. To complete the handshaking transaction, the DSP brings the data acknowledge signal high to request the next byte from the host. The boot table for this type of boot operation is created with the linker –cr option (RAM model) and hex conversion utility SECTIONS directive boot keyword — the same options used for other boot load procedures involving the on-chip boot loader program.

Figure 4–44. Boot From Host Using Serial Port (Linker –cr Option)

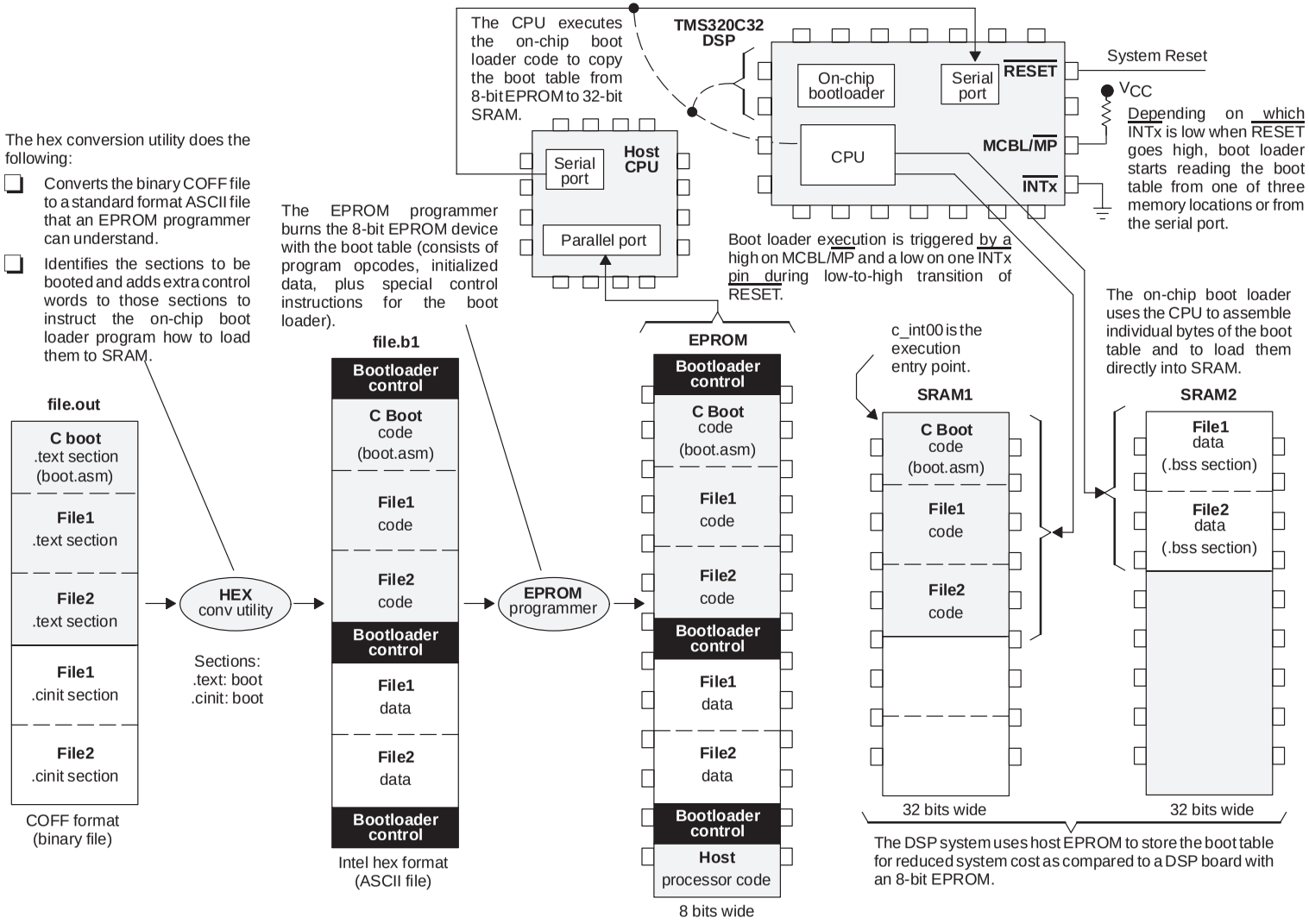


Figure 4–45. Boot From Host Using an 8-Bit Latch (Linker –cr Option)

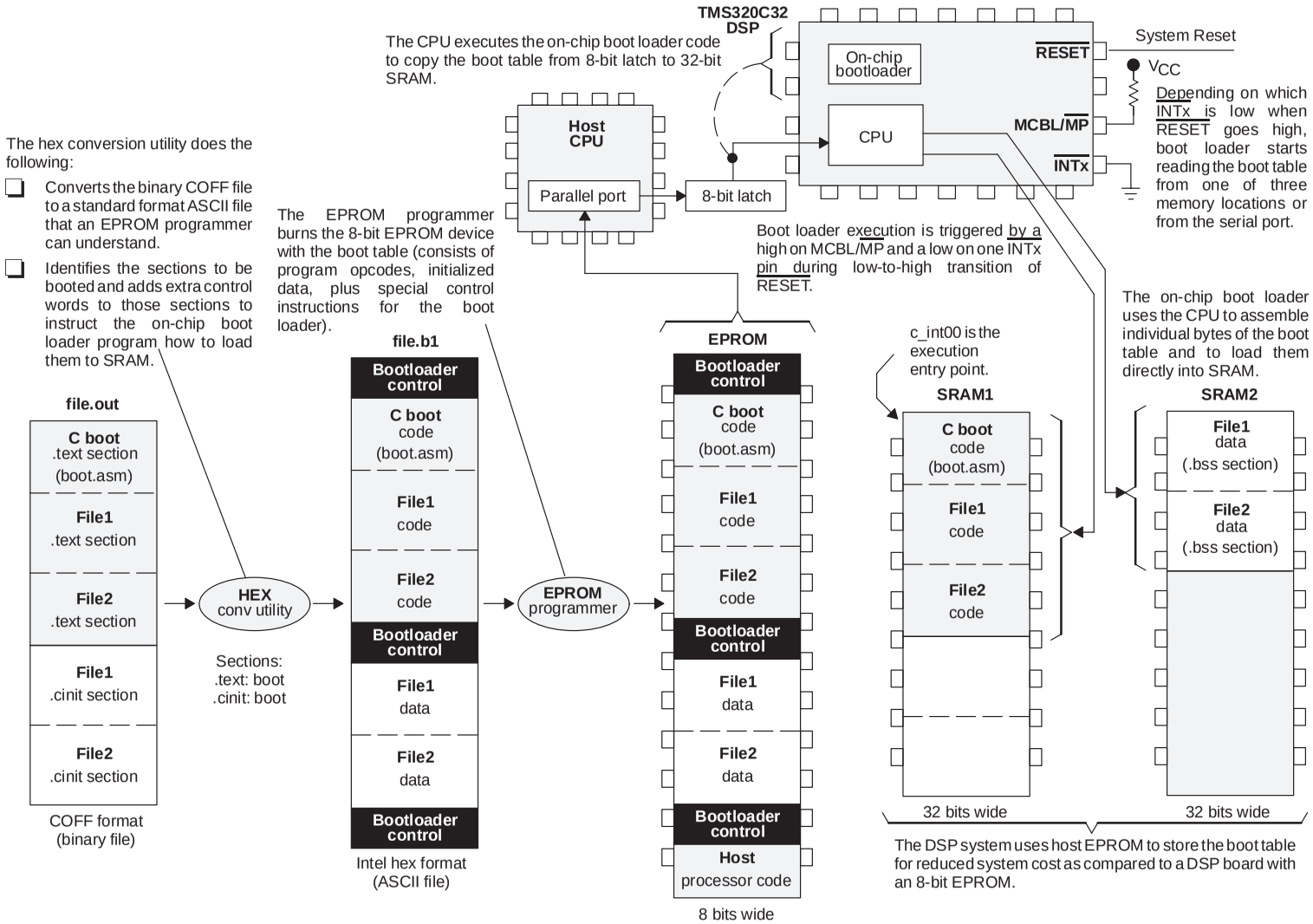
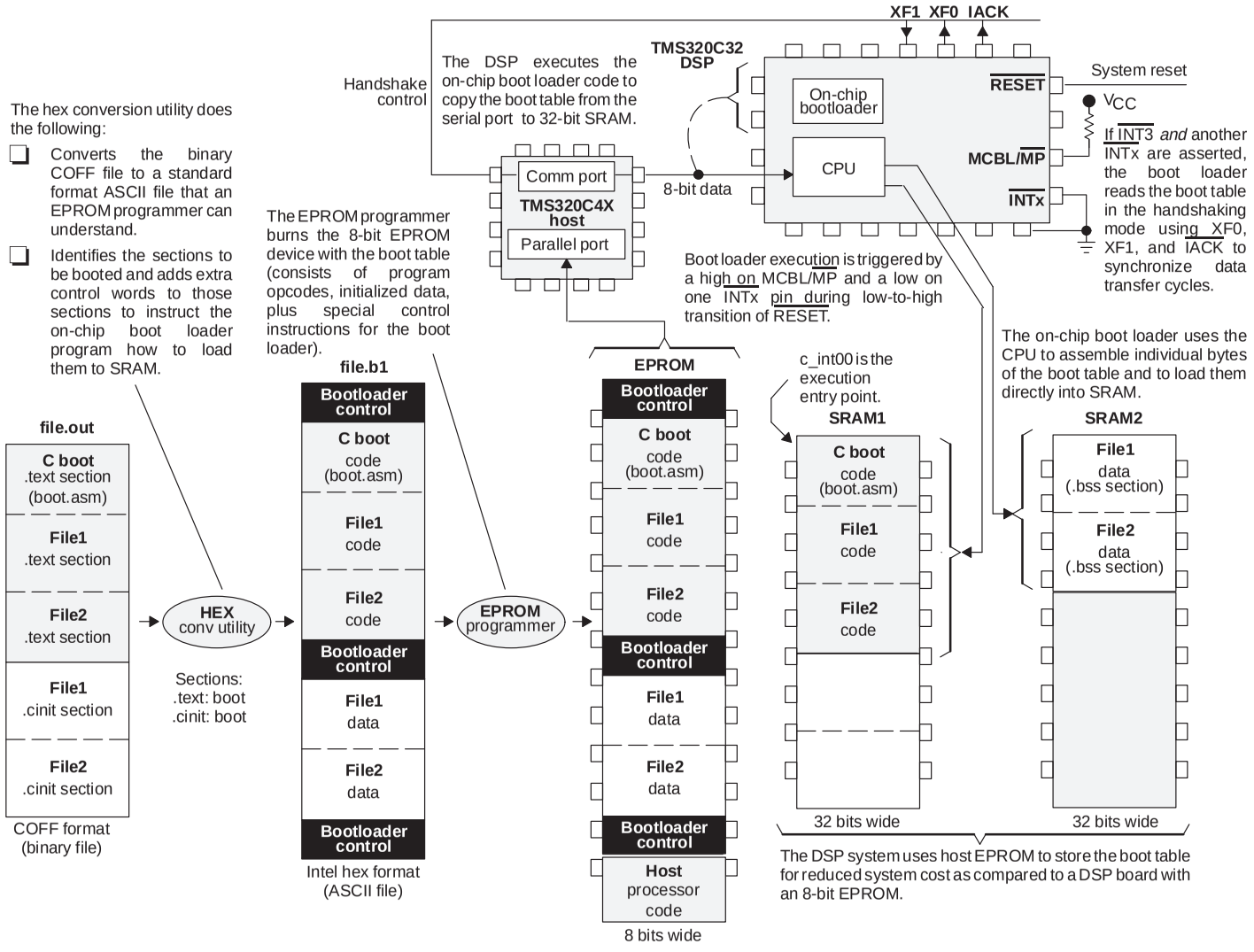


Figure 4–46. Boot From Host Using Asynchronous Communications Port (Linker –cr Option)



4.9 TMS320C30 Addressing up to 68 Gigawords

The ’C30 primary bus has 24 address lines which allow addressing up to 16 megawords of memory. The ’C30 expansion bus has 13 address lines addressing 8K words. These two busses, expansion bus address lines [XA(12-0)] and the primary lines [A(23-0)], can be used simultaneously to extend the address to 36 bits. This is accomplished by using the feature of the ’C3x family that holds the past address bits on an external bus until a new external access occurs. That means, the address bus works as a latch. Figure 4–47 shows how these two busses are combined together. The following parallel instruction accomplishes this task:

```
      STI   Rx,*ARn      ; address MSTRB while loading a
                         ; value from STRB memory
  ||  LDI   *ARp,Rq      ;
```

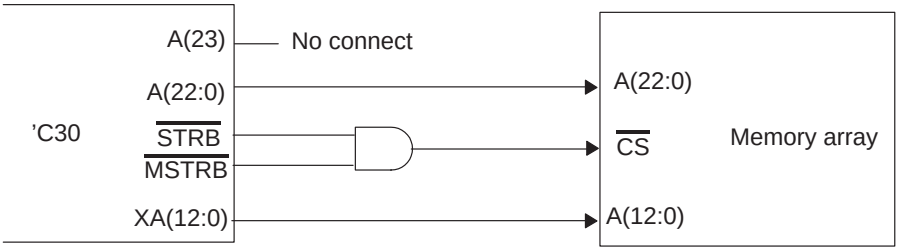
where:

Rx and Rq designate registers R0 to R7 (but not the same register)
ARn and ARp designate auxiliary registers AR0 to AR7 (but not the same register).

Note:

ARn contains the 8-Mword segment address plus 800000h. ARp contains the address within the 8-Mword segment and is between 0 and 7FFFFFh.

Figure 4–47. TMS320C30 Combination of Primary and Expansion Busses to Address 68 Gigawords



Chapter 5

Programming Tips

Programming style reflects personal preference. The purpose of this chapter is not to impose any particular style, but to highlight features of the ’C3x that can produce faster and/or shorter programs. The tips cover the C compiler, assembly language programming, and low-power mode wakeup.

Topic **Page**

5.1 Hints for Optimizing C Code

The ’C3x was designed with a large register file, software stack, and memory space that easily supports the floating point C compiler. The C compiler translates ANSI C programs into assembly language source code. It also increases code portability and decreases application porting time.

After writing your application in C language, debug the program and determine whether it runs efficiently. If the program does not run efficiently:

- Use the optimizer with –o2 or –o3 options when compiling
- Use registers to pass parameters (–ms compiling option)
- Use inlining (–x compiling option)
- Remove the –g option when compiling
- Follow some of the efficient code generation tips listed below

Identify places where most of the execution time is spent and optimize these areas by writing assembly language routines that implement the functions. Call the routines from the C program as C functions.

The efficiency of the code generated by the floating-point compiler depends to a large extent on the compiler options used 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

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

- **Precompute subexpressions.** This especially applies to 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 two 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 requires one extra instruction and incurs two 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

```
/* Inefficient Method  */
  int a[1000000]l; /* Inefficient */
  ...
  a[i] = 10;


/* Efficient Method  */

  int *a = (init *)malloc(1000000) ; /* Efficient */;
  ...
  a[i] = 10;
```

5.2 Hints for Assembly Coding

Each program has unique requirements. Not all possible optimizations are appropriate in every case. You can use the suggestions in this section as a checklist of available software tools.

- **Use delayed branches.** Delayed branches execute in a single cycle; regular branches execute in four cycles. The next three instructions are executed whether the branch is taken or not. If fewer than three instructions are required, use the delayed branch and append No-operation instructions (NOPs). A reduction in machine cycles still occurs.
- **Apply the repeat single/block construct.** In this way, loops are achieved with no overhead. Nesting such constructs does not normally increase efficiency, so try to use the feature on the most often performed loop. Note that the RPTS instruction is not interruptible and the executed instruction is not refetched for execution. This frees the buses for operand fetches.
- **Use parallel instructions.** It is possible to perform a multiply in parallel with an add (or subtract) and to execute stores in parallel with any multiply or arithmetic logic unit (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. It is possible to have loads in parallel with any multiply or add (or subtract) by multiplying by 1 or adding a 0. Therefore, to implement parallel instructions with a data load, substitute a multiply or an add instruction with one extra register containing 1 or 0, respectively, in place of a 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 the registers in addressing modes.
- **Use the cache.** This is especially important in conjunction 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 x 32 bits RAM and 4K x 32 bits ROM) is considerably faster to access. In a single cycle, two operands can be brought from internal memory. You can maximize performance if you use the direct memory access (DMA) in parallel with the CPU to transfer data to internal memory before you operate on it.
- **Avoid pipeline conflicts.** For time-critical operations, make sure you do not miss any cycles because of pipeline conflicts.

The preceding checklist is not exhaustive, and it does not address the detailed features outlined in other chapters of this manual. To learn how to exploit the full power of the ’C3x, study the architecture, hardware configuration, and instruction set of the device described in the *TMS320C3x User’s Guide*.

5.3 Low-Power Mode Wakeup Example

There are two instructions by which the ’C31, ’LC31, and ’C32 are placed in the low-power consumption mode:

- IDLE2
- LOPOWER

The LOPOWER instruction slows down the H1/H3 clock by a factor of 16 during the read phase of the instruction. The MAXSPEED instruction wakes the device from the low-power mode and returns it to full frequency during MAXSPEED’s read cycle. However, the H1/H3 clock may resume in the phase opposite to the one it was in before the clocks were shut down.

The IDLE2 instruction has the same functions that the IDLE instruction has, except that the clock is stopped during the execute phase of the IDLE2 instruction. The clock pin stops with H1 high and H3 low. The status of all the signals remains the same as in the execute phase of the IDLE2 instruction. In emulation mode, however, the clocks continue to run, and IDLE2 operates identically to IDLE. The external interrupts INT(0–3) are the only signals that start up the processor from the mode the device was in. Therefore, you must enable the external interrupt before going to IDLE2 power-down mode (see Example 5–4). If the proper external interrupt is not set up before executing IDLE2 to power down, the only way to wake up the processor is with a device reset.

Example 5–4. Setup of IDLE2 Power-Down Mode Wakeup

```
*
*   TITLE IDLE2 POWER-DOWN MODE WAKEUP ROUTINE SETUP
*
*   THIS EXAMPLE SETS UP THE EXTERNAL INTERRUPT 0, INT0, BEFORE
*   EXECUTING THE IDLE2 INSTRUCTION. WHEN THE INT0 SIGNAL IS RECEIVED
*   LATER, THE PROCESSOR WILL RESUME FROM ITS PREVIOUS
*   STATE. NOTE: THE “INTRPT” SECTION IS MAPPED FROM THE
*   ADDRESS 0 FROM THE RESET AND INTERRUPT VECTORS.
*
        .sect  “INTRPT”
RESET   .word  START      ;  Reset vector
INT0    .word  INT0_ISR   ;  INT0 interrupt vector
INT1    .word  INT1_ISR   ;  INT1 interrupt vector
INT2    .word  INT2_ISR   ;  INT2 interrupt vector
INT3    .word  INT3_ISR   ;  INT3 interrupt vector
        :   :
        :   :
        .text
        :   :
        :   :
    LDP    @SP_ADR
    LDI    @SP_ADR,SP     ;  Set up stack pointer
    OR     01h, IE        ;  Enable INT0
    IDLE2                 ;  Set GIE = 1 and stop clock
        :   :
        :   :
        :   :
        :   :
INT0_ISR  RETI            ;  Return to instruction after IDLE2#define N 16
```

There is one cycle of delay while waking up the processor from the IDLE2 power-down mode before the clocks start up. This adds one extra cycle from the time the interrupt pin goes low until the interrupt is taken. The interrupt pin needs to be low for at least two cycles. The clocks may start up in the phase opposite the phase that they were in before the clocks were stopped.

5.4 Bit-Reversed Addressing in C

The C language does not have any construct to take advantage of the bit-reversed addressing feature of the ’C3x. To take advantage of this feature, Figure 5–1 shows the assembly instructions added to the C code to use bit-reversed addressing.

Figure 5–1. Bit-Reversed Addressing in C Code

```
#define N 16
int x[N] = { 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15 };
int y[N] = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
/* int bitrev(int m, intn); */

void main()
{
   int i;
   asm(”    PUSH    AR5”);
   asm(”    PUSH    AR0”);
   asm(”    LDI    8,IR0;          ; Initialize IR0 TO 1/2 N”);
   asm(”    LDI    @CONST+0,AR5    ; AR5 <– address of X[] ”);
   asm(”    LDI    @CONST+1,AR0    ; AR0 <– address of Y[] ”);

   for ( i=0; i<n; i++ ){
      /* y[bitrev(i,N) ] = x[i]; */
      asm(”    LDI    *AR5++(IR0)b, R0”);
      asm(”    STI    R0, *AR0++”);
      }
   asm(”  POP AR0”);
   asm(”  POP AR5”);
}
         /* These statements place x and y in .bss and make their
           addresses available via the CONST table. */
asm(”     .bss      CONST,2     ”);
asm(”     .sect     \”.cinit\”  ”);
asm(”     .word     2,CONST     ”);
asm(”     .word     _x          ”);
asm(”     .word     _y          ”);
```

5.5 Sharing Header Files in C and Assembly

Sometimes it is useful to be able to define named constants that can be used in both C and assembly language.

One method is to have separate header files that define the same symbols: a C include file with #define directives and an assembler include file with .set or .asg directives. However, it is more convenient to have a single, shared header file that defines symbols once for C and assembly.

Figure 5–2 shows how a file can be used normally as a C include file and also to generate an assembler include file. By compiling it and defining ASMDEFS, an assembler include file is generated from this file with the following command:

```
c130 –dASMDEFS –k defs.h
```

Figure 5–2. Input File defs.h

```
#define PI 3.14
#define E  2.72
#ifdef ASMDEFS /* IF DEFINED, CREATE .asg DIRECTIVES */
#define ASM_ASG(sym) asm(”\t.asg\t” VAL(sym) ”,” #sym
#define VAL(sym)     #sym
ASM_ASG(PI);
ASM_ASG(E);
#endif /*ASMDEFS*/
```

The output is the file defs.asm, which contains .asg directives for your symbols (see Figure 5–3).

Figure 5–3. Output File defs.asm

```
; ... <compiler–generated header stuff> ...
.asg 3.14,PI
.asg 2.72,E
```

You can then use .include in your assembly modules. The same technique can be used to create .set directives rather than .asg.

5.6 Addressing Peripherals as Data Structures in C

A data structure is usually assigned to the .bss section by the C compiler. A .bss section stores global and statically allocated variables. A peripheral, such as a serial port, has memory-mapped control registers with addresses different from .bss. To manipulate a memory-mapped peripheral register in C, follow one of the methods listed below.

- **Method 1:** Use a pointer to the peripheral.

Peripheral as memory locations

1) Declare a structure that logically represents the memory locations of the peripheral.

```
struct controller {
                unsigned int status;
                ...
               };
```

2) Declare a pointer to the structure and initialize it to the peripheral's address.

```
struct controller *IFperipheral = (struct controller *)0x808000;
```

3) In your code, access the peripheral's memory values indirectly.

```
IFperipheral–>status = 0;
```

- **Method 2:** Place the structure in its own section.

1) Declare a peripheral instead of a pointer.

```
struct controller IFperiph;
```

2) Use inline assembly to give the structure its own section.

```
asm("_IFperiph .usect  \"periph\", 128);
      /* 128 is size of struct */
```

This creates a user-defined section that can be linked to any address.

3) Use your linker command file to map the section to memory.

```
periph:  load = 0x808000
```

4) Address the structure elements directly.

```
IFperiph.status = 0;
```

Method 1 is very useful for addressing peripheral or memory buffers that are device specific. Method 2 is preferred for addressing peripherals or memory buffers which are not device specific (that is, peripherals are user specified). This method ensures the task of mapping and aligning user-specific peripherals and/or memory buffers to the linker. The choice depends on your individual application.

See section 5.7 for another method of placing the structure in its own section using #pragma directives.

5.7 Linking C Data Objects Separate From the .bss Section

The TMS320 DSP C compilers produce several relocatable blocks of code and data when C code is compiled. These blocks are called sections and can be allocated into memory in a variety of ways to conform to a variety of system configurations. The .bss section is used by the compiler for global and static variables; it is one of the default COFF sections that is used to reserve a specified amount of space in the memory map that can later be used for storing data. It is normally unitialized. All global and static variables in a C program are placed in the .bss section. For example, on the floating-point DSPs, you might want to link all of your variables into off-chip memory but place a frequently-used array in on-chip RAM block 0.

- **Method A:** Declare variable in a separate section.

  1) Declare the variable that is to be separated from the .bss section in a separate file. For example, declare a 32-word array, tapDelay [ ], in a file called *array.c* as follows:

```
/* File: ARRAY.C */
int tapDelay[32]
/* End of file */
```

  2) Declare the variable as extern in any file that makes a reference to it. Consider the following file, *test.c*, that makes a reference to the array declared in file *array.c* as follows:

```
/* File: TEST.C */
.
extern int tapDelay[ ];
.
void main(void)
{
  int i;
  .
  tapDelay[i] = 0;
  .
}
/* End of file */
```

3) In the linker command file, link this variable separate from the .bss section in the SECTIONS section. The following linker command file segment illustrates how to link the array tapDelay [ ] onto the 'C3x on-chip, dual-access data RAM block 0 while linking the rest of the global and static variables into part of a similar data RAM block 1:

```
/* File: TEST.CMD */
.
test.obj
array.obj
.
MEMORY
{
  .
  RAMB0:    origin = 0x809800,    length = 0x400
  RAMB1:    origin = 0x809c00,    length = 0x400
  .
}

SECTIONS
{
  .
  .bss          :{}                    >RAMB1
  .
  tapdelayline  :  {array.obj(.bss) }  > RAMB0

}
/* End of file */
```

❑ **Method B:** Declare variable in a #pragma DATA_SECTION.

1) Declare the variable that is to be separated from the .bss section in a #pragma DATA_SECTION. Consider the example described in Method A. The following code segment uses the DATA_SECTION pragma to declare a 32-word array, tapDelay [ ], that is placed separate from the other global and static variables:

```
/* File: TEST.C */
#pragma DATA_SECTION (tapDelay, ".tapdelayline")
int tapDelay[32];
.
.
void main(void)
{
  int i;
  .
  tapDelay[i] = 0;
  .
}
/* End of file */
```

2) In the linker command file, use the section name .tapdelayline to place the array tapDelay [ ] in RAM block 0. Separate it from the other global and static variables that are in the .bss section as follows:

```
/* File: TEST.CMD */
.
test.obj
array.obj
.
MEMORY
{
  .
  EXT0:    origin = 0x100,       len = 0x3f00
  RAM0:    origin = 0x809800,    len = 0x400
.
}

SECTIONS
{
  .
  .bss          : {}    EXT0
  .
  .tapdelayline  : {}    RAM0

}
/* End of file */
```

Method B is available in the floating-point DSP C compiler version 4.60 or greater. It is described in the *TMS320 Floating-Point DSP Code Generation Tools Release 4.70 Getting Started Guide*.

5.8 Interrupts in C

To use interrupts in C, you must write an interrupt service routine (ISR), initialize the interrupt vector table, and link these parts with the linker command file. These steps are described below.

Step 1: Write a C language interrupt service routine (ISR).

The C compiler requires that each ISR be named as follows:

```
void c_int0n(void)   /* n is the int number */
{
      /* a C function that is an ISR */
}
```

The interrupt routine must not return a value and has no arguments. The C compiler recognizes this naming convention and treats it as a normal ISR. This means it performs a context save of the necessary registers and returns from the routine via an RETI instruction.

A good practice is to include the interrupts in a separate file called ints.c or something similar. This allows a modular style, simpler maintenance, and software that is easy to understand.

Step 2: Initialize the interrupt vector table using either C or assembly language.

In microprocessor mode of ’C30 and ’C31, the first 0x40 addresses are reserved for the interrupt and trap vectors. Address 0 (zero) holds the address of the reset routine. If using the –C linker option, the RTS30.lib function boot.asm takes care of defining the reset function, but the vector table initialization is left to the user.

An assembly language routine might look like this:

```
; file name is vectors.asm
; .sect ”vectors”    ; a new section begins here
  .word _c_int00      ; the address of the reset
vector
  .word _c_int01     ; the ISR for interrupt 0
  .word _c_int02     ; the ISR for interrupt 1
; etc.
; end
```

This routine creates a new section that is merely a list of addresses where the interrupt routines can be found. It can be written in C by encapsulating each line in an asm statement.

For example:

```
asm(”   .sect  \”vectors\” ”);
A C function that is an ISR.
```

Step 3: Link the interrupt service routine (ISR) and the initialized interrupt vector table with the linker command file.

The linker command file provides the mechanism for including the vectors.asm object and the ints.c object.

```
/* file name == mylink.cmd */
vectors.obj
ints.obj
```

The MEMORY section needs to identify the location of the int vectors.

```
MEMORY
{
   VECTORS: origin = 0h, length = 40h
   ...
}
```

The SECTIONS section needs to map the user-defined section called vectors to the memory location.

```
SECTIONS
{  vectors: > VECTORS
   ...
}
```

Chapter 6

DSP Algorithms

Certain features of the ’C3x architecture and instruction set facilitate the solution of numerically intensive problems. This chapter presents examples of applications using these features, such as companding, filtering, fast Fourier transforms (FFTs), and matrix arithmetic.

Topic **Page**

6.1 Companding

In telecommunications, conserving channel bandwidth while preserving speech quality is a primary concern. This is achieved by quantizing the speech samples logarithmically. An 8-bit logarithmic quantizer produces speech quality equivalent to 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 standard (used in the United States and Japan), and the A-law standard (used in Europe). Detailed descriptions of μ law and A law companding are included in Volume 1 of the book *Digital Signal Processing Applications With the TMS320 Family*.

During transmission, logarithmically compressed data in sign-magnitude form is transmitted along the communications channel. If any processing is necessary, you must expand this data to a 14-bit (for μ law) or 13-bit (for A law) linear format. This operation is performed when the data is received at the digital signal processor (DSP). After processing, the result is compressed back to 8-bit format and transmitted through the channel to continue transmission.

Example 6–1 and Example 6–2 show μ-law compression and expansion (that is, 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. A look-up table trades memory space for speed of execution. Since the compressed data is eight bits long, you can construct a table with 256 entries containing 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
```

You could use the same look-up table approach for compression, but the required table length would be 16384 words for μ-law and 8192 words for A-law. If this memory size is not acceptable, use the subroutines presented in Example 6–1 or Example 6–3.

Example 6–1. μ-Law Compression

```
*
*   TITLE U±LAW COMPRESSION
*
*   SUBROUTINE MUCMPR
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*
*   NOTE:  SINCE THE STACK POINTER 'SP' IS USED IN THE COMPRESSION
*          ROUTINE 'MUCMPR', MAKE SURE TO INITIALIZE IT IN THE
*          CALLING PROGRAM.
*
*   CYCLES: 20       WORDS: 17
*
*   .global  MUCMPR
*
MUCMPR    LDI    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

          LDI    0,R2
          LDI    R1,R1         ;  If number is negative,
          LDILT  80H,R2        ;     set sign bit
          ADDI   R2,R0         ;  R0 = compressed number
          NOT    R0            ;  Reverse all bits for transmission
          RETS
```

Example 6–2. μ-Law Expansion

```
*   TITLE U-LAW EXPANSION
*
*   SUBROUTINE  MUXPND
*   ARGUMENT  ASSIGNMENTS:
*
*   ARGUMENT  |  FUNCTION
*   ----------+-------------------------------------
*   R0        |  NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*
*   CYCLES: 20 (WORST CASE)WORDS: 14
    .global MUXPND
*
MUXPND    NOT    R0,R0     ;  Complement bits
          LDI    R0,R1
          AND    0FH,R1    ;  Isolate quantization bin
          LSH    1,R1
          ADDI   33,R1     ;  Add bias to introduce 1xxxx1
          LDI    R0,R2     ;  Store for sign bit
          LSH    ±4,R0
          AND    7,R0      ;  Isolate segment code
          LSH3   R0,R1,R0  ;  Shift and put result in R0
          SUBI   33,R0     ;  Subtract bias
          TSTB   80H,R2    ;  Test sign bit
          RETSZ
          NEGI   R0        ;  Negate if a negative number
          RETS
```

Example 6–3. A-Law Compression

```
*   TITLE  A±LAW COMPRESSION
*
*   SUBROUTINE  ACMPR
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |  FUNCTION
*   ----------+-----------------------------------
*   R0        |  NUMBER TO BE CONVERTED
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*   NOTE:  SINCE THE STACK POINTER 'SP' IS USED IN THE COMPRESSION
*   ROUTINE 'ACMPR', MAKE SURE TO INITIALIZE IT IN THE
*   CALLING PROGRAM.
*   CYCLES:22 WORDS: 19
    .global  ACMPR
ACMPR     LDI       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       LDI       0,R2
          LDI       R1,R1         ;  If number is negative,
          LDILT     80H,R2        ;     set sign bit
          ADDI      R2,R0         ;  R0 = compressed number
          XOR       0D5H,R0       ;  Invert even bits
                                  ;     for transmission
          RETS
*
```

Example 6–4. A-Law Expansion

```
*   TITLE A-LAW EXPANSION
*
*   SUBROUTINE  AXPND
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*
*      CYCLES: 25 (WORST CASE)WORDS: 16
*
       .global AXPND
*
AXPND  XOR    D5H,R0    ;  Invert even bits
       LDI    R0,R1
       AND    0FH,R1    ;  Isolate quantization bin
       LSH    1,R1
       LDI    R0,R2     ;  Store for bit sign
       LSH    ±4,R0
       AND    7,R0      ;  Isolate segment code
       BZ     SKIP1
       SUBI   1,R0
       ADDI   32,R1     ;  Create 1xxxx1
SKIP1  ADDI   1,R1      ;  OR 0xxxx1
       LSH3   R0,R1,R0  ;  Shift and put result in R0
       TSTB   80H,R2    ;  Test sign bit
       RETSZ
       NEGI   R0        ;  Negate if a negative number
       RETS
```

6.2 FIR, IIR, and Adaptive Filters

Digital filters are a common requirement for DSPs. There are two types of digital filters: finite impulse response (FIR) and infinite impulse response (IIR). Both of these types can have either fixed or adaptable coefficients. This section presents the fixed-coefficient filters first, followed by the adaptive filters.

6.2.1 FIR Filters

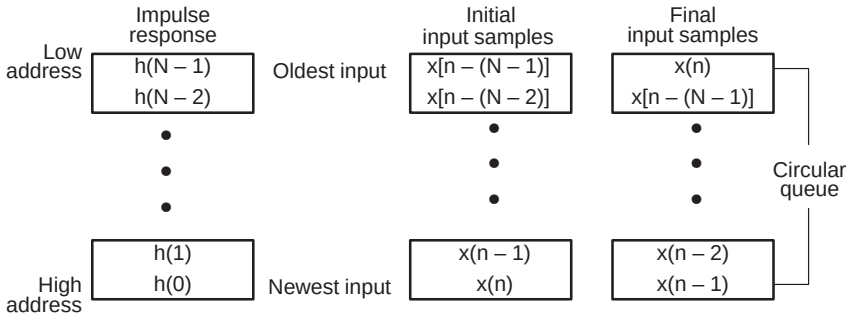
If the FIR filter has an impulse response h [0], h [1],..., h [N – 1], and x [n] represents the input of the filter at time n, the output y [n] at time n is given by this equation:

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

Two features of the ’C3x that facilitate the implementation of the FIR filters are parallel multiply/add operations and circular addressing. The former permits the performance of a multiplication and an addition in a single machine cycle, while the latter makes a finite buffer of length N sufficient for the data x.

Figure 6–1 shows the arrangement of memory locations necessary to implement circular addressing, while Example 6–5 presents the ’C3x 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. Start the locations for signal x 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 is a multiple of 32 (the lowest five bits of the beginning address are 0). See the *Circular Addressing* section in the *Addressing* chapter of the *TMS320C3x User’s Guide* for more information.

In Example 6–5, the pointer to the input sequence x is incremented and is assumed to be moving from an older input to a newer input. At the end of the subroutine, AR1 points 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
*   LOAD   AR1
*   LOAD   RC
*   LOAD   BK
*   CALL   FIR
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   AR0       |   ADDRESS OF h(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: AR0, AR1, RC, BK
*   REGISTERS MODIFIED: R0, R2, AR0, AR1, RC
*   REGISTER CONTAINING RESULT: R0
*
*   CYCLES: 11 + (N±1)  WORDS: 6
*
       .global FIR
*                                    ;  Initialize R0:
   FIR    MPYF3  *AR0++(1),*AR1++(1)%,R0
*                                    ;  h(N±1) * x(n±(N±1)) ±> R0
       LDF    0.0,R2                ;  Initialize R2
*
*   FILTER (1 <= i < N)
*
```

Example 6–5. FIR Filter (Continued)

```
        RPTS   RC                          ;  Set up the repeat cycle
        MPYF3  *AR0++(1),*AR1++(1)%,R0     ;  h(N±1±i)*x(n±(N±1±i))±>R0
||      ADDF3  R0,R2,R2                    ;  Multiply and add operation
*
        ADDF   R0,R2,R0                    ;  Add last product
*
*   RETURN SEQUENCE
*
        RETS                               ;  Return
*
*   end
*
    .end
```

6.2.2 IIR Filters

The transfer function of the IIR filters has both poles and 0s. Its output depends on both the input and the past output. As a rule, the IIR 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 shows the implementation for one biquad.

This is the equation for a single biquad:

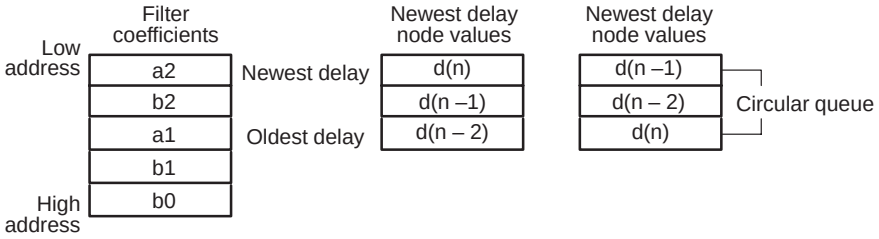
$$y[n] = a1\ \ y[n-1] + a2\ \ y[n-2] + b0\ x[n] + b1\ x[n-1] + b2\ x[n-2]$$

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, and Example 6–7 shows the implementation for any number of biquads.

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



As in the case of FIR filters, the address for the start of the d values must be a multiple of 4; that is, the last two bits of the beginning address must be 0. The block-size register BK must be initialized to 3.

Example 6–6. IIR Filter (One Biquad)

```
*   TITLE IIR FILTER
*
*   SUBROUTINE  IIR 1
*
*   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
*      load   AR0
*      load   AR1
*      load   BK
*      CALL   IIR1
*

*   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
*
```

Example 6–6. IIR Filter (One Biquad) (Continued)

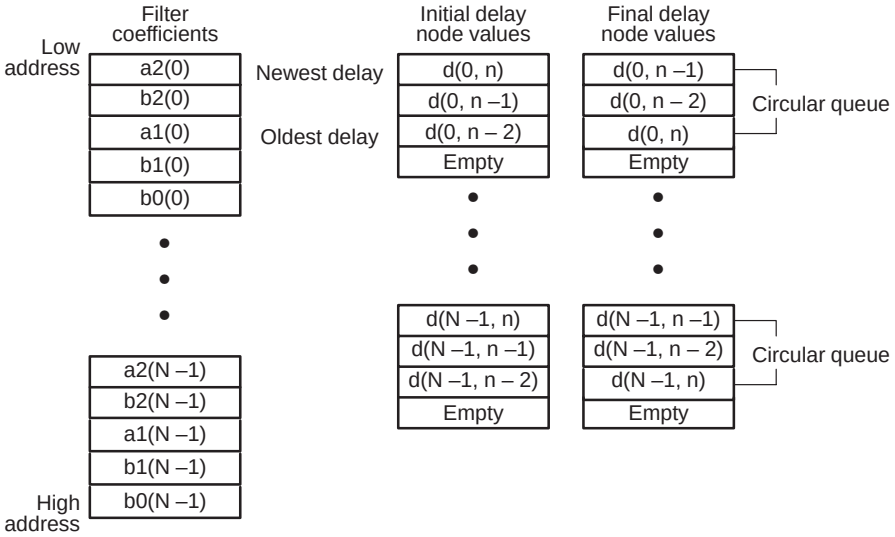
```
*   REGISTERS USED AS INPUT: R2, AR0, AR1, BK
*   REGISTERS MODIFIED: R0, R1, R2, AR0, AR1
*   REGISTER CONTAINING RESULT: R0
*
*   CYCLES: 11   WORDS: 8
*
*   FILTER
*
       .global  IIR1
*
IIR1   MPYF3  *AR0,*AR1,R0
*                                       ;   a2 * d(n±2) ±> R0
       MPYF3  *++AR0(1),*AR1––(1) % ,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
*
       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
*
*      RETURN SEQUENCE
*
       RETS                             ;   Return
*
*   end
*
    .end
```

In the more general case, 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 corresponding memory organization, while Example 6–7 shows the 'C3x assembly-language code.

Figure 6–3. Data Memory Organization for N Biquads



You must initialize the block register BK to 3; the beginning of each set of d values (that is, d [i,n], i = 0 ... N – 1) must be at an address that is a multiple of 4 (where the last two bits are 0).

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

```
*   TITLE IIR FILTERS (N > 1 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)
*      TYPICAL CALLING SEQUENCE:
*      }
*   y(n) = y(N±1,n)
*
*   TYPICAL CALLING SEQUENCE:
*
*      load   R2
*      load   AR0
*      load   AR1
*      load   IR0
*      load   IR1
*      load   BK
*      load   RC
*      CALL   IIR2
*
*   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
*
```

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

```
*   CYCLES: 17 + 6N  WORDS: 17
*
        .global IIR2
*
IIR2    MPYF3  *AR0, *AR1, R0
*                                       ;   a2(0) * d(0,n±2) ±> R0
        MPYF3  *AR0++(1), *AR1−−(1)%, R1
*                                       ;   b2(0) * d(0,n±2) ±> R1
*
        MPYF3  *++AR0(1),*AR1,R0        ;   a1(0) * D(0,n±1) ±> R0
||      ADDF   R0, R2, R2               ;   First sum term of d(0,n)
*
        MPYF3  *++AR0(1),*AR1−−(1)%,R0  ;   b1(0) * d(0,n±1) ±> R0
||      ADDF3  R0, R2, R2               ;   Second sum term of d(0,n)
        MPYF3  *++AR0(1),R2             ;   b0(0) * d(0,n) ±> R2
||      STF    R2, *AR1−−(1)%
*
*                                       ;   Store d(0,n) ;
                                        ;       point to;
                                        ;       d(0,n±2)
        RPTB   LOOP                     ;   Loop for 1 <= i < n
*
        MPYF3  *++AR0(1),*++AR1(IR0),R0 ;   a2(i) * d(i,n±2) ±> R0
||      ADDF3  R0,R2,R2                 ;   First sum term of y(i±1,n)
*
        MPYF3  *++AR0(1),*AR1−−(1)%R1   ;   b2(i) * D(i,n±2) ±> R1
||      ADDF3  R1,R2,R2                 ;   Second sum term
                                        ;       of y(i±1,n)
*
        MPYF3  *++AR0(1),*AR1,R0        ;   a1(i) * d(i,n±1) ±> R0
||      ADDF3  R0,R2,R2                 ;   First sum of d(i,n)
*
        MPYF3  *++AR0(1),*AR1−−(1)%,R0  ;   b1(i) * d(i,n±1) ±> R0
||      ADDF3  R0,R2,R2                 ;   Second sum term of d(i,n)
*
        STF    R2, *AR1−−(1)%
*                                       ;   Store d(i,n) ;
                                        ;       point to d(i,n±2)
LOOP    MPYF3  *++AR0(1), R2,R2
*                                       ;   b0(i) * d(i,n) ±> R2
*

*   FINAL SUMMATION
*
```

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

```
       ADDF   R0,R2                    ;  First sum term of y(n±1,n)
       ADDF3  R1,R2,R0                 ;  Second sum term
                                       ;     of y(n±1,n)
*
       NOP    *AR1––(IR1)              ;  Return to first biquad
       NOP *AR1––(1)%                  ;  Point to d(0,n±1)
*
*   RETURN SEQUENCE
*
       RETS                            ;  Return
*   end
*
    .end
```

6.2.3 Adaptive Filters (Least Mean Squares Algorithm)

In some applications in digital signal processing, you must adapt a filter over time to keep track of changing conditions. This is accomplished by adapting a coefficient to a filter and creating a new coefficient by means of a least mean squares (LMS) algorithm. The equations for this process are described below.

The book *Theory and Design of Adaptive Filters* 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:

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

The filter coefficients are time-dependent and updated through LMS algorithms. In a LMS algorithm, the coefficients are updated by an equation in this form:

$$h[n+1,i] = h[n,i] + \beta c[n]\,x[n-i],\ i = 0, 1, \ldots, N-1$$

where $c[n] = d[n] - y[n]$ β is a constant for the computation and d[n] is the desired signal. You can interleave the updating of the filter coefficients with the computation of the filter output so that it takes three 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 ’C3x. The memory organization and the positioning of the data in memory follows the same rules that apply to the FIR filter described in section 6.2.1 on page 6-7.

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

```
;  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))
;             e(n) = d(n) – y(n)
;             for (i = 0; i < N; i++)
;                h(n+1,i) = h(n,i) + mu * e(n) * x(n±i)

;  TYPICAL CALLING SEQUENCE:

;  load   R4
;  load   AR0
;  load   AR1
;  load   AR6
;  load   RC
;  load   BK
;  CALL   FIR

;  ARGUMENT  ASSIGNMENTS:

;  ARGUMENT  |  FUNCTION

;  ----------+-------------------------------------

;  R4        |  scale factor (2 * mu * err)
;  AR0       |  address of h(n,N±1)
;  AR1       |  address of x(n±(N±1))
;  AR6       |  address of d(n)
;  RC        |  length of filter ± 2 (N±1)
;  BK        |  length of filter (N)

;  REGISTERS USED AS INPUT: R4, AR0, AR1, RC, BK
;  REGISTERS MODIFIED: R0, R1, R2, R5, AR0, AR1, RC
;  REGISTER CONTAINING RESULT: R0

;  PROGRAM SIZE: 11 words

;  EXECUTION CYCLES: 13 + 3N

;=============================================================================
```

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

```
;  setup (i = 0)

   .  .text
LMS:
       ldf    *ar6++,r5                  ; Get desired sample
       mpyf3  *ar0--%, *ar1++(1)%,r0     ; h(n,N-1)  * x(n-(N-1)) -> R0
||     subf   r2,r2,r2                   ; init r2
*                                        ;  Initialize R0:
   LMS    MPYF3  *AR0, *AR1, R0
*                                        ;  h(n,N±1) * x(n±(N±1)) ±> R0
          LDF    0.0,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)
*

          RPTB   LOOP                    ;  Set up the repeat block
*
*                                        ;  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))
*
          LOOP   ADDF3  *AR0++(1), R1, R1
*                                        ;  h(n,N±1±i) + x(n±(N±1±i))
                                         ;      *tmuerr ±> R1
*
          ADDF3  R0,R2,R0                ;  Add last product
          STF    R1,*±AR0(1)             ;  h(n,0) + x(n)
                                         ;      * tmuerr ±> h(n+1,0)
*

*     RETURN SEQUENCE

*
          RETS                           ;  Return
*

*  end
*
   .end
```

6.3 Lattice Filters

The lattice form is an alternative way of implementing digital filters. It has found 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) is a filter having only 0s, and is 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-th 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 during speech synthesis.

Figure 6–4. Structure of the Inverse Lattice Filter

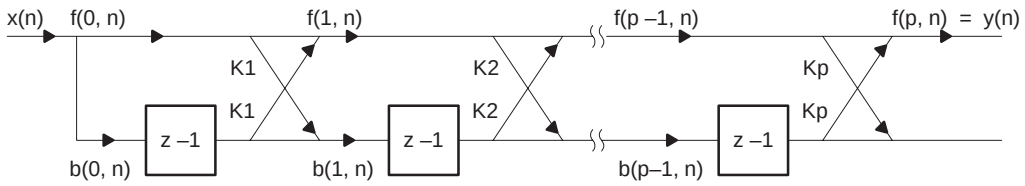
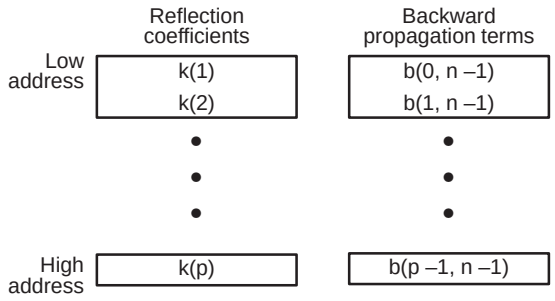


Figure 6–5 shows the data memory organization of the inverse lattice filter on the ’C3x.

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



Example 6–9 shows the implementation of an inverse lattice filter.

Example 6–9. Inverse Lattice Filter

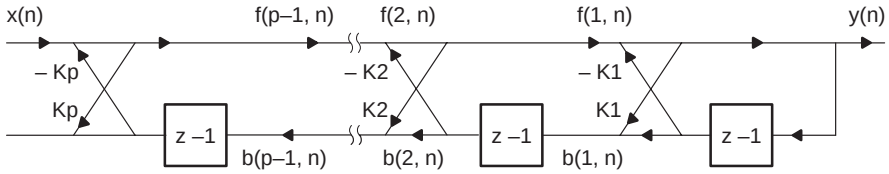
```
*   TITLE INVERSE LATTICE FILTER
*
*   SUBROUTINE LATINV
*
*   LATINV == LATTICE FILTER (LPC INVERSE FILTER ± ANALYSIS)
*
*   TYPICAL CALLING SEQUENCE:
*

*   load   R2
*   load   AR0
*   load   AR1
*   load   RC
*   CALL   LATINV
*

*   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))
*
```

Example 6–9. Inverse Lattice Filter (Continued)

```
*
*   PROGRAM SIZE: 10 WORDS
*
*   EXECUTION CYCLES: 13 + 3 * (p±1)
*
        .global LATINV
*
*   i = 1
*
LATINV 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
*

*   2 <= i <= p

*
        RPTB   LOOP
        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,b±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)
        ADDF3  R2,R0,R2             ;   f(p±1,n)+k(p)*b(p±1,n±1)
*                                   ;       = f(p,n) ±> R2
*
*                                   ;   b(p±1,n±1)+k(p)*f(p±1,n)
        ADDF3  *AR1, R1, R3         ;   = b(p,n) ±> R3
||      STF    R3, *AR1             ;   b(p±1,n) ±> b(p±1,n±1)
*
*   RETURN SEQUENCE
*
        RETS                        ;   RETURN

*
*   end
*
.end
```

The forward lattice filter is similar in structure to the inverse filter, as shown in Figure 6–6.

Figure 6–6. Structure of the (Forward) Lattice Filter



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:

$f(p,n) = x(n),\ b(i,n-1) = 0 \qquad \text{for } i = 1, \ldots, p$

Final conditions:

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

The data memory organization is identical to that of the inverse filter, as shown in Figure 6–5 on page 6-19. Example 6–10 shows the implementation of the lattice filter on the ’C3x.

Example 6–10. Lattice Filter

```
*   TITLE LATTICE FILTER
*
*   SUBROUTINE LATICE
*

*      LOAD   AR0
*      LOAD   AR1
*      LOAD   RC
*      CALL   LATICE
*

*   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))
    IR0       |   3
*   RC        |   RC = P ± 3
*
*   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))
*

*   STACK USAGE: NONE
*
*   PROGRAM SIZE: 12 WORDS
*
*   EXECUTION CYCLES: 15 + 3 * (P±2)
*

       .global  LATICE
*
LATICE MPYF3  *AR0,*AR1,R0
*                              ;   K(P) * B(P±1,N±1) ±> R0
                               ;   Assume F(P,N) ±> R2
       SUBF3  R0,R2,R2         ;   F(P,N)±K(P)*B(P±1,N±1)
                               ;       = F(P±1,N) ±> R2
||     MPYF3  *--AR0(1),*--AR1(1),R0
                               ;   K(P–1) * B(P±2,N±1) ±> R0
       SUBF3  R0,R2,R2         ;   F(P–1,N)±K(P–1)*B(P±2,N±1)
                               ;       = F(P±2,N) ±> R2
```

Example 6–10. Lattice Filter (Continued)

```
||     MPYF3  *−−AR0(1),*−−AR1(1),R0
                              ;  K(P−2) * B(P−3,N−1) ±> R0
       MPYF3  R2,*+AR0(1),R1  ;  F(P−2,N) * K(P−1) ±> R1
       ADDF3  R1,*+AR1(1),R3  ;  F(P±2,N) * K(P−1) + B(P±2,N−1)
                              ;     = B(P−1,N) ±> R3
                              ;  1 <= I <= P−2
*
       RPTB   LOOP
       SUBF3  R0,R2,R2        ;  F(I,N) − K(I) * B(I−1,N−1)
                              ;     = F(I−1,N) ±> R2
||     MPYF3  *−−AR0(1),*−−AR1(1),R0
                              ;  K(I−1) * B(I±2,N±1) ±> R0
       STF R3,*+AR1(IR0)      ;  B(I+1,N) ±> B(I+1,N−1)
||     MPYF3  R2,*+AR0(1),R1  ;  F(I−1,N) * K(I) ±> R1
LOOP   ADDF3  R1,*+AR1(1),R3  ;  F(I−1,N) * K(I) + B(I−1,N−1)
                              ;     = B(I,N) ±> R3
       STF    R3,*+AR1(2)     ;  B(1,N) ±> B(1,N±1)
       STF    R2,*+AR1(1)     ;  F(0,N) ±> B(0,N±1)

*   RETURN SEQUENCE
*
       RETS
*
*   END
*
       .end
```

6.4 Matrix-Vector Multiplication

In matrix-vector multiplication, a K x N matrix of elements m(i,j) having K rows and N columns is multiplied by an N x 1 vector to produce a K x 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:

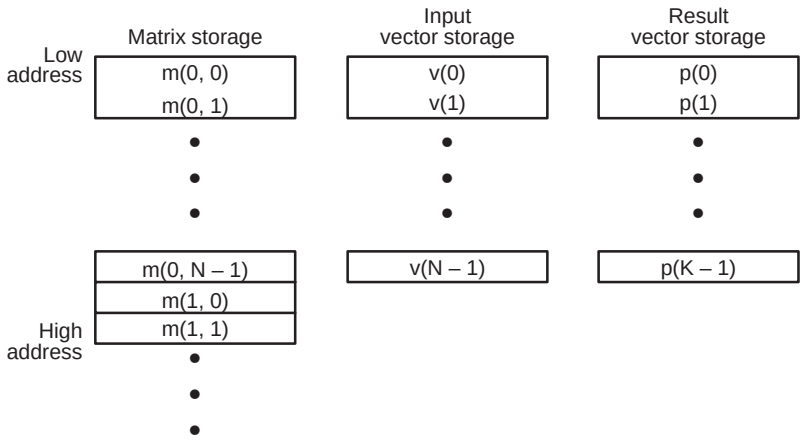
$$p(i) = m(i,0)\,v(0) + m(i,1)\,v(1) + \ldots + m(i,N-1)\,v(N-1) \quad i = 0,1,\ldots,K-1$$

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. 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 ’C3x assembly code that implements it. Note that in Example 6–11, K (number of rows) must be greater than 0 and N (number of columns) must 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)
*   R1        |   NUMBER OF COLUMNS ± 2 (N±2)
*
*   REGISTERS USED AS INPUT: AR0, AR1, AR2, AR3, R1
*   REGISTERS MODIFIED: R0, R2, AR0, AR1, AR2, AR3, IR0,
*      RC, RSA, REA
*
*   PROGRAM SIZE: 11
*
*   EXECUTION CYCLES: 6 + 10 * K + K * (N ± 1)
*
       .global  MAT
*
*   SETUP
*
MAT    LDI    R1,IR0                ;  Number of columns±2 ±> IR0
       ADDI   2,IR0                 ;  IR0 = N
*
*   FOR (i = 0; i < K; i++) LOOP OVER THE ROWS
*
```

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

```
ROWS   LDF    0.0,R2                  ;  Initialize R2
       MPYF3  *AR0++(1),*AR1++(1),R0
*                                     ;  m(i,0) * v(0) ±> R0
*

*   FOR (j = 1; j < N; j++) DO DOT PRODUCT OVER COLUMNS
*
       RPTS   R1                      ;  Multiply a row by a column
*
       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 no. of rows left
*
       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 Vector Maximum Search

In vector maximum search, a vector of N elements is searched for its greatest element:

```
max { p(i) }
```

In pseudo-C format, the search is expressed by:

```
max = 0
max location = 0
for ( i=0; i < N; itt) }
   if ( max < p [i]}
      max = p[i];
      max location = i;
   }
}
```

Example 6–12 shows an example.

Example 6–12. vecmax.asm

```
;  Vector maximum search

;  EQUATIONS: max = max {p(i) }

;  TYPICAL CALLING SEQUENCE:

;      load   AR0
;      load   RC
;      load   R1
;      CALL   vecmax

;  ARGUMENT  ASSIGNMENTS:

;  argument  |  function

;  ----------+-------------------------------------

;  AR0       |  address of vector
;  RC        |  length of filter ± 2 (N±2)
;  R1        |  length of filter – 1 (N–1)

;  REGISTERS USED AS INPUT: AR0, R1, RC
;  REGISTERS MODIFIED: R0, R1, AR0, RC
;  REGISTER CONTAINING RESULT:
;         R0  maximum value
;         R1  index of maximum value

;  PROGRAM SIZE: 5 words

;  EXECUTION CYCLES: 2 + 3N

;===========================================================================

          .text
vecmax    ldf       *ar0--,r0       ; last value

          rptb      loop            ;

          cmpf3     *ar0,r0         ; Compare input value to maximum

          ldile     rc,r1           ; Write index of loop

loop      ldfle     *ar0--,r0       ; Load new max value

          end
```

6.6 Fast Fourier Transforms (FFTs)

Fourier transforms are an important tool often used in digital signal processing (DSP) systems. The purpose of the transform is to convert 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*, and *Digital Signal Processing Applications With the TMS320 Family*.

Fast Fourier transform is a label for a collection of algorithms that implement efficient conversion from time to frequency domain. Distinctions are made among FFTs based on the following characteristics:

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

Certain ’C3x features that increase the efficiency of numerically intensive algorithms are particularly well suited for FFTs. The high speed of the device (33-ns cycle time) makes implementation of real-time algorithms easier, while floating-point capability eliminates the problems associated with dynamic range. The powerful indirect-addressing indexing scheme facilitates the access of FFT butterfly legs with different spans. The repeat block implemented by the RPTB instruction reduces the looping overhead in algorithms heavily dependent on loops (such as FFTs). This construct provides the efficiency of in-line coding in loop form. The FFT reverses the bit order of the output; therefore, the output must be reordered. This reordering does not require extra cycles, because the device has a special mode of indirect addressing (bit-reversed addressing) for accessing the FFT output in the original order.

The examples in this section are based on programs contained in the *DFT/FFT and Convolution Algorithms* book and in the paper *Real-Valued Fast Fourier Transform Algorithms*.

6.6.1 FFT Definition

The FFT is an efficient implementation of the discrete fourier transform (DFT) equation:

$$X_N(k) = \sum_{n=0}^{N-1} x(n)\ e^{j\,2\frac{\pi}{N}kn}$$

The inverse DFT equation is:

$$x_N(n) = \frac{1}{N} \sum_{k=0}^{N-1} X_N(k)\ e^{j\,2\frac{\pi}{N}nk}$$

The FFT takes advantage of the periodic nature of the complex exponential $e^{j\,2\frac{\pi}{N}}$ to reduce redundancy and number of calculations. The FFT expresses the original DFT using two smaller DFTs of length $\frac{N}{2}$. This definition is applied until the original DFT has been expressed in terms of a 2-point DFT, which is normally referred to as radix-2 FFT.

There are two ways this decomposition process occurs:

- By decimation in time where the signals are split into several shorter interleaved sequences (see Figure 6–8).
- By decimation in frequency where the signals are split into several smaller interleaved frequency components (see Figure 6–9).

Figure 6–8. Decimation in Time for an 8-Point FFT

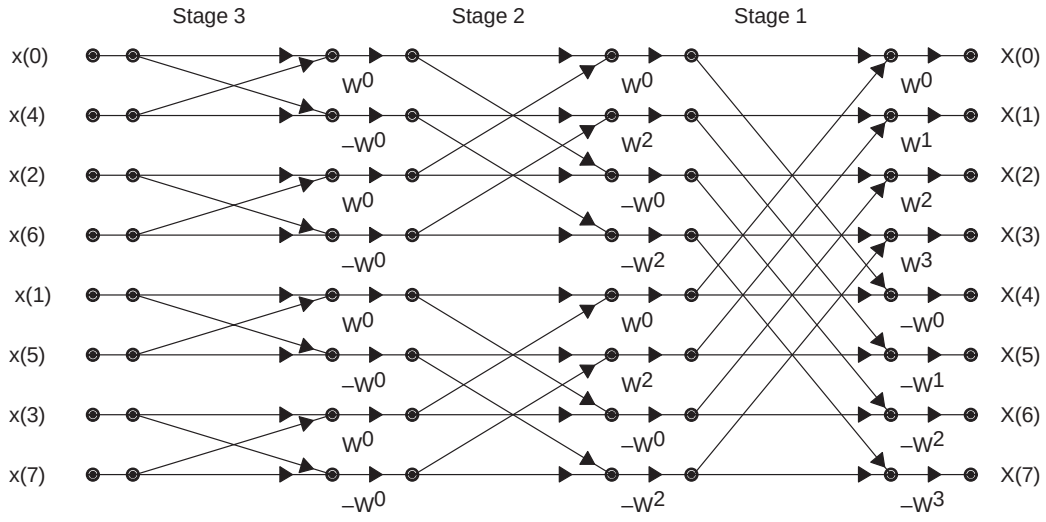
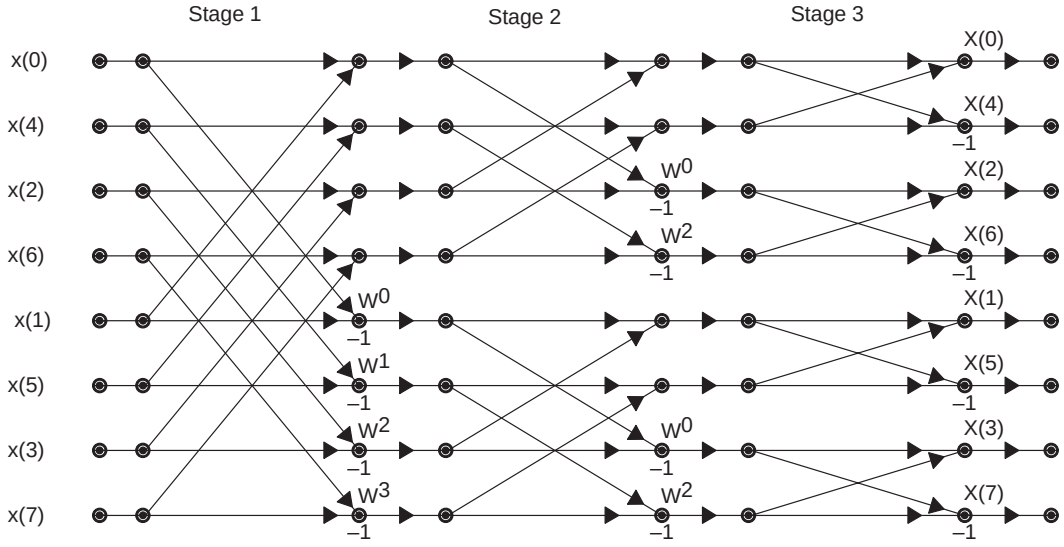


Figure 6–9. Decimation in Frequency for 8-Point FFT



$$W^k = e^{j\frac{\pi}{8}k}$$

6.6.2 Complex Radix-2 DIF FFT

Example 6–13 and Example 6–14 show the implementation of a complex radix-2 DIF FFT on the ’C3x. Example 6–13 contains the generic code of the FFT, which can be used with a FFT of any length. However, for the complete implementation of an FFT, you need a table of twiddle factors (sines/cosines); the length of the table depends on the size of the transform. A table with twiddle factors (containing 1-1/4 complete cycles of a sine) is presented separately in Example 6–14 as a 64-point FFT. This retains the generic form of the radix-2 DIF FFT in Example 6–13. A full sine wave must have an equal number of samples as the length of the FFT. Example 6–14 uses two variables: N, which is the FFT length, and M, which is the logarithm of N to a base equal to the radix. In other words, M is the number of stages of the FFT. For example, in a 64-point FFT, M = 6 when using a radix-2 algorithm, and M = 3 when using a radix-4 algorithm. If the table with the twiddle factors and the FFT code are kept in separate files, they will be connected at link time.

Example 6–13. Complex Radix-2 DIF FFT

```
*
*   TITLE COMPLEX, RADIX–2, DIF FFT
*
*   GENERIC PROGRAM FOR LOOPED±CODE RADIX±2 FFT COMPUTATION IN TMS320C3x
*
*   THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 111.
*   THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY. THE COMPUTATION
*   IS DONE IN  PLACE, BUT THE RESULT IS MOVED TO ANOTHER MEMORY
*   SECTION TO DEMONSTRATE THE BIT±REVERSED ADDRESSING.
*
*   THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE THAT IS PUT IN A .DATA
*   SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE
*   GENERIC NATURE OF THE PROGRAM. FOR THE SAME PURPOSE, THE SIZE OF
*   THE FFTN AND LOG2(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED
*   DURING LINKING.
*
*

       .globl    FFT             ;  Entry point for execution
       .globl    N               ;  FFT size
       .globl    M               ;  LOG2(N)
       .globl    SINE            ;  Address of sine table

INP    .usect “IN”,1024           ;  Memory with input data
.BSS   OUTP,1024                  ;  Memory with output data

              .text

*   INITIALIZE

FFTSIZ .word  N
LOGFFT .word  M
SINTAB .word  SINE
INPUT  .word  INP
OUTPUT .word  OUTP


FFT:      LDP    FFTSIZ           ;  Command to load data page pointer

LDI       @FFTSIZ,IR1
LSH       ±2,IR1                 ;  IR1 = N/4, pointer for SIN/COS table
LDI       0,AR6                  ;  AR6 holds the current stage number
LDI       @FFTSIZ,IR0
LSH       1,IR0                  ;  IR0 = 2*N1 (because of real/imag)
LDI       @FFTSIZ,R7             ;  R7 = N2
LDI       1,AR7                  ;  Initialize repeat counter
                                 ;      of first loop
LDI       1,AR5                  ;  Initialize IE index (AR5 = IE)
```

Example 6–13. Complex Radix-2 DIF FFT (Continued)

```
*   OUTER LOOP

LOOP:     NOP    *++AR6(1)          ;  Current FFT stage
          LDI    @INPUT,AR0         ;  AR0 points to X(I)
          ADDI   R7,AR0,AR2         ;  AR2 points to X(L)
          LDI    AR7,RC
          SUBI   1,RC               ;  RC should be one less than desired #

*   FIRST LOOP

          RPTB   BLK1
          ADDF   *AR0,*AR2,R0       ;  R0 = X(I)+X(L)
          SUBF   *AR2++,*AR0++,R1   ;  R1 = X(I)±X(L)
          ADDF   *AR2,*AR0,R2       ;  R2 = Y(I)+Y(L)
          SUBF   *AR2,*AR0,R3       ;  R3 = Y(I)±Y(L)
          STF    R2,*AR0--          ;  Y(I) = R2 and...
   ||     STF    R3,*AR2--          ;  Y(L) = R3
   BLK1   STF    R0,*AR0++(IR0)     ;  X(I) = R0 and...
   ||     STF    R1,*AR2++(IR0)     ;  X(L) = R1 and AR0,2 = AR0,2 + 2*n

*  IF THIS IS THE LAST STAGE, YOU ARE DONE

          CMPI   @LOGFFT,AR6
          BZD    END

*   MAIN INNER LOOP

          LDI    2,AR1              ;  Init loop counter for
                                    ;     inner loop
          LDI    @SINTAB,AR4        ;  Initialize IA index (AR4 = IA)
INLOP:    ADDI   AR5,AR4            ;  IA = IA+IE; AR4 points to
                                    ;     cosine
          LDI    AR1,AR0
          ADDI   2,AR1              ;  Increment inner loop counter
          ADDI   @INPUT,AR0         ;  (X(I),Y(I)) pointer
          ADDI   R7,AR0,AR2         ;  (X(L),Y(L)) pointer
          LDI    AR7,RC
          SUBI   1,RC               ;  RC should be 1 less than
                                    ;     desired #
          LDF    *AR4,R6            ;  R6 = SIN

*   SECOND LOOP

          RPTB      BLK2
          SUBF      *AR2,*AR0,R2    ;  R2 = X(I)±X(L)
          SUBF      *+AR2,*+AR0,R1
*                                   ;  R1 = Y(I)±Y(L)
          MPYF      R2,R6,R0        ;  R0 = R2*SIN and...
||        ADDF      *+AR2,*+AR0,R3
*                                   ;  R3 = Y(I)+Y(L)
          MPYF      R1,*+AR4(IR1),R3;  R3 = R1*COS and ...
||     STF          R3,*+AR0        ;  Y(I) = Y(I)+Y(L)
       SUBF         R0,R3,R4        ;  R4 = R1 * COS±R2 * SIN
       MPYF         R1,R6,R0        ;  R0 = R1 * SIN and...
```

Example 6–13. Complex Radix-2 DIF FFT (Continued)

```
||     ADDF      *AR2,*AR0,R3     ;  R3 = X(I) + X(L)
       MPYF      R2,*+AR4(IR1),R3 ;  R3 = R2 * COS and...
||     STF       R3,*AR0++(IR0)
*                                 ;  X(I) = X(I)+X(L) and AR0 = AR0+2*N1
       ADDF      R0,R3,R5         ;  R5 = R2*COS+R1*SIN
BLK2   STF R5,*AR2++(IR0)         ;  X(L) = R2 * COS+R1 * SIN,
                                  ;     incr AR2 and...
||     STF R4,*+AR2               ;  Y(L) = R1*COS±R2*SIN

       CMPI      R7,AR1
       BNE       INLOP            ;  Loop back to the inner loop

       LSH       1,AR7            ;  Increment loop counter for next time
       BRD       LOOP             ;  Next FFT stage (delayed)
       LSH       1,AR5            ;  IE = 2*IE
       LDI       R7,IR0           ;  N1 = N2
       LSH       ±1,R7            ;  N2 = N2/2

*   STORE RESULT OUT USING BIT-REVERSED ADDRESSING

       END:   LDI       @FFTSIZ,RC       ;  RC = N
              SUBI      1,RC             ;  RC should be one less than desired #
              LDI       @FFTSIZ,IR0      ;  IR0 = size of FFT = N
              LDI       2,IR1
              LDI       @INPUT,AR0
              LDI       @OUTPUT,AR1

              RPTB      BITRV
              LDF       *+AR0(1),R0
       ||     LDF       *AR0++(IR0)B,R1
       BITRV  STF       R0,*+AR1(1)
       ||     STF       R1,*AR1++(IR1)


       SELF   BR        SELF             ;  Branch to itself at the end
              .end
```

Example 6–14. 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 FFT *

          .globl SINE
          .globl N
          .globl M

N         .set   64
M         .set   6

          .data

SINE
          .float       0.000000
          .float       0.098017
          .float       0.195090
          .float       0.290285
          .float       0.382683
          .float       0.471397
          .float       0.555570
          .float       0.634393
          .float       0.707107
          .float       0.773010
          .float       0.831470
          .float       0.881921
          .float       0.923880
          .float       0.956940
          .float       0.980785
          .float       0.995185
COSINE
          .float       1.000000
          .float       0.995185
          .float       0.980785
          .float       0.956940
          .float       0.923880
          .float       0.881921
          .float       0.831470
          .float       0.773010
          .float       0.707107
          .float       0.634393
          .float       0.555570
          .float       0.471397
          .float       0.382683
          .float       0.290285
          .float       0.195090
          .float       0.098017
```

Example 6–14. Table With Twiddle Factors for a 64-Point FFT (Continued)

```
        .float      0.000000
        .float ±    0.098017
        .float ±    0.195090
        .float ±    0.290285
        .float ±    0.382683
        .float –    0.471397
        .float      –0.555570
        .float –    0.634393
        .float –    0.707107
        .float –    0.773010
        .float –    0.831470
        .float –    0.881921
        .float –    0.923880
        .float –    0.956940
        .float –    0.980785
        .float –    0.995185
        .float      –1.000000
        .float –    0.995185
        .float –    0.980785
        .float –    0.956940
        .float –    0.923880
        .float –    0.881921
        .float –    0.831470
        .float –    0.773010
        .float –    0.707107
        .float –    0.634393
        .float –    0.555570
        .float –    0.471397
        .float –    0.382683
        .float –    0.290285
        .float –    0.195090
        .float –    0.098017
        .float      0.000000
        .float      0.098017
        .float      0.195090
        .float      0.290285
        .float      0.382683
        .float      0.471397
        .float      0.555570
        .float      0.634393
        .float      0.707107
        .float      0.773010
        .float      0.831470
        .float      0.881921
        .float      0.923880
        .float      0.956940
        .float      0.980785
        .float      0.995185
```

6.6.3 Complex Radix-4 DIF FFT

The radix-2 algorithm has tutorial value because the functioning of the FFT algorithm is relatively easy to understand. However, radix-4 implementation can increase execution speed by reducing the amount of arithmetic required. Example 6–15 shows the generic implementation of a complex DIF FFT in radix-4. A companion table, such as the one in Example 6–14, must have a value of M equal to the logN, where the base of the logarithm is 4.

Example 6–15. Complex Radix-4 DIF FFT

```
*
*   TITLE COMPLEX, RADIX-4, DIF FFT
*
*   GENERIC PROGRAM TO PERFORM A LOOPED±CODE RADIX±4 FFT COMPUTATION
*   IN THE TMS320C3x
*
*   THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 117.
*   THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY, AND THE COMPUTATION
*   IS DONE IN PLACE.
*
*   THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE THAT IS PUT IN A .DATA
*   SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE
*   GENERIC NATURE OF THE PROGRAM. FOR THE SAME PURPOSE, THE SIZE OF
*   THE FFT N AND LOG4(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND
*   SPECIFIED DURING LINKING.
*
*   IN ORDER TO HAVE THE FINAL RESULT IN BIT±REVERSED ORDER, THE TWO
*   MIDDLE BRANCHES OF THE RADIX±4 BUTTERFLY ARE INTERCHANGED DURING
*   STORAGE. NOTE THIS DIFFERENCE WHEN COMPARING WITH THE PROGRAM IN
*   P. 117 OF THE BURRUS AND PARKS BOOK.
*
*
       .globl    FFT       ;  Entry point for execution
       .globl    N         ;  FFT size
       .globl    M         ;  LOG4(N)
       .globl    SINE      ;  Address of sine table

       .usect    "IN",1024 ;  Memory with input data

       .text
*   INITIALIZE

TEMP   .word     $+2
STORE  .word     FFTSIZ    ;  Beginning of temp storage area
       .word     N
       .word     M
       .word     SINE
       .word     INP
```

Example 6–15. Complex Radix-4 DIF FFT (Continued)

```
        .BSS      FFTSIZ,1  ;  FFT size
        .BSS      LOGFFT,1  ;  LOG4(FFTSIZ)
        .BSS      SINTAB,1  ;  Sine/cosine table base
        .BSS      INPUT,1   ;  Area with input data to process
        .BSS      STAGE,1   ;  FFT stage #
        .BSS      RPTCNT,1  ;  Repeat counter
        .BSS      IEINDX,1  ;  IE index for sine/cosine
        .BSS      LPCNT,1   ;  Second±loop count
        .BSS      JT,1      ;  JT counter in program, P. 117
        .BSS      IA1,1     ;  IA1 index in program, P. 117

                  FFT:
*   INITIALIZE DATA LOCATIONS
    LDP    TEMP             ;  Command to load data page counter
    LDI    @TEMP,AR0
    LDI    @STORE,AR1
    LDI    *AR0++,R0        ;  Xfer data from one memory to the other
    STI    R0,*AR1++
    LDI    *AR0++,R0
    STI    R0,*AR1++
    LDI    *AR0++,R0
    STI    R0,*AR1++
    LDI    *AR0,R0
    STI    R0,*AR1

    LDP    FFTSIZ           ;  Command to load data page pointer
    LDI    @FFTSIZ,R0
    LDI    @FFTSIZ,IR0
    LDI    @FFTSIZ,IR1
    LDI    0,AR7
    STI    AR7,@STAGE       ;  @STAGE holds the current stage number
    LSH    1,IR0            ;  IR0 = 2*N1 (because of real/imag)
    LSH    ±2,IR1           ;  IR1 = N/4, pointer for SIN/COS table
    LDI    1,AR7
    STI    AR7,@RPTCNT      ;  Init repeat counter of first loop
    STI    AR7,@IEINDX      ;  Init. IE index
    LSH    ±2,R0            ;  JT = R0/2+2
    ADDI   2,R0
    STI    R0,@JT
    SUBI   2,R0
    LSH    1,R0             ;  R0 = N2

*   OUTER LOOP

LOOP:
    LDI    @INPUT,AR0       ;  AR0 points to X(I)
    ADDI   R0,AR0,AR1       ;  AR1 points to X(I1)
    ADDI   R0,AR1,AR2       ;  AR2 points to X(I2)
    ADDI   R0,AR2,AR3       ;  AR3 points to X(I3)
    LDI    @RPTCNT,RC
    SUBI   1,RC             ;  RC should be one less than desired #
```

Example 6–15. Complex Radix-4 DIF FFT (Continued)

```
*   FIRST LOOP

    RPTB   BLK1
    ADDF   *+AR0,*+AR2,R1
*                           ;   R1 = Y(I)+Y(I2)
        ADDF   *+AR3,*+AR1,R3
*                           ;   R3 = Y(I1)+Y(I3)
        ADDF   R3,R1,R6     ;   R6 = R1+R3
        SUBF   *+AR2,*+AR0,R4
*                           ;   R4 = Y(I)±Y(I2)
        STF    R6,*+AR0     ;   Y(I) = R1+R3
        SUBF   R3,R1        ;   R1 = R1±R3
        LDF    *AR2,R5      ;   R5 = X(I2)
||      LDF    *+AR1,R7     ;   R7 = Y(I1)
        ADDF   *AR3,*AR1,R3 ;   R3 = X(I1)+X(I3)
        ADDF   R5,*AR0,R1   ;   R1 = X(I)+X(I2)
||      STF    R1,*+AR1     ;   Y(I1) = R1±R3
        ADDF   R3,R1,R6     ;   R6 = R1+R3
        SUBF   R5,*AR0,R2   ;   R2 = X(I)±X(I2)
||      STF    R6,*AR0++(IR0);  X(I) = R1+R3
        SUBF   R3,R1        ;   R1 = R1±R3
        SUBF   *AR3,*AR1,R6 ;   R6 = X(I1)±X(I3)
        SUBF   R7,*+AR3,R3  ;   ±R3 = Y(I1)±Y(I3)
||      STF    R1,*AR1++(IR0);  X(I1) = R1±R3
        SUBF   R6,R4,R5     ;   R5 = R4±R6
        ADDF   R6,R4        ;   R4 = R4+R6
        STF    R5,*+AR2     ;   Y(I2) = R4±R6
||      STF    R4,*+AR3     ;   Y(I3) = R4+R6
        SUBF   R3,R2,R5     ;   R5 = R2±R3
        ADDF   R3,R2        ;   R2 = R2+R3
BLK1    STF    R5,*AR2++(IR0);  X(I2) = R2±R3
||      STF R2,*AR3++(IR0)  ;   X(I3) = R2+R3

*   IF THIS IS THE LAST STAGE, YOU ARE DONE

        LDI    @STAGE,AR7
        ADDI   1,AR7
        CMPI   @LOGFFT,AR7
        BZD    END
        STI    AR7,@STAGE   ;   Current FFT stage

*   MAIN INNER LOOP

        LDI    1,AR7
        STI    AR7,@IA1     ;   Init IA1 index
        LDI    2,AR7
        STI    AR7,@LPCNT   ;   Init loop counter for inner loop
                            ;       INLOP:
        LDI    2,AR6        ;   Increment inner loop counter
        ADDI   @LPCNT,AR6
        LDI    @LPCNT,AR0
        LDI    @IA1,AR7
```

Example 6–15. Complex Radix-4 DIF FFT (Continued)

```
        ADDI   @IEINDX,AR7      ;  IA1 = IA1+IE
        ADDI   @INPUT,AR0       ;  (X(I),Y(I)) pointer
        STI    AR7,@IA1
        ADDI   R0,AR0,AR1       ;  (X(I1),Y(I1)) pointer
        STI    AR6,@LPCNT
        ADDI   R0,AR1,AR2       ;  (X(I2),Y(I2)) pointer
        ADDI   R0,AR2,AR3       ;  (X(I3),Y(I3)) pointer
        LDI    @RPTCNT,RC
        SUBI   1,RC             ;  RC should be one less than desired #
        CMPI   @JT,AR6          ;  If LPCNT = JT, go to
        BZD    SPCL             ;     special butterfly
        LDI    @IA1,AR7
        LDI    @IA1,AR4
        ADDI   @SINTAB,AR4      ;  Create cosine index AR4
        SUBI   1,AR4            ;  Adjust sine table pointer
        ADDI   AR4,AR7,AR5
        SUBI   1,AR5            ;  IA2 = IA1+IA1±1
        ADDI   AR7,AR5,AR6
        SUBI   1,AR6            ;  IA3 = IA2+IA1±1

*   SECOND LOOP

        RPTB   BLK2
        ADDF   *+AR2,*+AR0,R3
*                               ;  R3 = Y(I)+Y(I2)
        ADDF   *+AR3,*+AR1,R5
*                               ;  R5 = Y(I1)+Y(I3)
        ADDF   R5,R3,R6         ;  R6 = R3+R5
        SUBF   *+AR2,*+AR0,R4
*                               ;  R4 = Y(I)±Y(I2)
        SUBF   R5,R3            ;  R3 = R3±R5
        ADDF   *AR2,*AR0,R1     ;  R1 = X(I)+X(I2)
        ADDF   *AR3,*AR1,R5     ;  R5 = X(I1)+X(I3)
        MPYF   R3,*+AR5(IR1),R6 R6 = R3*CO2
||      STF    R6,*+AR0         ;  Y(I) = R3+R5
        ADDF   R5,R1,R7         ;  R7 = R1+R5
        SUBF   *AR2,*AR0,R2     ;  R2 = X(I)±X(I2)
        SUBF   R5,R1            ;  R1 = R1±R5
        MPYF   R1,*AR5,R7       ;  R7 = R1*SI2
||      STF R7,*AR0++(IR0)      ;  X(I) = R1+R5
        SUBF   R7,R6            ;  R6 = R3*CO2±R1*SI2
        SUBF   *+AR3,*+AR1,R5
*                               ;  R5 = Y(I1)±Y(I3)
        MPYF   R1,*+AR5(IR1),R7 ;  R7 = R1*C02
||      STF    R6,*+AR1         ;  Y(I1) = R3*CO2±R1*SI2
        MPYF   R3,*AR5,R6       ;  R6 = R3*SI2
        ADDF   R7,R6            ;  R6 = R1*CO2+R3*SI2
        ADDF   R5,R2,R1         ;  R1 = R2+R5
        SUBF   R5,R2            ;  R2 = R2±R5
        SUBF   *AR3,*AR1,R5     ;  R5 = X(I1)±X(I3)
        SUBF   R5,R4,R3         ;  R3 = R4±R5
        ADDF   R5,R4            ;  R4 = R4+R5
        MPYF   R3,*+AR4(IR1),R6 ;  R6 = R3*CO1
```

Example 6–15. Complex Radix-4 DIF FFT (Continued)

```
||     STF R6,*AR1++(IR0)      ;  X(I1) = R1*CO2+R3*SI2
       MPYF   R1,*AR4,R7       ;  R7 = R1*SI1
       SUBF   R7,R6            ;  R6 = R3*CO1±R1*SI1
       MPYF   R1,*+AR4(IR1),R6 ;  R6 = R1*CO1
||     STF R6,*+AR2            ;  Y(I2) = R3*CO1±R1*SI1
       MPYF   R3,*AR4,R7       ;  R7 = R3*SI1
       ADDF   R7,R6            ;  R6 = R1*C O1+R3*SI1
       MPYF   R4,*+AR6(IR1),R6 ;  R6 = R4*CO3
||     STF R6,*AR2++(IR0)      ;  X(I2) = R1*CO1+R3*SI1
       MPYF   R2,*AR6,R7       ;  R7 = R2*SI3
       SUBF   R7,R6            ;  R6 = R4*CO3±R2*SI3
       MPYF   R2,*+AR6(IR1),R6 ;  R6 = R2*CO3
||     STF R6,*+AR3            ;  Y(I3) = R4*CO3±R2*SI3
       MPYF   R4,*AR6,R7       ;  R7 = R4*SI3
       ADDF   R7,R6            ;  R6 = R2*CO3+R4*SI3

BLK2   STF    R6,*AR3++(IR0)
*                              ;  x(i3) = R2*CO3+R4*SI3

       CMPI   @LPCNT,R0
       BP  INLOP               ;  Loop back to the inner loop
       BR  CONT

*   SPECIAL BUTTERFLY FOR W = J

SPCL   LDI IR1,AR4
       LSH ±1,AR4              ;  Point to SIN(45)
       ADDI   @SINTAB,AR4      ;  Create cosine index AR4 = CO21

       RPTB   BLK3
       ADDF   *AR2,*AR0,R1     ;  R1 = X(I)+X(I2)
       SUBF   *AR2,*AR0,R2     ;  R2 = X(I)±X(I2)
       ADDF   *+AR2,*+AR0,R3
*                              ;  R3 = Y(I)+Y(I2)
       SUBF   *+AR2,*+AR0,R4
*                              ;  R4 = Y(I)±Y(I2)
       ADDF   *AR3,*AR1,R5     ;  R5 = X(I1)+X(I3)
       SUBF   R1,R5,R6         ;  R6 = R5±R1
       ADDF   R5,R1            ;  R1 = R1+R5
       ADDF   *+AR3,*+AR1,R5
*                              ;  R5 = Y(I1)+Y(I3)
       SUBF   R5,R3,R7         ;  R7 = R3±R5
       ADDF   R5,R3            ;  R3 = R3+R5
       STF    R3,*+AR0         ;  Y(I) = R3+R5
||     STF    R1,*AR0++(IR0)   ;  X(I) = R1+R5
       SUBF   *AR3,*AR1,R1     ;  R1 = X(I1)±X(I3)
       SUBF   *+AR3,*+AR1,R3
*                              ;  R3 = Y(I1)±Y(I3)
       STF    R6,*+AR1         ;  Y(I1) = R5±R1
```

Example 6–15. Complex Radix-4 DIF FFT (Continued)

```
||     STF    R7,*AR1++(IR0)   ;  X(I1) = R3±R5
       ADDF   R3,R2,R5         ;  R5 = R2+R3
       SUBF   R2,R3,R2         ;  R2 = ±R2+R3
       SUBF   R1,R4,R3         ;  R3 = R4±R1
       ADDF   R1,R4            ;  R4 = R4+R1
       SUBF   R5,R3,R1         ;  R1 = R3±R5
       MPYF   *AR4,R1          ;  R1 = R1*CO21
       ADDF   R5,R3            ;  R3 = R3+R5
       MPYF   *AR4,R3          ;  R3 = R3*CO21
||     STF    R1,*+AR2         ;  Y(I2) = (R3±R5)*CO21
       SUBF   R4,R2,R1         ;  R1 = R2±R4
       MPYF   *AR4,R1          ;  R1 = R1*CO21
||     STF    R3,*AR2++(IR0)   ;  X(I2) = (R3+R5)*CO21
       ADDF   R4,R2            ;  R2 = R2+R4
       MPYF   *AR4,R2          ;  R2 = R2*CO21
BLK3   STF    R1,*+AR3         ;  Y(I3) = ±(R4±R2)*CO21
||     STF R2,*AR3++(IR0)      ;  X(I3) = (R4+R2)*CO21
       CMPI   @LPCNT,R0
       BPD    INLOP            ;  Loop back to the inner loop

CONT   LDI    @RPTCNT,AR7
       LDI    @IEINDX,AR6
       LSH    2,AR7            ;  Increment repeat counter for
*                               ;     next time
       STI    AR7,@RPTCNT
       LSH    2,AR6            ;  IE = 4*IE
       STI    AR6,@IEINDX
       LDI    R0,IR0           ;  N1 = N2
       LSH    -3,R0
       ADDI   2,R0
       STI    R0,@JT           ;  JT = N2/2+2
       SUBI   2,R0
       LSH    1,R0             ;  N2 = N2/4
       BR     LOOP             ;  Next FFT stage

*   STORE RESULT USING BIT±REVERSED ADDRESSING

END:   LDI    @FFTSIZ,RC       ;  RC = N
       SUBI   1,RC             ;  RC should be one less than desired #
       LDI    @FFTSIZ,IR0      ;  IR0 = size of FFT = N
       LDI    2,IR1
       LDI    @INPUT,AR0
       LDP    STORE
       LDI    @STORE,AR1
       RPTB   BITRV
       LDF    *+AR0(1),R0
||     LDF    *AR0++(IR0)B,R1
BITRV  STF    R0,*+AR1(1)
||     STF    R1,*AR1++(IR1)
SELF   BR SELF                 ;  Branch to itself at the end
       .end
```

6.6.4 Real Radix-2 FFT

In many cases, the data to be transformed is usually a sequence of real numbers. This real input data has properties that reduce the computational load of the FFT algorithm even further. The FFT algorithm that exploits such properties is called a real radix-2 FFT. Example 6–16 shows the generic implementation of a real-valued, forward radix-2 FFT. For such an FFT, the total storage required for a length-N transform is only N locations; in a complex FFT, 2N locations are necessary. Recovery of the rest of the points is based on the symmetry conditions.

Example 6–16. Real Forward Radix-2 FFT

```
*************************************************************************
*  FILENAME         : ffft_rl.asm
*
*  WRITTEN BY       : Alex Tessarolo
*                     Texas Instruments, Australia
*
*  DATE             : 23rd July 1991
*
*  VERSION          : 2.0
*
*************************************************************************

*  VER    DATE             COMMENTS
*  ---    ------------     ------------------------------------------------
*  1.0    18th July 91     Original release.
*  2.0    23rd July 91     Most stages modified.
*                          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.
*************************************************************************

*  SYNOPSIS: int       ffft_rl( FFT_SIZE, LOG_SIZE, SOURCE_ADDR, DEST_ADDR,
*                          SINE_TABLE, BIT_REVERSE );
*
*            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, input bit is provided, reversed
                                     ; 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 6–16. Real Forward Radix-2 FFT (Continued)

```
*   DESCRIPTION: Generic function to do a radix–2 FFT computation on the C30.
*                The data array is FFT_SIZE–long with only real data. The out-
*                put is stored in the same locations 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 begin-
*                ning of the function.
*                If bit reversal is selected (bit reverse ≠ 0), the data
*                input is expected in bit-reverse order
*                The sine/cosine table for the twiddle factors is ex-
*                pected to be supplied in the following format:
*
*                SINE_TABLE[0]s               → 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.
*
*                Stack structure upon call:
*
*
*                –FP(7)  BIT_REVERSE
*                –FP(6)  SINE_TABLE
*                –FP(5)  DEST_ADDR
*                –FP(4)  SOURCE_ADDR
*                –FP(3)  LOG_SIZE
*                –FP(2)  FFT_SIZE
*                –FP(1)  returne
*                –FP(0)  addr
*                        old FP
*
*******************************************************************************
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
*
*                NOTE:     Calling C program can be compiled using either large
*                          or small model.
*
*                WARNING:  DP initialized only once in the program. Be wary
*                          with interrupt service routines. Make sure interrupt
*                             service routines save the DP pointer.
*
*                WARNING:  The DEST_ADDR must be aligned such that the first
*                          LOG_SIZE bits are zero (this is not checked by the
*                          program).
*
*************************************************************************
*
*   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 primary/expansion bus RAM
*                               – Sine/cosine tables in RAM0
*                               – Processing and data destination in RAM1.
*                               – Primary/expansion bus RAM, 0 wait state.
*
*                FFT Size  Bit Reversing    Data Source  Cycles(C30)
*                --------  -------------    -----------  -----------
*                1024         OFF              RAM1      19816 approx.
*                Note:  This number does not include the C callable overheads.
*                       Add 57 cycles for these overheads.
*
*************************************************************************

FP               .set      AR3

                 .global   _ffft_rl     ; Entry execution point.

FFT_SIZE:        .usect    ".fftdata",1 ; Reserve memory for arguments.
LOG_SIZE:        .usect    ".fftdata",1
SOURCE_ADDR:     .usect    ".fftdata",1
DEST_ADDR:       .usect    ".fftdata",1
SINE_TABLE:      .usect    ".fftdata",1
BIT_REVERSE:     .usect    ".fftdata",1
SEPARATION:      .usect    ".fftdata",1
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
                                        ;
                                        ; Initialize C function.
                                        ;
                .sect     ".ffttext"

_ffft_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      ; Init. DP pointer.

                LDI       *-FP(2),R0    ; Move arguments from stack.
                STI       R0,@FFT_SIZE
                LDI       *-FP(3),R0
                STI       R0,@LOG_SIZE
                LDI       *-FP(4),R0
                STI       R0,@SOURCE_ADDR
                LDI       *-FP(5),R0
                STI       R0,@DEST_ADDR
                LDI       *-FP(6),R0
                STI       R0,@SINE_TABLE
                LDI       *-FP(7),R0
                STI       R0,@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
            CMPI      0,R0
            BZ        MOVE_DATA

                                        ;
                                        ; Check bit reversing type.
                                        ;
                                        ; If SourceAddr = DestAddr, then in place
                                        ;  bit reversing.
                                        ; If SourceAddr <> DestAddr, then
                                        ;  standard bit reversing.
                                        ;
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
             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
             LDI       @FFT_SIZE,IR0
             LSH       –1,IR0           ; IRO  =  half FFT size.
             LDI       @SOURCE_ADDR,AR0
             LDI       @DEST_ADDR,AR1

             LDF       *AR0++,R1

             RPTS      R0
             LDF       *AR0++,R1
       ||    STF       R1,*AR1++(IR0)B

             STF       R1,*AR1++(IR0)B

             BR        START

                                    ;
                                    ; In-place bit reversing.
                                    ;
                                    ; Bit reversing on even locations,
                                    ;  1st half only.
IN_PLACE:    LDI       @FFT_SIZE,IR0
             LSH       –2,IR0           ; IRO  =  quarter FFT size.
             LDI       2,IR1

             LDI       @FFT_SIZE,RC
             LSH       –2,RC
             SUBI      3,RC
             LDI       @DEST_ADDR,AR0
             LDI       AR0,AR1
             LDI       AR0,AR2

             NOP       *AR1++(IR0)B
             NOP       *AR2++(IR0)B
             LDF       *++AR0(IR1),R0
             LDF       *AR1,R1
             CMPI      AR1,AR0          ; Xchange locs only if AR0<AR1.
             LDFGT     R0,R1
             LDFGT     *AR1++(IR0)B,R1
```

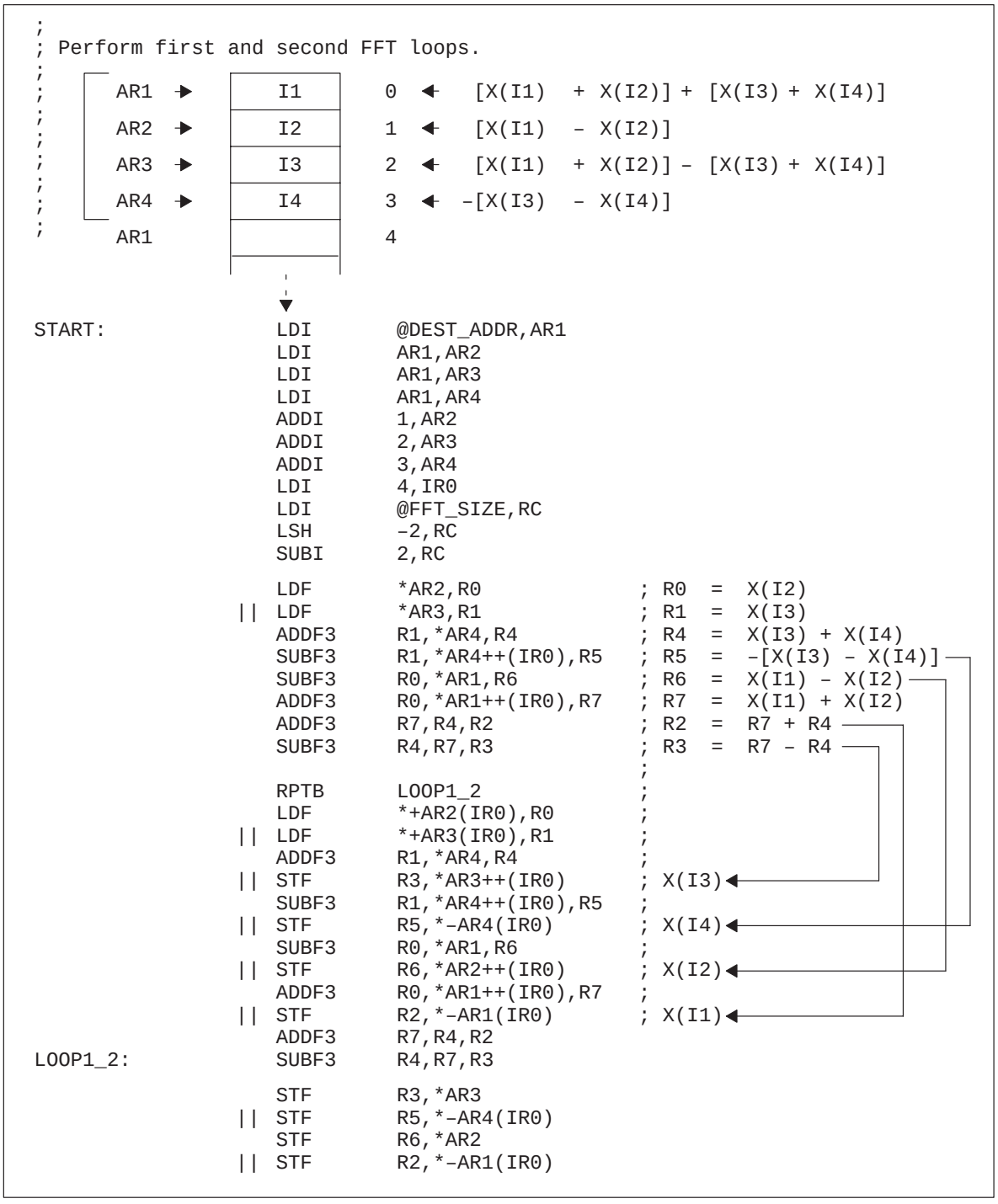
Example 6–16. Real Forward Radix-2 FFT (Continued)

```
              RPTB      BITRV1
              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 on odd
                                    ;  locations, 2nd half only.

              LDI       @FFT_SIZE,RC
              LSH       -1,RC
              LDI       @DEST_ADDR,AR0
              ADDI      RC,AR0
              ADDI      1,AR0
              LDI       AR0,AR1
              LDI       AR0,AR2
              LSH       -1,RC
              SUBI      3,RC

              NOP       *AR1++(IR0)B
              NOP       *AR2++(IR0)B
              LDF       *++AR0(IR1),R0
              LDF       *AR1,R1
              CMPI      AR1,AR0           ;  Xchange locs only if AR0<AR1.
              LDFGT     R0,R1
              LDFGT     *AR1++(IR0)B,R1

              RPTB      BITRV2
              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 on odd
          ;   locations, 1st half only.
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

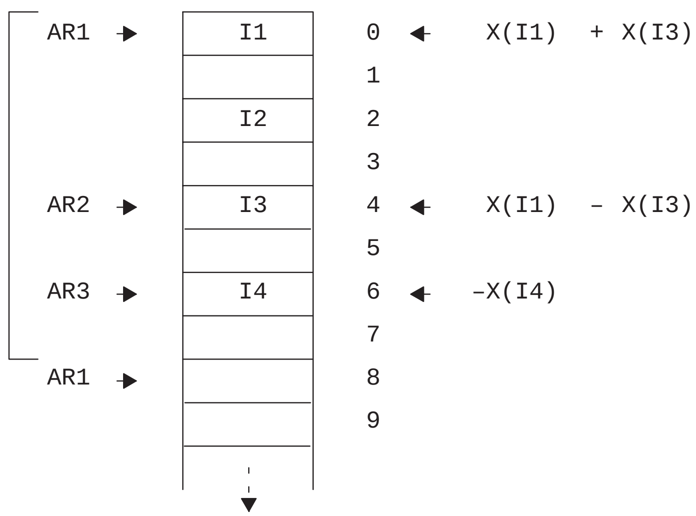
```
              LDI    @FFT_SIZE,RC
              LSH    -1,RC
              LDI    RC,IR0
              LDI    @DEST_ADDR,AR0
              LDI    AR0,AR1
              ADDI   1,AR0
              ADDI   IR0,AR1
              LSH    -1,RC
              LDI    RC,IR0
              SUBI   2,RC

              LDF    *AR0,R0
              LDF    *AR1,R1

              RPTB   BITRV3
              LDF    *++AR0(IR1),R0
       ||     STF    R0,*AR1++(IR0)B
BITRV3:       LDF    *AR1,R1
       ||     STF    R1,*-AR0(IR1)

              STF    R0,*AR1
              STF    R1,*AR0

              BR     START

          ;
          ;   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    START

              LDI    @FFT_SIZE,R0
              SUBI   2,R0
              LDI    @SOURCE_ADDR,AR0
              LDI    @DEST_ADDR,AR1

              LDF    *AR0++,R1

              RPTS   R0
              LDF    *AR0++,R1
       ||     STF    R1,*AR1++

              STF    R1,*AR1
```

Example 6–16. Real Forward Radix-2 FFT (Continued)



Example 6–16. Real Forward Radix-2 FFT (Continued)

```
;
;   Perform third FFT loop.
;   Part A:
;
```




```
                     LDI      @DEST_ADDR,AR1
                     LDI      AR1,AR2
                     LDI      AR1,AR3
                     ADDI     4,AR2
                     ADDI     6,AR3
                     LDI      8,IR0
                     LDI      @FFT_SIZE,RC
                     LSH      –3,RC
                     SUBI     2,RC

                     SUBF3    *AR2,*AR1,R1
                     ADDF3    *AR2,*AR1,R2
                     NEGF     *AR3,R3

                     RPTB     LOOP3_A
                     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)      ;
LOOP3_A:             NEGF     *AR3,R3             ; R3  =  –X(I4)
                                                  ;
                     STF      R2,*AR1             ; X(I1)
                     STF      R1,*AR2             ; X(I3)
                     STF      R3,*AR3             ; X(I4)
```

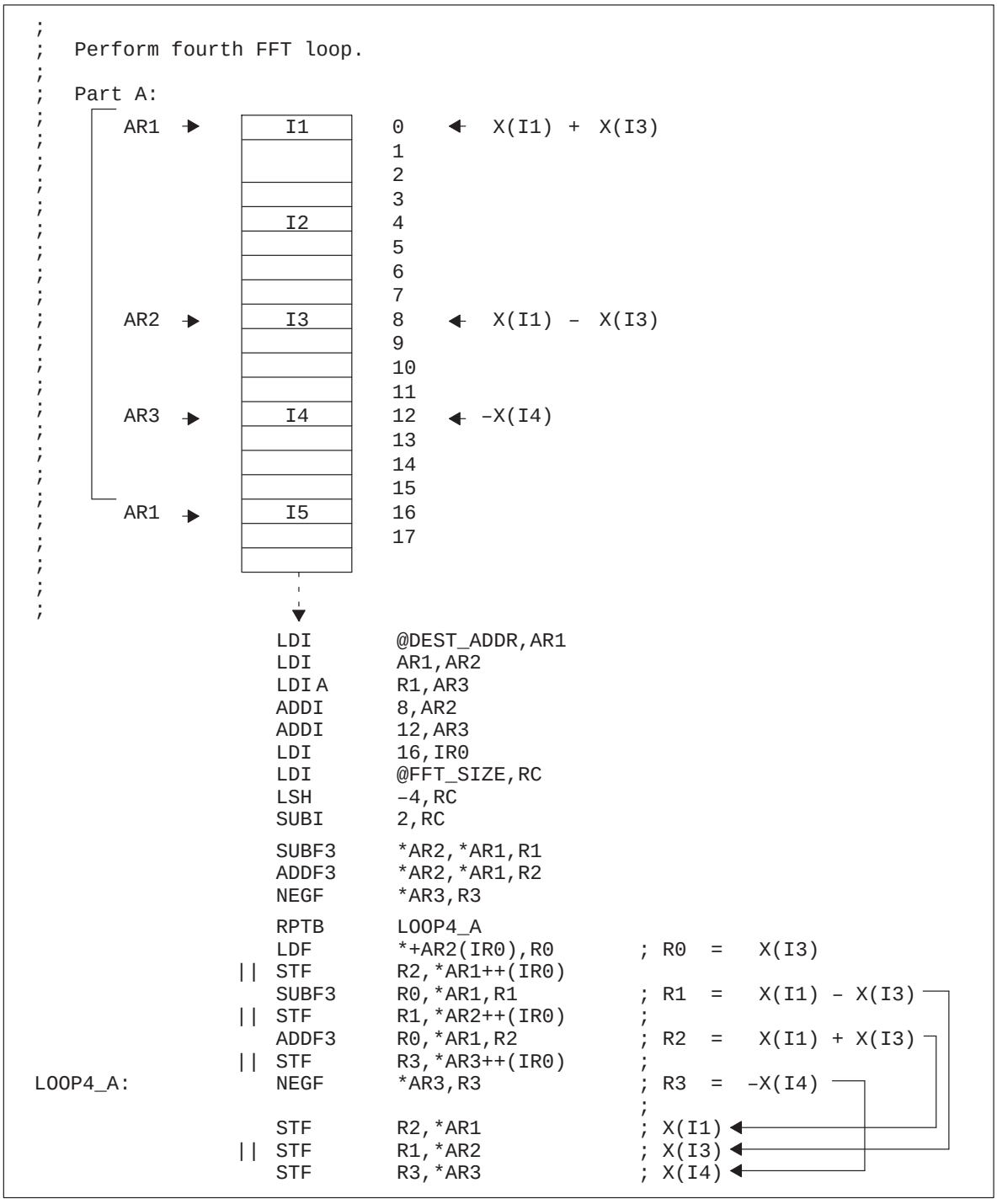
Example 6–16. Real Forward Radix-2 FFT (Continued)

```
;
;   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
                 LDI      RC,IR1
                 SUBI     3,RC
                 LDI      8,IR0
                 LDI      @DEST_ADDR,AR0
                 LDI      AR0,AR1
                 LDI      AR0,AR2
                 LDI      AR0,AR3
                 ADDI     1,AR0
                 ADDI     3,AR1
                 ADDI     5,AR2
                 ADDI     7,AR3

                 LDI      @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
              || SUBF3    R0,R1,R3            ; R3 = –[X(I3)*COS – X(I4)*COS]
                 SUBF3    *AR1,R3,R4          ; R4 = –X(I2) + R3
                 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)
                                              ;
                 RPTB     LOOP3_B             ;
                 MPYF3    *AR3,R7,R1          ;
              || STF      R4,*AR0++(IR0)      ; X(I1)
                 ADDF3    R0,R1,R2
                 MPYF3    *AR7,*+AR2(IR0),R0
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
                  || 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
```

Example 6–16. Real Forward Radix-2 FFT (Continued)



Example 6–16. Real Forward Radix-2 FFT (Continued)

```
;
; 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)*COS– 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
                    └──────────┘
                         ¦
                         ▼

          LDI       @FFT_SIZE,RC
          LSH       –4,RC
          LDI       RC,IR1
          LDI       2,IR0
          SUBI      3,RC
          LDI       @DEST_ADDR,AR0
          LDI       AR0,AR1
          LDI       AR0,AR2
          LDI       AR0,AR3
          LDI       AR0,AR4
          ADDI      1,AR0
          ADDI      7,AR1
          ADDI      9,AR2
          ADDI      15,AR3
          ADDI      11,AR4

          LDI       @SINE_TABLE,AR7
          LDF       *++AR7(IR1),R7      ; R7   =  SIN(1*[2*pi/16])
                                        ; *AR7  =  COS(3*[2*pi/16])
          LDI       AR7,AR6
          LDF       *++AR6(IR1),R6      ; R6   =  SIN(2*[2*pi/16])
                                        ; *AR6  =  COS(2*[2*pi/16])
          LDI       AR6,AR5
          LDF       *++AR5(IR1),R5      ; R5   =  SIN(3*[2*pi/16])
                                        ; *AR5  =  COS(1*[2*pi/16])

          LDI       16,IR1
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
        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)
        ADD     F3 *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
        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)
        ADDF3   *AR0,R2,R4
    || STF      R4,*AR1++(IR1)

        RPTB    LOOP4_B
        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
```

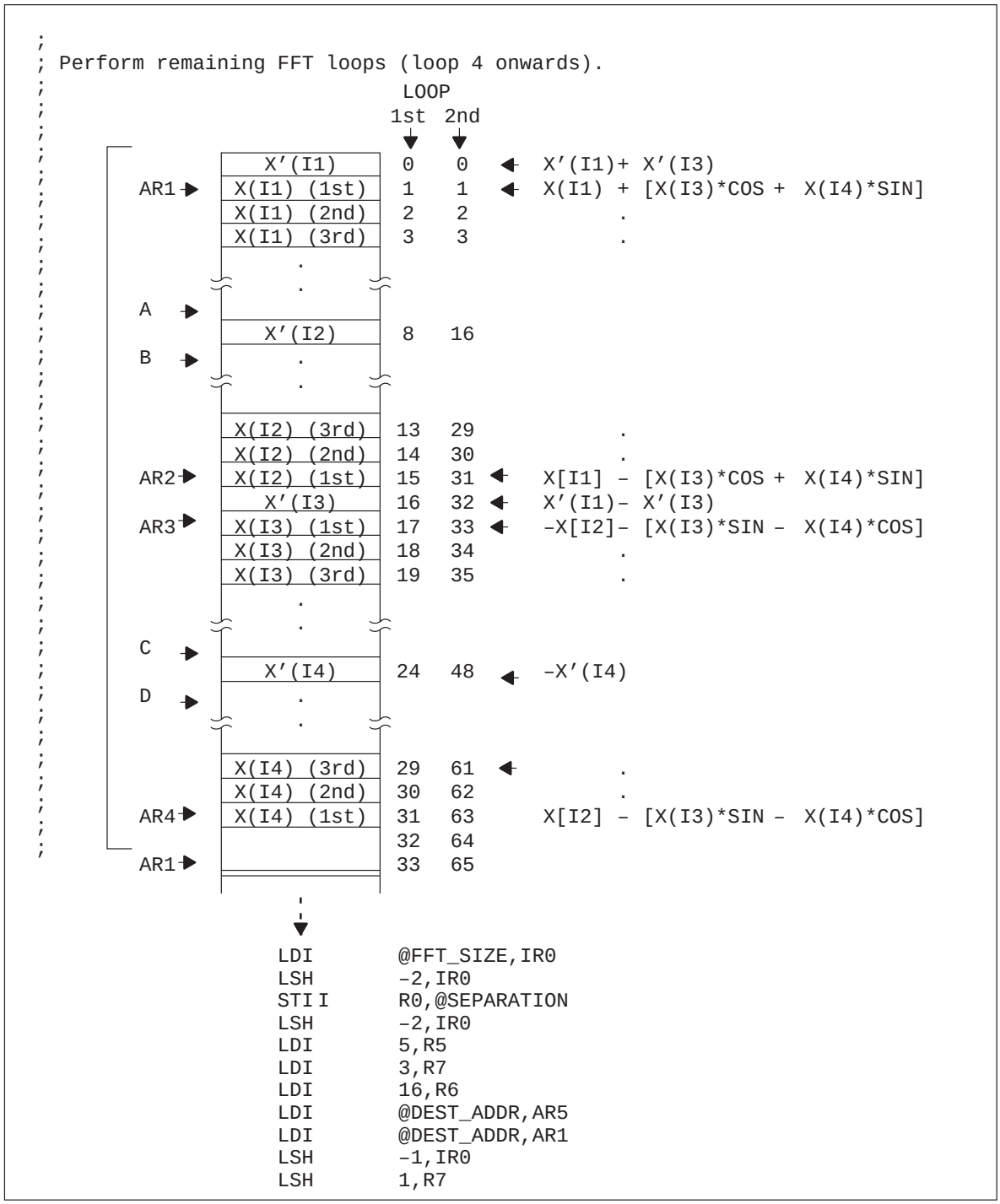
Example 6–16. Real Forward Radix-2 FFT (Continued)

```
                || 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
                || 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
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
||  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
```

Example 6–16. Real Forward Radix-2 FFT (Continued)



Example 6–16. Real Forward Radix-2 FFT (Continued)

```
LOOP:       ADDI    1,R7
            LSH     1,R6
            LDI     AR1,AR4
            ADDI    R7,AR1                  ; AR1 points at A.
            LDI A   R1,AR2
            ADDI    2,AR2                   ; AR2 points at B.
            ADDI    R6,AR4
            SUBI    R7,AR4                  ; AR4 points at D.
            LDI     AR4,AR3
            SUBI    2,AR3                   ; AR3 points at C.
            LDI     @SINE_TABLE,AR0         ; AR0 points at SIN/COS table.
            LDI     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)
            LDI     @SEPARATION,IR1             ; IR1=SEPARATION
                                                BETWEEN SIN/COS TBLS
            SUBI    3,RC
            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)
            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)
                                                ;
            RPTB    IN_BLK                      ;
            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--
```

Example 6–16. Real Forward Radix-2 FFT (Continued)

```
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
              LDI       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
              LDI       @SINE_TABLE,AR0     ; AR0 POINTS TO
                                              SIN/COS TABLE
              LDI       R7,IR1
              LDI       R7,RC

              ADDI      1,R5
              CMPI      @LOG_SIZE,R5
              BLED      LOOP
              LDI       @DEST_ADDR,AR1
              LSH       -1,IR0
              LSH       1,R7

                                            ; 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
```

Example 6–17 shows the implementation of a radix-2 real inverse FFT. The inverse transformation assumes that the input data is in the same order as the output of the forward transformation. It also produces a time signal in the proper order. In other words, bit reversing takes place at the end of the program.

Example 6–17. Real Inverse Radix-2 FFT

```
*   Real Inverse FFT
*******************************************************************************
*
*   FILENAME  :  ifft_rl.asm
*
*   WRITTEN BY    :  Daniel Mazzocco
*                    Texas Instruments, Houston
*
*   DATE   :  18th Feb 1992
*
*   VERSION   :  1.0
*
*******************************************************************************
*   VER    DATE                                COMMENTS
*   ---    ------------     ---------------------------------------------------
*   1.0    18th Feb 92      Original release. Started from forward real FFT
*                           routine written by Alex Tessarolo, rev 2.0 .
*
*******************************************************************************
*
*   SYNOPSIS:     int       ifft_rl( FFT_SIZE, LOG_SIZE, SOURCE_ADDR,
*                           DEST_ADDR, SINE_TABLE, BIT_REVERSE );
*
*                 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).
*
```

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

```
*  DESCRIPTION: Generic function to do an inverse radix–2 FFT computation
*               on the C30.
*               The data array is FFT_SIZE long with real and imaginary
*               points R and I as follows:
*
*               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] s➔in((FFT_SIZE/2–1)*2*pi/FFT_SIZE)
*
*               NOTE: The table is the first half period of a sine wave.
*
*               Stack structure upon call:
*
*
*                         –FP(7)  | BIT_REVERSE |
*                         –FP(6)  | SINE_TABLE  |
*                         –FP(5)  | DEST_ADDR   |
*                         –FP(4)  | SOURCE_ADDR |
*                         –FP(3)  | LOG_SIZE    |
*                         –FP(2)  | FFT_SIZE    |
*                         –FP(1)  | returne     |
*                         –FP(0)  | addr        |
*                                 | old FP      |
*
*******************************************************************************
```

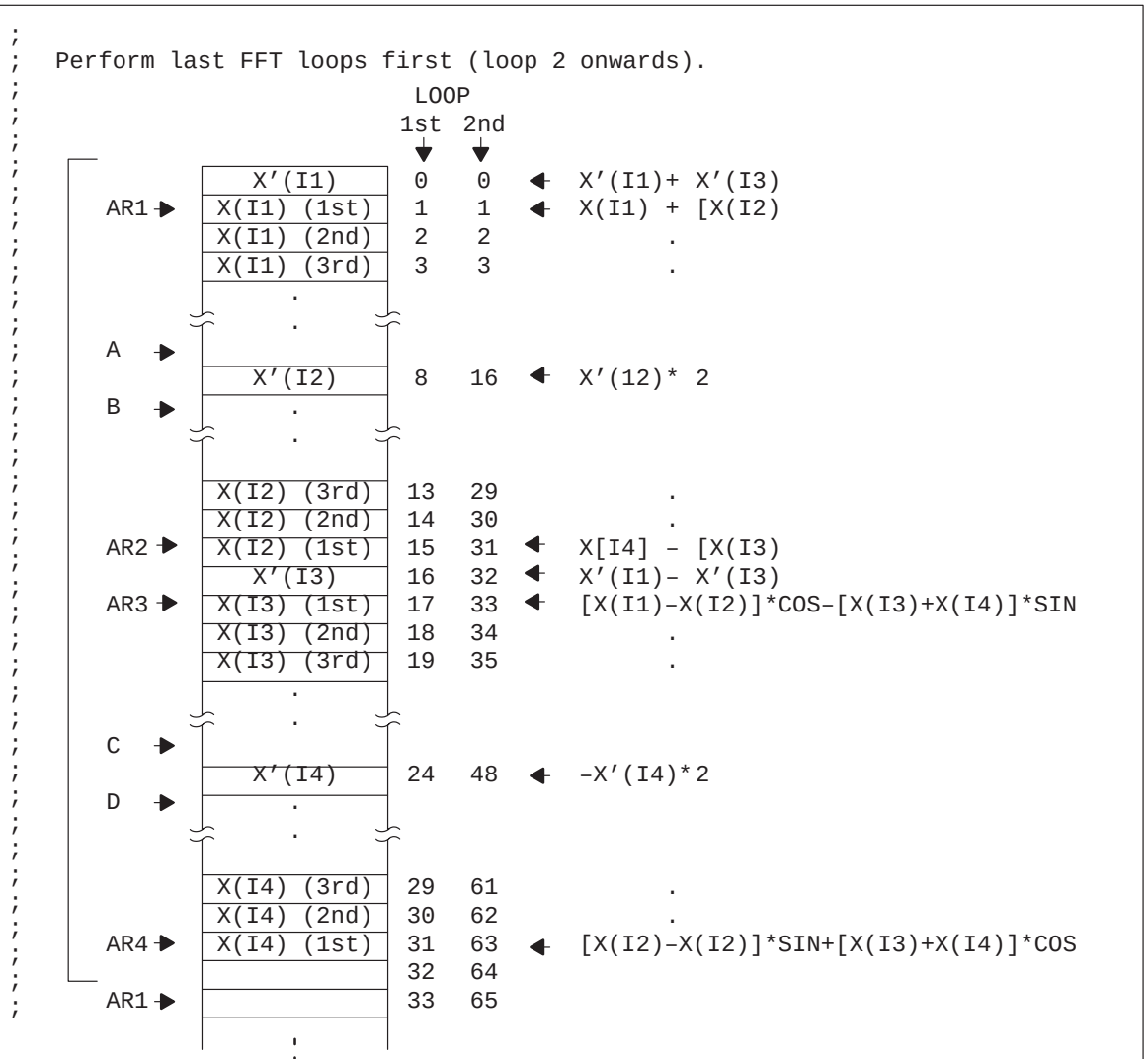
Example 6–17. Real Inverse Radix-2 FFT (Continued)

```
*         NOTE:     Calling C program can be compiled using either large
*                   or small model.
*
*         WARNING:  DP initialized only once in the program. Be wary
*                   with interrupt service routines. Make sure interrupt
*                   service routines save the DP pointer.
*
*         WARNING:  The SOURCE_ADDR must be aligned such that the first
*                   LOG_SIZE bits are zero (this is not checked by the
*                   program).
*
*************************************************************************
*
*   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
*                          Stack    =  12 words
*
*************************************************************************

*
* BENCHMARKS:       Assumptions   – Program in RAM0
*                                 – Reserved data in RAM0
*                                 – Stack on primary/expansion bus RAM
*                                 – Sine/cosine tables in RAM0
*                                 – Processing and data destination in RAM1
*                                 – Primary/expansion bus RAM, 0 wait state
*
*                   FFT Size      Bit Reversing    Data Source      Cycles(C30)
*                   --------      -------------    -----------      -----------
*                   1024          OFF              RAM1             25892 approx.
*                   Note:  This number does not include the C callable overheads.
*                          Add 57 cycles for these overheads.
*************************************************************************

FP                  .set   AR3

                    .global      _ifft_rl         ; Entry execution point.

FFT_SIZE:           .usect ”   .ifftdata”,1       ; Reserve memory for arguments.
LOG_SIZE:           .usect ”   .ifftdata”,1
SOURCE_ADDR:        .usect ”   .ifftdata”,1
DEST_ADDR:          .usect ”   .ifftdata”,1
SINE_TABLE:         .usect ”   .ifftdata”,1
BIT_REVERSE:        .usect ”   .ifftdata”,1
SEPARATION:         .usect ”   .ifftdata”,1
```

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

```
                                                ;
                                                ; Initialize C Function.
                                                ;
                    .sect     ".iffttext"

_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.

                    LDI       *-FP(2),R0        ; Move arguments from stack.
                    STI       R0,@FFT_SIZE
                    LDI       *-FP(3),R0
                    STI       R0,@LOG_SIZE
                    LDI       *-FP(4),R0
                    STI       R0,@SOURCE_ADDR
                    LDI       *-FP(5),R0
                    STI       R0,@DEST_ADDR
                    LDI       *-FP(6),R0
                    STI       R0,@SINE_TABLE
                    LDI       *-FP(7),R0
                    STI       R0,@BIT_REVERSE
```

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

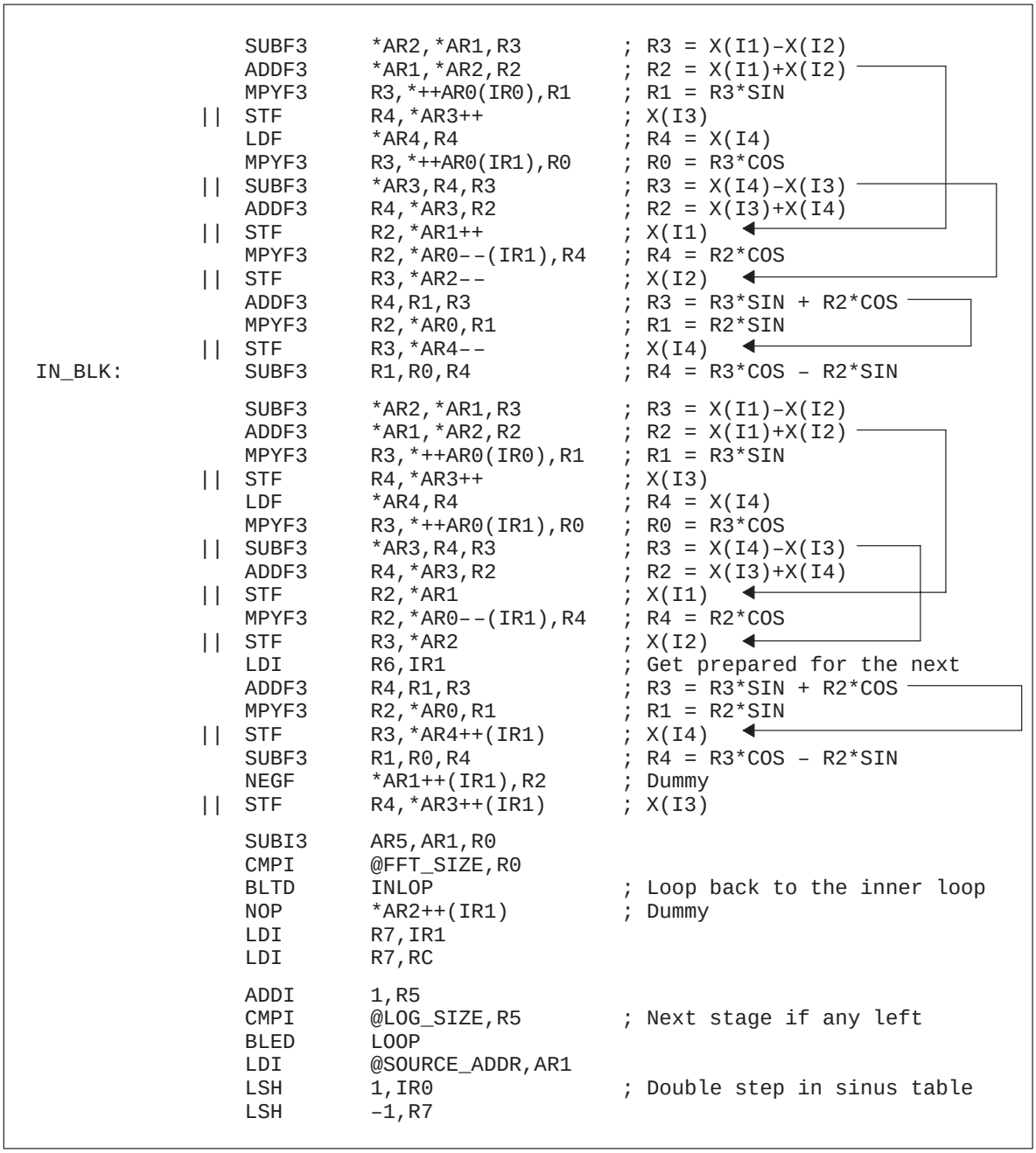



```
                LDI     1,IR0           ; Step between two consecutive sines
                LDI     4,R5            ; Stage number from 4 to M.
                LDI     @FFT_SIZE,R7
                LSH     –2,R7           ; R7 is FFT_SIZE/4–1 (ie 15 for 64 pts)
                SUBI    1,R7            ; and will be used to point at A & D.
                LDI     @FFT_SIZE,R6    ; R6 will be used to point at D.
                LSH     1,R6
                LDI     @SOURCE_ADDR,AR5
                LDI     @SOURCE_ADDR,AR1
LOOP:           LSH     –1,R6           ; R6 is FFT_SIZE at the 1st loop.
                LDI     AR1,AR4
                ADDI    R7,AR1          ; AR1 points at A.
```

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

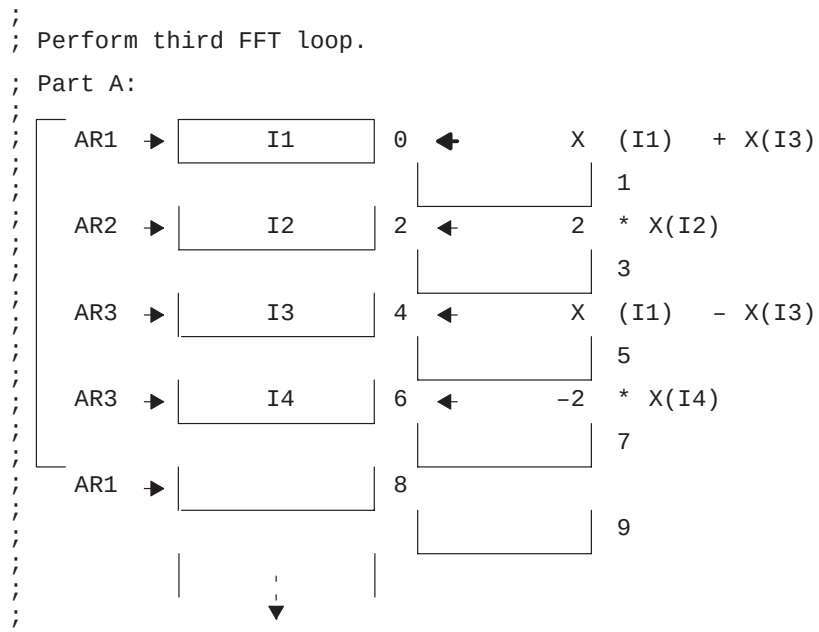
```
                LDI     AR1,AR2
                ADDI    2,AR2               ; AR2 points at B.
                ADDI    R6,AR4
                SUBI    R7,AR4 ; AR4 points at D.
                LDI     AR4,AR3
                SUBI    2,AR3  ; AR3 points at C.

                LDI     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 R   1,*AR3++            ; X'(I3)
                MPYF    2.0,R3              ; R3 = 2*X'(I2)
                STF     R3,*AR2++(IR1)      ; X'(I2)
             || STF     R2,*AR4++(IR1)      ; X'(I4)

                LDI     @FFT_SIZE,IR1       ; IR1=separation between SIN/
                                            ; COS tbls
                LDI     @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)
                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

                RPTB    IN_BLK
```

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



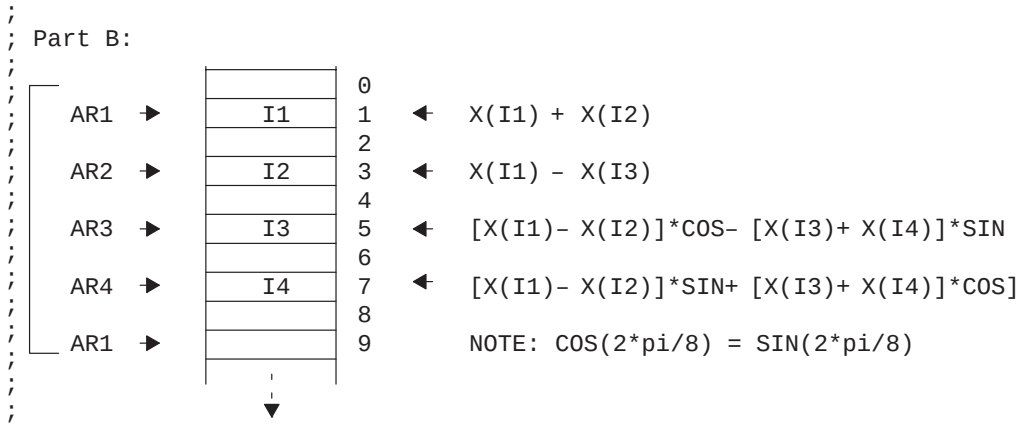
```
                SUBF3     *AR2,*AR1,R3        ; R3 = X(I1)–X(I2)
                ADDF3     *AR1,*AR2,R2        ; R2 = X(I1)+X(I2)
                MPYF3     R3,*++AR0(IR0),R1   ; R1 = R3*SIN
             || STF       R4,*AR3++           ; X(I3)
                LDF       *AR4,R4             ; R4 = 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)
                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)
                LDI       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
                LDI       R7,IR1
                LDI       R7,RC

                ADDI      1,R5
                CMPI      @LOG_SIZE,R5        ; Next stage if any left
                BLED      LOOP
                LDI       @SOURCE_ADDR,AR1
                LSH       1,IR0               ; Double step in sinus table
                LSH       –1,R7
```

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



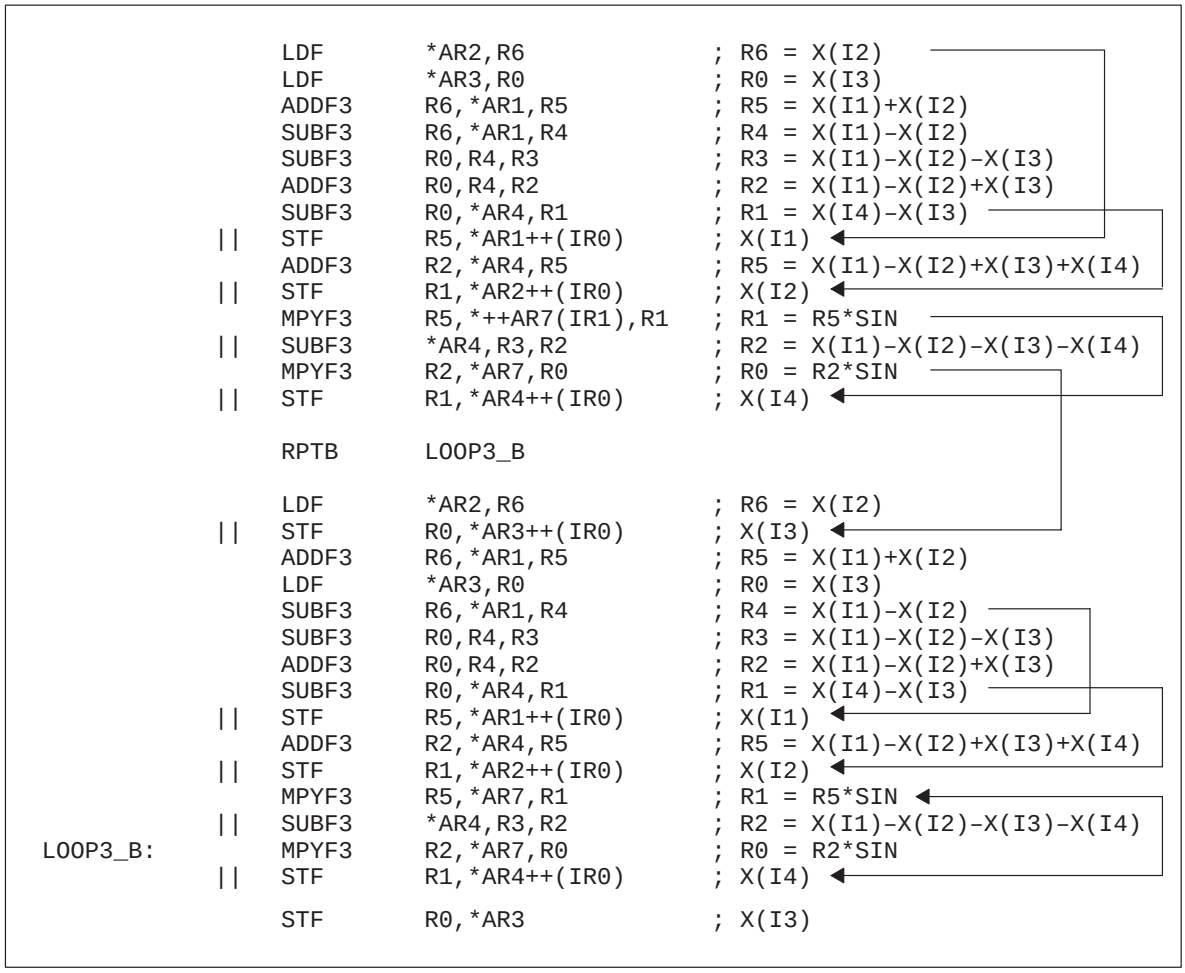

```
                LDI     @SOURCE_ADDR,AR1
                LDI     AR1,AR2
                LDI     AR1,AR3
                LDI     AR1,AR4
                ADDI    2,AR2
                ADDI    4,AR3
                ADDI    6,AR4
                LDI     8,IR0
                LDI     @FFT_SIZE,RC
                LSH     –3,RC
                SUBI    1,RC
                LDI     @SINE_TABLE,AR0     ; AR0 points at SIN/COS table.

                RPTB    LOOP3_A
                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)
```

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



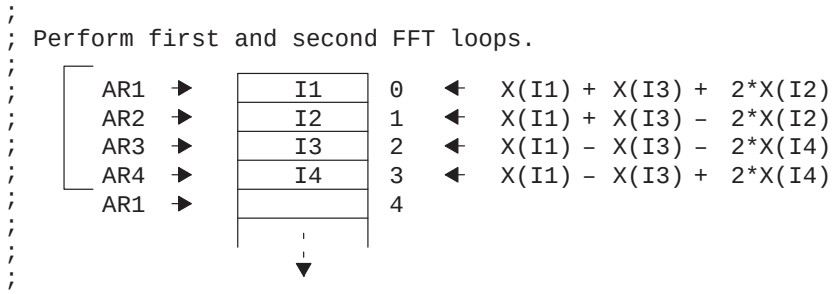

```
        LDI     @SOURCE_ADDR,AR1
        LDIA    R1,AR2
        LDIA    R1,AR3
        LDIA    R1,AR4
        ADDI    1,AR1
        ADDI    3,AR2
        ADDI    5,AR3
        ADDI    7,AR4
        LDI     @SINE_TABLE,AR7     ; AR7 points at SIN/COS table.
        LDI     @FFT_SIZE,RC
        LSH     –3,RC
        LDI     RC,IR1
        SUBI    2,RC
```

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



```
                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)
                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)

                RPTB    LOOP3_B

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




```
                LDI     @SOURCE_ADDR,AR1
                LDI     AR1,AR2
                LDI     AR1,AR3
                LDI     AR1,AR4
                ADDI    1,AR2
                ADDI    2,AR3
                ADDI    3,AR4
                LDI     4,IR0
                LDI     @FFT_SIZE,RC
                LSH     –2,RC
                SUBI    2,RC
```

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

```
                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)
                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)
                                            ;
                RPTB    LOOP1_2             ;
                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)
              ||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)
```

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

```
;
; Check bit reversing mode (on or off).
;
; BIT_REVERSING =  0, then OFF (no bit reversing).
; BIT_REVERSING <> 0, then ON.
;
                LDI       @BIT_REVERSE,R0
                CMPI      0,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
                LDI       @FFT_SIZE,IR0
                LSH       –1,IR0 ; IRO = half FFT size.
                LDI       @SOURCE_ADDR,AR0
                LDI       @DEST_ADDR,AR1

                LDF       *AR0++,R1

                RPTS      R0
                LDF       *AR0++,R1
             || STF       R1,*AR1++(IR0)B

                STF       R1,*AR1++(IR0)B

                BR        DIVISION
```

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

```
                                        ;
                                        ; In-place bit reversing.
                                        ;
                                        ; Bit reversing on even locations, 1st half
                                        ;  only.
IN_PLACE:     LDI       @FFT_SIZE,IR0
              LSH       –2,IR0          ; IRO = quarter FFT size.
              LDI       2,IR1

              LDI       @FFT_SIZE,RC
              LSH       –2,RC
              SUBI      3,RC
              LDI       @DEST_ADDR,AR0
              LDI A     R0,AR1
              LDI A     R0,AR2

              NOP       *AR1++(IR0)B
              NOP       *AR2++(IR0)B
              LDF       *++AR0(IR1),R0
              LDF       *AR1,R1
              CMPI      AR1,AR0         ; Xchange locations only if AR0<AR1.
              LDFGT     R0,R1
              LDFGT     *AR1++(IR0)B,R1

              RPTB      BITRV1
              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 on odd locations,
                                        ;  2nd half only.

              LDI       @FFT_SIZE,RC
              LSH       –1,RC
              LDI       @DEST_ADDR,AR0
              ADDI      RC,AR0
              ADDI      1,AR0
              LDI       AR0,AR1
              LDI       AR0,AR2
              LSH       –1,RC
              SUBI      3,RC

              NOP       *AR1++(IR0)B
              NOP       *AR2++(IR0)B
              LDF       *++AR0(IR1),R0
```

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

```
               LDF       *AR1,R1
               CMPI      AR1,AR0             ; Xchange locations only if AR0<AR1.
               LDFGT     R0,R1
               LDFGT     *AR1++(IR0)B,R1

               RPTB      BITRV2
               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 on odd
                                           ;  locations, 1st half only.

               LDI       @FFT_SIZE,RC
               LSH       -1,RC
               LDI       RC,IR0
               LDI       @DEST_ADDR,AR0
               LDI       AR0,AR1
               ADDI      1,AR0
               ADDI      IR0,AR1
               LSH       -1,RC
               LDI       RC,IR0
               SUBI      2,RC

               LDF       *AR0,R0
               LDF       *AR1,R1

               RPTB      BITRV3
               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.
                                           ;
```

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

```
MOVE_DATA:          LDI     @SOURCE_ADDR,R0
                    CMPI    @DEST_ADDR,R0
                    BEQ D   IVISION

                    LDI     @FFT_SIZE,R0
                    SUBI    2,R0
                    LDI     @SOURCE_ADDR,AR0
                    LDI     @DEST_ADDR,AR1

                    LDF     *AR0++,R1

                    RPTS    R0
                    LDF     *AR0++,R1
                 || STF     R1,*AR1++

                    STF     R1,*AR1

DIVISION:           LDI     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
                    LDI     @DEST_ADDR,AR1
                    LDI     @DEST_ADDR,AR2
                    NOP     *AR2++
                    LDI     @FFT_SIZE,RC
                    LSH     -1,RC
                    SUBI    2,RC
                    MPYF3   R0,*AR1,R1          ; 1st location
                    RPTB    LAST_LOOP
                    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
```

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

```
                                        ; 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.
*
*******************************************************************************
*
```

The ’C3x quickly executes FFT lengths up to 1024 points (complex) or 2048 (real), covering most applications. It performs this task almost entirely in on-chip memory. See Table 6–2 on page 6-79 for the number of CPU clock cycles and the execution time required for FFT lengths between 64 and1024 points for the four algorithms.

6.7 TMS320C3x 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 algorithms in Example 6–13, Example 6–15, Example 6–16, and Example 6–17 beginning on page 6-31.

The benchmarks are given in clock 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 60 MHz 'C3x, multiply by 33 ns.

Table 6–1. TMS320C3x Application Benchmarks

Application	Words	Cycles
Inverse of a floating-point number (32-bit precision)	31	31
Square root	38	46
Double precision integer add/subtract	2	2
Double precision integer multiply	24	24
IEEE to 'C3x format conversion (fast)	12	9
IEEE to 'C3x format conversion (complete)	33	19
'C3x to IEEE format conversion (fast)	14	10
'C3x to IEEE format conversion (complete)	24	27
FIR filter	5	6+N
IIR filter (one biquad)	7	7
IIR filter (N >1 biquads)	16	13+6N
LMS adaptive FIR filter	11	13+3N
Matrix-vector multiplication	10	2+10K+K (N–1)
Vector dot product	6	N+4
Vector maximum	5	2+3N
Forward LPC lattic filter	11	5+3P
Inverse LPC lattice filter	9	6+3P
μ-law (A-law) compression	16(18)	16(18)
μ-law (A-law) expansion	13(15)	16(21)

Table 6–2. TMS320C3x FFT Timing Benchmarks (Assumes Data On Chip and No Bit Reversing)

	Number of CPU Clock Cycles			
Number of Points	**Radix-2 (Complex)**	**Radix-4 (Complex)**	**Radix–2 (Real)**	**Radix-2 (Real Inverse)**
64	1481	2050	791	1064
128	3445	–	1746	2369
256	7865	10400	3925	5282
512	17 709 17 709 (’C31) 42 210 (’C32)	–	8840	11731
1024	39 600 (’C30) 40 100 (’C31) 94 519 (’C32)	50 670	19 820	25 900
512	25 688 (’C32)			
1024	64 781 (’C32)			
2048	11 611 (’C30) 117 400 (’C31)			
4096	280 800 (’C30) 283 600 (’C31)			

These benchmarks include C overhead: they represent the number of cycles between the standard C-compiler _main and _exit labels.

These benchmarks do not include the final bit-reversing stage. If bit-reversing is required, it is implemented in a serial fashion in off-chip memory.

6.8 Sliding FFT

SFFT.ASM uses a technique known as a sliding FFT (SFFT) to calculate the spectrum of a signal on a sample-by-sample basis. The SFFT is particularly well-suited for applications where signal analysis, filtering, modulation, demodulation, or other forms of signal manipulation in the frequency domain must be performed in real time. The SFFT algorithm is similar to the discrete Fourier transform (DFT). The SFFT is equivalent to overlapped FFTs with an overlap of 1 sample, in that the past frequency data is reused to calculate the frequency spectra of the next sample window. The calculation is performed by adding the frequency domain spectra of a new sample, while simultaneously subtracting the frequency domain spectra of the oldest sample. The SFFT does not require first-hand knowledge of the DFT or FFT. In addition, the SFFT can be used to derive the DFT equation, which can be used by DSP beginners or by DSP experts looking for a different approach to solve a problem.

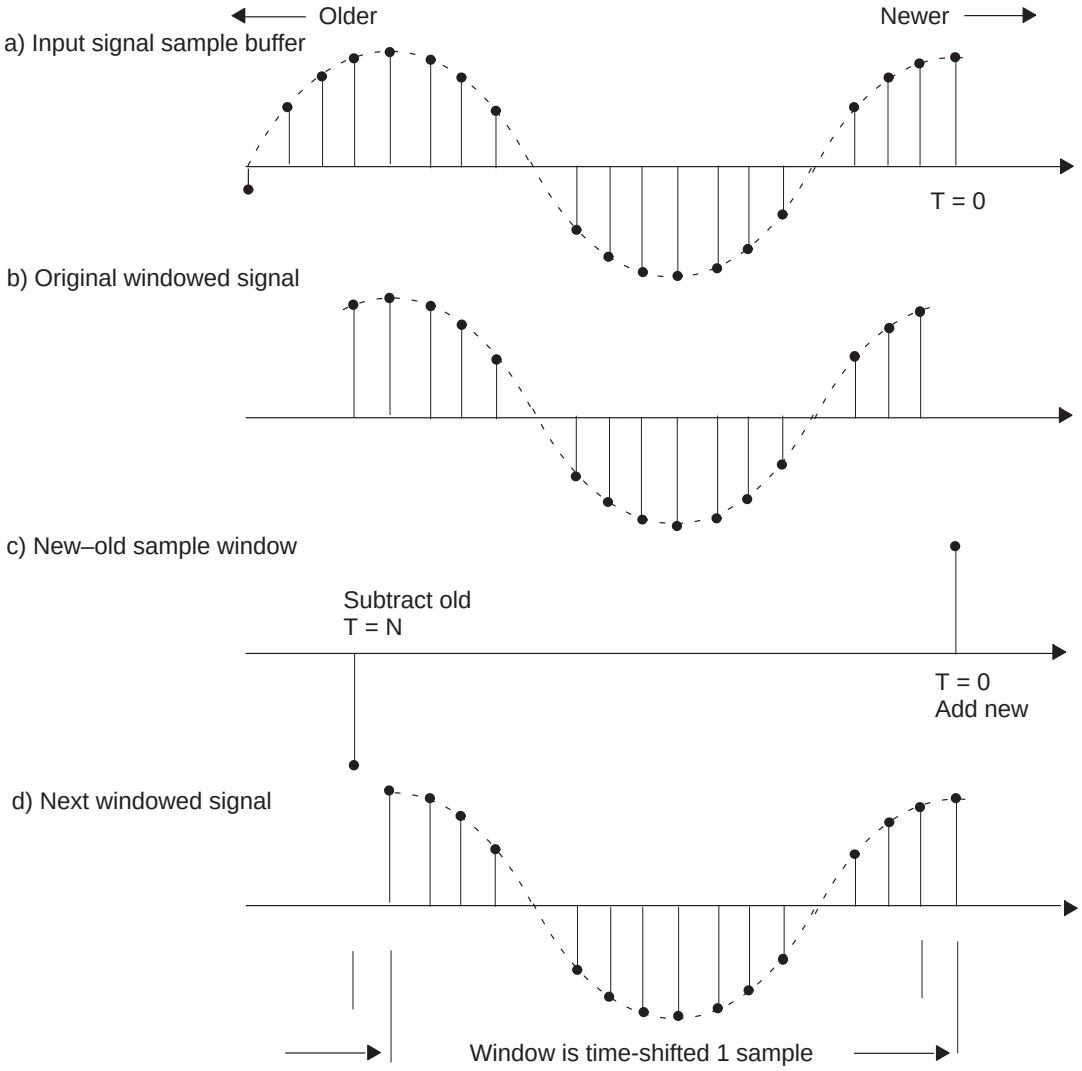
6.8.1 SFFT Theory: A Better Way to Use the Impulse Response

The SFFT is based on the following simple concepts:

1) The property of superposition allows two or more signals to be added linearly to create a new signal. A sampled time domain signal is the summation of a series of individual input samples or impulses of varying magnitude (Figure 6–10a). Similarly, signals, or impulses, can be subtracted.

   If an input signal sample buffer (Figure 6–10a) of data is kept in memory, a sliding rectangular window of data samples (Figure 6–10b and Figure 6–10d) can be constructed by adding the newest sample and subtracting the oldest sample (Figure 6–10c) from the previous original windowed signal (Figure 6–10b). The following diagram shows how the addition and subtraction of samples can 'slide' a window of data samples from those shown in Figure 6–10b to those shown in Figure 6–10d.

Figure 6–10. Input Signal Sample Buffer



Note: T = time

2) The frequency domain response of an impulse, or single sample point where all other data points are zero, results in a flat frequency response with a magnitude in each frequency bin equal to the impulse input magnitude. Conversely, the impulse is the additive result of many sinusoidal frequency components. The time when the impulse occurs within the sample window is determined by the phase angles of the individual component frequencies. An impulse's time of arrival is determined by a linear phase shift between each frequency bin.

3) In the frequency domain, the addition of frequency samples also follows the rules of superposition.

   The spectra of Figure 6–10c, the new–old sample window, is added to the spectra of Figure 6–10b, the original windowed signal, to create the new spectra of Figure 6–10d. The difference is that complex data is used in the frequency domain to represent the phase information of the individual component frequencies.

4) The summation of a series of simple impulse transforms, which have correspondingly simple frequency domain transforms, results in the composite frequency domain transform of the signal.

5) A sliding rectangular window is created by subtracting the Nth oldest sample, which, in the frequency domain, will have gone through a multiple of 2 $\times$ pi radian rotations.

Note:

In some applications, complex time domain inputs may also useful. For this application, only the REAL data from an ADC is used.

6.8.2 Frequency Response Calculation

If an impulse sample occurs at T = 0, the frequency response calculation is further simplified since the response contains only REAL and no IMAG components. The transform of an impulse at T = 0 is simply to store the magnitude of the impulse into each REAL bin, and zero the IMAG bin.

If T != 0, the time shift creates a phase shift or complex vector rotation within each frequency bin. The phase rotation angle is proportional to the time shift and the frequency of interest.

If the time shift is one sample period, as used in the SFFT, special conditions can be applied. At low frequencies, the amount of phase shift from sample to sample is low, or in the case of 0 Hz, zero radians of phase. At higher frequencies, the phase rotation is greatest. At the Nyquist frequency, the vector rotation is pi/2 radians per sample, which corresponds to 2 samples per sine wave cycle. Vector rotation for bins between DC and the Nyquist rate are proportional to the bin frequency.

A Fourier transform also produces both negative and positive frequencies, which are mirror images of each other. Only positive frequencies need to be computed. This is suitable for spectrum analysis and filtering. The ranges for n and the resulting complex rotation vectors (twiddle factors) for each bin are:

```
Positive frequencies      0   <= n < N/2
Negative frequencies     –N/2 <= n < 0
complex(R_phase,I_phase) = exp–j*2*pi*n/N
REAL_tw[n] = cos(n*2*pi/N)
IMAG_tw[n] = sin(n*2*pi/N)
```

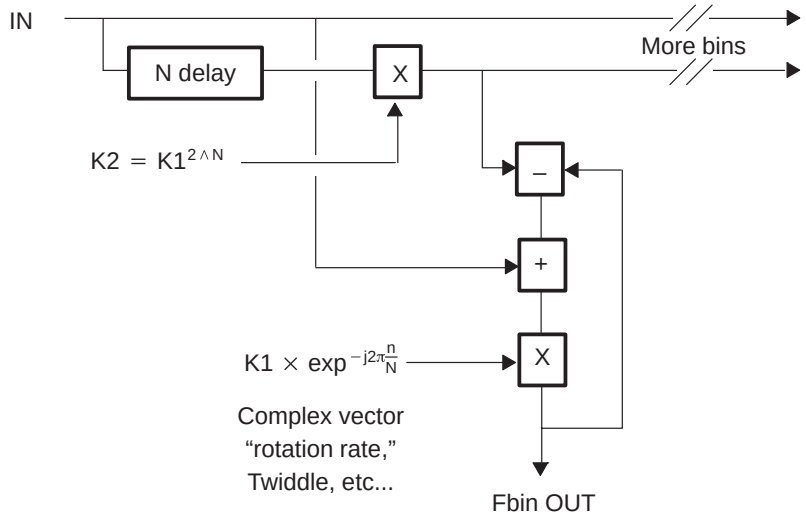
The basic SFFT operation is a vector rotate of each previous bin value; that is, add the newest sample and subtract the oldest sample. Although it is a simple operation, all bins must be computed before the next input sample is ready.

```
NewBinVal = (New – Old) + (OldBinval * vect_rotate)
Bin[n] = (Sample[0]–Sample[N–1]) + (Bin[n] * exp–j*2*pi*n/N)
```

6.8.3 Visualizing the SFFT

The easiest way to visualize the SFFT is to consider that each new sample occurs at T = 0, making each new sample all REAL in the frequency domain. Then, since the past summation is time-shifted by one sample, a vector rotation proportional to the frequency is applied. A schematic representation for an SFFT bin is shown in Figure 6–11.

Figure 6–11. Frequency Bin Diagram (Equivalent to an IIR Filter)



Where: Vector_rotation_rate[n–th Freq] = 2*PI * n / (N*Fs)
K1 & K2 force convergence (see section 6.8.4)

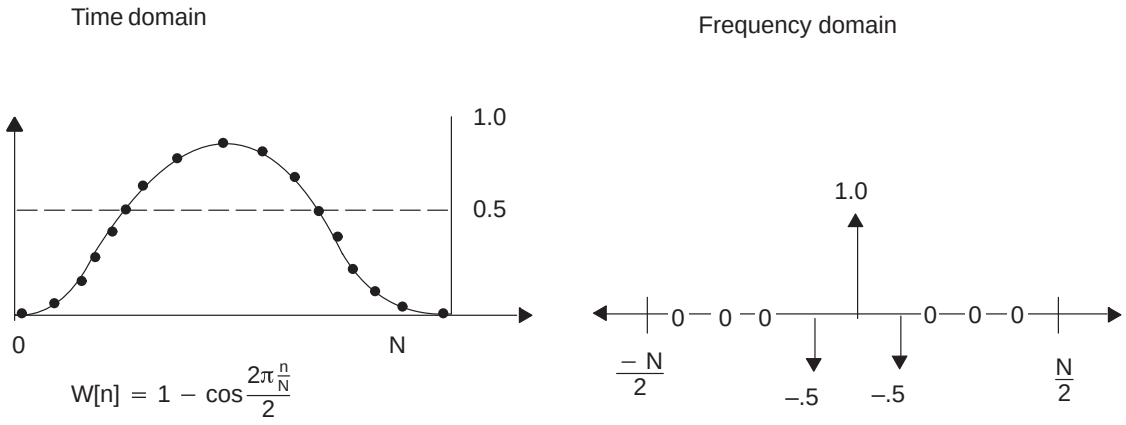
6.8.4 Fbin Convergence and Stability

One aspect of the SFFT is that there is a feedback loop which affects the stability of the bin values. This is similar to an IIR filter where, in the Z domain, a pole sites on the unit circle. To maintain stability and keep the bin values from growing out of control, the magnitude of the complex vector rotation twiddles must be set to slightly less than 1, placing the pole inside the unit circle. This causes the impulse energy magnitude in each bin to decay exponentially towards zero. By adding a stability factor, by Nth bin rotation an impulse decays to $K1^N$ of its original magnitude. To subtract the Nth oldest sample, the Nth oldest sample is scaled by a second coefficient $K2 = K1^N$. A side effect of the exponential decay is that the SFFT is now windowed by an exponentially decaying window. To minimize this effect, keep K1 close to 1.000 (0.999, for example).

6.8.5 SFFT Windowing

Unlike the FFT and DFT, SFFT windowing cannot be performed in the time domain; the input window is moving in time and, therefore, the window function must also move in time. The SFFT windowing operation is performed in the frequency domain using a technique known as convolution. The desirable effect of windowing is a multiplicative process in the time domain whereby the sharp discontinuities at the endpoints, that accompany a rectangular data window, are smoothed out. Without a smoothing window, these abrupt changes smear the frequency spectrum over many bins. In the frequency domain, the coefficients of most windowing functions are simple and do not require large storage arrays. For the raised cosine window function, the coefficients are particularly simple (–.5, +1.0, –.5) and are easily imbedded into the code as addition and subtraction. However, frequency domain (or convolutional window filtering) is applied to the REAL and IMAG data separately before the REAL/IMAG data is combined into a magnitude. The operation is fast and only occurs during output. Furthermore, other window functions are rapidly and easily implemented by selecting different convolution coefficients.

Figure 6–12. Raised Cosine Window



6.8.6 Using SFFT.ASM for Spectrum Analysis

If the SPECT_EN variable is set to 1 (true), the DSK analog output is configured to be the computed spectrum of the analog input beginning at BIN_START and ending at BIN_END. The output is then viewed using an oscilloscope, which is triggered on a positive synch pulse. The DAC output voltage is proportional to the log magnitude of each frequency bin.

To help pass impulses with minimal magnitude errors, each DAC output sample can be repeated up to DAC_RPT times. Also, the AIC TA register value can be programmed to have a very high pass band. This increases the DAC output distortion, which is a problem if used for audio applications, but is acceptable for visual purposes.

Also, the BIN_START and BIN_END values do not need to begin at zero or end at SFFTSIZE/2. This can be used to show that the frequency bins repeat in the frequency domain, as predicted by the discrete Fourier transform. The only restrictions are the availability memory and CPU processing power.

6.8.7 Using SFFT.ASM for Hilbert Transforms and Arbitrary Phase Angles Filters

If SPECT_EN is set to 0, the output is configured to be the summation of the reconstructed REAL and IMAG components.

An arbitrary output phase angle is implemented by performing a complex multiplication of the REAL and IMAG components by a complex vector determined by the ANGLE parameter. If ANGLE = 90°, the Hilbert transform is reconstructed from the pass-band SFFT bins covering BIN_START to BIN_END. If ANGLE = 0.0, no phase shift occurs.

The 0° and matched 90° phase shift Hilbert transform is useful in telecommunications applications, where the quadrature outputs are used to shift the spectrum of a signal or in radio and modem modulation schemes.

6.8.8 Raised Cosine Windowed Filters

By applying the raised cosine window to the summation of bin values, the REAL or IMAG filter response ripple is improved.

The method implemented uses a series of coefficients that are applied to each frequency bin and then added much like an FIR filter, except in the frequency domain.

The coefficient values result from both:

- The convolution of the response of a raised cosine function with the signal response
- The multiplication of a rectangular bandpass filter, also applied in the frequency domain

A group delay, or time shift, is also seen which is equal to N/2 plus the time it takes a signal to make it through the ADC/DAC conversion process.

In Figure 6–13 through Figure 6–16, the number of bins required is actually WIDTH + 2 for a given pass-band bandwidth and the signs of the coefficients alternate (+, –, +, –). The endpoints, which are also scaled by 50%, are the result of the window coefficients and define the edge characteristics of the filter.

Figure 6–13. Raised Cosine Window Function (Length = 1 Bin)

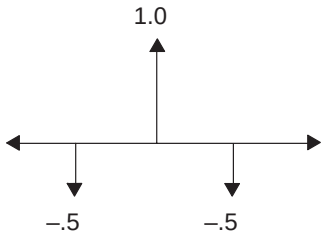


Figure 6–14. Raised Cosine Window Function (Length = 2 Bins)

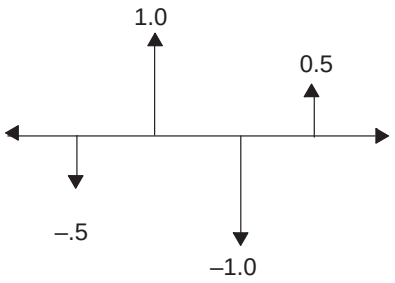


Figure 6–15. Raised Cosine Window Function (Length = 3 Bins)

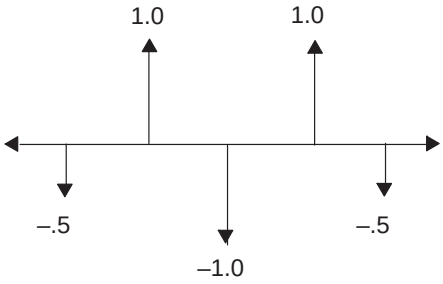
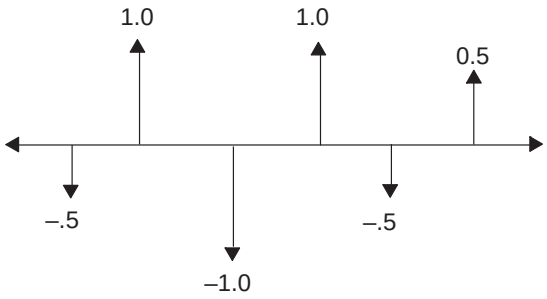


Figure 6–16. Raised Cosine Window Function (Length = 4 Bins)



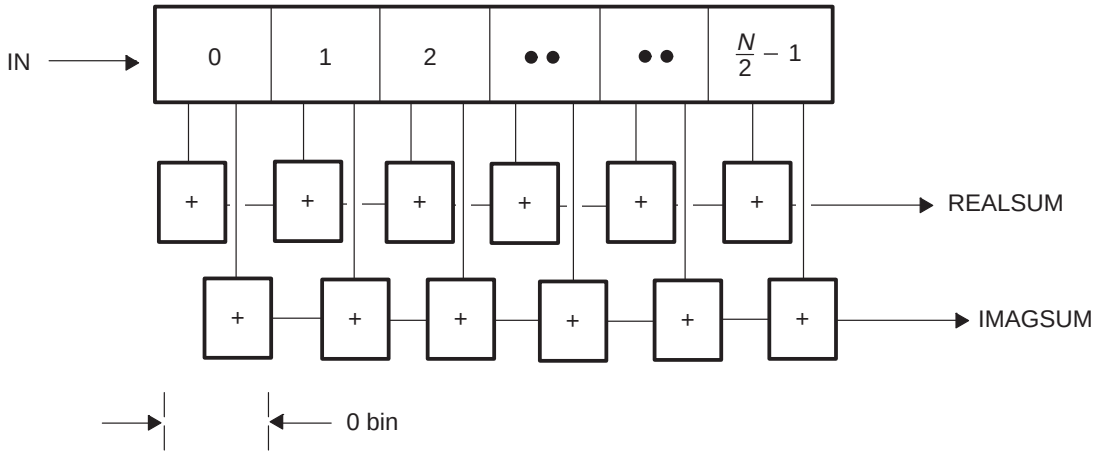
6.8.9 Non-Windowed SFFT

A special case occurs when the SFFT is used to compute the all pass 0' and 90' Hilbert transforms of a non-windowed synchronized signal. Frequency bin spreading occurs if the signal is not harmonically related to the sample window.

For REAL summations, the input is reconstructed by scaling the 0 or DC bin by 50%. This scaling compensates for a 2:1 rise in signal level since all bin data energy, except for the 0 bin, is split equally between the positive and negative frequencies.

At the 0 bin, there is no IMAG information, since no phase shift is applied to that bin. A DC component for an IMAG reconstruction, therefore, does not exist.

Figure 6–17. N/2 SFFT R/I Bins



6.8.10 Performance

Since the SFFT needs only to compute the bins of interest within the span of one time sample, narrow band analysis or filtering is very efficient, even when the effective FFT size is very large. If large numbers of bins and/or high sampling rates are impractical for a single processor, a traditional block style FFT or filter may be more practical.

For example, in a filter application, only a few frequency bins may be required; the unused bins are zero since they are not needed for reconstruction. The maximum sampling rate (or the number of bins that can be calculated) is shown in the following equation.

Ts(min) = (SFFT_cycles_per_bin * bins + loop_overhead) * nS/cycle Ts(min)
= (7 * N/2 + 52) 40 nS

Note:

The loop overhead value is the time consumed by interrupt routines, data formatting, input, and output. SFFT.ASM is not highly optimized, since it is for educational purposes.

The loop can be optimized by inlining the three major functions—Input, SFFT, and Output— to remove 3 calls and 3 returns (or 24 cycles) from the loop overhead.

6.8.11 Loop Unrolling for High Speed Filtering

The inner loop of the SFFT consumes 5 computational cycles, but executes in 6 cycles. The conflict occurs from a data bus bandwidth limitation and results from the STF||STF operation immediately preceding a double load of data for the MPYF3 instruction.

This null cycle is filled by moving the filter summations within the loop. The summation can be done entirely within registers and requires no data path access.

The +1, –1 convolutional filter coefficients for raised cosine windowing can be hard coded within the loop by performing subtractions that invert the sum each time it goes through the loop. This avoids fetching coefficients from the data bus.

Overall, the forward and reverse SFFT are computed at 6–7 cycles per bin, depending on whether both REAL and IMAG outputs are required. The general case educational example SFFT.ASM is slightly slower, while SFFT2.ASM which is written for filtering.

6.8.12 Fitting the Code and Data Into Memory

If the effective desired SFFT/FFT size is 512 points, then only 256 positive frequencies need to be computed. With R/I twiddle and R/I SFFT data associated with each bin, 1024 words of memory are required. In addition, 512 words of input buffer data are needed.

To maximize speed, the inner loop of the SFFT uses dual access on-chip memory to access data at the rate of two data moves per CPU cycle. To avoid program fetch conflicts, the SFFT code is loaded into the second on-chip SRAM block, which also holds the data buffer.

If off-chip memory is available, excellent performance is achieved by placing as much SFFT bin data on-chip as possible. The input window sample buffer and code can be external since the main code loop easily fits inside the cache and the sample buffer is only accessed twice per SFFT cycle.

Note:

The SFFT only needs to calculate the difference of the input of the most recent and the oldest data sample one time. This value is reused for all bin calculations and is kept in a register.

If circular or bit-reversed data storage is used, the data and twiddle buffers are forced to 2^N word boundaries. In addition, the circular addressing registers are consumed. Since the overhead of checking and reloading the buffer pointers is minimal and allows non-2^N sizes, explicit pointer testing is used in SFFT.ASM.

6.8.13 Using This Code With 'C'

To use the functions in this code with a high level language such as C, you must perform context save and restore operations at the beginning and end of each function.

6.8.14 TLC32040 ADC and DAC Considerations

The application file SFFT.ASM is written to use a TLC32040 analog interface chip (AIC) connected as used in a TMS320C31 DSP Starter Kit or DSK (TMDS3200031). Further documentation for the DSK is available in the DSK or by downloading from the Texas Instruments FTP site.

Files	Location
Main TMS320 FTP mirror site	ftp://ftp.ti.com/mirrors/tms320bbs
C3x DSK files subdirectory	ftp://ftp.ti.com/mirrors/tms320bbs/c3xdskfiles

6.8.15 SFFT Summary

- A time signal is comprised of a series of samples.
- Each sample is an impulse.
- The time signal is a time summation of a series of impulses.
- The frequency spectra of a single impulse at T = 0 is trivial to calculate, since it is only a REAL component in each frequency bin whose magnitude is that of the impulse.
- The frequency spectra of a signal is the summation of the individual impulse responses.

- A shift in time is a shift in phase (or phase rotate) in the frequency domain.
- Consider each new impulse as occurring at T = 0 and perform the time shift on the past summation of samples as a whole.
- At each bin, the amount of phase rotation or twiddle factor that is applied to each bin is proportional to the frequency of the bin. The phase shift is zero at DC (n = 0) and pi radians at Fnyq (n = N/2).
- After phase rotating each bin, simply add the new sample/impulse value. (Don't forget to start with each bin magnitude as zero.)
- At this point, the Fourier transform is a forever expanding series in both the time and frequency domains.
- The Nth oldest sample is rotated n multiples of 2 $\times$ pi radians, making the Nth oldest sample completely REAL with no IMAG component.
- At N samples of age, phase rotation = N $\times$ (n $\times$ 2 $\times$ pi/N) = n $\times$ 2 $\times$ pi.
- A sliding rectangular window is created by subtracting the T = Nth oldest sample while adding the newest T = 0 sample. At T = N, each frequency bin has rotated N times and is back to 0 radians of phase and can be properly subtracted.

6.8.16 SFFT Algorithm

SFFT.ASM (Example 6–18 on page 6-94) is written for the DSP beginner, but contains features that also make it useful to the experienced DSP programmer. SFFT.ASM implements a continuous time Fourier transform which can be used to construct filters and analyze spectra. It can also be used as a general-purpose DSP teaching platform.

SFFT.ASM uses a technique known as a sliding FFT (SFFT) to efficiently calculate the spectrum of a signal on a sample-by-sample basis. The SFFT is particularly well-suited for applications where signal analysis, filtering, modulation, demodulation, or other forms of signal manipulation in the frequency domain must be performed in real time. The SFFT algorithm is similar to the DFT.

Further reading and other information includes:

- Designer Notebook page 22 'Fast Logrithms on a Floating Point Device'
- APPHELP1.TXT and APPHELP2.TXT included with the DSK software

- Texas Instruments' FTP site:

Files	Location
Main TMS320 FTP mirror site	ftp://ftp.ti.com/mirrors/tms320bbs
C3x DSK files subdirectory	ftp://ftp.ti.com/mirrors/tms320bbs/c3xdskfiles
TMS320C3x code examples	ftp://ftp.ti.com/mirrors/tms320bbs/c3xfiles
TMS320C4x code examples	ftp://ftp.ti.com/mirrors/tms320bbs/c4xfiles

The following section sets the SFFT parameters which determine the SFFT output characteristics. The following rules apply:

- BIN_LEN = BIN_END – BIN–START > 0
- ((SFFTBINS $\times$ 4) + SFFTSIZE) < Free data space
- Sampling period < time to compute all bins

Be careful not to set the sampling rate too high while calculating many bin values. The SFFT must finish calculating all of its bin values within the time span of one sample.

The effective Fourier series size is determined by the size of the time window of samples. Although this does not affect the calculation rate, it does consume internal memory.

Creating a pass band around a particular signal is easy, since the signal can be viewed either in frequency or time by changing the setting of SPECT_EN. With practice, you can you can zoom in on particular segments of frequency by changing the start and stop bins, window size, and sampling rate.

The DAC output signal fidelity is largely determined by the TA register value that is programmed into the AIC. No one value seems to fit all applications. However, the following rules generally apply. If TA is small, the DAC reconstruction filter is clocked at a faster rate. This pushes the upper pass-band limit higher in frequency, resulting in faster slew times. This is desireable for a spectrum analyzer output where fast impulse response to frequency peaks are needed for suitable viewing. For audio applications, a larger TA value is desired, since the overclocking of the DAC reconstruction filter results in significant distortions.

The AIC master clock input is derived from the timer output pin of internal timer 0. If the timer reference is set higher than the TLC32040 maximum clock rate of 10 MHz, additional distortion occurs.

A TLC32040 analog interface circuit is used on the DSK since it responds favorably when used beyond its tested limits. However, predicting performance depends on many factors; experimentation may be required.

AIC setup registers are programmed into the AIC using a data word which is tagged with xxxx11b in the bottom 2 LSBs to signal the AIC to accept a secondary transmit (or register program) word.

The DAC switch cap filter rate high is set by the TA divisor. A low TA value, used to overclock the DAC reconstruction filter, trades signal fidelity for faster impulse response times.

This application was designed and tested using a 50 MHz TMS320C31 DSP Starter Kit (TMDS3200031) which includes a TLC32040 14-bit ADC/DAC.

Example 6–18. SFFT.ASM

```
;=========================================================================
; SFFT2.ASM
; Keith Larson
; TMS320 DSP Applications
; (C) Copyright 1996,1997,1998
; Texas Instruments Incorporated
;
; This is unsupported freeware with no implied warranties or
; liabilities.  See the C3x DSK disclaimer document for details
;=========================================================================
; Default setup
; -------------
;   SPECT_EN =  1
;   Fs       =  20.8 khz  (4.8 uS)
;   Hz/bin   =  40.7 hz
;   Range    =  1.3 Khz – 3.9 Khz
;
; If this file is re–assembled with SPECT_EN set to 0, this will give a
; bandpass filter from 1.3 – 3.9 Khz having 90 degrees phase shift at all
; frequencies.
;=========================================================================
SFFTSIZE  .set     512                ; Sample Window length (FFT size)
BIN_START .set     32                 ; Start computing SFFT at this bin
BIN_END   .set     96                 ; End computing SFFT at this bin
;- - - - - - - - - - - - - - - - - -
ANGLE     .set     90.0               ; Filter reconstruction angle (degrees)
;- - - - - - - - - - - - - - - - - -
SPECT_EN  .set     1                  ; Enable spectrum analyzer output
RATE      .set     2                  ; Write display points RATE times each
;- - - - - - - - - - - - - - - - - -
TIM0_prd  .set     2                  ; AIC reference clock is TIM0
TA        .set     6                  ; DAC setup
TB        .set     25                 ;
RA        .set     10                 ; ADC setup
RB        .set     15                 ;
;=========================================================================
; PARAMETERS BELOW THIS LINE ARE COMPUTED FROM THE INFORMATION
; ABOVE.  THERE IS NO NEED TO MODIFY ANYTHING BELOW THIS POINT
;=========================================================================
BIN_LEN   .set     BIN_END–BIN_START  ; Filter length in bins
SFFTBINS  .set     BIN_LEN+1          ;
N         .set     SFFTSIZE           ; 'N' used as shorthand for SFFTSIZE
TR        .set     0                  ; Real twiddle offset in each cell
TI        .set     1                  ; Imag
DR        .set     0                  ; Real data offset in each cell
DI        .set     1                  ; Imag
RIBINSIZE .set     2                  ; Size of R/I element pair
pi        .set     3.14159265         ; Useful in making apple pie
w         .set     2.0*pi/N           ; angle = F * 2*pi/Fs
OVM       .set     0x80               ; Use overflow mode to saturate results
```

Example 6–18. SFFT.ASM (Continued)

```
;=======================================================================
; If the input parameters won't work, generate a descriptive error
; for the user letting them know what to look for and maybe fix
;=======================================================================
      .if (BIN_LEN < 1)
APP MESSAGE: Calculated BIN_LEN must be >1
      .endif
      .if  ((SFFTBINS*4) + SFFTSIZE) > (0xE40–0x800)
APP MESSAGE: The Fbin and data storage buffers are too big for the DSK
      .endif
;=====================================================================;
; The SFFT twiddles, data, and input buffer arrays are allocated     ;
; to be placed into RAM0 to avoid bus conflicts with program fetching;
;=====================================================================;
          .include "C3XMMRS.ASM"      ;
          .start   "DATA",0x809800    ; Data arrays are placed at start of RAM0
          .sect    "DATA"             ;
TWIDCOEF   ;------------------------ ;
n         .set     BIN_START          ;
          .loop    SFFTBINS           ; R/I phase or twiddle coefficients
          .float   K1*cos(n*w)        ;
          .float   K1*sin(n*w)        ;
n         .sdef    n+1.0              ; next 'n'
          .endloop                    ;
SFFTDATA  ;-------------------------;
          .loop    SFFTBINS           ; R/I frequency bin data
          .float   0,0                ;   Pre–Zeroing bin data removes
          .endloop                    ;   startup glitches
BUF       ;-------------------------;
          .loop    N/2                ; N samples of ADC input delay data
          .float   0,0                ;
          .endloop                    ;
;=====================================================================;
; The application code begins here, beginning with constants that    ;
; are used in various routines.                                       ;
;=====================================================================;
Tbase     .word    TWIDCOEF           ; Location of twiddle coefficients
Bbase     .word    SFFTDATA           ; Location of R/I SFFT Bin data
CircAddr  .word    BUF                ; Current pointer into sample data
BUFSTART  .word    BUF                ; Start address of sample data
BUFEND    .word    BUF+N              ; End address of sample data
OutBin    .float   0                  ; Current spectrum analyzer bin
MAX       .float   32000.0            ; Used synch pulse and scaling
          ;- - - - - - - - - - - - - -
A_REG     .word    (TA<<9)+(RA<<2)+0  ; Packed AIC register values
B_REG     .word    (TB<<9)+(RB<<2)+2  ;
C_REG     .word    00000011b          ;
;0gctrl   .word    0x0E970300         ; Sport setup, noninverted clkx/clkr
S0gctrl   .word    0x0E973300         ; Sport setup,    inverted clkx/clkr
S0xctrl   .word    0x00000111         ;
```

Example 6–18. SFFT.ASM (Continued)

```
S0rctrl   .word    0x00000111         ;
NewMnsOld .word    0                  ;
K1        .set     0.99995            ; Use a value slightly less than 1.0
K2        .float   pow(K1,N)          ; K1^N oldest sample scale factor
FILTEROUT .float   0.0                ; Temp storage for SFFT filter output
Scale     .float   4.0/N              ; SFFT growth scale factor
REAL_VEC  .float  –cos(pi*ANGLE/180.0); filtered REAL scale factor
IMAG_VEC  .float  –sin(pi*ANGLE/180.0); filtered IMAG scale factor
FLOG2SC   .float   pow(2.0,–24.0)     ; Scale factor for log2 calculations
bigval    .word    0x00010000         ; Used in overflow mode saturation
;=====================================================================;
; The main loop consists of waiting for a new ADC sample.            ;
; When an receive interrupt occurs, the new data is loaded into the  ;
; data delay line buffer, followed by the SFFT and output routines.  ;
; Four dummy writes to the external bus have been added in the main  ;
; loop to allow real time benchmarking of the three functions using  ;
; and oscilloscope to monitor the address bus LSB's                  ;
;=====================================================================;
          .start   "CODE",0x809E40    ; Start in last 512 words of RAM0
          .sect    "CODE"             ;  (also includes DSK kernel)
main      ldi      0xE4,IE            ;   Enable XINT/RINT/INT2
          idle                        ;   Wait for Receive Interrupt
          ;- - - - - - - - - - - - - -
          ldi      @S0_rdata,R0       ; The first interrupt occurs shortly
          ldi      0,R0               ; after AIC init is complete, which
          sti      R0,@S0_xdata       ; will not leave enough time for SFFT
          ;- - - - - - - - - - - - - -
loop      idle                        ;   Wait for Receive Interrupt
          sti      R0,@0x80A000       ;<1
          call     Input              ;   Put ADC sample in delay buffer
          sti      R0,@0x80AF03       ;<2
          call     SFFT               ;   Calculate SFFT
          sti      R0,@0x80AF0F       ;<3
          call     Output             ;   Output result
          sti      R0,@0x80AF3F       ;<4
          b        loop               ;   Loop back and do forever
;=====================================================================;
; The ADC data is read and buffered here                              ;
;=====================================================================;
Input     ldi      @S0_rdata,R0       ; get ADC data
          ash      –16,R0             ; Sign extend previous sample in MSB's
          float    R0,R0              ; Convert the ADC data to float
          ldi      @CircAddr,AR0      ; Load present circ buf address
          ldf      *AR0,R7            ; Multiply by 'K2' for bin stability
          mpyf     @K2,R7             ;  (see text)
          stf      R0,*AR0++          ;
          cmpi     @BUFEND,AR0        ; If at end of buffer, point to start
          ldige    @BUFSTART,AR0      ;
          subrf    R0,R7              ; R7 =  X[–N] – X[0]
          sti      AR0,@CircAddr      ; save new 'circular' modified ptr
          stf      R7,@NewMnsOld      ;
          rets                        ;
```

Example 6–18. SFFT.ASM (Continued)

```
;=====================================================================;
; The forward and reverse SFFT are calculated within this one loop   ;
; The loop itself is unrolled to achieve an inner loop cycle count   ;
; of 7 cycles per bin calculation.  The inner loop contains both the ;
; REAL and IMAG filter summations, so if the output is for spectrum  ;
; analysis or only one filter sum is required, one or both summations;
; can be removed giving an inner loop speed of 6 cycles/bin          ;
;=====================================================================;
SFFT      ldi      @Tbase,AR0         ; R/I twiddle ptr
          ldi      @Bbase,AR1         ; R/I SFFT array ptr
          ldi      @Bbase,AR2         ; SFFT output (usualy in place)
          ldi      SFFTBINS–1,RC      ; Number of bins to calculate
          ldi      RIBINSIZE,IR0      ; Size of R/I pair in array
          ldf      @NewMnsOld,R7      ; R7 = (New – K2*Old)
          ;– – – – – – – – – – – – –
          ldf      0,R4               ; Zero the REAL filter sum
          ldf      0,R5               ; Zero the IMAG filter sum
          ;– – – – – – – – – – – – – –
          mpyf3 *+AR0(TR),*+AR1(DR)    ,R0 ; TR*DR <– unroll from main loop
          rptb  EndSFFT                    ;
          ;– – – – – – – – – – – – – –
Loop      mpyf3 *+AR0(TR)  ,*+AR1(DI)  ,R1 ; TR*DI
          mpyf3 *+AR0(TI)  ,*+AR1(DI)  ,R0 ; TI*DI
       || addf3 R7,R0                  ,R3 ; (TR*DR + DELTA)
          mpyf3 *+AR0(TI)  ,*+AR1(DR)  ,R0 ; TI*DR
       || subf3 R0,R3                  ,R3 ; TR*DR – TI*DI + DELTA
          mpyf3 *++AR0(IR0),*++AR1(IR0),R0 ; TR*DR (used in next loop)
       || addf3 R1,R0                  ,R2 ; TR*DI + TI*DR
          stf   R2,*+AR2(DI)               ; Save the new Fbin values
       || stf   R3,*AR2++(IR0)             ;
          ;– – – – – – – – – – – – – –
          subf3    R4,R3,R4       ;REAL sum; sum'=R–sum  alternates sign of
EndSFFT   subf3    R5,R2,R5       ;IMAG sum; raised cosine window coeficients
       ;-------------------------------------------------------------
       ; For raised cosine window filters the endpoint bin values
       ; are scaled to 1/2 relative to the pass bins
       ;-------------------------------------------------------------
          addf     R4,R4              ; Double inner +/–1 sum loop
          addf     R5,R5              ;
          subf     R3,R4              ; Subtract endpoints at 50%
          subf     R2,R5              ;
          ldi      @Bbase,AR1         ; ptr to start of R/I SFFT array
          ldf      *+AR1(DI),R2       ;
       || ldf      *+AR1(DR),R3       ;
          .if      SFFTBINS&1         ; If the loop count was odd, the
          mpyf     –1,R4              ; +,–,+,– sum result is negative
          mpyf     –1,R5              ;
          .endif                      ;
          addf     R3,R4              ;
          addf     R2,R5              ;
```

Example 6–18. SFFT.ASM (Continued)

```
          ;-------------------------------------------------------------
          ; When the SFFT is finished, the REAL/IMAG sums are scaled
          ; accordingly for the desired output phase angle.  A 'growth'
          ; scale factor is also applied since the summation occurs
          ; over N data points.
          ;-------------------------------------------------------------
ExitSFFT  mpyf     @REAL_VEC,R4       ; Rotate to desired output phase
          mpyf     @IMAG_VEC,R5       ;
          addf3    R4,R5,R0           ; Sum the R/I into a REAL output
          mpyf     @Scale,R0          ; inverse of N/2 growth
          stf      R0,@FILTEROUT      ;
          rets                        ;
;=====================================================================;
; The output section is written for both Spectrum analyzer output    ;
; as well as REAL/IMAG filter sum outputs                             ;
;=====================================================================;
Output:   .if      SPECT_EN=0         ; If SPECT_EN=0 (disable) output either
          ldf     @FILTEROUT,R0       ; Output REAL/IMAG bin sum
          .else                       ;
          ;-----------------------------------------------------
          ; The Spectrum analyzer output section is bypassed
          ; if the spectrum analyzer is not enabled
          ;-----------------------------------------------------
          ldf      @OutBin,R0         ; Point to next output bin
          addf     1.0/RATE,R0        ; increment analyzer output pointer
          cmpf     BIN_LEN,R0         ;
          ldfge    0,R0               ;
          stf      R0,@OutBin         ;
          fix      R0,R0              ;
          bzd      Out                ;
          mpyi     RIBINSIZE,R0       ; Fbins are 2 words (R/I) per bin
          ldfz     @MAX,R0            ; If at base Fbin 0 Hz, output a synch
          ldi      @Bbase,AR0         ;
          subi     2,AR0              ; point to output bin–1 to perform
          addi     R0,AR0             ; –.5,1.0,–.5 convolutional window
          ;- - - - - - - - - - - - -
          ldf      *+AR0(DI+0),R0     ; Perform convolutional window filter
       || ldf      *+AR0(DR+0),R2     ; on the R/I pairs for this output
          addf     *+AR0(DI+4),R0     ;
          addf     *+AR0(DR+4),R2     ;
          mpyf     –0.5,R0            ; Scaling coefficient for –1,+1 bins
          mpyf     –0.5,R2            ;
          addf     *+AR0(DI+2),R0     ;
          addf     *+AR0(DR+2),R2     ;
          ;- - - - - - - - - - - - -
```

Example 6–18. SFFT.ASM (Continued)

```
          mpyf     R0,R0           ; Calculate REAL^2 + IMAG^2 magnitude
          mpyf     R2,R2           ;
          addf     R2,R0           ;
          call     FLOG2           ; Convert to log2(), then scale
          mpyf     32,R0           ; and shift for best display
          mpyf     32,R0           ;
          subf     @MAX,R0         ;
          ;- - - - - - - - - - - - -
          .endif                   ;
Out       fix      R0,R0           ; Convert to integer DAC output
          mpyi     @bigval,R0      ; Use Overflow mode ALU saturation
          ash      -16,R0          ;
          andn     3,R0            ; Do not request a 2nd xmit
          sti      R0,@S0_xdata    ; Output DAC value to serial port
          rets                     ;
;=========================================================================;
; FLOG2() Ultra Fast LOG2 function                                        ;
; computes log2(R0) and returns e8/s1/m4 accuracy float value in R0       ;
;=========================================================================;
FLOG2:    cmpf     0.0,R0          ; Exit if value is <= Zero
          ldfle    -1,R0           ; if x<=0 return -1 (error)
          retsle                   ; return if X<=0
          lsh      1,R0            ; Concatenate mantissa to exponent
          pushf    R0              ; Convert 'fast log' to int, then float
          pop      R0              ; Value is accurate but scaled by 2^24
          float    R0,R0           ;
          mpyf     @FLOG2SC,R0     ; Mpy by scale factor
          rets                     ;
;=========================================================================;
; The startup stub is used during initialization only and can be          ;
; overwritten by the stack or data after initialization is complete.      ;
; Note: A DSK or RTOS communications kernel may also use the stack.       ;
; In this case be sure to not put the stack here during debug.            ;
;=========================================================================;
          .entry   ST_STUB         ; Debugger starts here
ST_STUB   ldp      T0_ctrl         ; Use kernel data page and stack
          ldi      @stack,SP       ;
          ldi      0,R0            ; Halt TIM0 & TIM1
          sti      R0,@T0_ctrl     ;
          sti      R0,@T0_count    ; Set counts to 0
          ldi      TIM0_prd,R0     ; Set period
          sti      R0,@T0_prd      ;
          ldi      0x2C1,R0        ; Restart both timers
          sti      R0,@T0_ctrl     ;
          ;- - - - - - - - - - - - -
```

Example 6–18. SFFT.ASM (Continued)

```
          ldi     @S0xctrl,R0     ;
          sti     R0,@S0_xctrl    ; transmit control
          ldi     @S0rctrl,R0     ;
          sti     R0,@S0_rctrl    ; receive control
          ldi     0,R0            ;
          sti     R0,@S0_xdata    ; DXR data value
          ldi     @S0gctrl,R0     ; Setup serial port
          sti     R0,@S0_gctrl    ; global control
;=======================================================================;
; This section of code initializes the AIC                              ;
;=======================================================================;
AIC_INIT  LDI     0x10,IE         ; Enable only XINT interrupt
          andn    0x34,IF         ;
          ldi     0,R0            ;
          sti     R0,@S0_xdata    ;
          RPTS    0x040           ;
          LDI     2,IOF           ; XF0=0 resets AIC
          rpts    0x40            ;
          LDI     6,IOF           ; XF0=1 runs AIC
          ;- - - - - - - - - - - -
          ldi     @C_REG,R0       ; Setup control register
          call    prog_AIC        ;
          ldi     0xfffc  ,R0     ; Program the AIC to be real slow
          call    prog_AIC        ;
          ldi     0xfffc|2,R0     ;
          call    prog_AIC        ;
          ldi     @B_REG,R0       ; Bump up the Fs to final rate
          call    prog_AIC        ; (smaller divisors should be sent last)
          ldi     @A_REG,R0       ;
          call    prog_AIC        ;
          or      OVM,ST          ; Use the overflow mode for fast saturate
          b       main            ; the DRR before going to the main loop
;=======================================================================;
; prog_AIC is used to transmit new timing configurations to the AIC.    ;
; If you single step this routine, the AIC timing will be corrupted     ;
; causing AIC programming to fail.                                      ;
; STEP OVER THIS ROUTINE USING THE F10 FUNCTION STEP                    ;
;=======================================================================;
prog_AIC  ldi     @S0_xdata,R1    ; Use original DXR data during 2 ndy
          sti     R1,@S0_xdata    ;
          idle                    ;
          ldi     @S0_xdata,R1    ; Use original DXR data during 2 ndy
          or      3,R1            ; Request 2 ndy XMIT
          sti     R1,@S0_xdata    ;
          idle                    ;
          sti     R0,@S0_xdata    ; Send register value
          idle                    ;
          andn    3,R1            ;
          sti     R1,@S0_xdata    ; Leave with original safe value in DXR
          ;- - - - - - - - - - - -
          ldi     @S0_rdata,R0    ; Fix receiver underrun by dummy read
          rets                    ;
```

Example 6–18. SFFT.ASM (Continued)

```
;======================================================================;
; By placing the stack at the end of the users runtime code, the       ;
; maximum space is made available for applications.  Essentialy once   ;
; used initialization code or data can be reclaimed after it is used.;
; However, use this configuration for debug purposes                   ;
;======================================================================;
          .start   "STACK",$        ; This is a reminder to put the stack
          .sect    "STACK"          ; stack in a safe place.  $ places
stack     .word    stack            ; section at the current assy address
;======================================================================;
; Install the XINT/RINT ISR branch vectors                             ;
;======================================================================;
       .start "SP0VECTS",0x809FC5; Place ISR returns directly into
        .sect  "SP0VECTS"         ; secondary branch table
          reti                    ; XINT0
          reti                    ; RINT0
```

Chapter 7

Programming the DMA Channel

The direct memory access (DMA) coprocessor is an on-chip peripheral that can read from or write to any location in the memory map without interfering with the CPU operation. The DMA channel contains its own address generators, source and destination registers, and transfer counters. The DMA channel can be easily programmed in C or in assembly language.

The ’C30 and ’C31 coprocessors each have one DMA channel, while the ’C32 coprocessor has two DMA channels. Each channel of the ’C32 DMA channel is similar to those of the ’C30 and ’C31, with the addition of user-configurable priorities.

This chapter provides examples for programming the DMA for the ’C3x.

Topic **Page**

7.1 Hints for DMA Programming

The *Peripherals* chapter of the *TMS320C3x User's Guide* describes the DMA channel and its operation in detail. Use the following techniques to program your DMA more efficiently and to avoid unexpected results:

- Reset the DMA register before starting it. This clears any previously latched interrupts that may no longer exist.
- After starting the DMA, set the IE register to enable interrupts for sync transfer.
- If a conflict occurs when the CPU and DMA access the memory simultaneously on the 'C30 or 'C31, the CPU always prevails. Carefully allocate the sections of the program in memory for faster execution. If a CPU program access conflicts with a DMA access, enabling the cache helps if the program is located in external memory. DMA on-chip access happens during the H3 phase. Refer to the *Pipeline Operation* chapter in the *TMS320C3x User's Guide* for details on CPU accesses.

  If a conflict occurs during CPU-DMA access on the 'C32, the priority set between the CPU and DMA is used to arbitrate conflicts. If the DMA channel has lower priority than the CPU, the DMA may fail to finish a block transfer if conflicts occur. To avoid this condition, use CPU/DMA rotating priority in the corresponding DMA control register.

Note: Expansion and Peripheral Buses

The expansion and peripheral buses on the 'C30 cannot be accessed simultaneously because they are multiplexed into a common port. Therefore, DMA access to the peripheral bus along with CPU access to the expansion bus can cause CPU-DMA conflicts. (See the *TMS320C3x User's Guide* for more information.)

- When you use interrupt synchronization, ensure that interrupts are actually generated; otherwise, the DMA will never complete the block transfer.
- Use read/write synchronization when reading from or writing to serial ports to guarantee data validity.

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. The following are indications that the DMA has finished a set of transfers:

- **The DINT bit in the IIF register is set to 1 (interrupt polling)**. This requires that the TCINT bit in the DMA control register be set first. This interrupt-polling method does not cause any additional conflict during CPU-DMA access.
- **The transfer counter has a zero value**. The transfer counter is decremented after the DMA read operation finishes (not after the write operation). Nevertheless, a transfer counter with a zero value can be used as an indication of a transfer completion.
- **The STAT bits in the DMA channel control register are set to 00_2**. You can poll the DMA channel-control register for this value. However, because the DMA registers are memory-mapped into the peripheral bus address space, this option can cause further conflicts during CPU-DMA access.

7.3 DMA Assembly Programming Examples

Example 7–1, Example 7–2, and Example 7–3 illustrate how to program the DMA channel using assembly language.

When linking the examples, allocate section memory addresses carefully to avoid CPU-DMA conflict. In the ’C30 or ’C31, the CPU always prevails in cases of conflict. If a conflict occurs between a CPU program and DMA data, you can enable the cache if the .text section is in external memory. For example, when linking the code in Example 7–1, Example 7–2, and Example 7–3, allocate the following sections into memory (RAM0 corresponds to on-chip RAM block 0 and RAM1 corresponds to on-chip RAM block 1):

- .text section into RAM0
- .data section into RAM1
- .bss section into RAM1

Example 7–1. Array Initialization With DMA

```
* TITLE: ARRAY INITIALIZATION WITH DMA
*
          .GLOBAL START
          .DATA
DMA       .WORD  808000H          ; DMA GLOBAL CONTROL REG ADDRESS
RESET     .WORD  0C40H            ; DMA GLOBAL CONTROL REG RESET VALUE
CONTROL   .WORD  0C43H            ; DMA GLOBAL CONTROL REG INITIALIZATION
SOURCE    .WORD  ZERO             ; DATA SOURCE ADDRESS
DESTIN    .WORD  _ARRAY           ; DATA DESTINATION ADDRESS
COUNT     .WORD  128              ; NUMBER OF WORDS TO TRANSFER
ZERO      .FLOAT 0.0              ; ARRAY INITIALIZATION VALUE 0.0 = 0X80000000
          .BSS   _ARRAY,128       ; DATA ARRAY LOCATED IN .BSS SECTION
          .TEXT

START  LDP    DMA                ; LOAD DATA PAGE POINTER
       LDI    @DMA,AR0           ; POINT TO DMA GLOBAL CONTROL REGISTER
       LDI    @RESET,R0          ; RESET DMA
       STI    R0,*AR0
       LDI    @SOURCE,R0         ; INITIALIZE DMA SOURCE ADDRESS REGISTER
       STI    R0,*+AR0(4)
       LDI    @DESTIN,R0         ; INITIALIZE DMA DESTINATION ADDRESS REGISTER
       STI    R0,*+AR0(6)
       LDI    @COUNT,R0          ; INITIALIZE DMA TRANSFER COUNTER REGISTER
       STI    R0,*+AR0(8)
       OR     400H,IE            ; ENABLE INTERRUPT FROM DATA TO CPU
       OR     2000H,ST           ; ENABLE CPU INTERRUPTS GLOBALLY
       LDI    @CONTROL,R0        ; INITIALIZE DMA GLOBAL CONTROL REGISTER
       BU     $
       .END
```

In Example 7–1, the DMA initializes a 128-element array to 0. The DMA sends an interrupt to the CPU after the transfer is completed. This program assumes previous initialization of the CPU interrupt vector table (specifically the DMA-to-CPU interrupt). The ST and IE registers are initialized for interrupt processing.

In Example 7–2, the serial port 0 is initialized to receive 32-bit data words with an internally generated receive-bit clock and a bit-transfer rate of 8H1 cycles/bit.

This program assumes previous initialization of the CPU interrupt vector table (specifically the DMA-to-CPU interrupt). The serial-port interrupt directly affects only the DMA; therefore, no CPU serial-port interrupt vector setting is required.

Example 7–2. DMA Transfer With Serial-Port Receive Interrupt

```
* TITLE DMA TRANSFER WITH SERIAL PORT RECEIVE INTERRUPT
*
          .GLOBAL START
          .DATA
DMA       .WORD  808000H          ; DMA GLOBAL CONTROL REG ADDRESS
CONTROL   .WORD  0D43H            ; DMA GLOBAL CONTROL REG INITIALIZATION
SOURCE    .WORD  80804CH          ; DATA SOURCE ADDRESS: SERIAL PORT INPUT REG
DESTIN    .WORD  _ARRAY           ; DATA DESTINATION ADDRESS
COUNT     .WORD  128              ; NUMBER OF WORDS TO TRANSFER
IEVAL     .WORD  002000400H       ; IE REGISTER VALUE
RESET1    .WORD  0D40H            ; DMA RESET

          .BSS   _ARRAY,128       ; DATA ARRAY LOCATED IN .BSS SECTION
                                  ; THE UNDERSCORE USED IS JUST TO MAKE IT
                                  ; ACCESSIBLE FROM C (OPTIONAL)

START     LDP    DMA              ; LOAD DATA PAGE POINTER
* DMA INITIALIZATION

      LDI    @DMA,AR0            ; POINT TO DMA GLOBAL CONTROL REGISTER
      LDI    @SPORT,AR1
      LDI    @RESET,R0
      STI    R0,*+AR1(4)         ; RESET SPORT TIMER
      LDI    @RESET1,R0
      STI    R0,*AR0             ; RESET DMA
      LDI    @SPRESET,R0
      STI    R0,*AR1             ; RESET SPORT
      LDI    @SOURCE,R0          ; INITIALIZE DMA SOURCE ADDRESS REGISTER
      STI    R0,*+AR0(4)
      LDI    @DESTIN,R0          ; INITIALIZE DMA DESTINATION ADDRESS REGISTER
      STI    R0,*+AR0(6)
      LDI    @COUNT,R0           ; INITIALIZE DMA TRANSFER COUNTER REGISTER
      STI    R0,*+AR0(8)
      OR     @IEVAL,IE           ; ENABLE INTERRUPTS
      OR     2000H,ST            ; ENABLE CPU INTERRUPTS GLOBALLY
      LDI    @CONTROL,R0         ; INITIALIZE DMA GLOBAL CONTROL REGISTER
      STI    R0,*AR0             ; START DMA TRANSFER

* SERIAL PORT INITIALIZATION

      LDI    @SRCTRL,R0       ; SERIAL-PORT RECEIVE CONTROL REG INITIALIZATION
      STI    R0,*+AR1(3)
      LDI    @STPERIOD,R0     ; SERIAL-PORT TIMER PERIOD INITIALIZATION
      STI    R0,*+AR1(6)
      LDI    @STCTRL,R0       ; SERIAL-PORT TIMER CONTROL REG INITIALIZATION
      STI    R0,*+AR1(4)
      LDI    @SGCCTRL,R0      ; SERIAL-PORT GLOBAL CONTROL REG INITIALIZATION
      STI    R0,*AR1
      BU     $
      END
```

Example 7–3 sets up the DMA to transfer data (128 words) from an array buffer to the serial-port-0 output register with serial-port transmit interrupt XINT0. The DMA sends an interrupt to the CPU when the data transfer completes.

Serial port 0 is initialized to transmit 32-bit data words with an internally generated frame sync and a bit-transfer rate of 8H1 cycles/bit. The receive-bit clock is internally generated and equal in frequency to one half of the 'C3x H1 frequency.

This program assumes previous initialization of the CPU interrupt vector table (specifically the DMA-to-CPU interrupt). The serial-port interrupt directly affects only the DMA; therefore, no CPU serial-port interrupt vector setting is required.

Note: Serial Port Transmit Synchronization

The DMA uses serial port transmit interrupt XINT0 to synchronize transfers. Because the XINT0 is generated when the transmit buffer has written the last bit of data to the shifter, an initial CPU write to the serial port is required to trigger XINT0 to enable the first DMA transfer.

Example 7–3. DMA Transfer With Serial-Port Transmit Interrupt

```
* TITLE: DMA TRANSFER WITH SERIAL PORT TRANSMIT INTERRUPT
*         .GLOBAL START
          .DATA
DMA       .WORD  808000H       ; DMA GLOBAL CONTROL REG ADDRESS
CONTROL   .WORD  0E13H         ; DMA GLOBAL CONTROL REG INITIALIZATION
SOURCE    .WORD  (_ARRAY+1)    ; DATA SOURCE ADDRESS
DESTIN    .WORD  80804CH       ; DATA DESTIN ADDRESS: SERIAL-PORT OUTPUT REG
COUNT     .WORD  127           ; NUMBER OF WORDS TO TRANSFER =(MSG LENGHT–1)
IEVAL     .WORD  00100400H     ; IE REGISTER VALUE
          .BSS   _ARRAY,128    ; DATA ARRAY LOCATED IN .BSS SECTION
                               ; THE UNDERSCORE USED IS JUST TO MAKE IT
                               ; ACCESSIBLE FROM C (OPTIONAL)
RESET1    .WORD  0E10H         ; DMA RESET
SPORT     .WORD  808040H       ; SERIAL-PORT GLOBAL CONTROL REG ADDRESS
SGCCTRL   .WORD  04880044H     ; SERIAL-PORT GLOBAL CONTROL REG INITIALIZATION
SXCTRL    .WORD  111H          ; SERIAL-PORT TX PORT CONTROL REG INITIALIZA-
TION
STCTRL    .WORD  00FH          ; SERIAL-PORT TIMER CONTROL REG INITIALIZATION
STPERIOD  .WORD  00000002H     ; SERIAL-PORT TIMER PERIOD
SPRESET   .WORD  00880044H     ; SERIAL-PORT RESET
RESET     .WORD  0H            ; SERIAL-PORT TIMER RESET
          .TEXT
START     LDP    DMA           ; LOAD DATA PAGE POINTER
```

Example 7–3. DMA Transfer With Serial-Port Transmit Interrupt (Continued)

```
* DMA INITIALIZATION

       LDI    @DMA,AR0         ; POINT TO DMA GLOBAL CONTROL REGISTER
       LDI    @SPORT,AR1
       LDI    @RESET,R0
       STI    R0,*+AR1(4)      ; RESET SPORT TIMER
       STI    R0,*AR0          ; RESET DMA
       STI    R0,*AR1          ; RESET SPORT
       LDI    @SOURCE,R0       ; INITIALIZE DMA SOURCE ADDRESS REGISTER
       STI    R0,*+AR0(4)
       LDI    @DESTIN,R0       ; INITIALIZE DMA DESTINATION ADDRESS REGISTER
       STI    R0,*+AR0(6)
       LDI    @COUNT,R0        ; INITIALIZE DMA TRANSFER COUNTER REGISTER
       STI    R0,*+AR0(8)
       OR     @IEVAL,IE        ; ENABLE INTERRUPT FROM DMA TO CPU
       OR     2000H,ST         ; ENABLE CPU INTERRUPTS GLOBALLY
       LDI    @CONTROL,R0      ; INITIALIZE DMA GLOBAL CONTROL REGISTER
       STI    R0,*AR0          ; START DMA TRANSFER

* SERIAL PORT INITIALIZATION

       LDI    @SXCTRL,R0       ; SERIAL-PORT TX CONTROL REG INITIALIZATION
       STI    R0,*+AR1(2)
       LDI    @STPERIOD,R0     ; SERIAL–PORT TIMER PERIOD INITIALIZATION
       STI    R0,*+AR1(6)
       LDI    @STCTRL,R0       ; SERIAL-PORT TIMER CONTROL REG INITIALIZATION
       STI    R0,*+AR1(4)
       LDI    @SGCCTRL,R0      ; SERIAL-PORT GLOBAL CONTROL REG INITIALIZATION
       STI    R0,*AR1

* CPU WRITES THE FIRST WORD (TRIGGERING EVENT ---> XINT IS GENERATED)

       LDI    @SOURCE,AR0
       LDI    *–AR0(1),R0
       STI    R0,*+AR1(8)
       BU     $
       .END
```

Other examples of DMA initialization include:

- Transfer a 256-word block of data from off-chip memory to on-chip memory and generate an interrupt on completion. Maintain the memory order.

DMA source address	800000h
DMA destination address	809800h
DMA transfer counter	00000100h
DMA global control	00000C53h
CPU/DMA interrupt enable (IE)	00000400h

- Transfer a 128-word block of data from on-chip memory to off-chip memory and generate an interrupt on completion. Invert the order of memory—the highest addressed member of the block becomes the lowest addressed member.

DMA source address	809800h
DMA destination address	800000h
DMA transfer counter	00000080h
DMA global control	00000C93h
CPU/DMA interrupt enable (IE)	00000400h

- Transfer a 200-word block of data from the serial port 0 receive register to on-chip memory and generate an interrupt on completion. Synchronize the transfer with the serial-port-0 receive interrupt.

DMA source address	80804Ch
DMA destination address	809C00h
DMA transfer counter	000000C8h
DMA global control	00000D43h
CPU/DMA interrupt enable (IE)	00200400h

- Transfer a 200-word block of data from off-chip memory to the serial port 0 transmit register and generate an interrupt on completion. Synchronize the transfer with the serial-port-0 transmit interrupt.

DMA source address	809C00h
DMA destination address	808048h
DMA transfer counter	000000C8h
DMA global control	00000E13h
CPU/DMA interrupt enable (IE)	00400400h

- Transfer data continuously between the serial port 0 receive register and the serial-port-0 transmit register to create a digital loop back. Synchronize the transfer with the serial-port-0 receive and transmit interrupts.

DMA source address	80804Ch
DMA destination address	808048h
DMA transfer counter	00000000h
DMA global control	00000303h
CPU/DMA interrupt enable (IE)	00300000h

Chapter 8

Analog Interface Peripherals and Applications

Analog interface peripherals are analog input/output devices that interface directly to the ’C3x. This chapter describes these devices and their applications in ’C3x-based systems.

Topic **Page**

8.1 Analog-to-Digital Converter Interface to the TMS320C30 Expansion Bus

Analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) are commonly required in DSP systems and interface efficiently to the I/O expansion bus. These devices are available in many speed ranges and with a variety of features. While some might require one or more wait states on the I/O bus, others can be used at full speed. Figure 8–1 illustrates a 'C30 interface to an Analog Device's AD1678 ADC. The AD1678 is a 12-bit, 5-µs converter that allows sample rates up to 200 kHz and has an input voltage range of 10 V, bipolar or unipolar. The converter is connected according to manufacturer's specifications to provide 0–10-V operation. This interface illustrates a common approach to connecting such devices to the 'C30. Note that the interface requires only a minimum amount of control logic.

The AD1678 is a very flexible converter and is configurable in a number of different operating modes. These operating modes include:

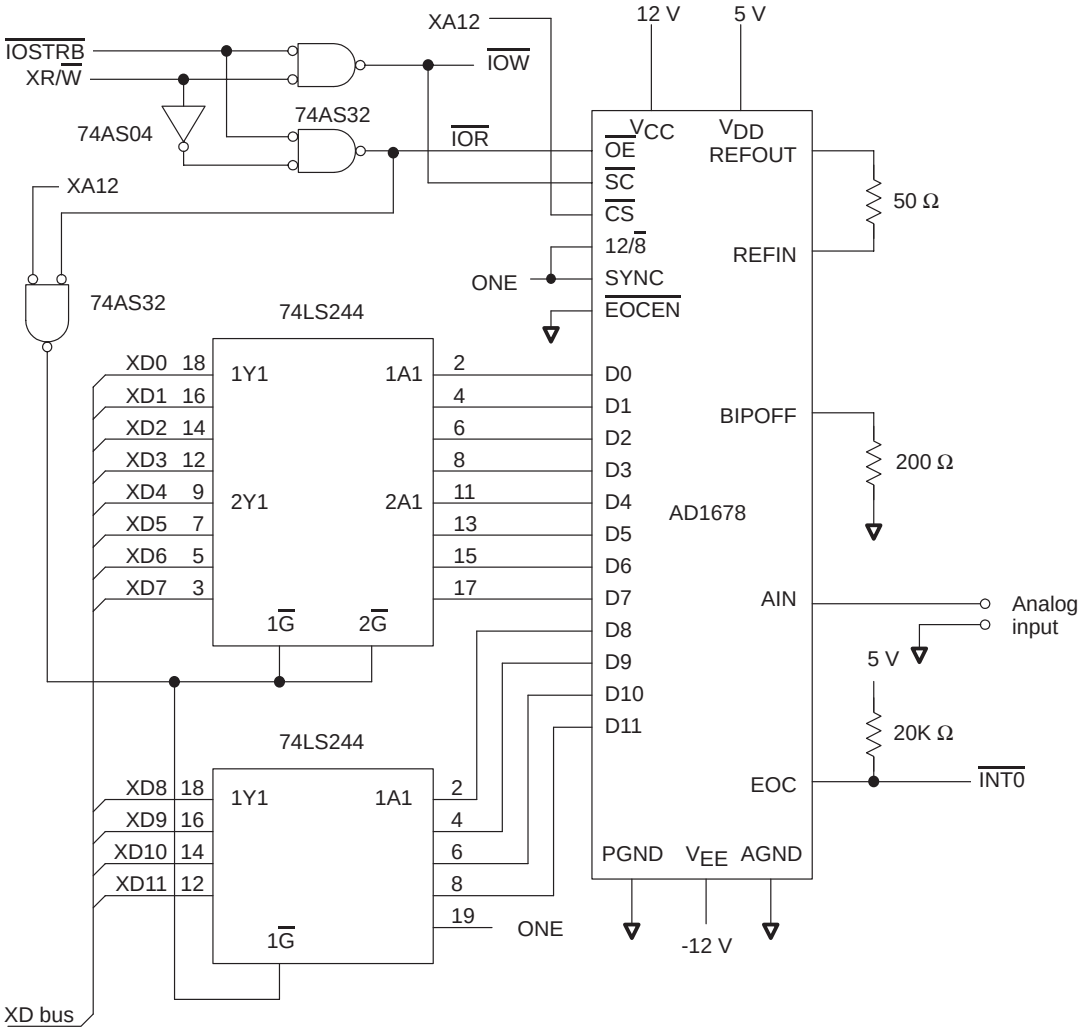
- Byte or word data format
- Continuous or noncontinuous conversions
- Enabled or disabled chip-select function
- Programmable end-of-conversion indication

This interface uses a data format of 12-bit words, rather than a byte format, to be compatible with the 'C3x. Noncontinuous conversions are selected so that variable sample rates can be used; continuous conversions occur at a fixed rate of 200 kHz. With noncontinuous conversions, the host processor determines the conversion rate by initiating conversions through write operations to the converter.

The chip-select input must be active when accessing the device. Enabling the chip-select function is necessary to isolate the AD1678 from other peripheral devices connected to the expansion bus. To establish the desired operating modes, the SYNC and 12/$\overline{8}$ inputs to the converter are pulled high and $\overline{\text{EOCEN}}$ is grounded, as specified in the *AD1678 Data Sheet*.

In this application, the converter's chip-select is driven by XA12, which maps this device at 804000h in I/O address space. Conversions are initiated by writing any data value to the device. The conversion results are obtained by reading from the device after the conversion is complete. To generate the device's start conversion ($\overline{\text{SC}}$) and output enable ($\overline{\text{OE}}$) inputs, the 74AS32 performs an AND operation on $\overline{\text{IOSTRB}}$ and R/$\overline{\text{W}}$ (see Figure 8–1). Therefore, the converter is selected whenever XA12 is low; $\overline{\text{OE}}$ is driven when reads are performed, and $\overline{\text{SC}}$ is driven when writes are performed.

Figure 8–1. Interface Between the TMS320C30 and the AD1678



As with many A/D converters, the AD1678 data output lines enter a high-impedance state at the end of a read cycle. This occurs after the output enable ($\overline{OE}$) or read control line goes inactive. Furthermore, the data output buffer often requires a substantial amount of time to actually attain a full high-impedance state. When used with the ’C30-33, device output must be fully disabled no later than 65 ns following the rising edge of $\overline{IOSTRB}$. This is because the ’C30 begins driving the data bus at this point if the next cycle is a write. If this timing is not met, bus conflicts between the ’C30 and the AD1678 can occur. This degrades system performance and may cause failure due to damaged data bus drivers. The actual disable time for the AD1678 can be as long as 80 ns; therefore, 74LS244 buffers are used to isolate the converter outputs

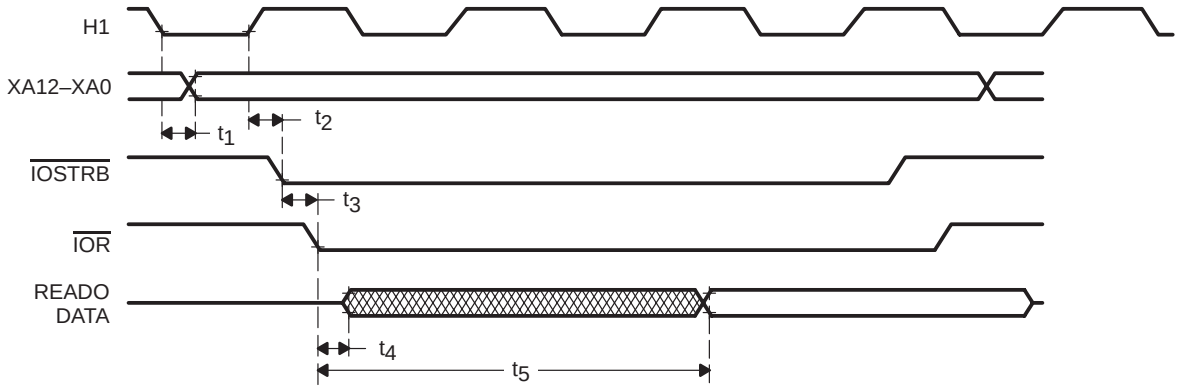
from the ’C30. The buffers are enabled when the AD1678 is read and are turned off 30.8 ns after $\overline{\text{IOSTRB}}$ goes high, meeting the ’C30-33 requirement of 65 ns.

When data is read following a conversion, the AD1678 takes 100 ns after its $\overline{\text{OE}}$ control line is asserted to provide valid data at its outputs. Thus, including the propagation delay of the 74LS244 buffers, the total access time for reading the converter is 118 ns. This requires two wait states on the ’C30-33 expansion I/O bus.

The two wait states required in this case are implemented using software wait states. However, depending on the overall system configuration, you can implement a separate wait-state generator for the expansion bus (for example, in a case where multiple devices that require different numbers of wait states are connected to the expansion bus). See section 4.5 *Wait States and Ready Generation* on page 4-10.

Figure 8–2 shows the timing for read operations between the ’C30-33 and the AD1678. At the beginning of the cycle, the address and XR/$\overline{\text{W}}$ lines become valid at 10 ns (t_1) following the falling edge of H_1. Then, after 10 ns (t_2) from the next rising edge of H_1, $\overline{\text{IOSTRB}}$ goes low. This begins the active portion of the read cycle. After the control logic propagation delay at 5.8 ns (t_3), the $\overline{\text{IOR}}$ signal goes low, asserting the $\overline{\text{OE}}$ input to the AD1678. The 74LS244 buffers take 30 ns (t_4) to enable their outputs. Then, after the converter access delay and the buffer propagation delay at 118 ns (t_5 which equals 100 + 18), data is provided to the ’C30. This provides approximately 46 ns of data setup time before the rising edge of $\overline{\text{IOSTRB}}$. Therefore, this design easily satisfies the ’C30-33’s requirement of 15 ns of data setup time for reads.

Figure 8–2. Read Operations Timing Between the TMS320C30 and the AD1678



Unlike the primary bus, read and write cycles on the I/O expansion bus are timed the same but have the following exceptions:

- XR/$\overline{W}$ is high for reads and low for writes
- The data bus is driven by the ’C30 during writes (reads are the same)

When writing to the AD1678, the 74LS244 buffers do not turn on and no data is transferred. The purpose of writing to the converter is only to generate a pulse on the converter’s $\overline{SC}$ input, which initiates a conversion cycle. When a conversion cycle is completed, the AD1678’s end of conversion (EOC) output generates an interrupt on the ’C30 to indicate that the converted data can be read.

The TLC1225 is a self-calibrating 12-bit-plus-sign bipolar or unipolar converter, which features 10-µs conversion times. The TLC1550 is a 10-bit, 6-µs converter with a high-speed DSP interface. Both converters are parallel-interface devices.

8.2 Digital-to-Analog Converter Interface to the TMS320C30 Expansion Bus

In many DSP systems, the requirement for generating an analog output signal is a consequence of sampling an analog waveform with an ADC so that it can be processed digitally. This digitally processed signal is then reproduced with a digital-to-analog converter (DAC). Interfacing the DAC to the ’C30 on the expansion I/O bus is also straightforward.

Various types of DACs may be distinguished by whether or not the converters include:

- Latches to store the digital value to be converted to an analog quantity
- The interface to control those latches

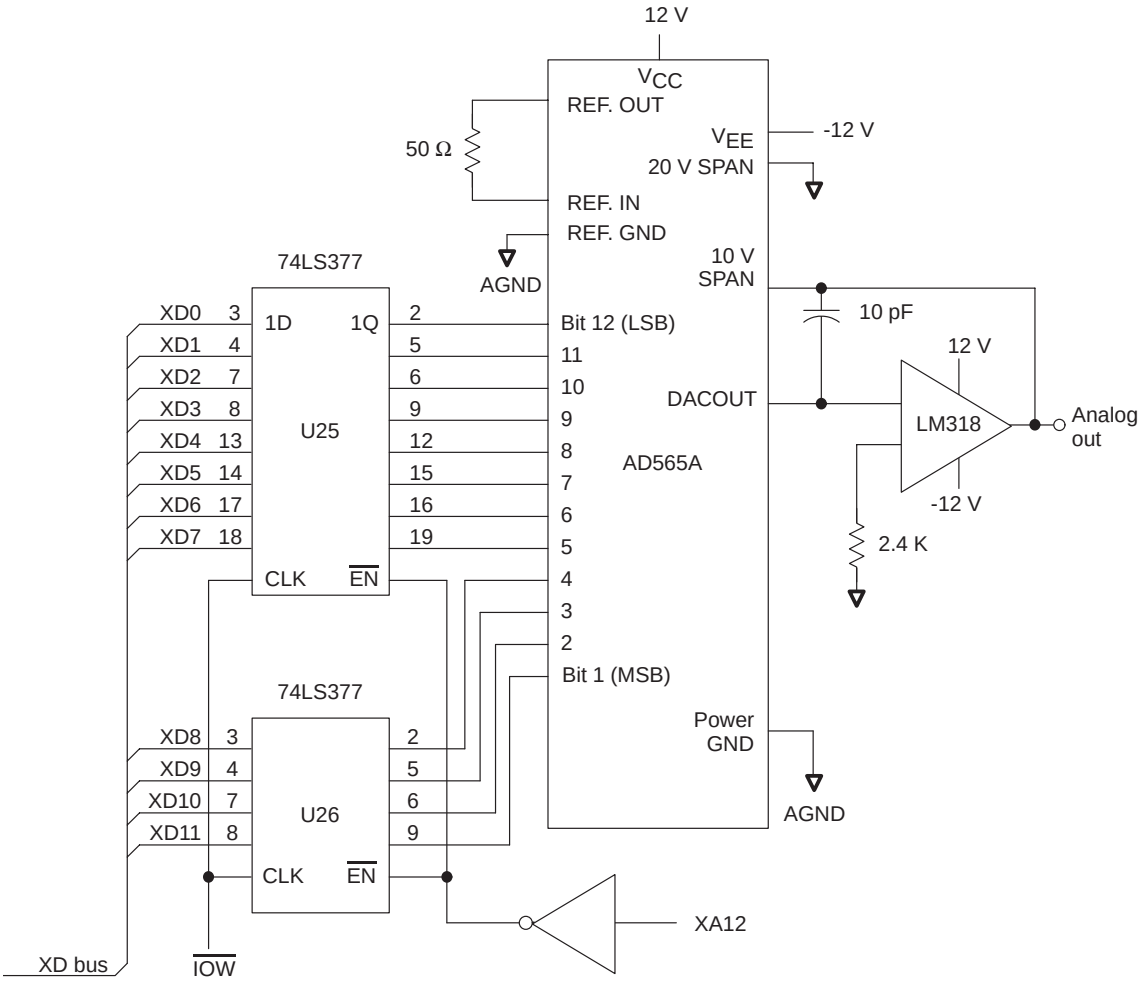
When latches and control logic are included, interface design is often simplified; however, internal latches are often included only in slower DACs.

Although slower converters limit signal bandwidth, the converter design described in Figure 8–3 allows a reasonably wide range of signal frequencies to be processed and illustrates the technique of interfacing to a converter that uses external data latches.

Figure 8–3 shows an interface to an Analog Device, AD565A DAC. This device is a 12-bit, 250-ns current output DAC with an on-chip 10-V reference. Using an off-chip current-to-voltage conversion circuit connected according to the manufacturer’s specifications, the converter exhibits output signal ranges of 0–10 V, which is compatible with the conversion range of the ADC discussed in the previous section.

Because this DAC essentially performs continuous conversions based on the digital value provided at its inputs, periodic sampling is maintained by updating the value stored in the external latches at regular intervals. Therefore, between updates, the digital value is stored and maintained at the latch outputs that provide the input to the DAC. This results in a stable analog output until the next sample update is performed.

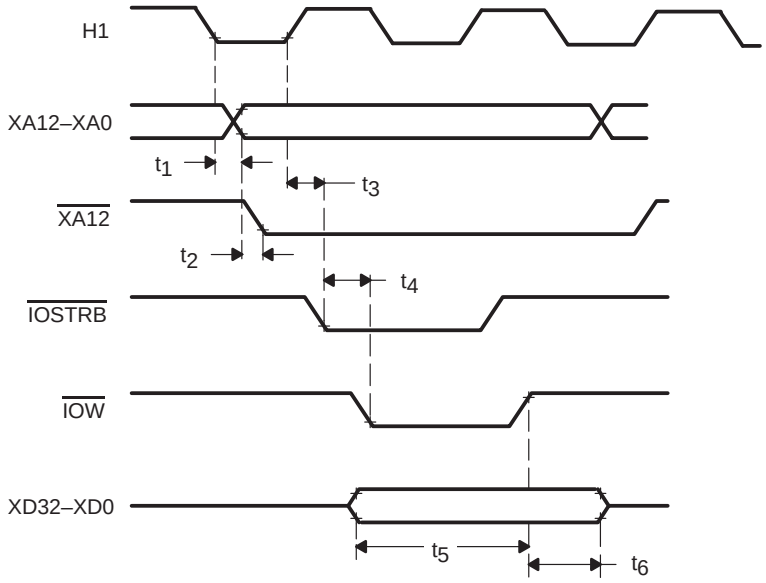
Figure 8–3. Interface Between the TMS320C30 and the AD565A



The external data latches are 74LS377 devices that have both clock and enable inputs. These latches serve as a convenient interface with the ’C30; the enable inputs provide a device select function and the clock inputs latch the data. The enable input driven by inverted XA12 and the clock input driven by $\overline{\text{IOW}}$ (which is the AND of $\overline{\text{IOSTRB}}$ and XR/$\overline{\text{W}}$). Therefore, data is stored in the latches when a write is performed to I/O address 805000h. Reading this address has no effect on the circuit.

Figure 8–4 shows the timing diagram of a write operation to the DAC latches.

Figure 8–4. Timing Diagram for Write Operation to the DAC



Because the data is written to the latches, rather than to the DAC, the timing requirements for these devices are fundamental to the operation of the interface. At a minimum, these latches require:

- Data setup time of 20 ns
- Enable setup time of 25 ns
- Disable setup time of 10 ns
- Data and enable hold times of 5 ns

This design provides approximately 60 ns of enable setup, 30 ns of data setup, and 7.2 ns of data hold time. Therefore, the setup and hold times provided by this design exceed those required by the latches. The key timing parameters for this interface are summarized in Table 8–1.

Table 8–1. Key Timing Parameters for DAC Write Operation

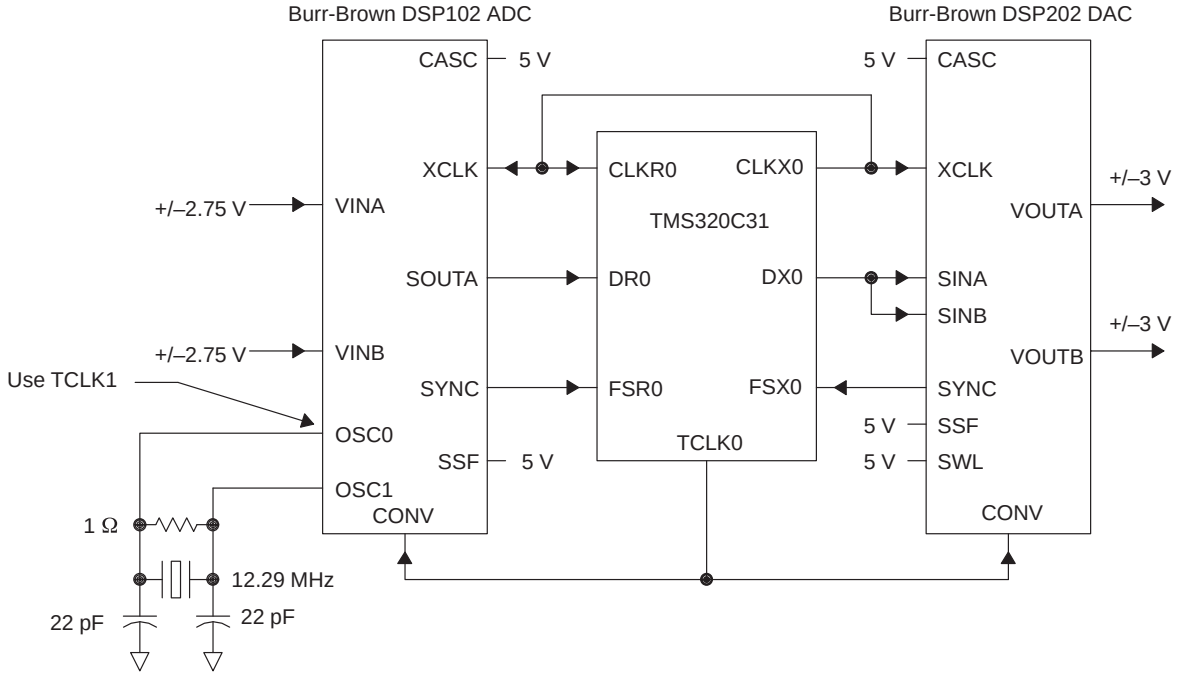
Time Interval	Event	Time Period†
t_1	H1 falling to address valid	10 ns
t_2	XA12 to $\overline{\text{XA12}}$ delay	5 ns
t_3	H1 rising to $\overline{\text{IOSTRB}}$ falling	10 ns
t_4	$\overline{\text{IOSTRB}}$ to $\overline{\text{IOW}}$ delay	5.8 ns
t_5	Data setup to $\overline{\text{IOW}}$	30 ns
t_6	Data hold from $\overline{\text{IOW}}$	7.2 ns

† Timing for the ’C30-33

8.3 Burr-Brown DSP101/2 and DSP201/2 Interface to TMS320C3x

Figure 8–5 shows how to interface the ’C3x with zero glue logic to Burr-Brown’s DSP201/2 and DSP101/2 family of 16-bit DAC and ADC. Using a ’C3x and the DSP202 and DSP102 dual-channel DAC and ADC chips provides an efficient, low-cost, stereo, digital audio interface.

Figure 8–5. TMS320C31 Zero Glue-Logic Interface to Burr-Brown ADC and DAC



The DSP102 ADC is interfaced to the ’C3x serial port receive side; the DSP202 DAC is interfaced to the transmit side. The ADC and DAC are hard-wired to run in cascade mode. In this mode, when the ’C3x initiates a convert command (CONV) to the ADC through its TCLK0 pin, both analog inputs are converted into two 16-bit words that are concatenated to form one 32-bit word. The ADC signals the ’C3x that serial data from the last conversion is being transmitted through the ADC’s SYNC signal. The 32-bit word is then serially transmitted, most significant bit (MSB) first, through the SOUTA serial pin of the DSP102 to the DR0 pin of the ’C3x serial port. The ’C3x is programmed to drive the analog interface bit clock from its CLKX0 pin. The bit clock drives both the ADC and DAC XCLK input.

The ’C3x transmit clock can also act as the input clock on the receive side of the ’C3x serial port. Since the receive clock is synchronous to the ’C3x’s internal clock, the receive clock can run at full speed (even though it is an external clock).

Similarly, upon receiving a convert command (CONV), the DAC converts the last word received from the ’C3x. It signals the ’C3x, through the SYNC signal, to begin transmitting a 32-bit word representing the two channels of data to be converted. The data, transmitted from the ’C3x DX0 pin, is input to both the SINA and SINB inputs of the DAC.

The ’C3x is set up to transfer bits at the maximum rate of about 8 Mbytes/s. It uses a dual-channel sample rate of about 44.1 KHz by setting the following registers (assuming a 32 MHz CLKIN):

Serial Port:

Port global control register	0x0EBC0040
FSX/DX/CLKX port control register	0x00000111
FSR/DR/CLKR port control register	0x00000111
Receive/transmit timer control register	0x0000000F

Timer:

Timer global control register	0x000002C1
Timer period register	0x000000B5

A synchronous receive interrupt service routine is sufficient for parsing and transferring data between the serial ports and memory. Source code for setting up the serial port and timers of the ’C3x for interfacing to the DSP102 and DSP202 can be found on the TI BBS (file name: C3XBB.EXE). This code is listed in Example 8–1 through Example 8–4.

Example 8–1. TMS320C3x / BB – DSP102/202 Driver Header File

```
/*************************************************************************/
/*   BB.H                                                                */
/*                                                                       */
/*   TMS320C3x – BB DSP102/202 DRIVER HEADER FILE                        */
/*************************************************************************/
#include <serprt30.h>
#include <timer30.h>
#include <dma30.h>
#include <bus30.h>
#include <general.h>
/*************************************************************************/
/* COMMON STRUCTURES                                                     */
/*************************************************************************/
typedef volatile int VI;
typedef volatile float VF;
typedef VF * volatile VPVF;
typedef VI * volatile VPVI;
/*************************************************************************/
/* FUNCTION PROTOTYPES                                                   */
/*************************************************************************/
void c_int99(void);
void heap_overflow(void);
void init_c30(void);
void error_in_real_time(void);
/*************************************************************************/
/* MACROS                                                                */
/*************************************************************************/
#define      BLOCK_SIZE    64           /* BUFFER SIZE */
#define      GEN_OSC       OFF          /* GENERATE OSCILLATOR */
#define      GEN_CC        ON           /* GENERATE CONVERT COMMAND */
#define      SER_NUM       SERIAL_PORT_ONE
#define      OSC_TIMER_NUM TIMER_ZERO
#define      CC_TIMER_NUM  TIMER_ONE
#define      XF_NUM        1
#define      ERROR_CHECK   ON

#define      WAIT_BUFFERS  while(!buffer_rcvd || !buffer_xmtd);
#define      RESET_FLAGS   buffer_rcvd = buffer_xmtd = FALSE
#define      INIT_ARRAYS   init_arrays(t_buffer,r_buffer)
#if          XF_NUM
#define      RESET_BB      asm(" AND 2Fh,IOF"); asm(" OR  20h,IOF")
#define      UN_RESET_BB   asm(" OR  60h,IOF")
#else
#define      RESET_BB      asm(" AND 0F2h,IOF"); asm(" OR  2h,IOF")
#define      UN_RESET_BB   asm(" OR  6h,IOF")
#endif
```

Example 8–1.TMS320C3x / BB – DSP102/202 Driver Header File (Continued)

```
/* TIMER PERIOD VALUES ARE BASED ON AN INPUT CLOCK OF 30 MHz */
#define CD              0xAA
#define DAT             0x9C
#define TIMER_PERIOD    CD

#define WAIT(A)         for(i=0;i<A;i++);

/*************************************************************************/
/* STRUCTURES                                                            */
/*************************************************************************/
typedef union
{
    unsigned int _intval;
    struct {
      signed int chan0   :16;
      signed int chan1   :16;
    } _bitval;
} BB_CASC_WORD;
/*************************************************************************/
/* GLOBAL VARIABLES                                                      */
/*************************************************************************/
extern int  t_buffer;                              /* OUTPUT BUFFER SIZE */
extern int  r_buffer;                               /* INPUT BUFFER SIZE */
extern VPVF output0;            /* OUTPUT DATA BUFFER FOR PROCESSOR      */
extern VPVF input0;             /* INPUT DATA BUFFER FOR PROCESSOR       */
extern VPVF output_xfer0;       /* OUTPUT DATA BUFFER FOR ISR/BB         */
extern VPVF input_xfer0;        /* INPUT DATA BUFFER FOR ISR/BB          */
extern VPVF output1;            /* OUTPUT DATA BUFFER FOR PROCESSOR      */
extern VPVF input1;             /* INPUT DATA BUFFER FOR PROCESSOR       */
extern VPVF output_xfer1;       /* OUTPUT DATA BUFFER FOR ISR/BB         */
extern VPVF input_xfer1;        /* INPUT DATA BUFFER FOR ISR/BB          */
extern VI   buffer_rcvd;        /* CPU–ISR COMM FLAG (INPUT)             */
extern VI   buffer_xmtd;        /* CPU–ISR COMM FLAG (OUTPUT)            */
extern VI   r_index;            /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS */
extern VI   t_index;            /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS */
extern VI   i;                  /* GENERIC COUNTER VARIABLE              */
/*************************************************************************/
/* FUNCTION PROTOTYPES                                                   */
/*************************************************************************/
/***********************/
/* BB DRIVER FUNCTIONS */
/***********************/
void init_arrays(int t_buffer_size, int r_buffer_size);
void init_bb(int period_value);
#if SER_NUM
void c_int07(void);
#else
void c_int05(void);
#endif
```

Example 8–2. TMS320C3x – BB DSP102/202 Driver

```
/*************************************************************************/
/*   BBDRVR.C                                                            */
/*                                                                       */
    /*   TMS320C3x – BB DSP102/202 DRIVER
*/
/*************************************************************************/
#include <math.h>
#include <stdlib.h>
#include <bb.h>
/*************************************************************************/
/* GLOABL VARS                                                           */
/*************************************************************************/
int  t_buffer = BLOCK_SIZE;                        /* OUTPUT BUFFER SIZE */
int  r_buffer = BLOCK_SIZE;                         /* INPUT BUFFER SIZE */
VPVF output0;                  /* OUTPUT DATA BUFFER FOR PROCESSOR       */
VPVF input0;                   /* INPUT DATA BUFFER FOR PROCESSOR        */
VPVF output_xfer0;             /* OUTPUT DATA BUFFER FOR ISR/BB          */
VPVF input_xfer0;              /* INPUT DATA BUFFER FOR ISR/BB           */
VPVF output1;                  /* OUTPUT DATA BUFFER FOR PROCESSOR       */
VPVF input1;                   /* INPUT DATA BUFFER FOR PROCESSOR        */
VPVF output_xfer1;             /* OUTPUT DATA BUFFER FOR ISR/BB          */
VPVF input_xfer1;              /* INPUT DATA BUFFER FOR ISR/BB           */
VI   buffer_rcvd = FALSE;      /* CPU–ISR COMM FLAG (INPUT)              */
VI   buffer_xmtd = FALSE;      /* CPU–ISR COMM FLAG (OUTPUT)             */
VI   r_index = 0;              /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS */
VI   t_index = 0;              /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS */
VI   i;                        /* GENERIC COUNTER VARIABLE               */
/*************************************************************************/
/* FUNCTION DECLARATIONS                                                 */
/*************************************************************************/
/*************************************************************************/
/* VOID C_INT05() OR C_INT07():                                          */
/*                 ISR FOR HANDLING DATA TRANSFER BETWEEN C3X SERIAL PORT */
/*                 ONE AND THE A/D,D/A. ASSUMES SYNCHRONOUS OPERATION.    */
/*************************************************************************/
#if SER_NUM
void c_int05(void) {}
void c_int07(void)
#else
void c_int07(void) {}
void c_int05(void)
#endif
{
     BB_CASC_WORD temp;
     VPVF swap;
```

Example 8–2.TMS320C3x – BB DSP102/202 Driver (Continued)

```
     /* DSP102/202 TRANSFER TWO SIXTEEN BIT WORDS REPRESENTING   */
     /* BOTH CHANNELS IN ONE THIRTYTWO BIT WORD. EXTRACT INTO    */
     /* THE INPUT_XFER BUFFERS                                   */
     temp._intval = SERIAL_PORT_ADDR(SER_NUM)->r_data;
     input_xfer0[r_index] = temp._bitval.chan0;
     input_xfer1[r_index] = temp._bitval.chan1;

     /* WRITE OUTPUT_XFER BUFFER VALUE BY CASCADING BOTH CHANNELS */
     temp._bitval.chan0 = output_xfer0[t_index];
     temp._bitval.chan1 = output_xfer1[t_index];
     SERIAL_PORT_ADDR(SER_NUM)->x_data = temp._intval;

     /* CHECK IF BUFFERS ARE FULL */
     if(++r_index == r_buffer)
     {
         /* CHECK CPU SYNCHRONIZATION FLAG */
#if ERROR_CHECK
/*       if(buffer_rcvd == TRUE) error_in_real_time(); */
         if(buffer_rcvd == TRUE) for(;;);
#endif
         swap          = input0;
         input0        = input_xfer0;
         input_xfer0   = swap;
         swap          = input1;
         input1        = input_xfer1;
         input_xfer1   = swap;
         r_index       = 0;
         buffer_rcvd   = TRUE;
     }
     if(++t_index == t_buffer)
     {
         /* CHECK CPU SYNCHRONIZATION FLAG */
#if ERROR_CHECK
/*       if(buffer_xmtd == TRUE) error_in_real_time(); */
         if(buffer_xmtd == TRUE) for(;;);
#endif
         swap           = output0;
         output0        = output_xfer0;
         output_xfer0   = swap;
         swap           = output1;
         output1        = output_xfer1;
         output_xfer1   = swap;
         t_index = 0;
         buffer_xmtd = TRUE;
     }
}
```

Example 8–2.TMS320C3x – BB DSP102/202 Driver (Continued)

```
/*************************************************************************/
/* INIT_ARRAYS(): INITIALIZE DATA ARRAY PARAMETERS                       */
/*************************************************************************/
void init_arrays(int t_buffer, int r_buffer)
{
     int i;
     /*********************************/
     /* INITIALIZE AND ZERO FILL ARRAYS */
     /*********************************/
     if(!(input0         = (float *) calloc(r_buffer,sizeof(float))))
        heap_overflow();
     if(!(output0        = (float *) calloc(t_buffer,sizeof(float))))
        heap_overflow();
     if(!(input_xfer0    = (float *) calloc(r_buffer,sizeof(float))))
        heap_overflow();
     if(!(output_xfer0   = (float *) calloc(t_buffer,sizeof(float))))
        heap_overflow();
     if(!(input1         = (float *) calloc(r_buffer,sizeof(float))))
        heap_overflow();
     if(!(output1        = (float *) calloc(t_buffer,sizeof(float))))
        heap_overflow();
     if(!(input_xfer1    = (float *) calloc(r_buffer,sizeof(float))))
        heap_overflow();
     if(!(output_xfer1   = (float *) calloc(t_buffer,sizeof(float))))
        heap_overflow();

     for(i = 0; i < t_buffer; i++)
     {
       output0[i] = output_xfer0[i] = 0.0;
       output1[i] = output_xfer1[i] = 0.0;
     }
}

/*************************************************************************/
/* INIT_BB(): INITIALIZE COMMUNICATIONS TO DSP102/202                    */
/*************************************************************************/
void init_bb(int period_value)
{
     /* RESET D/A, MAKE SURE RESET IS HELD LOW SUFFICIENTLY (?) LONG */
     RESET_BB;
     WAIT(50);

#if GEN_OSC
     /* CONFIGURE C3X TIMER AS BB A/D OSC */
     TIMER_ADDR(OSC_TIMER_NUM)->gcontrol = 0x0;
     TIMER_ADDR(OSC_TIMER_NUM)->counter  = 0x0;
     TIMER_ADDR(OSC_TIMER_NUM)->period   = 0x0;
     TIMER_ADDR(OSC_TIMER_NUM)->gcontrol = FUNC | GO | HLD_ | CP_ | CLKSRC;
#endif
```

Example 8–2.TMS320C3x – BB DSP102/202 Driver (Continued)

```
     /* CONFIGURE SERIAL PORT */
     SERIAL_PORT_ADDR(SER_NUM)->gcontrol      = 0x0;
     SERIAL_PORT_ADDR(SER_NUM)->s_x_control   = CLKXFUNC | DXFUNC | FSXFUNC;
     SERIAL_PORT_ADDR(SER_NUM)->s_r_control   = CLKRFUNC | DRFUNC | FSRFUNC;
     SERIAL_PORT_ADDR(SER_NUM)->s_rxt_control = 0x0F;
     SERIAL_PORT_ADDR(SER_NUM)->s_rxt_period  = 0x0;
     SERIAL_PORT_ADDR(SER_NUM)->gcontrol      = XCLKSRCE | XLEN_32 | RLEN_32 |
                                                XINT | XRESET | RRESET;
     /* CLEAR SERIAL TRANSMIT DATA    */
     SERIAL_PORT_ADDR(SER_NUM)->x_data = 0x0;

     /* TAKE A/D,D/A OUT OF RESET, (OPTIONALY) CLEAR THE INT FLAG REG, */
     /* ENABLE THE APPROPRIATE SERIAL PORT TRANSMIT INT AND ENABLE     */
     /* GLOBAL INTERRUPTS                                              */
     UN_RESET_BB;
     CL_INT_FL_REG;

#if SER_NUM
     EN_SER_PORT_XMT_INT_1;
#else
     EN_SER_PORT_XMT_INT_0;
#endif

     EN_GLOBAL_INTS;

#if GEN_CC
     /* CONFIGURE C3X TIMER 1 AS BB A/D,D/A CONVERT CLOCK  */
     TIMER_ADDR(CC_TIMER_NUM)->gcontrol = 0x0;
     TIMER_ADDR(CC_TIMER_NUM)->counter  = 0x0;
     TIMER_ADDR(CC_TIMER_NUM)->period = period_value;
     TIMER_ADDR(CC_TIMER_NUM)->gcontrol = FUNC | GO | HLD_ | CLKSRC;
#endif
}
```

Example 8–3. General Macro Definitions

```
/*****************************************************************************/
/* general.h v4.2                                                            */
/* Copyright (c) 1991 Texas Instruments Incorporated                         */
/*****************************************************************************/
#ifndef _GENERAL
#define _GENERAL
/*****************************************************************************/
/* COMMON MACRO DEFINTIONS                                                   */
/*****************************************************************************/
#ifndef OFF
#define OFF                   0x00
#endif

#ifndef ON
#define ON                    0x01
#endif

#ifndef FALSE
#define FALSE                 0x00
#endif

#ifndef TRUE
#define TRUE                  0x01
#endif

#ifndef CLEAR
#define CLEAR                 0x00
#endif

#ifndef SET
#define SET                   0x01
#endif
```

Example 8–3.General Macro Definitions (Continued)

```
/*************************************************************************/
/* GENERAL C3x MACROS                                                    */
/*************************************************************************/
#ifndef INIT_XF_PINS
#define INIT_XF_PINS          asm(" LDI  00h,IOF")
#endif

#ifndef CL_INT_FL_REG
#define CL_INT_FL_REG         asm(" LDI  0h,IF")
#endif

#ifndef EN_GLOBAL_INTS
#define EN_GLOBAL_INTS        asm(" OR   2000h,ST")
#endif

#ifndef EN_SER_PORT_XMT_INT_0
#define EN_SER_PORT_XMT_INT_0 asm(" OR 10h,IE")
#endif

#ifndef EN_SER_PORT_RCV_INT_0
#define EN_SER_PORT_RCV_INT_0 asm(" OR 20h,IE")
#endif

#ifndef EN_SER_PORT_XMT_INT_1
#define EN_SER_PORT_XMT_INT_1 asm(" OR 40h,IE")
#endif

#ifndef EN_SER_PORT_RCV_INT_1
#define EN_SER_PORT_RCV_INT_1 asm(" OR 80h,IE")
#endif

#ifndef ENABLE_CACHE
#define ENABLE_CACHE          asm(" OR 800h,ST")
#endif

#endif /* #ifndef _GENERAL */
```

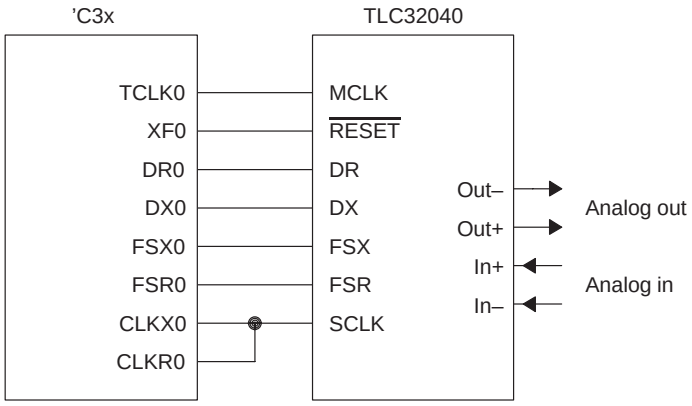
Example 8–4. Common Driver Header File

```
/*************************************************************************/
/*   COMMDRVR.H                                                          */
/*                                                                       */
/*   TMS320C3x – COMMOM DRIVER HEADER FILE                               */
/*************************************************************************/
#include <c30_per.h>
/*************************************************************************/
/* COMMON STRUCTURES                                                     */
/*************************************************************************/
typedef volatile int VI;
typedef volatile float VF;
typedef VF * volatile VPVF;
typedef VI * volatile VPVI;

/*************************************************************************/
/* FUNCTION PROTOTYPES                                                   */
/*************************************************************************/
void c_int99(void);
void heap_overflow(void);
void init_c30(void);
void error_in_real_time(void);
```

8.4 TLC32040 Interface to the TMS320C3x

Figure 8–6 shows how to interface the ’C3x with zero glue logic to a Texas Instruments’ TLC32040 14-bit analog interface circuit (AIC). The following sections describe the steps required to initialize and set up the ’C3x timer and serial port, and to reset and program the TLC32040.

Figure 8–6. TM320C3x-to-TLC32040 Interface



8.4.1 Resetting the Analog Interface Circuit

The ’C31’s XF0 signal is connected to the $\overline{\text{RESET}}$ signal of the AIC. By toggling the $\overline{\text{RESET}}$ signal, the ’C31 can reset the AIC. This is achieved by executing the following instructions:

```
rpts 40              ;  Execute next instruction 40x
ldi 2h,IOF           ;  Pull AIC into reset
ldi 6h,IOF           ;  Pull AIC out of reset
```

8.4.2 Initializing the TMS320C31 Timer

The ’C31’s timer (TCLK0) signal is connected to the AIC’s master clock (MCLK) signal. The MCLK signal drives all the key logic signals of the AIC, such as the shift clock, the switched-capacitor filter clocks, and the ADC and DAC timing signals. The timer pulses the TCLK0 signal whenever the ’C31 timer counter register (which is memory mapped to 0x808024) counts up to the value in the timer period register (which is memory mapped to 0x808028). Then, the timer counter register resets to 0 and repeats. (For a detailed description of the ’C31 timer, see the *TMS320C3x User’s Guide*.) Because of differences between the maximum frequency of the ’C31’s timer and the maximum and minimum frequencies of the AIC, observe the following constraints:

- **Minimum Timer Period Register Value**.The ’C31 running at 50 MHz can generate a maximum timer frequency of 12.5 MHz (CLKIN/4), which is above the AIC’s tested master clock frequency maximum of 10 MHz. If you use frequencies beyond those listed in the *TLC32040 Data Sheet*, the resulting performance can be unpredictable. If the timer is run in pulse mode (control value is 0x2C1) the minimum period of 1 results in 12.5-MHz master pulse rate and a period of 2 results in 6.25 MHz. See the *TLC32040 Data Sheet* for more information.

- **Maximum Timer Period Register Value**. The AIC’s minimum master clock frequency is 75 kHz. Taking into account the ’C31 maximum timer frequency of 12.5 MHz and the AIC’s minimum master clock frequency, the maximum value in the ’C31’s timer counter register must be 165 (12.5 MHz / 75 kHz = 166.7). The ’C31’s timer counts down to 0; therefore, you must subtract 1 from this number (166 – 1 = 165). The TLC32040 specification describes a minimum clock frequency, since the internal signals of the AIC are stored in capacitors that must be periodically updated.

The following ’C31 assembly code initializes the timer in clock mode with a timer period of 1. The following code initializes timer 0 to generate a square wave (clock mode) on the TCLK0 pin at a frequency of 6.25 MHz (timer period = 1):

```
TGCR0   .set    808020h     ; Timer 0 global control register
TCNT0   .set    808024h     ; Timer 0 counter register
TPR0    .set    808028h     ; Timer 0 period register
TIMVAL  .word   3c1h        ; Timer global control register value
        ldp     @TGCR0      ; Set Data Page
        ldi     0h,R4       ; Initialize R4 to zero
        ldi     1h,R0       ; Initialize R0 to 1
        sti     R4,@TGCR0   ; Reset timer0
        sti     R0,@TPR0    ; Store timer0 period
        sti     R4,@TCNT0   ; Reset timer0 counter
        ldi     @TIMVAL,R7  ; Load timer control value
        sti     R7,@TGCR0   ; Start timer 0
```

A period of 0 is not allowed in pulse mode. If the timer is run in clock mode, the resulting output is a square wave with a frequency of half that of pulse mode. A period of 0 is allowed in clock mode resulting in a 12.5-MHz clock.

8.4.3 Initializing the TMS320C31 Serial Port

This section explains how to initialize the:

- ❑ ’C31 serial port
- ❑ ’C31 serial-port control register (memory mapped to 0x808040)
- ❑ FSX/DX/CLKX control register (memory mapped to 0x808042)
- ❑ FSR/DR/CLKR control register (memory mapped to 0x808043)

For a detailed description of the ’C31 serial port, see the *TMS320C3x User’s Guide*.

Example 8–5 shows the assembly code to initialize the serial port global control register (SGCR0) for the ’C31 in the following manner:

1) Issue transmit and receive resets
2) Enable receive and transmit interrupts
3) Set 16-bit receive and transmit transfers
4) Set FSX and FSR, CLKX and CLKR active low
5) Set continuous mode
6) Set variable data rate transfers

See the example code supplied with the DSP for help on setting up the AIC.

Example 8–5. Initialize the Serial Port Global Control Register

```
SGCR0     .set   808040h      ; Serial port 0 global control register  ;
SPCX0     .set   808042h      ; Serial port 0 FSX/DX/CLKX control reg. ;
SPCR0     .set   808043h      ; Serial port 0 FSR/DR/CLKR control reg. ;
SINIT0    .word  0e973300h    ; Enable RINT & 16–bit transfers
SINIT1    .word  111h         ; Configure as serial port pins
          ldp    @SGCR0       ; Set Data Page
          ldi    0h,R4        ; Initialize R4 to zero
          sti    R4,@SGCR0
          ldi    @SINIT1,R7   ; Reset and
          sti    R7,@SPCX0    ;   initialize serial port
          sti    R7,@SPCR0    ;   initialize serial port
          ldi    @SINIT0,R7   ; Reset and
          sti    R7,@SGCR0    ;   initialize serial port
```

8.4.4 Initializing the AIC

Once the ’C31 supplies MCLK, initializes its serial port, and resets the AIC, you can initialize the AIC to a specified sample rate. The AIC sampling rate is determined by the values of two registers (Tx counter A and Tx counter B) in the AIC’s transmit and receive sections. These values are loaded into the respective counter whenever the counter counts down to 0. The Tx counters A and B determine the D/A conversion timing. The Rx counters A and B determine the A/D conversion timing. For more information, see the *TLC32040 AIC Data Sheet*. The formula for the conversion frequency is given in Equation 8–1.

Equation 8–1. Conversion Frequency

$$Conversion\_\ frequency = \frac{MCLK}{2 \times A \times B}$$

To ensure that the switched-capacitor lowpass and bandpass filters meet their transfer function characteristics, the frequency of the clock inputs of the switched-capacitor filter must be 288 kHz. Otherwise, the upper and lower cutoff frequencies of the lowpass and bandpass are scaled accordingly. Equation 8–2 shows the switched-capacitor filter frequency.

Equation 8–2. Switched Capacitor Filter Frequency

$$SCF\_Clock\_\ frequency = \frac{MCLK}{2 \times A}$$

For example, using this equation for an 8-kHz sampling rate with an MCLK of 6.25 MHz results in a Tx counter A of 11 [A = MCLK / (2 × SCF)]. Using Equation 8–2, Tx counter B results in 36 [B = MCLK / (2 × A × *Conversion_Frequency*)].

To initialize the AIC’s Tx counter A and B registers, you must send a primary communication followed by a secondary communication (as explained in the following sections). Primary communications load values into the D/A while secondary communications load A/D internal registers, such as the control register, Tx counters A and B, and Rx counters A and B.

8.4.4.1 Primary Communications

Primary communications have a data value in the 14 MSBs (D15–D2) of data and a mode selection in the two least significant bits (LSBs) (D1–D0). This format is shown in Figure 8–7.

The AIC sends the data value to the DAC and enables one of the modes shown in Table 8–2, depending on the two LSBs.

Figure 8–7. Primary Communication Data Format

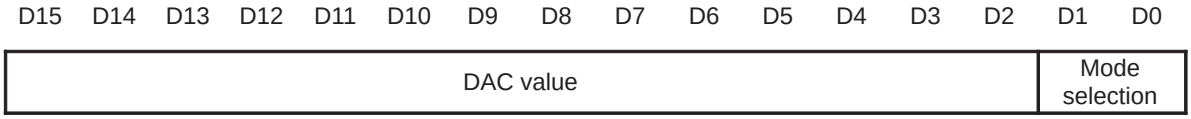


Table 8–2. Primary Communications Mode Selection

LSBs	Mode
00	Tx counter A ← TA, Rx counter A ← RA Tx counter B ← TB, Rx counter B ←RB
01	Tx counter A← TA + TA', Rx counter A ← RA + RA' Tx counter B ←TB, Rx counter B ← RB
10	Tx counter A ←TA - TA', Rx counter A ← RA + RA' Tx counter B ←TB, Rx counter B ← RB
11	Tx counter A ← TA, Rx counter A ← RA Tx counter B ←TB, Rx counter B ← RB

The second and third modes use the TA' and RA' registers to advance or slow down the sampling frequency by respectively shortening or lengthening the sample period. This is particularly useful in modem applications, where it can enhance the signal-to-noise performance, perform frequency-tracking functions, and generate nonstandard modem frequencies.

8.4.4.2 Secondary Communications

Secondary communication follows a primary communication that has the two LSBs set to 11 together. This secondary communication programs the AIC by loading the A, A', B, or control registers. Figure 8–8 shows the secondary communication data format. The TA, RA, TB, and RB values are unsigned. The TA' and RA' values are in signed 2s-complement format. The control register enables bandpass filters and asynchronous transmit/receive, enables and disables auxiliary inputs, and changes input gain.

Table 8–3 describes the control register bit fields.

Figure 8–8. Secondary Communication Data Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	TA register value (unsigned)					X	X	RA register value (unsigned)					0	0
X	TA' register value (signed 2s complement)						X	RA' register value (signed 2s complement)						0	1
X	TB register value (unsigned)						X	RB register value (unsigned)						1	0
X	X	X	X	X	X	X	X	Control register						1	1

Table 8–3. Control Register Bit Fields

D7 D6	D5	D4	D3	D2
Input gain 0 0 = 1X for ± 6-V analog input 0 1 = 2X for ± 3-V analog input 1 0 = 4X for ± 1.5-V analog input 1 1 = 1X for ± 6-V analog input	**Transmit/receive** 0 = asynchronous 1 = enables	**AUX IN pins** 0 = disables 1 = enables	**Loopback function** 0 = disables 1 = enables	**Bandpass filter** 0 = deletes 1 = inserts

The assembly code in Example 8–6 sets the TA and TB registers of the AIC. This code transmits a 16-bit word to the AIC and then waits until the transmit interrupt is generated by the serial port. Four commands are transmitted starting with a 0, then the TB and RB values, followed by the TA and RA values, and finally the control word. TA and RA values should be the last values transmitted, since they change the AIC sample rate. By transmitting these values last, the sample rate is not changed until the AIC receives the last program word. In this way, very high sample rates can be achieved. Each command transmits three 16-bit words: a primary communication, a secondary communication, and a zero-data word.

Example 8–6.Setting the TA and TB Registers

```
;-------------------------------------------------------------------------
; LOOPAIC.ASM is an example program which shows how to initialize and use
; the TLC32040.  The analog output (DAC output) is either a ramp signal
; (RAMPEN=1) or a loopback of the analog input (RAMPEN=0).
;-------------------------------------------------------------------------
;-----------------------------------
; Define constants used by program
;-----------------------------------
RAMPEN    .set   1                  ; Set to 1 to generate ramp at AOUT
T0_ctrl   .set   0x808020           ; TIM0 gl control
T0_count  .set   0x808024           ; TIM0 count
T0_prd    .set   0x808028           ; TIM0 prd
S0_gctrl  .set   0x808040           ; SP 0 global control
S0_xctrl  .set   0x808042           ; SP 0 FSX/DX/CLKX port ctl
S0_rctrl  .set   0x808043           ; SP 0 FSR/DR/CLKR port ctl
S0_xdata  .set   0x808048           ; SP 0 Data transmit
S0_rdata  .set   0x80804C           ; SP 0 Data receive
TA        .set   12                 ; AIC timing register values
TB        .set   15                 ;
RA        .set   12                 ;
RB        .set   15                 ;
GIE       .set   0x2000             ; This bit in ST turns on interrupts
;-----------------------------------
; Define some constant storage data
;-----------------------------------
A_REG           .word     (TA<<9)+(RA<<2)+0   ; A registers
B_REG           .word     (TB<<9)+(RB<<2)+2   ; B registers
C_REG           .word     10000011b           ; control
S0_gctrl_val    .word     0x0E970300          ; Serial port control register
                                              ; values
S0_xctrl_val    .word     0x00000111          ;
S0_rctrl_val    .word     0x00000111          ;
RAMP            .word     0                   ; RAMP count value
ADC_last        .word     0                   ; Last received ADC value
```

Example 8–6.Setting the TA and TB Registers (Continued)

```
;*************************************************
; Begin main code loop here
;*************************************************
main      or    GIE,ST          ; Turn on INTS
          ldi   0x3,IE         ; Enable XINT/RINT
          call  INIT
          b     main            ; Do it again!
;------------------------------
DAC2      push  ST              ; DAC Interrupt service routine
          push  R3              ;
          .if   RAMPEN          ; If RAMPEN=1 assemble this code
          ldi   @RAMP,R3        ;
          addi  256,R3          ; Add a value to RAMP
          sti   R3,@RAMP        ;
          .else                 ; Else assemble this
          ldi   @ADC_last,R3    ;
          .endif                ;
          andn  3,R3            ;
          sti   R3,@S0_xdata    ; Output the new DAC value
          pop   R3              ;
          pop   ST              ;
          reti                  ;
;------------------------------
ADC2      push  ST              ;
          push  R3              ;
          ldi   @S0_rdata,R3    ;
          sti   R3,@ADC_last    ;
          pop   R3              ;
          pop   ST              ;
          reti                  ;
;************************************************;
; The startup stub is used during initialization only ;
; and can be safely overwritten by the stack or data  ;
;************************************************;
          .entry ST_STUB        ; Debugger starts here
INIT      ldp    T0_ctrl        ; Use kernel data page and stack
          ldi    0,R0           ; Halt TIM0 & TIM1
          sti    R0,@T0_ctrl    ;
          sti    R0,@T0_count   ; Set counts to 0
          ldi    1,R0           ; Set periods to 1
          sti    R0,@T0_prd     ;
          ldi    0x2C1,R0       ; Restart both timers in pulse mode
          sti    R0,@T0_ctrl    ;
          ;--------------------
          ldi    @S0_xctrl_val,R0;
          sti    R0,@S0_xctrl   ; transmit control
          ldi    @S0_rctrl_val,R0;
          sti    R0,@S0_rctrl   ; receive control
          ldi    0,R0           ;
          sti    R0,@S0_xdata   ; DXR data value
          ldi    @S0_gctrl_val,R0; Setup serial port
          sti    R0,@S0_gctrl   ; global control
```

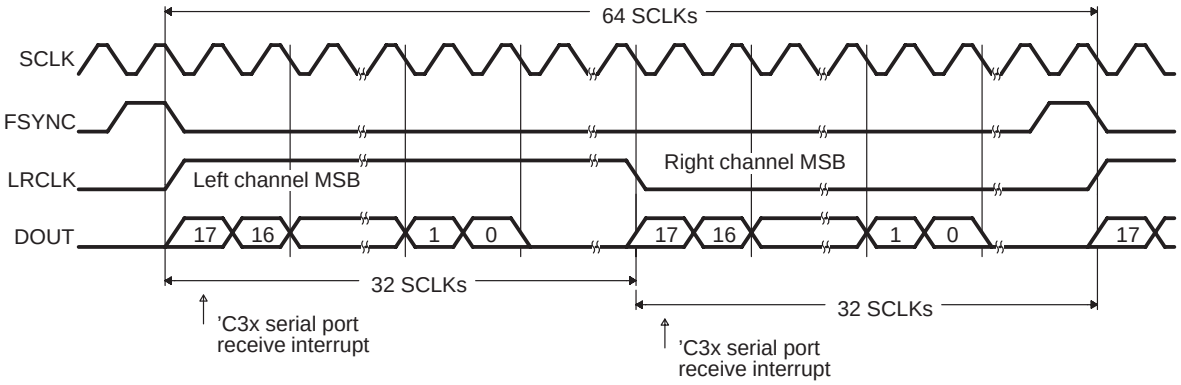
Example 8–6.Setting the TA and TB Registers (Continued)

```
;=======================================================;
; This section of code initializes the AIC             ;
;=======================================================;
AIC_INIT  LDI    0x10,IE          ; Enable only XINT interrupt
          andn   0x34,IF          ;
          ldi    0,R0             ;
          sti    R0,@S0_xdata     ;
          RPTS   0x040            ;
          LDI    2,IOF            ; XF0=0 resets AIC
          rpts   0x40             ;
          LDI    6,IOF            ; XF0=1 runs AIC
          ;---------------------
          ldi    @C_REG,R0        ; Setup control register
          call   prog_AIC         ;
          ldi    0xfffc  ,R0      ; Program the AIC to be real slow
          call   prog_AIC         ;
          ldi    0xfffc|2,R0      ;
          call   prog_AIC         ;
          ldi    @B_REG,R0        ; Bump up the Fs to final rate
          call   prog_AIC         ; (smallest divisor should be last)
          ldi    @A_REG,R0        ;
          call   prog_AIC         ;
          b      main
;-------------------------------
prog_AIC  ldi    @S0_xdata,R1     ; Use original DXR data during 2 ndy
          sti    R1,@S0_xdata     ;
          idle
          ldi    @S0_xdata,R1     ; Use original DXR data during 2 ndy
          or     3,R1             ; Request 2 ndy XMIT
          sti    R1,@S0_xdata     ;
          idle                    ;
          sti    R0,@S0_xdata     ; Send register value
          idle                    ;
          andn   3,R1             ;
          sti    R1,@S0_xdata     ; Leave with original safe value in DXR
          ;---------------------
          ldi    @S0_rdata,R0     ; Fix the receiver underrun by reading
          rets   main             ; the DRR before going to the main loop
;*************************************************;
; Install the XINT/RINT ISR handler directly into    ;
; the vector RAM location it will be used for        ;
;*************************************************;
          .start "SP0VECTS",0x809FC5
          .sect  "SP0VECTS"
          B      DAC2             ; XINT0
          B      ADC2             ; RINT0
```

8.5 TLC320AD58 Interface to the TMS320C3x

The TLC320AD58C serial interface provides several master and slave modes for 16-bit or 18-bit data output. This allows it to be compatible to a wide range of DSPs. To interface with the ’C3x 32-bit floating-point DSP, the 18-bit master mode “100” was chosen to get an 18-bit resolution result and meet the ’C3x serial port requirements. The timing diagram is shown in Figure 8–9.

Figure 8–9. TLC320AD58C Serial Interface 18-bit Master Mode “100” Timing Diagram



The frame sync signal (FSYNC) is then used to designate valid data from the ADC and is active for one shift clock period. After the falling edge of FSYNC, the left channel data is shifted out on the falling edge of SCLK with the MSB (D17) first. When the last data bit is shifted out, the output remains low for another 14 SCLKs to get a total of 32 SCLK periods each channel. After 32 SCLKs, LRCLK goes low and the right channel data is then shifted out. FSYNC and LRCLK frequency are fixed to the sampling frequency (F_S = MCLK/256 or MCLK/384, depending on the status of the CMODE input pin). The conversion cycle is synchronized to the rising edge of LRCLK and, therefore, to the falling edge of FSYNC. Although data is shifted out in two separate time packets representing the left and right channel digital outputs, the analog inputs are sampled and converted simultaneously. In the master mode, SCLK, FSYNC, and LRCLK are generated internally from MCLK, depending on the status of the CMODE input pin, as shown in Table 8–4.

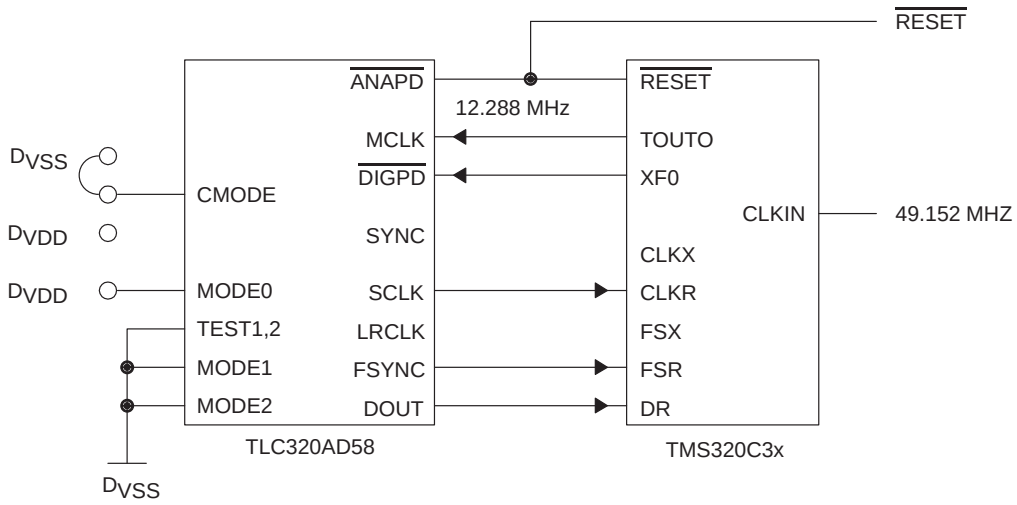
Table 8–4. Master-Clock-to-Sample-Rate Conversion

MCLK (MHz)	CMODE	SCLK (MHz)	Sample Rate (kHz)
12.288 18.432	Low High	3.072	48
11.290 16.934	Low High	2.8224	44.1
8.129 12.288	Low High	2.048	32
0.256 0.384	Low High	0.064	1

The ’C30 uses two bidirectional serial ports; the ’C31 and ’C32 each have one. Each serial port controls six port pins for receiving/transmitting data: FSR/FSX, CLKR/CLKX, and DR/DX. Figure 8–10 shows the glueless interface to the TLC320AD58C using the SCLK, FSYNC, and DOUT signals. Mode “100” is set by pulling the MODE1 and MODE2 pins low and the MODE0 pin high. The master clock is derived from the ’C3x to make sure all clock signals are synchronized. The ’C3x is running at 49.152 MHz and provides the required MCLK frequency of 12.288 MHz at the timer 0 output pin in order to get a 48-kHz sample rate. CMODE must be pulled low. If other sample rates are required, see Table 8–4.

The TLC320AD58C analog function blocks are initialized together with the DSP by a system reset after all supply voltages are stable. The digital function blocks are initialized by pulling down $\overline{\text{DIGPD}}$ for several microseconds. After the rising edge of $\overline{\text{DIGPD}}$, the device resumes normal operation. When $\overline{\text{DIGPD}}$ is low, the TLC320AD58C digital function blocks are shut down and power consumption is reduced. However, if power down mode is not required, this signal can be tied to $\overline{\text{ANAPD}}$. In both cases, refer to the TI *Data Acquisition Circuits Data Book* for setup timing requirements. All digital inputs and outputs of the ’C3x and the TLC320AD58C are 5-V TTL compatible. To reduce ringing and overshot, a serial damping resistor (50 Ω) is recommended for the master clock signal.

Figure 8–10. Interface Between the-TMS320C3x and the TLC320AD58C



The ’C3x can be configured to receive a maximum of 32 bits of data per word. But, the TLC320AD58C transmits a total of 64 bits after the FSYNC pulse appears. This forces the DSP to read the left and right channels back-to-back. To accomplish this, the ’C3x serial port configuration is toggled between continuous mode and burst mode. In burst mode, FSYNC indicates the start of a new data transfer. In continuous mode, the new data transfer starts immediately after the last bit of the previous transfer has been shifted out. Both the serial port and the timer registers are memory mapped. Eight memory-mapped registers are provided for each serial port:

- One global control register—defines the serial port configuration
- Two control registers—set the function of the CLKX/CLKR and FSX/FSR pins
- Three receive/transmit timer registers
- One data receive register
- One data transmit register

If the serial port shift clock (CLKR/CLKX) is generated externally, the corresponding timer can be used as a general-purpose timer. See the *TMS320C3x User’s Guide* for more information on the ’C3x serial port.

Example 8–7 shows the C code for interfacing a TLC320AD58 to the ’C3x. Example 8–8 (page 8-36) shows the header file for the C code of Example 8–7. Example 8–9 (page 8-38) shows the interrupt table vector listing. These examples perform the following tasks:

- Initialize the TLC320AD58C and the ’C30 serial port 1 to meet the TLC320AD58C serial interface timing requirements
- Set up the timer 0 period register to generate the required MCLK frequency

On a serial port 1 receive interrupt, which occurs after receiving 32 bits from either the left channel or right channel, the program reads from the serial port receive register and converts the input signal into a floating-point number within the range of –1.0 and 1.0. It then changes the serial port configuration from burst to continuous mode when the right channel has been received, or from continuous to burst mode when the left channel has been received. The transmit port is configured as the receive port for connection to the 18-bit TMS57014A stereo DAC. Remember that the data has to be written to the data transmit register no later than three CLKX cycles before the FSYNC pulse occurs (in burst mode) or the next transfers starts (in continuous mode).

Example 8–7. Interfacing the 18-bit TLC320AD58 to TMS320C3x

```
/*************************************************************************/
/* File: AD58. C                                                         */
/* interfacing the 18–Bit TLC320AD58 to TMS320C3x                        */
/*************************************************************************/

/*include files */
/*--------------*/
#include “vectors.h”
#include “c3x.h”

/* global variables */
/*------------------*/
float   Ichannel;
float   r_channel;

/*-----------------------------------------------------------------------*/
/* main program                                                          */
/*-----------------------------------------------------------------------*/
void main(void)
```

Example 8–7.Interfacing the 18-bit TLC320AD58 to TMS320C3x (Continued)

```
{
  asm("         ldi   1000h,ST");      /* clear and enable cache */
  asm("         ldi      0h,IE");      /* clear all interrupt masks*/
  asm("         ldi      0h,IF");      /* clear all pending interrupt*/
  init_t0();                           /* Generate AD58 MCLK, if required */
  init_sl();                           /* Initialize serial port 1 */
  init_ad58();
  asm("         ldi _ERINT1_CPU,IE");  /* enable serial port 1 receive int */
  asm("         or      _GIEBIT,ST:);  /* global enable interrupts */
  while(1);                            /* wait on interrupt */
}

/*-------------------------------------------------------------------------*/
/* Subroutine to initialize Serial Port 1 to communicate with TLC320AD58   */
/*-------------------------------------------------------------------------*/
void init_s1 (void)
{
serial_port[l][X_PORT] = X1_MODE;
serial_port[1][R_PORT] = R1_MODE;
serial_port[l][GLOBRL] = S1_CONFIG;
}

/*-------------------------------------------------------------------------*/
/* Subroutine to initialize Timer 0 to generate TLC320AD58 MCLK            */
/*-------------------------------------------------------------------------*/

void init_t0(void)
{
  timer[0][GLOBAL] = T0_HOLD;
  timer[0][T_COUNTER] = 0X0;
  timer[0][T_PERIOD] = T0_PERIOD;
  timer[0][GLOBAL] = T0_HOLD;
}

/*-------------------------------------------------------------------------*/
/* Serial Port Receive Interrupt Service Routine                           */
/*-------------------------------------------------------------------------*/
void c_int08(void)
{
/* reconfigure serial port to receive both channels within one frame sync  */
   if (serial_port[l][GLOBAL] & 0x0C00)
   {
      /* read LEFT channel and normalize within –1.0..1.0 */
      1_channel = ((float) (serial_port[l][R_DATA] >> 14))/(4.0*65536);
      /* switch to burst mode*/
      serial_port[1][GLOBAL] = serial_port[1][GLOBAL] & 0xFFFFF3FF;
      /* if transmitting to DAC, make sure to write to the transmit register no
         later than 3 SCLK=CLKX cycles before the rising edge of FSYNC */
   }
```

Example 8–7.Interfacing the 18-bit TLC320AD58 to TMS320C3x (Continued)

```
   else
   {
      /* read RIGHT channel and normalize within –1.0..1.0 */
      r_channel = ((float) (serial_port[l][R_DATA] >> 14))/4.0*65536

      /* switch to continuous made */
      serial_port[ll[GLOBAL] = serial_port[1][GLOBAL] | 0x0C00;

      /* if transmitting to DAC, make sure to write to the transmit register no
         later than 3 SCLK=CLKX cycles before the next transfer */
    }
}

/*-------------------------------------------------------------------------*/
/* Subroutine to initialize TLC320AD58                                     */
/*-------------------------------------------------------------------------*/
void init_ad58(void)

{
   asm(”       ldi      0010b,IOF”);   /* reset XF0, power down AD58 */
   asm(”       rpts       2500   ”);   /* wait for 100 usee before */
   asm(”       nop               ”);   /*         asserting DigPwd */
   asm(”       ldi      0110b,IOF”);   /* AD58 normal operation */
}
```

Example 8–8. C3x.h, Header File Listing

```
/*-------------------------------------------------------------------------*
/
/*    FILE: C3X.H
*/
/*    TMS320C3X CONTROL REGISTER SETTINGS TO SETUP INTERFACE WITH
*/
/*    TLC320AD58 18 BIT MASTER MODE
*/
/*-------------------------------------------------------------------------*
/

/*----------------------------------*/
/*  Serial Port 1 Initialization    */
/*----------------------------------*/
#define X1_MODE   0x000000111        /* FSX/DX/CLKX are serial port pins   */
#define R1 MODE   0x000000111        /* FSX/DX/CLKX are serial port pins   */
#define S1_CONFIG 0x00EBC3C00        /* SerialPort Configutration          */
                                     /* FSX/FSR input                      */
                                     /* FSX/FSR signals active high        */
                                     /* external CLKX/R                    */
                                     /* CLIM/CLKR active low               */
                                     /* fixed data rate mode               */
                                     /* 32–bit data width                  */
                                     /* TX/RX interrupts are enabled       */
                                     /* XRESET/RRESET set to O             */
                                     /*        (take out of reset)         */

/*----------------------------*/
/*  Timer 0 Initialization    */
/*----------------------------*/
/T TOUT Frequency (clock mode) = 1/[8*CLKIN*TO_PERIOD], if TO_PERIOD period>0
*/
/*                             = 1/[4*CLKINI. if TO_PERIOD period ; 0    */
#define TO_PERIOD   0       /* TOUTO = 12.288 MHz for 49.152 MHz CLKIN   */
#define TO_HOLD     0x0301  /* clock mode, 50% duty cycle   */
#define TO_GO       0x03C1

/*-------------------*/
/*  Interrupt Mask   */
/*-------------------*/

asm(”_ERINT1_CPU   .set    80h:);      /* enable serial port 1 receive int */
asm(”_GIEBIT)      .set    2000h”);    /* global enable interrupts */
```

Example 8–8.C3x.h, Header File Listing (Continued)

```
/*--------------------------------------------------------------------------*
/
/*    TMS320C3X CONTROL REGISTER LOCATIONS
*/
/*--------------------------------------------------------------------------*
/

/*-----------------*/
/*  Serial Ports  */
/*-----------------*/
/* SERIAL PORT BASE LOCATION */
volatile int (*serial_port)[16 = (volatile int (*)[16]) 0x808040;

/* SERIAL PORT CONTROL REGISTERS */
#define GLOBAL  0                                        /* GLOBAL CONTROL   */
#define X_PORT  2                                        /* TRANSMIT CONTROL */
#define R_PORT  3                                        /* RECEIVE CONTROL  */
Rdefine X_DATA  8                                        /* TRANSMIT DATA    */
#define R_DATA 12                                        /* RECEIVE DATA     */

/*-----------------*/
/*  Timer         */
/*-----------------*/
/* TIMER BASE LOCATION   */
volatile int (*timer)[16] = (volatile int (*)[16]) 0x808020;
#define T_COUNTER 4
#define T_PERIOD  8
```

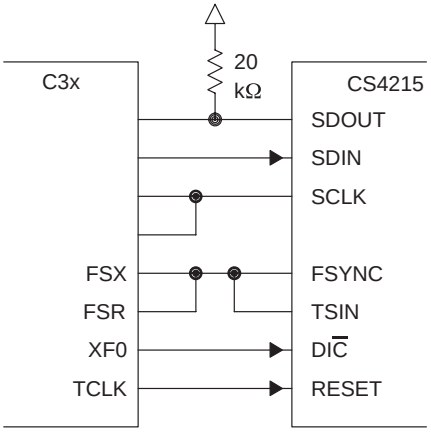
Example 8–9. TMS320C3x Interrupt Vector Table Listing

```
/*--------------------------------------------------------------------------*/
/* Filename: vectors.h   Defines interrupt vectors and trap vectors         */
/*                     for C programs                                       */
/*                                                                          */
/* Usage:    #include vectors.h                                             */
/*
/* Modifications: If you add interrupt service routines, modify             */
/*              this file to insert the vectors at the proper               */
/*              location in the vector table.                               */
/*--------------------------------------------------------------------------*/

  asm("          .global _c_int00          ");
  asm("          .global _c_int08          ");

  asm("          .sect \"vectors\"         ");
  asm("RESET     .word _c_int00 ; external RESET-    ");
  asm("INT0      .word _c_int99 ; external INT0-     ");
  asmi("INT1     .word _c_int99 ; external INT1-     ");
  asm("INT2      .word _c_int99 ; external INT2-     ");
  asm("INT3      .word _c_int99 ; external INT3-     ");
  asm("XINTO     .word _c_int99 ; Serial port 0 XMT  ");
  asm("RINTO     .word _c_int99 ; Serial port 0 RCV  ");
  asm("XINT1     .word _c_int99 ; Serial port 1 XMT  ");
  asm("RINT1     .word _c_int08 ; Serial port 1 RCV  ");
  asm("TINTO     .word _c_int99 ; Timer 0            ");
  asm("TINT1     .word _c_int99 ; Timer 1            ");
  asm("DINT      .word _c_int99 ; DMA complete       ");

  asm("          .space 20      ; Reserved space     ");
  asm("TRAPO                                         ");
  asm("          .loop 28       ; TRAPS 0-27 are     ");
  asm("          .word _c_int99 ; undefined traps    ");
  asm("          .endloop                            ");

  asm("          .space 4       ; TRAPS 28-31 reserved");

/*--------------------------------------------------------------------------*/
/* NOTE: Put all interrupt handlers AFTER this next statement!              */
/*                                                                          */
/*--------------------------------------------------------------------------*/

  asm("          .text                               ");

void c_int99() { }    /* Spurious interrupt handler */
```

8.6 CS4215 Interface to the TMS320C3x

Figure 8–11 shows how to interface the ’C3x with zero glue logic to Crystal Semiconductor’s CS4216 16-bit stereo codec.

Figure 8–11. TMS320C3x-to-CS4216 Interface



Example 8–10 through Example 8–16 show the assembly and C language codes with their respective header files that program and interface the ’C3x to the CS4215. Example 8–10 shows the CS4215 driver interrupt vector table. Example 8–11 (page 8-41) shows the ’C3x serial port transmit interrupt service routine. Example 8–12 (page 8-44) and Example 8–13 (page 8-46) display the C code header files. Example 8–14 (page 8-47) shows the C language common driver routines. Example 8–15 (page 8-49) is the C code header file for Example 8–16 (page 8-59), which displays the C language driver routines for the CS4215.

These files can be downloaded from Texas Instrument’s BBS or ftp site (file-name C3x4215.EXE).

Example 8–10. vecs.asm

```
;***********************************************************
;
;                 vecs.asm
;
;                 staff
;
;                 01–03–92
;
;           (C) Texas Instruments Inc., 1992
;
;           Refer to the file 'license.txt' included with this
;           this package for usage and license information.
;
;***********************************************************
*******************************************************
*   VECS.ASM                                          *
*                                                     *
*   C3x – CS4215 DRIVER INTERRUPT VECTOR TABLE        *
*                                                     *
*   (C) 1991 TEXAS INSTRUMENTS, HOUSTON               *
*******************************************************
*   INTERRUPT AND RESET VECTORS                       *
*******************************************************
        .sect "vecs"      ; interrupt and reset vectors

        .ref _c_int00   ; compiler defined C initialization reset
        .ref _c_int06   ; serial port transmit interrupt service routine
        .ref _c_int08   ; serial port transmit interrupt service routine
        .ref _c_int99   ; unexpected interrupt handler

reset:  .word   _c_int00
int0:   .word   _c_int99
int1:   .word   _c_int99
int2:   .word   _c_int99
int3:   .word   _c_int99
xint0:  .word   _c_int99
rint0:  .word   _c_int06
xint1:  .word   _c_int99
rint1:  .word   _c_int08
tint0:  .word   _c_int99
tint1:  .word   _c_int99
dint:   .word   _c_int99
```

Example 8–11. C_int.asm

```
;************************************************************
;                 c_int.asm
;
;                 Leor Brenman
;
;                 03–16–92
;
;           (C) Texas Instruments Inc., 1992
;
;           Refer to the file 'license.txt' included with this
;           this package for usage and license information.
;************************************************************
*********************************************************
* C_INT08(VOID)
*
* Hand–coded assembly language interrupt service routine.
* This serial port transmitt ISR supports the CS4215 zero
* chip I/F to the C3x serial port
* This ISR has been hand–coded for speed optimization.
*
* Leor Brenman, DSP Applications
* (C) 1991 TEXAS INSTRUMENTS, HOUSTON
*********************************************************

        .globl  _c_int08

******************
* global variables
******************
        .global _first_half, _input_xfer0, _input_xfer1, _buffer_size
        .global _buffer_index, _output_xfer0
        .global _output_xfer1, _output0, _output1, _data_control
        .global _buffer_rdy, _input0, _input1

******************
* global variables
******************
        .data
SER_1   .word   808050h    ;place in same page as .bss
                           ;to eliminate push/pop of DP when loading
                           ;serial port one's base address

*************************************************
* FUNCTION DEF : _c_int08
*************************************************
        .text
_c_int08:
        PUSH    ST
        PUSH    R0
        PUSHF   R0
        PUSH    AR0
```

Example 8–11. C_int.asm (Continued)

```
*****************************************************************
* if this is the first half of the transmission then goto FRST_HALF
*****************************************************************

        LDI     @_first_half,R0
        BNZ     FRST_HALF

*********************************************
* else, this the second half of the transmission
*********************************************
SCND_HALF:

*************************************************************************
* load AR0 with serial port base address
* do dummy read of serial port to empty control info from serial port
*************************************************************************

        LDI     @SER_1,AR0
        LDI     *+AR0(12),R0

*************************************************************************
* get control value and write to serial port while branching to end of ISR
* and set first_half flag to TRUE
*************************************************************************

        LDI     @_data_control+1,R0
        BD      FIN_S
        STI     R0,*+AR0(8)
        LDI     1,R0
        STI     R0,@_first_half

*************************************
* This the second half of the transmission
*************************************
FRST_HALF:

************************
* push remaining registers
************************

        PUSH    R1
        PUSHF   R1
        PUSH    AR1
        PUSH    IR0
```

Example 8–11. C_int.asm (Continued)

```
*****************************
* set first_half flag to FALSE
*****************************

        LDI     0,R0
        STI     R0,@_first_half



                                                C_int.asm

POP     AR0
POPF    R0
POP     R0
POP     ST

RETI
```

Example 8–12. General.h

```
/*****************************************************************************/
/* general.h v4.2                                                            */
/* Copyright (c) 1991 Texas Instruments Incorporated                         */
/*****************************************************************************/
#ifndef _GENERAL
#define _GENERAL

/*****************************************************************************/
/* COMMON MACRO DEFINTIONS                                                   */
/*****************************************************************************/
#ifndef OFF
#define OFF                   0x00
#endif

#ifndef ON
#define ON                    0x01
#endif

#ifndef FALSE
#define FALSE                 0x00
#endif

#ifndef TRUE
#define TRUE                  0x01
#endif

#ifndef CLEAR
#define CLEAR                 0x00
#endif

#ifndef SET
#define SET                   0x01
#endif
```

Example 8–12. General.h (Continued)

```
/*************************************************************************/
/* GENERAL C3x MACROS                                                    */
/*************************************************************************/
#ifndef INIT_XF_PINS
#define INIT_XF_PINS          asm(" LDI  00h,IOF")
#endif

#ifndef CL_INT_FL_REG
#define CL_INT_FL_REG         asm(" LDI  0h,IF")
#endif

#ifndef EN_GLOBAL_INTS
#define EN_GLOBAL_INTS        asm(" OR   2000h,ST")
#endif

#ifndef EN_SER_PORT_XMT_INT_0
#define EN_SER_PORT_XMT_INT_0 asm(" OR 10h,IE")
#endif

#ifndef EN_SER_PORT_RCV_INT_0
#define EN_SER_PORT_RCV_INT_0 asm(" OR 20h,IE")
#endif

#ifndef EN_SER_PORT_XMT_INT_1
#define EN_SER_PORT_XMT_INT_1 asm(" OR 40h,IE")
#endif

#ifndef EN_SER_PORT_RCV_INT_1
#define EN_SER_PORT_RCV_INT_1 asm(" OR 80h,IE")
#endif

#ifndef ENABLE_CACHE
#define ENABLE_CACHE          asm(" OR 800h,ST")
#endif

#endif /* #ifndef _GENERAL */
```

Example 8–13. Commdrvr.h

```
/*************************************************************************/
/*   COMMDRVR.H                                                          */
/*                                                                       */
/*   TMS320C3x – COMMOM DRIVER HEADER FILE                               */
/*      :TMS320C3x CODE                                                  */
/*       Compile and archive into appropriate driver library             */
/*                                                                       */
/*   (C) 1991 TEXAS INSTRUMENTS, HOUSTON                                 */
/*************************************************************************/
#include <c30_per.h>

/*************************************************************************/
/* COMMON STRUCTURES                                                     */
/*************************************************************************/
typedef volatile int VI;
typedef volatile float VF;
typedef VF * volatile VPVF;
typedef VI * volatile VPVI;

/*************************************************************************/
/* FUNCTION PROTOTYPES                                                   */
/*************************************************************************/
void c_int99(void);
void heap_overflow(void);
void init_c30(void);
void error_in_real_time(void);
```

Example 8–14. Commdrvr.c

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

                commdrvr.c

                staff

                01–15–92

          (C) Texas Instruments Inc., 1992

          Refer to the file 'license.txt' included with this
          this package for usage and license information.

************************************************************/
/*****************************************************************************/
/*   COMMDRVR.C                                                              */
/*                                                                           */
/*   TMS320C3x – COMMOM DRIVER ROUTINES                                      */
/*      :TMS320C3x CODE                                                      */
/*       Compile and archive into aic.lib                                    */
/*                                                                           */
/*   (C) 1991 TEXAS INSTRUMENTS, HOUSTON                                     */
/*****************************************************************************/
#include <commdrvr.h>

/*****************************************************************************/
/* C_INT99(): ERRONEOUS INTERRUPT SERVICE ROUTINE                            */
/*            THIS ROUTINE IDLES AFTER RECEIVING AN UNEXPECTED INTERRUPT     */
/*****************************************************************************/
void c_int99(void)
{
    for(;;);
}
/*****************************************************************************/
/* HEAP_OVERFLOW(): NOT ENOUGH MEMORY IN THE HEAP                            */
/*                  THIS ROUTINE IS AN ERROR HANDLER FOR WHEN MEMORY         */
/*                  CANNOT BE ALLOCATED FROM THE HEAP                        */
/*****************************************************************************/
void heap_overflow(void)
{
    for(;;);
}

/*****************************************************************************/
/* INIT_C30(): INITIALIZE TMS320C30                */
/*****************************************************************************/
void init_c30(void)
```

Example 8–14. Commdrvr.c (Continued)

```
{
    BUS_ADDR->exp_gcontrol  = 0x0;
    BUS_ADDR->prim_gcontrol = 0x0;
    INIT_XF_PINS;
    ENABLE_CACHE;
}

/*************************************************************************/
/* ERROR_IN_REAL_TIME(): ERROR HANDLER, PROCESSING TIME IS GREATER   */
/*         I/O TIME.                    */
/*************************************************************************/
void error_in_real_time(void)
{
    for(;;);
}
```

Example 8–15. CS4215.h

```
/************************************************************************/
/*   CS4215.H                                                           */
/*                                                                      */
/*   TMS320C3x – CRYSTAL 4215 MM CODEC                                  */
/*      :TMS320C3x CODE                                                 */
/*                                                                      */
/*   Leor Brenman, DSP Applications                                     */
/*   (C) 1991 TEXAS INSTRUMENTS, HOUSTON                                */
/************************************************************************/
#include <math.h>
#include <stdlib.h>
#include <c30_per.h>
#include <commdrvr.h>

/*======================================================================*/
/* MACROS *=============================================================*/
/*======================================================================*/
#define BLOCK_SIZE       64
#define SER_NUM          SERIAL_PORT_ONE
#define TIMER_NUM        TIMER_ONE
#define XF_NUM           1

#define INIT_ARRAYS      init_arrays(buffer_size)
#define WAIT_BUFFERS     while(!buffer_rdy);
#define RESET_FLAGS      buffer_rdy = FALSE
#define RESET_CODEC      TIMER_ADDR(TIMER_NUM)->gcontrol = I_O | HLD_
#define UN_RESET_CODEC   TIMER_ADDR(TIMER_NUM)->gcontrol = I_O | HLD_ | DATOUT
#if XF_NUM
#define DCB_LOW          asm("   AND  2fh,IOF"); asm("  OR 20h,IOF")
#define DCB_HI           asm("   OR   60h,IOF")
#else
#define DCB_LOW          asm("   AND 0F2h,IOF"); asm("  OR 2h,IOF")
#define DCB_HI           asm("   OR   6h,IOF")
#endif

#define WAIT(A)          for(i=0;i<A;i++);

#define C_ISR            ON
```

Example 8–15. CS4215.h (Continued)

```
/*************************************************************************/
/*         CS4215 DATA COMMAND BIT FIELD DATA STRUCTURES                 */
/*************************************************************************/
/*************************************************************************/
/*   CONTROL COMMAND                                                     */
/*************************************************************************/
typedef union
{
  unsigned int _intval[2];
  struct
  {
    /* Time slot 4 */
    unsigned int adl     :1;     /* Loopback mode                        */
    unsigned int enl     :1;     /* Enable loopback testing              */
    unsigned int d_r5    :6;     /* Unused – don't care bits: 2 – 7      */

    /* Time slot 3 */
    unsigned int xen     :1;     /* Transmitter enable                   */
    unsigned int xclk    :1;     /* Transmit clock                       */
    unsigned int bsel    :2;     /* Select bit rate                      */
    unsigned int mckf    :2;     /* Clock source select                  */
    unsigned int d_r4    :2;     /* Unused – don't care bits: 6 – 7      */

    /* Time slot 2 */
    unsigned int df      :2;     /* Data format selection                */
    unsigned int st      :1;     /* Stereo bit: 0–mono, 1–stereo         */
    unsigned int dfr     :3;     /* Data conversion freq selection       */
    unsigned int d_r3    :2;     /* Unused – don't care bits: 6 – 7      */

    /* Time slot 1 */
    unsigned int d_r1    :2;     /* Unused – don't cares bits: 0 – 1     */
    unsigned int dcb     :1;     /* Data control handshake bit           */
    unsigned int d_r2    :5;     /* Unused – don't cares bits: 3 – 7     */

    /* Time slot 8 */
    unsigned int d_r9    :8;     /* Unused – don't care bits: 0 – 7      */

    /* Time slot 7 */
    unsigned int rv      :4;     /* Revision level of the CS4215         */
    unsigned int d_r8    :4;     /* Unused – don't care bits: 4 – 7      */

    /* Time slot 6 */
    unsigned int d_r7    :8;     /* Unused – don't care bits: 0 – 7      */

    /* Time slot 5 */
    unsigned int d_r6    :6;     /* Unused – don't care bits: 0 – 5      */
    unsigned int pio     :2;     /* Parallel port control                */
  } _bitval;
} CONTROL;
```

Example 8–15. CS4215.h (Continued)

```
/*************************************************************************/
/*   DATA COMMANDS                                                       */
/*************************************************************************/
typedef union
{
  unsigned int _intval[2];
  struct
  {
    /* Time slots 3 & 4 */
      signed int right   :16;    /* Right channel 16 bit                */

    /* Time slots 1 & 2 */
      signed int left    :16;    /* Left channel 16 bit                 */

    /* Time slot 8 */
    unsigned int rg      :4;     /* Right input gain settings           */
    unsigned int ma      :4;     /* Monitor path selection              */

    /* Time slot 7 */
    unsigned int lg      :4;     /* Left input gain settings            */
    unsigned int is      :1;     /* Input selection                     */
    unsigned int ovr     :1;     /* Overange                            */
    unsigned int pio     :2;     /* Parallel I/O bits                   */

    /* Time slot 6 */
    unsigned int ro      :6;     /* Right output attenuation setting    */
    unsigned int se      :1;     /* Speaker output enable control       */
    unsigned int d_r1    :1;     /* Unused – don't care bit 7           */

    /* Time slot 5 */
    unsigned int lo      :6;     /* Left output attenuation setting     */
    unsigned int le      :1;     /* Parallel output enable control      */
    unsigned int he      :1;     /* Headphone output enable control     */
  } _bitval;
} STEREO_16;

typedef union
{
  unsigned int _intval[2];
  struct
  {
    /* Time slots 3 & 4 */
      signed int d_r1    :16;    /* Unused – don't care bits 0 – 15     */

    /* Time slots 1 & 2 */
      signed int left    :16;    /* Left channel 16 bit                 */

    /* Time slot 8 */
    unsigned int d_r3    :4;     /* Unused – don't care bits: 0 – 3     */
    unsigned int ma      :4;     /* Monitor path selection              */
```

Example 8–15. CS4215.h (Continued)

```
    /* Time slot 7 */
    unsigned int lg      :4;     /* Left input gain settings             */
    unsigned int is      :1;     /* Input selection                      */
    unsigned int ovr     :1;     /* Overange                             */
    unsigned int pio     :2;     /* Parallel I/O bits                    */

    /* Time slot 6 */
    unsigned int ro      :6;     /* Right output attenuation setting     */
    unsigned int se      :1;     /* Speaker output enable control        */
    unsigned int d_r2    :1;     /* Unused – don't care bit 7            */

    /* Time slot 5 */
    unsigned int lo      :6;     /* Left output attenuation setting      */
    unsigned int le      :1;     /* Parallel output enable control       */
    unsigned int he      :1;     /* Headphone output enable control      */
  } _bitval;
} MONO_16;

typedef union
{
  unsigned int _intval[2];
  struct
  {
    /* Time slots 4 */
      signed int d_r2    :8;     /* Unused – don't care bits 0 – 7       */

    /* Time slot 3 */
      signed int right   :8;     /* Right channel 8 bit                  */

    /* Time slots 2 */
      signed int d_r1    :8;     /* Unused – don't care bits 0 – 7       */

    /* Time slot 1 */
      signed int left    :8;     /* Left channel 8 bit                   */

    /* Time slot 8 */
    unsigned int rg      :4;     /* Right input gain settings            */
    unsigned int ma      :4;     /* Monitor path selection               */

    /* Time slot 7 */
    unsigned int lg      :4;     /* Left input gain settings             */
    unsigned int is      :1;     /* Input selection                      */
    unsigned int ovr     :1;     /* Overange                             */
    unsigned int pio     :2;     /* Parallel I/O bits                    */

    /* Time slot 6 */
    unsigned int ro      :6;     /* Right output attenuation setting     */
    unsigned int se      :1;     /* Speaker output enable control        */
    unsigned int d_r3    :1;     /* Unused – don't care bit 7            */
```

Example 8–15. CS4215.h (Continued)

```
    /* Time slot 5 */
    unsigned int lo      :6;     /* Left output attenuation setting       */
    unsigned int le      :1;     /* Parallel output enable control        */
    unsigned int he      :1;     /* Headphone output enable control       */
  } _bitval;
} STEREO_8;

typedef union
{
  unsigned int _intval[2];
  struct
  {
    /* Time slots 2 – 4 */
      signed int d_r1    :24;    /* Unused – don't care bits 0 – 23       */

    /* Time slot 1 */
      signed int left    :8;     /* Left channel 8 bit                    */

    /* Time slot 8 */
    unsigned int d_r3    :4;     /* Unused – don't care bits: 0 – 3       */
    unsigned int ma      :4;     /* Monitor path selection                */

    /* Time slot 7 */
    unsigned int lg      :4;     /* Left input gain settings              */
    unsigned int is      :1;     /* Input selection                       */
    unsigned int ovr     :1;     /* Overange                              */
    unsigned int pio     :2;     /* Parallel I/O bits                     */

    /* Time slot 6 */
    unsigned int ro      :6;     /* Right output attenuation setting      */
    unsigned int se      :1;     /* Speaker output enable control         */
    unsigned int d_r2    :1;     /* Unused – don't care bit 7             */

    /* Time slot 5 */
    unsigned int lo      :6;     /* Left output attenuation setting       */
    unsigned int le      :1;     /* Parallel output enable control        */
    unsigned int he      :1;     /* Headphone output enable control       */
  } _bitval;
} MONO_8;

typedef union
{
  unsigned int     _intval[2];
  CONTROL           control;
  STEREO_16         stereo_16;
  MONO_16           mono_16;
  STEREO_8          stereo_8;
  MONO_8            mono_8;
} CS4215_WORD;
```

Example 8–15. CS4215.h (Continued)

```
/*=========================================================================*/
/* GLOBAL VARIABLES *======================================================*/
/*=========================================================================*/
extern int  buffer_size;      /* SIZE OF I/O BUFFER(S)                     */
extern VPVF output0;          /* OUTPUT DATA BUFFER FOR PROCESSOR          */
extern VPVF input0;           /* INPUT DATA BUFFER FOR PROCESSOR           */
extern VPVF output_xfer0;     /* OUTPUT DATA BUFFER FOR ISR/AIC            */
extern VPVF input_xfer0;      /* INPUT DATA BUFFER FOR ISR/AIC             */
extern VPVF output1;          /* OUTPUT DATA BUFFER FOR PROCESSOR          */
extern VPVF input1;           /* INPUT DATA BUFFER FOR PROCESSOR           */
extern VPVF output_xfer1;     /* OUTPUT DATA BUFFER FOR ISR/AIC            */
extern VPVF input_xfer1;      /* INPUT DATA BUFFER FOR ISR/AIC             */
extern VI   buffer_rdy;       /* CPU–ISR COMM FLAG (INPUT)                 */
extern VI   buffer_index;     /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS */
extern VI   i;                /* GENERIC COUNTER VARIABLE                   */
extern VI   first_half;

extern CS4215_WORD     data_control;

/*************************************************************************/
/*      FUNCTION PROTOTYPES                                              */
/*************************************************************************/
/*************************/
/* CS4215 DRIVER FUNCTIONS */
/*************************/
void init_arrays(int buffer_size);
void init_4215(int crystal, int sample_rate);
#if SER_NUM
void c_int07(void);
#else
void c_int05(void);
#endif

/*************************************************************************/
/*         CS4215 DATA COMMAND BIT FIELD MACROS                          */
/*************************************************************************/
/*************************************************************************/
/*   CONTROL COMMAND MACROS                                              */
/*************************************************************************/
#define DATA               1
#define COMM               0
#define SIXTEEN_BIT_LINEAR 0
#define EIGHT_BIT_U_LAW    1
#define EIGHT_BIT_A_LAW    2
#define MONO_MODE          0
#define STEREO_MODE        1
```

Example 8–15. CS4215.h (Continued)

```
/* Data conversion Frequency Selections Assumes that XTAL1 = 24.576 MHz   */
/* And XTAL2 = 16.9344 MHz.                                               */
/*                                          XTAL1 (kHz) | XTAL2 (kHz)   */
/*                                          ========================== */
#define CONV_FREQ_0        0           /*         8.00000 |  5.5125       */
#define CONV_FREQ_1        1           /*        16.00000 | 11.0250       */
#define CONV_FREQ_2        2           /*        27.42857 | 18.9000       */
#define CONV_FREQ_3        3           /*        32.00000 | 22.0500       */
#define CONV_FREQ_4        4           /*        NA       | 37.8000       */
#define CONV_FREQ_5        5           /*        NA       | 44.1000       */
#define CONV_FREQ_6        6           /*        48.00000 | 33.0750       */
#define CONV_FREQ_7        7           /*         9.60000 |  6.6150       */

#define CS_ENABLE          0                    /* Data output enabled */
#define CS_DISABLE         1                    /* Data output disabled*/

#define CS_TCLOCK_EXT      0                  /* FSYNC and SCLK are inputs*/
#define CS_TCLOCK_INT      1                 /* FSYNC and SCLK are outputs*/


#define BPF_64             0                      /* 64 bits per frame */
#define BPF_128            1                     /* 128 bits per frame */
#define BPF_256            2                     /* 256 bits per frame */

#define CS_CLOCK_SCLK      0                 /* Clock source select: SCLK */
#define CS_CLOCK_XTAL1     1                 /* Clock source select: XTAL1*/
#define CS_CLOCK_XTAL2     2                 /* Clock source select: XTAL2*/
#define CS_CLOCK_EXT       3                 /* Clock source select: Ext  */

#define DIGITAL_LOOPBACK   0
#define ANALOG_LOOPBACK    1

#define LOOP_ENABLE        1
#define LOOP_DISABLE       0

/************************************************************************/
/*   DATA COMMAND MACROS                                                */
/************************************************************************/
/* Output attenuation is 1.5 dB per unit integer value                  */
/*                                        Attenuation (dB)              */
/*                                        ================              */
#define ATT_0              0           /*      0.0                      */
#define ATT_1              1           /*      1.5                      */
#define ATT_2              2           /*      3.0                      */
#define ATT_3              3           /*      4.5                      */
#define ATT_4              4           /*      6.0                      */
#define ATT_5              5           /*      7.5                      */
#define ATT_6              6           /*      9.0                      */
#define ATT_7              7           /*     10.5                      */
#define ATT_8              8           /*     12.0                      */
#define ATT_9              9           /*     13.5                      */
#define ATT_10             10          /*     15.0                      */
```

Example 8–15. CS4215.h (Continued)

```
#define ATT_11              11          /*     16.5                        */
#define ATT_12              12          /*     18.0                        */
#define ATT_13              13          /*     19.5                        */
#define ATT_14              14          /*     21.0                        */
#define ATT_15              15          /*     22.5                        */
#define ATT_16              16          /*     24.0                        */
#define ATT_17              17          /*     25.5                        */
#define ATT_18              18          /*     27.0                        */
#define ATT_19              19          /*     28.5                        */
#define ATT_20              20          /*     30.0                        */
#define ATT_21              21          /*     31.5                        */
#define ATT_22              22          /*     33.0                        */
#define ATT_23              23          /*     34.5                        */
#define ATT_24              24          /*     36.0                        */
#define ATT_25              25          /*     37.5                        */
#define ATT_26              26          /*     39.0                        */
#define ATT_27              27          /*     40.5                        */
#define ATT_28              28          /*     42.0                        */
#define ATT_29              29          /*     43.5                        */
#define ATT_30              30          /*     45.0                        */
#define ATT_31              31          /*     46.5                        */
#define ATT_32              32          /*     48.0                        */
#define ATT_33              33          /*     49.5                        */
#define ATT_34              34          /*     51.0                        */
#define ATT_35              35          /*     52.5                        */
#define ATT_36              36          /*     54.0                        */
#define ATT_37              37          /*     55.5                        */
#define ATT_38              38          /*     57.0                        */
#define ATT_39              39          /*     58.5                        */
#define ATT_40              40          /*     60.0                        */
#define ATT_41              41          /*     61.5                        */
#define ATT_42              42          /*     63.0                        */
#define ATT_43              43          /*     64.5                        */
#define ATT_44              44          /*     66.0                        */
#define ATT_45              45          /*     67.5                        */
#define ATT_46              46          /*     69.0                        */
#define ATT_47              47          /*     70.5                        */
#define ATT_48              48          /*     72.0                        */
#define ATT_49              49          /*     73.5                        */
#define ATT_50              50          /*     74.0                        */
#define ATT_51              51          /*     75.5                        */
#define ATT_52              52          /*     77.0                        */
#define ATT_53              53          /*     78.5                        */
#define ATT_54              54          /*     80.0                        */
#define ATT_55              55          /*     81.5                        */
#define ATT_56              56          /*     83.0                        */
#define ATT_57              57          /*     84.5                        */
#define ATT_58              58          /*     87.0                        */
#define ATT_59              59          /*     88.5                        */
#define ATT_60              60          /*     90.0                        */
#define ATT_61              61          /*     91.5                        */
```

Example 8–15. CS4215.h (Continued)

```
#define ATT_62              62          /*     93.0                         */
#define ATT_63              63          /*     94.5                         */

#define HEADPHONE_OFF       0
#define HEADPHONE_ON        1

#define LINE_OUT_OFF        0
#define LINE_OUT_ON         1

#define SPEAKER_OFF         0
#define SPEAKER_ON          1

/* Input gain is 1.5 dB per unit integer value                              */
/*                                        Gain (dB)                         */
/*                                        =========                         */
#define GAIN_0              0           /*      0.0                         */
#define GAIN_1              1           /*      1.5                         */
#define GAIN_2              2           /*      3.0                         */
#define GAIN_3              3           /*      4.5                         */
#define GAIN_4              4           /*      6.0                         */
#define GAIN_5              5           /*      7.5                         */
#define GAIN_6              6           /*      9.0                         */
#define GAIN_7              7           /*     10.5                         */
#define GAIN_8              8           /*     12.0                         */
#define GAIN_9              9           /*     13.5                         */
#define GAIN_10             10          /*     15.0                         */
#define GAIN_11             11          /*     16.5                         */
#define GAIN_12             12          /*     18.0                         */
#define GAIN_13             13          /*     19.5                         */
#define GAIN_14             14          /*     21.0                         */
#define GAIN_15             15          /*     22.5                         */

#define LINE_IN             0
#define MIKE_IN             1

#define OVERANGE_ENABLE     1
#define OVERANGE_CLEAR      0

/* Monitor path attenuation = 6 dB per unit integer value                   */
/*                                        Gain (dB)                         */
/*                                        =========                         */
#define MATT_0              0           /*      6.0                         */
#define MATT_1              1           /*     12.0                         */
#define MATT_2              2           /*     18.0                         */
#define MATT_3              3           /*     24.0                         */
#define MATT_4              4           /*     30.0                         */
#define MATT_5              5           /*     36.0                         */
#define MATT_6              6           /*     42.0                         */
#define MATT_7              7           /*     48.0                         */
#define MATT_8              8           /*     54.0                         */
#define MATT_9              9           /*     60.0                         */
#define MATT_10             10          /*     66.0                         */
```

Example 8–15. CS4215.h (Continued)

```
#define MATT_11             11          /*     72.0                        */
#define MATT_12             12          /*     78.0                        */
#define MATT_13             13          /*     84.0                        */
#define MATT_14             14          /*     90.0                        */
#define MATT_15             15          /*     96.0  (Mute Monitor Path)   */
```

Example 8–16. CS4215.c

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

                cs4215.c

                staff

                05–13–92

          (C) Texas Instruments Inc., 1992

          Refer to the file 'license.txt' included with this
          this package for usage and license information.

************************************************************/
/*************************************************************************/
/*   CS4215.C                                                            */
/*                                                                       */
/*   TMS320C3x – CRYSTAL 4215 MM CODEC                                   */
/*      :TMS320C3x CODE                                                  */
/*       Compile and archive into CS4215.lib                             */
/*                                                                       */
/*   Leor Brenman, DSP Applications                                      */
/*   (C) 1991 TEXAS INSTRUMENTS, HOUSTON                                 */
/*************************************************************************/
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <cs4215.h>

/*************************************************************************/
/* GLOBAL VARIABLES                                                      */
/*************************************************************************/
int  buffer_size = BLOCK_SIZE; /* SIZE OF I/O BUFFER(S)                  */
VPVF output0;                  /* OUTPUT DATA BUFFER FOR PROCESSOR       */
VPVF input0;                   /* INPUT DATA BUFFER FOR PROCESSOR        */
VPVF output_xfer0;             /* OUTPUT DATA BUFFER FOR ISR/CODEC       */
VPVF input_xfer0;              /* INPUT DATA BUFFER FOR ISR/CEDEC        */
VPVF output1;                  /* OUTPUT DATA BUFFER FOR PROCESSOR       */
VPVF input1;                   /* INPUT DATA BUFFER FOR PROCESSOR        */
VPVF output_xfer1;             /* OUTPUT DATA BUFFER FOR ISR/CEDEC       */
VPVF input_xfer1;              /* INPUT DATA BUFFER FOR ISR/CODEC        */
VI   buffer_rdy     = FALSE;   /* CPU–ISR COMM FLAG (INPUT)              */
VI   buffer_index   = 0;       /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS */
VI   first_half = TRUE;
VI   i;                        /* GENERIC COUNTER VARIABLE               */

CS4215_WORD     data_control;

#if C_ISR
```

Example 8–16. CS4215.c (Continued)

```
/*************************************************************************/
/* C_INT06() OR C_INT08()                                                */
/*        SERIAL PORT 0/1 RECEIVE INTERRUPT SERVICE ROUTINE              */
/*************************************************************************/
#if SER_NUM
void c_int06(void) {}
void c_int08(void)
#else
void c_int08(void) {}
void c_int06(void)
#endif
{
    VPVF swap;
    CS4215_WORD     in,out;

    if(first_half) /* First half of the 64 bit transmission */
    {
       first_half = FALSE;

       in._intval[0] = SERIAL_PORT_ADDR(SER_NUM)->r_data;
       input_xfer0[buffer_index] = in.stereo_16._bitval.right;
       input_xfer1[buffer_index] = in.stereo_16._bitval.left;

       out.stereo_16._bitval.left    = output_xfer1[buffer_index];
       out.stereo_16._bitval.right   = output_xfer0[buffer_index];
       SERIAL_PORT_ADDR(SER_NUM)->x_data  = out._intval[0];

       if(++buffer_index == buffer_size)
       {
           swap          = input0;
           input0        = input_xfer0;
           input_xfer0   = swap;

           swap          = input1;
           input1        = input_xfer1;
           input_xfer1   = swap;

           swap           = output0;
           output0        = output_xfer0;
           output_xfer0   = swap;

           swap           = output1;
           output1        = output_xfer1;
           output_xfer1   = swap;

           buffer_index   = 0;
           buffer_rdy     = TRUE;
       }
    }
```

Example 8–16. CS4215.c (Continued)

```
    else /* Second half of transmission */
    {
       SERIAL_PORT_ADDR(SER_NUM)->r_data;
       SERIAL_PORT_ADDR(SER_NUM)->x_data  = data_control._intval[1];
       first_half = TRUE;
    }
}
#endif /* C_ISR */

/*=========================================================================*/
/* INIT_ARRAYS(): INITIALIZE DATA ARRAY PARAMETERS                         */
/*=========================================================================*/

void init_arrays(int buffer_size)
{
     int i;
     /*-----------------------------------------------------------------------*/
     /* INITIALIZE AND ZERO FILL ARRAYS                                       */
     /*-----------------------------------------------------------------  */
     if(!(input0 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(output0 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(input_xfer0 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(output_xfer0 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(input1 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(output1 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(input_xfer1 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();
     if(!(output_xfer1 = (float *) calloc(buffer_size,sizeof(float))))
        heap_overflow();

     for(i = 0; i < buffer_size; i++)
     {
        output0[i] = output_xfer0[i] = 0.0;
        output1[i] = output_xfer1[i] = 0.0;
     }
}
```

Example 8–16. CS4215.c (Continued)

```
/*************************************************************************/
/* INIT_4215(): INITIALIZE COMMUNICATIONS TO CS4215                      */
/*             NOTE: i IS A VOLATILE TO FORCE TIME DELAYS AND TO FORCE    */
/*                  READS OF SERIAL PORT DATA RECEIVE REGISTER TO CLEAR   */
/*                  THE RECEIVE INTERRUPT FLAG                            */
/*************************************************************************/
void init_4215(int crystal, int sample_rate)
{
    VI i,j,dummy;
    CS4215_WORD     temp,in,out;

    RESET_CODEC;                                          /* RESET AIC */
    WAIT(50);                  /* KEEP RESET LOW FOR SOME PERIOD OF TIME */

    /*********************************************************************/
    /* CONFIGURE SERIAL PORT 1                                           */
    /*********************************************************************/
    SERIAL_PORT_ADDR(SER_NUM)->gcontrol = 0x0;

    SERIAL_PORT_ADDR(SER_NUM)->s_x_control = CLKXFUNC | DXFUNC | FSXFUNC;
    SERIAL_PORT_ADDR(SER_NUM)->s_r_control = CLKRFUNC | DRFUNC | FSRFUNC;

    SERIAL_PORT_ADDR(SER_NUM)->s_rxt_control = XGO | XHLD_ | XCP_ | XCLKSRC;

    /* THE FOLLOWING PERIOD REGISTER VALUE HAS BEEN TESTED ON A 50 MHz C30 */

    SERIAL_PORT_ADDR(SER_NUM)->s_rxt_period_bit.x_period = 0x3;

    SERIAL_PORT_ADDR(SER_NUM)->gcontrol = XCLKSRCE | XLEN_32 | XFSM | RFSM |
                                          RLEN_32 | XINT | RINT |
                                          FSXOUT | RRESET | XRESET;


    /* BUILD CONTROL WORDS */
    /* ALL BITS ARE 0 EXCEPT THOSE DEFINED OTHERWISE */

    temp._intval[0] = temp._intval[1] = 0;
    temp.control._bitval.st   = STEREO_MODE;
    temp.control._bitval.dfr  = sample_rate;
    temp.control._bitval.xclk = 1;
    temp.control._bitval.mckf = crystal;
    temp.control._bitval.pio  = 3;

    /* BUILD DATA CONTROL WORD */
    data_control._intval[0] = data_control._intval[1] = 0;
    data_control.stereo_16._bitval.lo   = ATT_0;
    data_control.stereo_16._bitval.le   = ON;
    data_control.stereo_16._bitval.ro   = ATT_0;
    data_control.stereo_16._bitval.ovr  = ON;
    data_control.stereo_16._bitval.ma   = MATT_15;
```

Example 8–16. CS4215.c (Continued)

```
    UN_RESET_CODEC;                        /* PULL 4215 OUT OF RESET        */


    DCB_LOW;
    /* Write out control word until dcb bit is low */
    do
    {
       out = temp;
       for(i=0;i<5;i++)
       {
          while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.xsrempty == 1);

          SERIAL_PORT_ADDR(SER_NUM)->gcontrol = 0x0;

          /* See note on XRESET/RRESET and three cycle delay in C3x U.G. */
          for(j=0;j<3;j++);

          SERIAL_PORT_ADDR(SER_NUM)->gcontrol = XCLKSRCE | XLEN_32 | XFSM |
                                                RFSM | RLEN_32 | XINT | RINT |
                                                FSXOUT | RRESET | XRESET;

          dummy = SERIAL_PORT_ADDR(SER_NUM)->r_data;

          SERIAL_PORT_ADDR(SER_NUM)->x_data = out._intval[0];

          /* See note on XRDY and three cycle delay in C3x U.G. */
          for(j=0;j<3;j++);

          while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.xrdy == 0);

          SERIAL_PORT_ADDR(SER_NUM)->x_data = out._intval[1];




          while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.rrdy == 0);

          in._intval[0] = SERIAL_PORT_ADDR(SER_NUM)->r_data;

          /* See note on RRDY and three cycle delay in C3x U.G. */
          for(j=0;j<3;j++);

          while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.rrdy == 0);

          in._intval[1] = SERIAL_PORT_ADDR(SER_NUM)->r_data;
      }
    } while(in.control._bitval.dcb != 0);
```

Example 8–16. CS4215.c (Continued)

```
    /* Write out control word twice with the dcb bit high */
    temp.control._bitval.dcb  = 1;
    out = temp;
    for(i=0;i<2;i++)
    {
       while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.xsrempty == 1);

       SERIAL_PORT_ADDR(SER_NUM)->gcontrol = 0x0;

       /* See note on XRESET/RRESET and three cycle delay in C3x U.G. */
       for(j=0;j<3;j++);

       SERIAL_PORT_ADDR(SER_NUM)->gcontrol = XCLKSRCE | XLEN_32 | XFSM |
                                             RFSM | RLEN_32 | XINT | RINT |
                                             FSXOUT | RRESET | XRESET;

       dummy = SERIAL_PORT_ADDR(SER_NUM)->r_data;

       SERIAL_PORT_ADDR(SER_NUM)->x_data = out._intval[0];

       /* See note on XRDY and three cycle delay in C3x U.G. */
       for(j=0;j<3;j++);

       while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.xrdy == 0);

       SERIAL_PORT_ADDR(SER_NUM)->x_data = out._intval[1];



       while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.rrdy == 0);

       in._intval[0] = SERIAL_PORT_ADDR(SER_NUM)->r_data;

       /* See note on RRDY and three cycle delay in C3x U.G. */
       for(j=0;j<3;j++);

       while(SERIAL_PORT_ADDR(SER_NUM)->gcontrol_bit.rrdy == 0);

       in._intval[1] = SERIAL_PORT_ADDR(SER_NUM)->r_data;
    }

    SERIAL_PORT_ADDR(SER_NUM)->gcontrol = 0x0;
    SERIAL_PORT_ADDR(SER_NUM)->gcontrol = XLEN_32 | RLEN_32 | XFSM | RFSM |
                                          RRESET | XRESET | XCLKSRCE;
```

Example 8–16. CS4215.c (Continued)

```
    while(SERIAL_PORT_ADDR(SER_NUM)–>gcontrol_bit.xrdy == 0);
    SERIAL_PORT_ADDR(SER_NUM)–>x_data = 0;
    /* See note on XRDY and three cycle delay in C3x U.G. */
    for(j=0;j<3;j++);

    while(SERIAL_PORT_ADDR(SER_NUM)–>gcontrol_bit.xrdy == 0);

    SERIAL_PORT_ADDR(SER_NUM)–>x_data = data_control._intval[1];

    dummy = SERIAL_PORT_ADDR(SER_NUM)–>r_data;

    SERIAL_PORT_ADDR(SER_NUM)–>gcontrol |= XINT | RINT;

    SERIAL_PORT_ADDR(SER_NUM)–>gcontrol &= ~XCLKSRCE;

    SERIAL_PORT_ADDR(SER_NUM)–>s_rxt_control = 0;

    CL_INT_FL_REG;

#if SER_NUM
    EN_SER_PORT_RCV_INT_1;
#else
    EN_SER_PORT_RCV_INT_0;
#endif

    EN_GLOBAL_INTS;

    DCB_HI;
}
```

8.7 Software UART Emulator for the TMS320C3x

By using the general-purpose I/O pins in conjunction with two timers and an external interrupt, you can develop a very flexible full-duplex universal asynchronous receive transmit (UART) emulator in software. This solution discusses the implementation of an interrupt-driven, 9 600-baud UART with eight data bits, one stop bit, and no parity. This solution was contributed by Ted Fried of Advanced Computer Communications.

8.7.1 Hardware

The hardware interface is relatively straightforward (see Figure 8–12). The receive line is connected to both the INT0 and IOF1 pins. This triggers an interrupt on the falling edge of the start bit. The transmit line is connected to the IOF0 pin and a pullup resistor.

8.7.2 Software

As shown in Example 8–17, the receive sequence begins when the start bit triggers the external interrupt. At the interrupt service routine, R_XINT0, timer0 is loaded with a value that results in a delay of one half of the bit time. The routine then loads the timer's interrupt vector, enables it, then exits to the main program. When the timer triggers its interrupt, R_X-TMR-INT, the main body of the receive code executes. At this time, the line is in the middle of the start bit. The CPU then samples IOF1 and verifies that the start bit has been read in. If the start bit is verified, the timer is then loaded with the full-bit time and started. The procedure then exits to the main program.

On successive timer0 interrupts, R_XINT0, the received bits are shifted into a storage area in memory until a byte is read in. On the ninth interrupt, if the stop bit is verified, the routine executes a software trap to inform the main program of the byte reception. If the stop bit is not verified, the BAD_STOP_BIT subroutine is called where the appropriate action is taken. After the received byte is processed, the external interrupt is then reenabled and the system waits for the next start bit.

The transmit routine begins when the main program loads a byte into the holding register and then calls TX_MAIN. This procedure loads timer1 with the full-bit time value, resets the transmit counter, sets the start bit, and enables the timer's interrupt. The routine then exits back to the main program. The main program does not call for another byte transmit until it finds the transmit counter equal to 0. On each subsequent timer1 interrupt, T_X-INT, the routine shifts out the transmit byte including the stop bit, until the transmit counter is 0.

Example 8–17. Full Duplex UART Emulator for TMS320C3x

```
half_bit_time       set 01ADh         ; assume 33–MHz TMS320C3x
whole_bit_time      set 0358h
timer_go            set 03Clh
timer_setup         set O?Dlh
int_setup           sec O301h
iof_setup           set 06h

timer0_vector       .word RX_TMR_INT  ; interrupt vector addresses
timerl_vector       .word TX_TNT
rx_int_vector       .word RX_INTO
timer0_period       .word 0808028h    ; on–chip RAM locations
timerl_period       .word 0808038h
timer0_control      .word 0808020h
timerl_control      .word 0808030h
timer0_int_vect     .word 0809FC9h
timerl_int_vect     .word 0809FCAh
intO_vector         .word 0809FC1h
rx_byte             .word 0809FF8h
tx_byte             .word 0809FF9h
rx_counter          .word 0809FFAh
tx_counter          .word 0809FFBh

; Main setup for asynchronous serial interface to be run at
  powerup.

SETUP_ASYNCH:     PUSH      AR7
     OR    iof_setup, IOF             ; iof seetup and iof0=1
     LDI   timer_setup, AR7           ; setup timer0 and timer1
     STI   AR7, @timer0_control       ;
     STI   AR7, @timer1_control       ;
     LDI   rx_int_vector, AR7         ; load int0 interrupt vector
     STI   AR7, @int0_vector          ;
     OR    int_setup, IE              ; enable interrupts
     POP   AR7
     RETS

; Start bit received. external interrupt service routine

RX_INT0:   PUSH   AR7
     XOR   01h,   Ie                  ; disable int0
     LDI   half_bit_time, AR7         ;
     STI   AR7, @timer0_period        ; rx_timer period
     LDI   timer0_vector, AR7         ;
     STI   AR7, @timer0_int_vect      ; rx_timer int vector
     LDI   timer_go, AR7              ;
     STI   AR7, @timer0_control       ; start rx_timer
     LDI   0Ah, AR7                   ;
     STI   AR7, @rx_counter           ; reset rx_counter
     POP   AR7
     RETI
```

Example 8–17. Full Duplex UART Emulator for TMS320C3x (Continued)

```
; Timer0 interrupt service routine for byte reception.

RX_TMR_INT:    PUSH   AR7
       LDI     @rx_counter, AR7
       CMPI    09h, AR7                ; are we at start bit?
       BNE     STOP                    ; nope, check for stop bit
       CMPI    080h, IOF               ; check rx_bit (IOF1)
       BLT     OK                      ; if less than 80h (IOF1=0)?
       OR      01h, IE                 ; bad start bit, reenable
INT0
       BR      CLEANUP2                ; go back to main
OK:    SUBI    01h, AR7                ; decrement rx_counter
       STI     AR7, @rx_counter        ; update counter in memory
       LDI     whole_bit_time, AR7     ;
       STI     AR7 @timer0_period      ; load bit time into rx_timer
       LDI     timer_go, AR7           ;
       STI     AR7, @timer0_crontrol   ; start rx_timer
       POP     AR7
       RETI
STOP:  PUSH    AR6
       LDI     @rx_byte, AR6
       DBNZ    AR7, NEXT               ; if rx_count !=0, get next bit
       CMPI    080h, IOF               ; check rx_bit (IOF1)
       BLT     BAD_STOP_BIT            ; GO TO INVALID STOP BIT MODULE
       LSH     –24, AR6                ; shift rx_byte 24 bits right
       STI     AR6, @rx_byte           ; TRAP RECEIVED BYTE!!
       OR      0lh, IE                 ; reenable INT0\
       BR      CLEANUP                 ;
NEXT:  CMPI    080h. IOF               ; check rx_bit (IOF1)
       OR      01h, ST                 ; force carry flag to 1
       BGE     ONE                     ; if rx_bit = 1
       XOR     01h, ST                 ; set carry flag to 0
ONE:   RORC    AR6                     ; shift in carry bit
       STI     AR6, @rx_byte           ; update rx_byte in memory
       STI     AR7, @rx_counter        ; update counter in memory
       LDI     timer_go, AR6           ;
       STI     AR6, @timer0_control    ; start rx_timer
CLEANUP:       POP    AR6
CLEANUP2:      POP    AR7     RETI
```

Example 8–17. Full Duplex UART Emulator for TMS320C3x (Continued)

```
; Transmit byte main subroutine

   TX_MAIN:   PUSH   AR7
         LDI  whole_bit_time,  AR7
         STI  AR7. @timer1_period         ; load timer period
         LDI  timer1_vector, AR7          ;
         STI  AR7, @timer1_int_vect       ; tx_timer int vector
         LDI  @tx_byte, AR7               ;
         OR   0FF00h, AR7                 ; mask stop bit to tx_byte
         STI  AR7, @tx_byte               ; update tx_byte
         AND  0FBh, IOF                   ; send out '0' to IOF0
         LDI  0Ah, AR7                    ;
         STI  AR7, @tx_counter            ; load counter in memory
         LDI  timer_go, AR7               ;
         STI  AR7, @timer1_control        ; start tx_timer
              POP  AR7
              RETS

      ; Timer1 interrupt service routine for byte transmission.

        TX_INT:    PUSH     AR7
              LDI  @tx_counter, AR7        ; load in tx_counter from mem
              DBNZ AR7, NEXT_OUT           ; if tx_counter not zero
              POP  AR7
              RETI
        NEXT OUT:  PUSH     AR6
              LDI  timer_go, AR7
              STI  AR7, @timer1_control    ; start tx_timer
              LDI  tx_byte, AR6            ; load in tx_byte from mem
             RORC  AR6                     ; next bit out is in carry
              BNC  OUT ZERO                ; carry=0. then send out '0'
              OR   04h, IOF                ; send out '1' to IOF0
              BR   CLEANUP3                ;
        OUT ZERO:  AND      0FBh, IOF      ; send out '0' to IOF0
        CLEANUP3:  STI      AR6, @tx_byte  ; update byte in memory
              STI  AR7, @tx_counter        ; update counter in memory
              POP  AR6
              POP  AR7
              RETI
```

8.8 Hardware UART for TMS320C3x

Section 8.7 discusses a software UART emulator, which allows the ’C3x to perform asynchronous communication. There are some applications that require a hardware UART. This section describes one possible design for a hardware UART (see Figure 8–12). This design, originally done in a field programmable gate array (FPGA), can be easily transferred to an application specific integrated circuit (ASIC). You can modify this design to accommodate faster data rates or different communication protocols.

Figure 8–12. TMS320C3x Serial Port to UART Interface

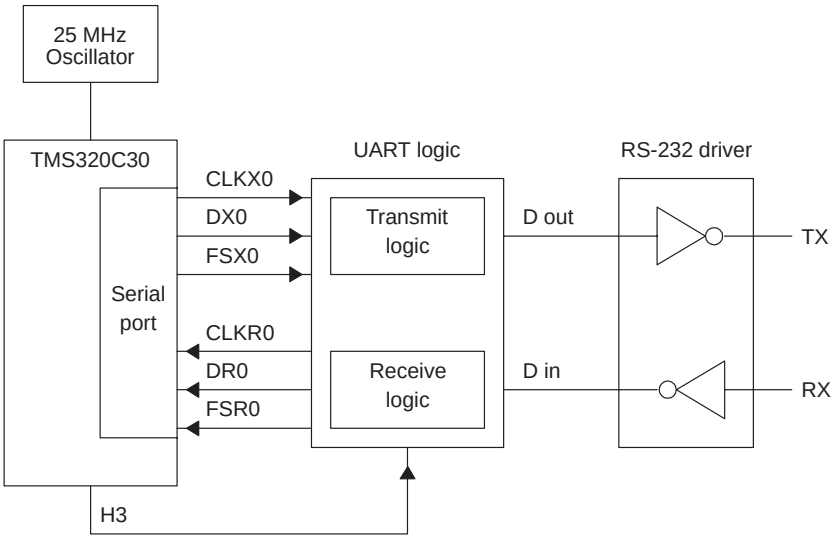
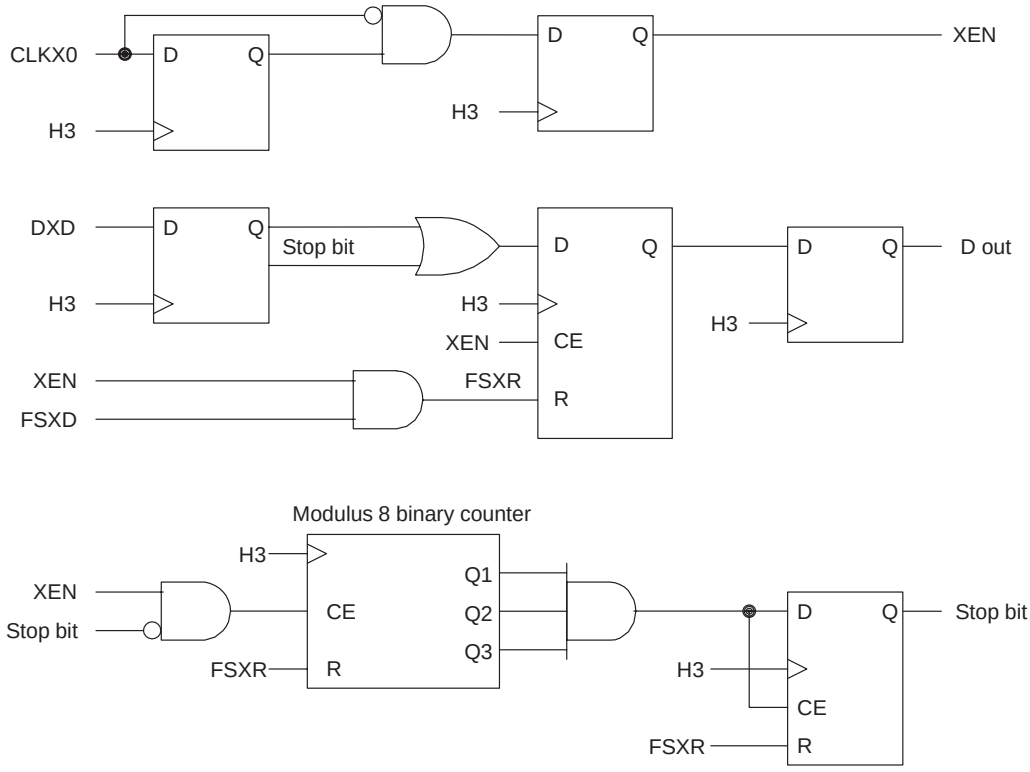


Figure 8–13 shows a 9,600-baud UART with one stop bit and one start bit. The clock signal, H3, is supplied to the circuit from the ’C3x. The DSP uses a 25-MHz clock.

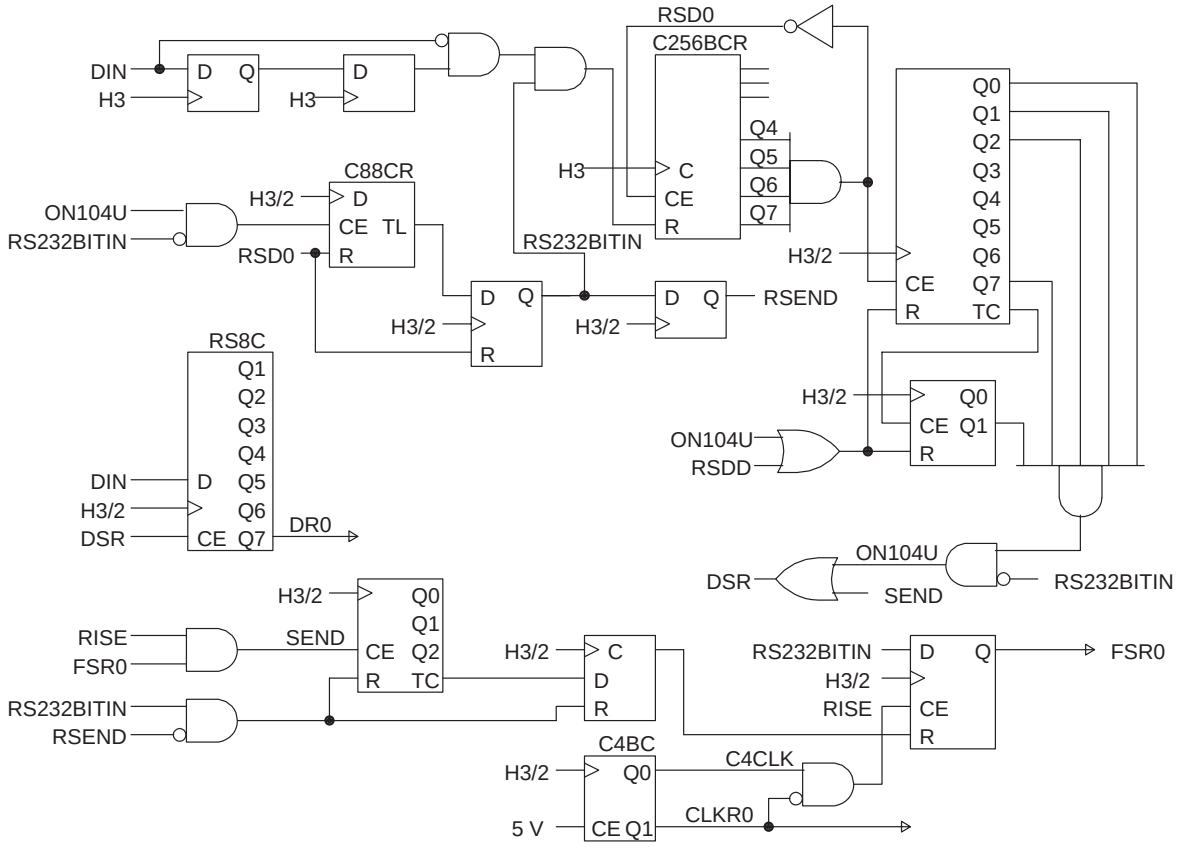
Figure 8–13. Transmit Circuitry



The ’C3x serial port transmit circuitry, shown in Figure 8–13, is configured to output eight bits of data at a rate of approximately 9.6 kHz. This is achieved by using one of the ’C30’s internal timers and programming it to the desired 9.6 kHz frequency. The transmitting port is configured in the first burst mode. This allows the leading FSX signals to help initiate a start bit for the UART protocols. The stop bit is generated at the end of the eighth bit by the UART circuitry.

The receive circuitry of the UART, shown in Figure 8–14, is activated when the circuit detects the start bit. The start bit is a logical 0. The delay circuit is activated on the falling edge of the start bit. The delay causes sampling of the incoming data bits to occur in the middle of each bit, thus, increasing the UART’s noise immunity.

Figure 8–14. Receive Circuitry



After the delay is performed, the timer is activated. The timer has a period of 104 μs, which corresponds to a baud rate of approximately 9.6 KHz. At each bit time, a data value is sampled into an 8-bit shift register. After all eight bits are received, the data is passed to the ’C30 over the serial port at 1/8 of the H3 clock rate. The FPGA circuitry interfaces the ’C30 in the fixed burst mode of operation to the serial port. Both the clock and the frame sync signals are generated by the FPGA circuitry.

This UART circuitry can also easily be designed to function as an ASIC or can be incorporated into a custom digital signal processor (cDSP). Modification to this circuit can be done for different serial communication protocols or even higher baud rates.

Chapter 9

Clock Oscillator and Ceramic Resonators

This chapter provides a general background on oscillators as well as information regarding crystal and ceramic resonators, their frequency characteristics, and the type of oscillator circuit used on the ’C3x. Also covered are design aspects of the ’C3x oscillator, including appropriate configuration of the external components, measured parameters for the on-board portion of the circuitry, use of the oscillator with overtone crystals, and general design considerations for choosing the external components for the oscillator. Finally, this chapter shows some design solutions for common frequencies.

Topic **Page**

9.1 Oscillators

The ’C3x is a member of the Texas Instruments’ family of high-speed DSPs. The ’C3x is capable of performing operations at a rate of up to 30 million instructions per second (MIPS). The wide variety of DSP applications requires a wide range of clocking frequencies. The ’C3x allows considerable flexibility in meeting these clocking requirements.

The ’C3x provides two modes for clock generation and control for use with different application needs. These include:

- External clock input with the capability to divide the clock frequency by 2
- Internal clock generation from an on-board oscillator with no external clock necessary (’C30 and ’C31 only)

The built-in oscillator provides a method for accurate clock generation that requires few external components (a crystal or ceramic resonator and two load capacitors). This saves board space and reduces system cost.

On the ’C3x devices, the on-board oscillator operates in a divide-by-2 mode. In this mode, the frequency of H1 or H3 (which indicates the actual machine cycles of the processor) is one half of the oscillator frequency.

9.1.1 Recommendations for Oscillator Use

The ’C3x family of devices provides several clock generation options based on cost, component count, and the required clock frequency for the application. The oscillator clocking option on the ’C3x provides a low-cost method of clock generation with as few as three external components (one crystal and two load capacitors), which helps to minimize board space consumed for clock generation. The crystal or ceramic resonator used determines the frequency of operation. This frequency can extend up to 60 MHz with third-overtone crystals.

CMOS-compatible integrated-circuit crystal oscillators are available across a wide frequency range. These are more expensive than the internal oscillator and usually consume more space on the board. CMOS oscillators also become more expensive with higher operating frequency.

9.2 Quartz Crystal and Ceramic Resonators

All oscillators require resonating components to determine the frequency of oscillation. A resonating component reacts more strongly within a certain frequency range than at other frequencies outside that range. A simple resonator consists of an inductor (L) and a capacitor (C). These components resonate or favor the frequency at which their individual reactances cancel each other. Figure 9–1 shows a simple series-LC resonator with impedance equations.

Figure 9–1. Series-LC Schematic

The impedance equations for the series-LC schematic are as follows:

$Z_L = j\omega L \qquad Z_c = 1/j\omega C \qquad Z_t = Z_L + Z_c = j(\omega L - 1/\omega C)$

Z_t is minimum where $\omega L = 1/\omega C$

so $\omega_s^2 = \frac{1}{LC} \quad \Rightarrow \quad \omega_s = \frac{1}{\sqrt{LC}}$

Consider the impedance of the series combination of these components. The impedance of the inductor $Z_L = j\omega L$, where ω is the angular frequency ($\omega = 2\pi f$), and the impedance of the capacitor $Z_c = 1/j\omega C$. The total impedance of the inductor-capacitor combination is $Z_t = Z_L + Z_c = j(\omega L - 1/\omega C)$. Therefore, the magnitude of the combined impedance of these two components is a minimum at the frequency where $\omega L = 1/\omega C$. This frequency (ω_s) is the resonant frequency and is determined by :

$$\omega_s = \frac{1}{\sqrt{LC}}$$

Although oscillators frequently consist of different combinations of inductors and capacitors as resonating elements, the accuracy of the frequency control with these components is limited. Changes in the values of L and C due to tolerance limitations and changes in the environment (such as temperature) strongly affect the frequency of the oscillator. Many applications in digital systems require precise clock timing and need more accurate resonators. Quartz crystal and ceramic resonators can provide a more stable and precise frequency control.

9.2.1 Behavior and Operation of Quartz Crystal and Ceramic Resonators

The oscillator circuitry built into the ’C3x devices is designed for use with a quartz crystal or ceramic resonator as the frequency-controlling element.

Quartz crystal and ceramic resonators are resonating components made with materials that have specific piezoelectric properties. Piezoelectric materials deform mechanically in the presence of an electric potential; this mechanical stress on the material produces a voltage. This property makes a very stable resonator, since the frequency of mechanical vibration is controlled precisely by the size, shape, and material properties of the crystal or ceramic used. In fact, many quartz crystal resonators are so precise that they operate within 10 parts per million (ppm) of the intended frequency.

Ceramic resonators are similar to quartz crystal resonators in physical structure, but they are made from a polycrystalline ceramic instead of monocrystalline quartz. The production process for the ceramic is much less expensive than for quartz, reducing the final cost of the resonator. However, the polycrystalline structure of the ceramic vibrates within a wider range of frequency than a quartz crystal does, and consequently, the frequency control is not as precise as it is with quartz. While quartz crystal resonators can operate within 10 ppm of the intended frequency, ceramic resonators generally operate within 5000 ppm. However, if accuracy greater than 5000 ppm is not necessary, ceramic resonators are a cost-effective alternative. Table 9–1 shows a comparison of three types of resonators.

Table 9–1. Comparison of Resonator Types

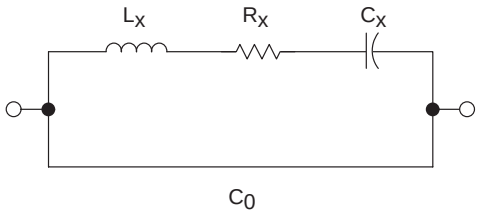
Type	Relative Price	Adjustment	Frequency Tolerance	Long-Term Stability
LC	Very low	Necessary	± 20000 ppm	Fair
Ceramic	Low	Not necessary	± 5000 ppm	Excellent
Crystal	High	Not necessary	± 10 ppm	Excellent

This document assumes that a quartz crystal is being used as the resonator; however, the information applies equally to ceramic resonators, unless otherwise specified.

Figure 9–2 shows a circuit model that is equivalent to a crystal. The graphs illustrate the behavior of the magnitude of the crystal impedance and the reactance of the crystal with frequency. The three components, L_X, R_X, and C_X, model the electrical behavior related to the mechanical vibration of the crystal. L_X and C_X control the resonant frequency according to the same equation shown in Figure 9–1. R_X models the mechanical energy loss in the crystal and

is related to the power dissipation in the crystal. C_0 is the capacitance of the two electrodes. The dielectric of the quartz physically separates the two electrodes. Together these components are a reasonably accurate electrical model for the behavior of the crystal. Values for these component models are usually available from the crystal manufacturer.

Figure 9–2. Crystal Equivalent Circuit Model

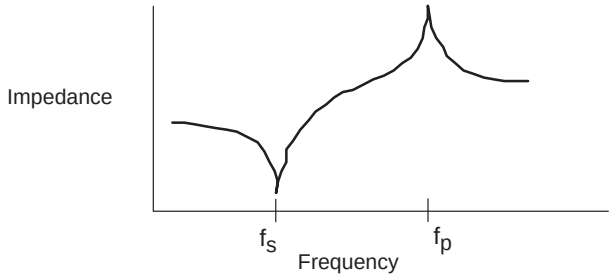


Notes: 1) C_0 is the capacitance of the two electrodes.

2) L_X, R_X, and C_X model the electrical behavior related to the mechanical vibration of the crystal; L_X and C_X control the resonant frequency according to the same equation shown in Figure 9–1 and R_X models the mechanical energy loss in the crystal.

Like the series LC resonator, crystals have an impedance minimum at a frequency determined by L_X and C_X. This is the series-resonant frequency (f_s). The presence of C_0 also introduces an impedance maximum at a frequency determined by L_X and C_0. This frequency is the parallel-resonant frequency (f_p). A graph of impedance magnitude that illustrates this behavior is also shown in Figure 9–3. The series-resonant frequency corresponds to the natural mechanical vibration frequency of the crystal. The parallel-resonant frequency is basically an electrical measurement phenomenon that results from the resonance between L_X and C_0 in the electrical model of the crystal and does not occur naturally. Consequently, all crystal oscillators operate at or near their series-resonant frequency.

Figure 9–3. Impedance Characteristics of Crystal

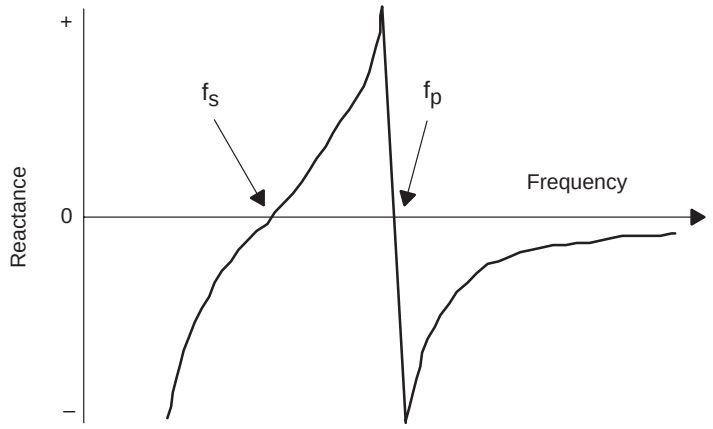


Notes: 1) f_S = series-resonant frequency

2) fp = parallel-resonant frequency

The graph in Figure 9–3 illustrates the behavior of the magnitude of the impedance of the crystal, but the crystal's phase response is also important in oscillator design. Figure 9–4 shows the reactance of the crystal with frequency. The reactance (and consequently the phase) is 0 at the series-resonant frequency (f_S), because at this frequency the reactances of L_X and C_X cancel each other. At this frequency, the total impedance of the crystal is equal to the resistance R_X.

Figure 9–4. Reactance Characteristics of Crystal



Notes: 1) f_S = series-resonant frequency
2) fp = parallel-resonant frequency

Below f_s, the crystal appears capacitive (negative reactance). Between f_s and f_p, the crystal appears inductive (positive reactance) and above f_p the crystal appears capacitive again. In an oscillator circuit, the crystal is always operated at or slightly above the series-resonant frequency in the inductive region. The capacitance C_0 has little effect on the series-resonant point (f_s), but in combination with the external load on the crystal, the capacitance C_0 affects the parallel-resonant point (f_p). For simplification of the circuit analysis, C_0 is sometimes considered part of the external load on the crystal.

When ordering a crystal, you must tell the manufacturer whether a *series-resonant* or *parallel-resonant* crystal is required. The nature of these terms is slightly different from the serial- and parallel-resonant frequency terms (f_S and f_p) previously described. A series-resonant crystal is intended to operate in a circuit with a low-load impedance across its terminals and, consequently, resonates very close to the series-resonant frequency (f_s). A parallel-resonant crystal is intended to operate in a circuit with a high-impedance load across its terminals and operates at some frequency slightly above f_s where the crystal's reactance is inductive. In this case, the

crystal attempts to resonate at the frequency at which its own inductive reactance exactly cancels the capacitive reactance of the combination of C_0 and an external-capacitive load. If supplied with the desired frequency and the external load to which the crystal will be connected, the manufacturer can produce a crystal that meets both of these requirements. The oscillator circuit used on the ’C3x devices requires a parallel-resonant crystal.

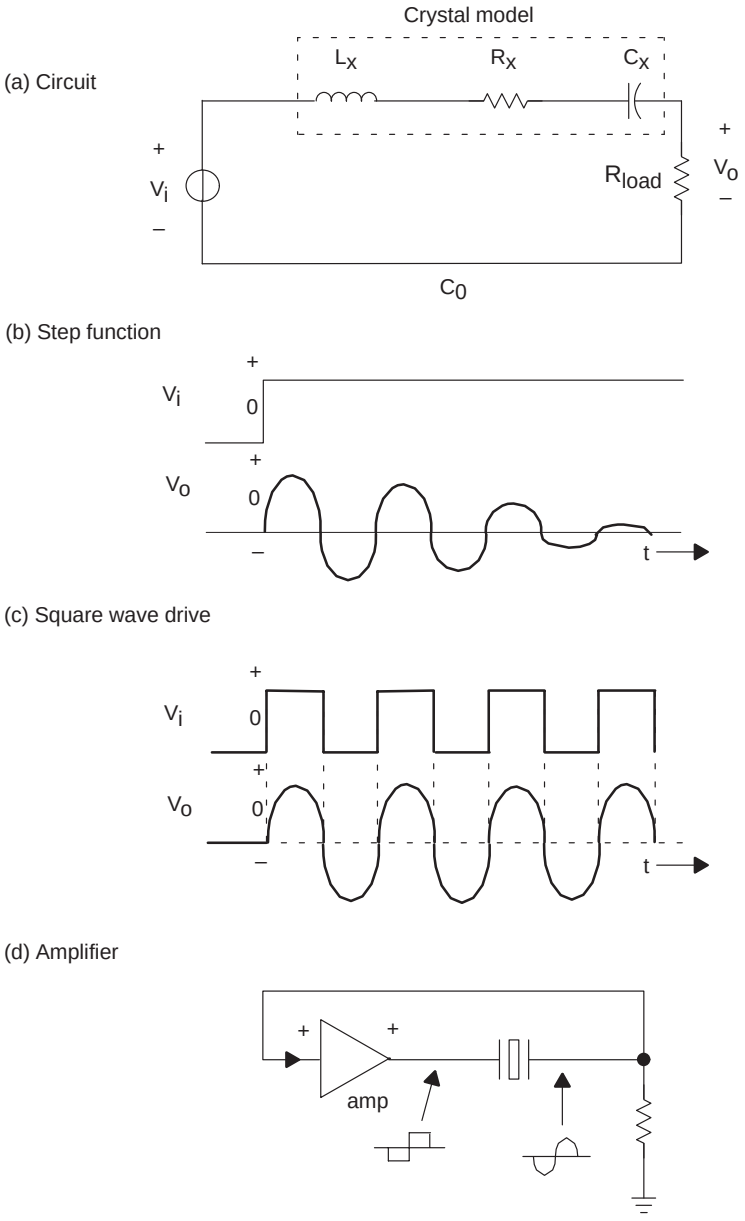
9.2.2 Crystal Response to Square-Wave Drive

Figure 9–5(a) shows the equivalent circuit model of a crystal driven by a step-function voltage source in series with a resistive load. In this figure, the capacitance, or C_0, of the crystal model is ignored because it is usually considered part of the load on the crystal and does not strongly affect the series-resonant frequency. When a step function excites a crystal, the crystal produces damped sinusoidal oscillation at its series-resonant frequency, as shown in Figure 9–5(b). The magnitude of the damping on the output waveform is proportional to the magnitude of R_X.

The lowest natural frequency of the crystal is the fundamental frequency. Depending on the design of the crystal, it can also have contributions to its output waveform from odd multiples of the fundamental frequency, or *overtones*. However, if the response at the fundamental frequency is considerably stronger than the response at these overtone frequencies, the contribution of the overtones to the output waveform is negligible.

If the step-function input is changed to a square-wave drive (a periodic set of step functions) at the frequency of the fundamental, the output of the crystal is sinusoidal, as shown in Figure 9–5(c). The source of the square wave provides enough energy to overcome the damping in each cycle. Although a square wave has a high content of odd overtones, the crystal resonates at its fundamental frequency and strongly attenuates all other frequencies. Consequently, the output of a crystal driven by a square wave is sinusoidal. If this sinusoidal output is fed back to the input of an appropriately designed amplifier, as shown in Figure 9–5(d), sustained oscillation is generated.

Figure 9–5. Crystal Response to a Square-Wave Drive



Notes: 1) C_0 is the capacitance of the two electrodes.

2) L_X, R_X, and C_X model the electrical behavior related to the mechanical vibration of the crystal; L_X and C_X control the resonant frequency according to the same equation shown in Figure 9–1 and R_X models the mechanical energy loss in the crystal.

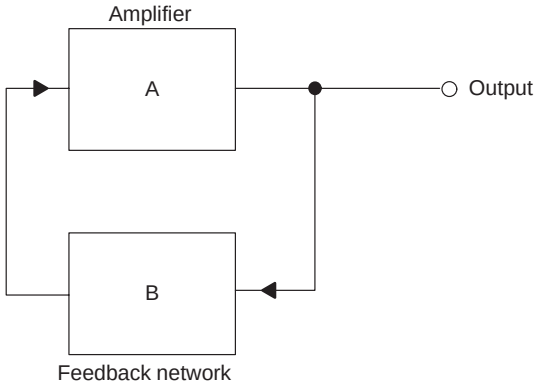
9.3 Pierce Oscillator Circuit

Figure 9–6 shows an oscillator circuit in its simplest form: an amplifier and a feedback network. This circuit must meet two requirements to sustain oscillation:

- The circuit must have positive feedback.
- The open loop gain must be greater than 1.

In Figure 9–6, A is the gain of the amplifier and B is the gain of the feedback network. For the circuit to have open-loop gain greater than 1, A $\times$ B must be greater than 1. For the circuit to have positive feedback, the phase shift around the loop must be 0 degrees (or n360°, where n = 0, 1, 2, 3, ...). If these conditions are met, the output oscillates at a frequency determined by the frequency selective feedback network and the amplitude increases until it reaches the linearity limitation of the amplifier.

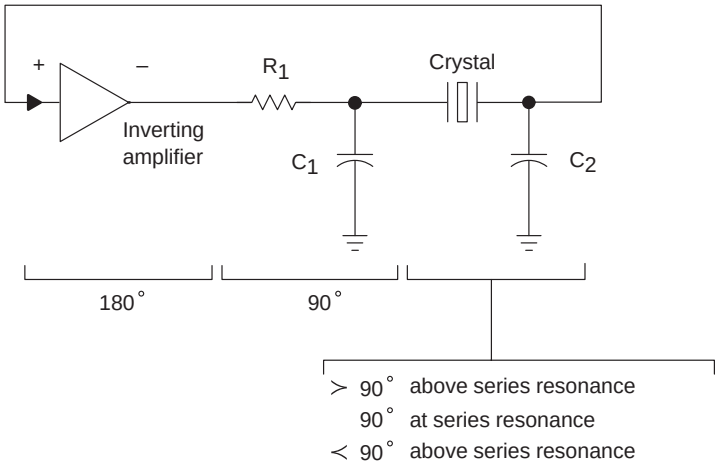
Figure 9–6. Simple Form of an Oscillator Circuit



There are many possible combinations of amplifiers, crystals, and phase-shifting components (inductors and capacitors) that meet the above-specified conditions for oscillation. One of the most common is a circuit based on the Pierce oscillator. Figure 9–7 shows an ideal version of this circuit. The Pierce oscillator uses an inverting amplifier, a parallel-resonant crystal as a resonator, and two capacitors as phase-shifting elements and load for the crystal. This circuit is used for several reasons:

- It has a large frequency range, from approximately 1 kHz to 200 MHz.
- It has high Q (because the load impedances are mostly capacitive and not resistive) and consequently exhibits very good stability.
- It maintains a high output signal while driving the crystal at a low-power level. This is important at higher frequencies, where crystals are physically thinner and therefore have lower power-dissipation limits.
- The low-pass RC networks formed by the crystal and load capacitors tend to filter transient noise spikes, giving the circuit good noise immunity.

Figure 9–7. Pierce Circuit: Ideal Operation



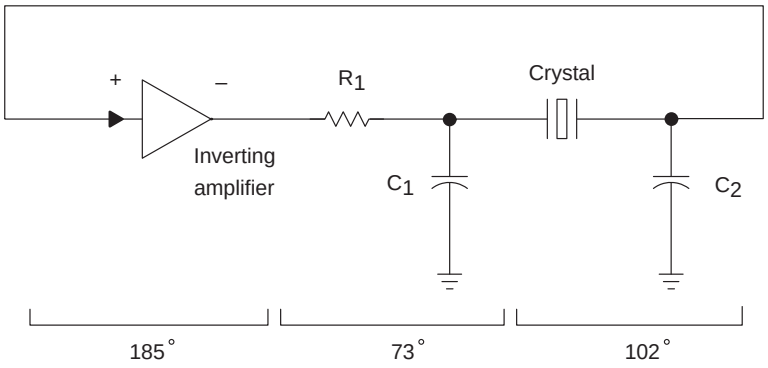
9.3.1 Oscillator Operation

The ideal circuit operates in the following manner. An input signal to the amplifier appears at the output, phase-shifted by approximately 180°. If it is assumed that at a certain frequency the impedance of C_1 is much greater than R_1, then the phase shift of this RC network introduces another approximately 90° phase shift. At the series-resonant frequency, the crystal appears to be a resistor and forms another RC network with C_2. If the impedance of C_2 is much greater than

the series resistance (R_X) of the crystal, this network provides another 90° phase shift. The total phase shift around the loop is now 180° + 90° + 90° = 360°**.** This phase shift meets one of the conditions for oscillation. If the gain of the amplifier is high enough to overcome the losses in the $R_1 - C_1 -$ crystal(R_X) $- C_2$ network for a total loop gain of greater than 1, then the circuit meets both oscillation conditions and oscillates.

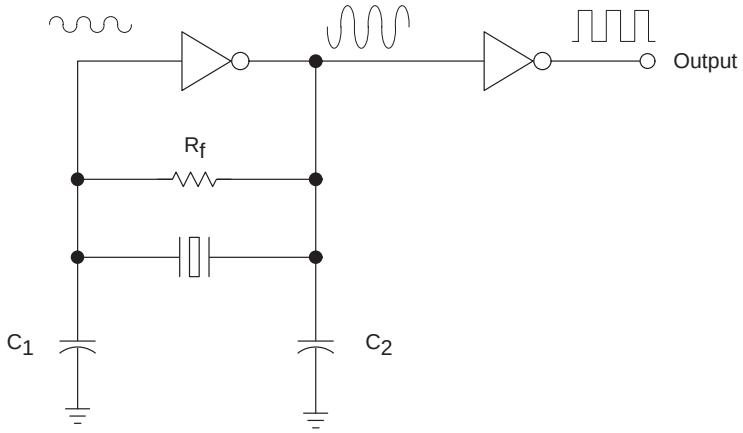
This explanation, however, is unrealistic because it ignores too many aspects of real-world circuit effects. Figure 9–8 illustrates a more typical example of the circuit behavior. In this case, the inverting amplifier has some phase delay, which causes it to produce a phase shift somewhat longer than 180°, depending on the frequency of operation. If oscillation is to occur, the passive components are forced to compensate for this phase difference. The only way the impedance of the load capacitances can change is when the frequency of operation changes. The frequency of operation tends to move above the series-resonant frequency, lowering the impedance of the load capacitances and raising the impedance of the crystal as it goes from being purely resistive to being both resistive and inductive (see Figure 9–2 (c) on page 9-5). When the frequency changes such that the loop phase shift once again equals 360°, the circuit oscillates at the higher frequency. For this reason, most Pierce circuits operate 5 – 40 ppm above the series-resonant frequency. This explanation clearly illustrates the circuit's actual behavior and explains why a parallel-resonant crystal always operates slightly above the series-resonant frequency.

Figure 9–8. Pierce Circuit: Actual Operation



When a square-wave output is desired (such as for a microprocessor clock source) the Pierce circuit sometimes is implemented in the manner shown in Figure 9–9. The crystal and load capacitances are in the same configuration as the circuit shown in Figure 9–8, with the exception that R_1 is replaced with the output impedence of the inverter. In the linear region, the inverter behaves like a linear inverting amplifier. The resistor (R_f) is introduced across the inverter to bias it into the linear region. This is the transition region between the two digital states, as shown in Figure 9–11 on page 9-14. Otherwise, the inverter output moves toward one of its two stable digital states and oscillation does not start because there is no gain in these regions (the output characteristic shown in Figure 9–11 on page 9-14 is flat).

Figure 9–9. Pierce Circuit for Square-Wave Output



The removal of R_1 from the circuit improves the loop gain and thus improves the likelihood of oscillation. However, removing R_1 also increases the drive level (power dissipation) on the crystal. The power dissipation limit of the crystal must not be exceeded under these conditions (power dissipation issues are discussed in section 9.4.4 on page 9-18.) Otherwise, the circuit operation is identical to that described for Figure 9–8.

The second inverter is added as a buffer and a waveshaping device. Since the output of the crystal is sinusoidal, the output of the first inverter also is sinusoidal. The second inverter provides a rail-to-rail square-wave output at the oscillation frequency to drive the microprocessor clock.

9.3.2 Pierce Oscillator Configuration for the TMS320C30 and TMS320C31

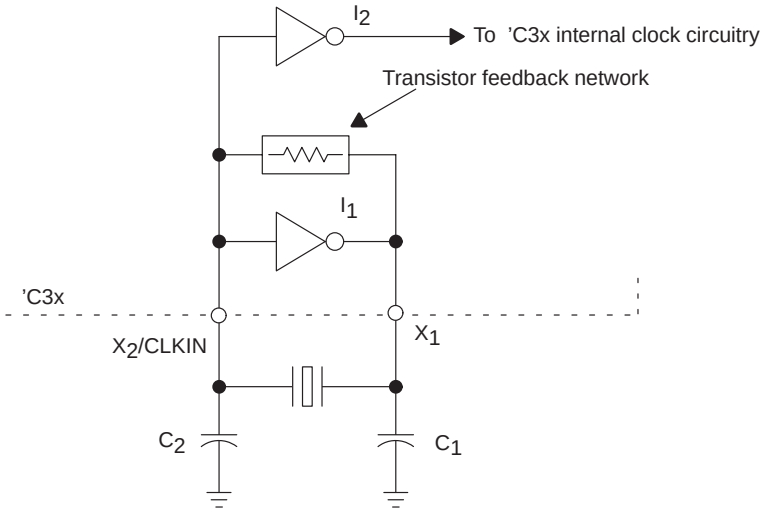
The ’C3x DSPs have two options for clocking the processor:

- Divide-by-2 operation of an externally supplied clock
- Divide-by-2 operation using the internal oscillator

To use the ’C3x internal oscillator, connect the crystal across the X2/CLKIN and X1 pins of the ’C30 and ’C31 (the ’C32 does not support the internal oscillator option.)

The ’C3x oscillator circuitry (with the exception of the crystal and the load capacitors) is integrated into the processor. Figure 9–10 shows the ’C3x oscillator circuitry, which is similar to the Pierce integrated circuit oscillator shown in Figure 9–9. On the ’C3x, the waveshaping inverter (I_2) takes its input from the input side of the inverter being used as the amplifier (I_1) rather than from the output as in the Pierce oscillator. This has little effect on the oscillator other than generating the digital complement of the clock that is generated in the circuit of Figure 9–9. Also, the feedback resistor in Figure 9–9 is integrated into the ’C3x as an active-load transistor-feedback network, so an external-feedback resistor is unnecessary. This feedback network ensures that the inverter I_1 is biased in its linear region.

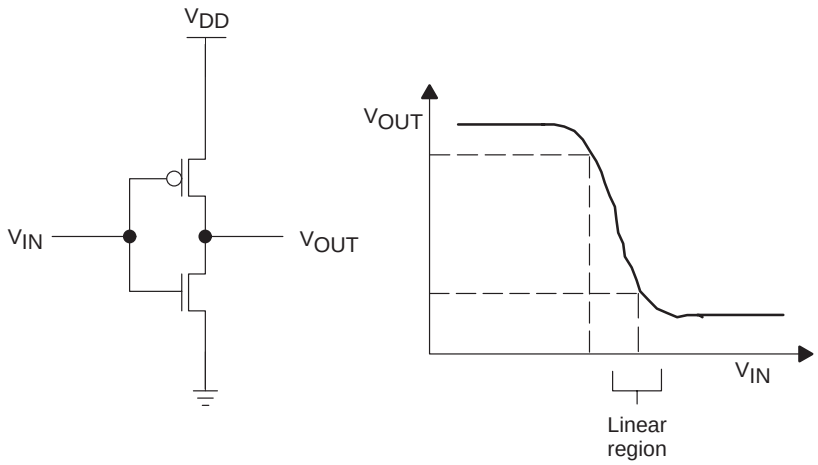
Figure 9–10. TMS320C3x Oscillator Circuitry



The inverters in the oscillator circuitry differ from the usual CMOS inverter configuration (shown in Figure 9–11) in that the p-channel transistor is biased as an active load instead of having the gate connected as the input of the inverter. This difference is part of the biasing scheme, which helps to ensure that the oscillator starts when power is applied. This design causes the rise and fall

times to be asymmetrical (for example, the rise time is longer than fall time), but since the oscillator output is divided by 2 before driving the internal-processor circuitry, the duty cycle of the final clock (H1 or H3) is 50%.

Figure 9–11. Digital Inverter Circuit and Its Transfer Characteristic

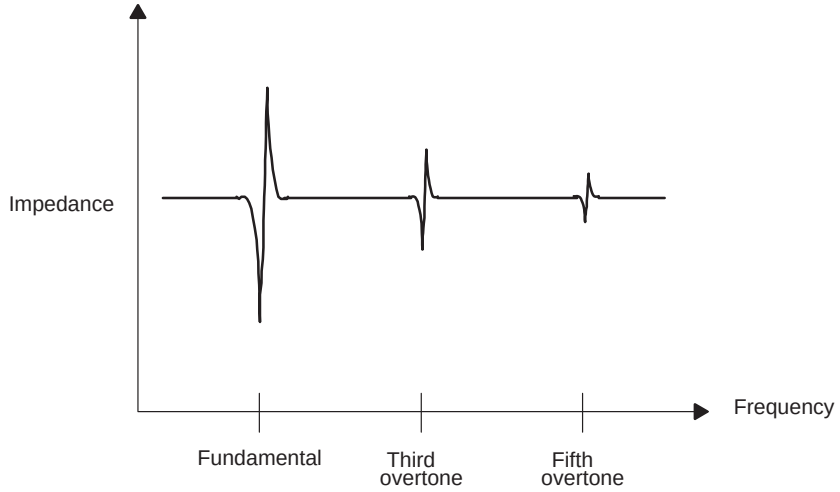


9.3.3 Overtone Operation of the Oscillator

Although crystals are usually considered to vibrate at only one frequency, they also resonate at odd multiples, or overtones, of the series-resonant frequency. The series-resonant frequency is the fundamental frequency of the crystal, and the odd overtones are odd multiples of the fundamental frequency (for example: 3×, 5×, 7×, ...). For low frequencies, it is common to operate crystals at their fundamental frequency. For higher frequencies, the crystal is made thinner. The thinner the crystal is, the more fragile and expensive it becomes. Thinner crystals also have a low-power dissipation limit and damage easily when overdriven.

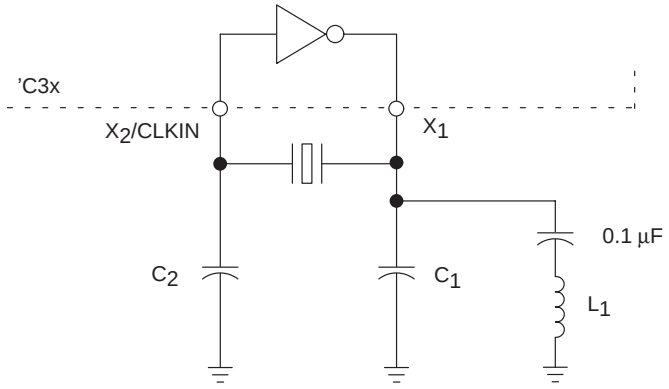
Most fundamental mode crystals operate at frequencies of 40 MHz or less. To generate frequencies higher than 40 MHz, it is common to use overtone crystals. Overtone crystals are optimized for operation at an overtone frequency with the fundamental frequency attenuated. Figure 9–12 illustrates the impedance of a crystal with respect to frequency. The strongest change in impedance is at the fundamental frequency, but there is also a response at the third and fifth overtones. If a crystal with the properties in Figure 9–12 is used in a Pierce circuit, it oscillates at the fundamental frequency. However, if the fundamental frequency is attenuated, the crystal circuit oscillates at the next higher odd overtone, in this case, the third overtone. High-frequency operation is achieved by using an overtone crystal and attenuating the fundamental frequency.

Figure 9–12. Impedance Characteristics of a Crystal



For the Pierce circuit used on the ’C3x, this attenuation of the fundamental frequency is achieved by capacitively coupling an inductor (L_1) in parallel with the load capacitor (C_1), as shown in Figure 9–13. The value of L_1 is chosen to resonate with C_1 at some intermediate frequency between the frequency of the desired overtone and the next lower odd overtone. At the desired overtone frequency, the impedance of L_1 is high enough compared to C_1 that L_1 is neglected and the network of C_1 and the inverter’s output impedance provides the near-90° phase lag desired. Since the phase conditions are met, the circuit oscillates at this frequency. At all lower overtones, L_1 is a lower impedance than C_1 and causes a 90° phase lead instead of phase lag. At any of these lower frequencies, the total phase shift around the feedback loop is 180°, not 360°, which is negative feedback, and stabilizes the circuit and prevents oscillation. L_1 is coupled with a 0.1 µF capacitor, which prevents the inductor from altering the dc bias of the inverter while causing negligible additional impedance at the oscillation frequency.

Figure 9–13. Oscillator Circuit for Overtone Crystal Operation



As an example, assume a 60-MHz third-overtone crystal is used with 10 pF load capacitors. The fundamental for this crystal is at 60/3 = 20 MHz. L_1 must be chosen to resonate with C_1 at a frequency between 20 and 60 MHz. If you choose the frequency halfway in between, 40 MHz, the value of L_1 is calculated as follows:

$$L_1 = 1/(\omega^2 C_1) = 1/(4\pi^2 f^2 C_1) = 1/(4\pi^2 (40 \times 10^6)^2 (10 \times 10^{-12})) = 1.58\ \mu H$$

Since the value of this inductance is not critical, the closest conveniently available inductor is used as long as the resonant frequency of $L_1 – C_1$ falls between the desired overtone and the next lower overtone.

A variety of crystals have been evaluated in this circuit. Although at higher frequencies, fifth-overtone crystals are more commonly available, they are not recommended for this circuit. The available gain from the internal inverting amplifier limits this configuration to third-overtone crystals. Several third-overtone crystal solutions for this circuit up to 60 MHz are listed in Table 9–2 on page 9-22.

9.4 Design Considerations

This section discusses some of the aspects of the design of the oscillator and their effects on its operation.

9.4.1 Crystal Series Resistance (R_X)

The series resistance of the crystal has a strong effect on the design of the oscillator, primarily in loop gain. R_X limits the crystal's minimum impedance value (seen at series resonance). Since the impedances of L_X and C_X cancel each other at this frequency, the impedance of the crystal is due entirely to R_X. The voltage divider formed by the crystal and C_2 influences the loop gain. As the impedance of the crystal becomes larger, the loss of gain due to the voltage divider becomes greater. Low-loop gain causes the oscillator to take longer to start up and prevents oscillation if the overall loop gain falls below 1. Higher crystal series resistance also reduces the overall oscillator circuit Q, resulting in poorer frequency stability. For these reasons, it is desirable to use the lowest R_X possible. Crystals with series resistance of 40 ohms or less are recommended.

9.4.2 Load Capacitors

In the Pierce circuit used on the 'C3x, the load capacitors have a strong effect on how far above the series-resonant frequency the crystal oscillates. The crystal's shunt-terminal capacitance, C_0, is considered part of the crystal's external-load capacitance as far as the frequency controlling elements (C_X and L_X) are concerned. A parallel-resonance oscillator circuit operates at the frequency where the reactances of the crystal (C_X and L_X) cancel the reactances from the load (C_0, C_1, C_2). Consequently, changes in the external-load capacitance cause the oscillator to change frequency to compensate for the phase change. The following formula gives an approximate value for the frequency shift from the series-resonant frequency:

$$\Delta f \approx \frac{f_s C_O}{2r(C_O + C_L)} \text{ where } r = \frac{C_O}{C_x} \text{ and } C_L = C_1 + C_2$$

The derivative of this formula, as shown below, is useful for determining the frequency variance due to changes in the load capacitance. This derivative is applied to find the frequency range implied by a load capacitance with a given tolerance. Also, if there is a need to adjust the operating frequency, use this formula to determine the appropriate value of a variable load capacitor.

$$\Delta f_r \approx \frac{\Delta C_L f_s C_O}{2r(C_O + C_L)^2}$$

Crystal manufacturers often accommodate requests for specific values for load capacitance to be used with their crystals. Values of 20 pF and 30 pF are commonly available. These load capacitance values are represented by $C_1 + C_2$, so for a crystal designed for load capacitance of 20 pF, $C_1 = C_2 = 10$ pF is used. Capacitance values higher than 30 pF increase attenuation, lowering the overall loop gain. Capacitance values this high can cause the circuit to stop oscillating. A load capacitance of 20–30 pF is recommended for high-frequency crystals. Ceramic resonators usually require higher load capacitance than high-frequency crystals (see the manufacturer's recommendations). Load capacitance values are included in Table 9–2 on 9-22.

9.4.3 Loop Gain

Loop gain primarily affects the startup time of the oscillator. Overall loop gain must be greater than 1 for oscillation to be sustained. Higher loop gain causes the oscillation amplitude to increase rapidly, therefore reducing the time necessary for the oscillator to reach its steady state.

The minimum gain measured for the 'C3x inverter is 5.6. To maintain an overall loop gain of 1, the external component network of C1-crystal-C2 must not introduce a loss of greater than 5.6. For this reason, the values of the load capacitance and crystal-series resistance have a strong effect on whether the circuit oscillates.

9.4.4 Drive Level/Power Dissipation

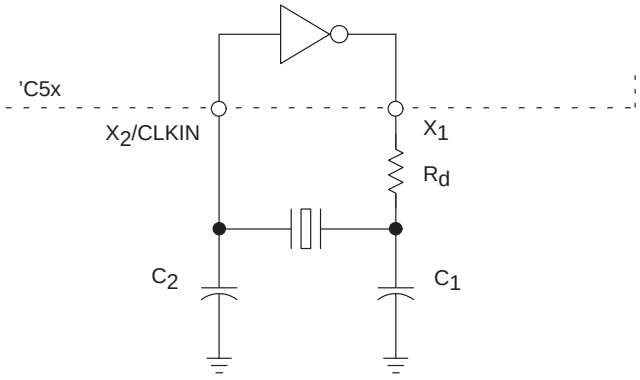
Another parameter specified when ordering a crystal is the drive level or power dissipation. Higher frequency crystals generally have lower power dissipation ratings because the crystal is physically thinner and is damaged by excessive voltages. Power dissipation also affects frequency stability because the crystal's frequency of operation is dependent on temperature. Excessive power dissipation causes crystal heating and results in frequency drift.

There is not a convenient way to measure the power dissipation in the crystal. The series resistance (R_X) is the only power-dissipating component in the crystal. Measuring the external voltage on the crystal includes the voltage across L_X and C_X. Therefore, the power dissipation in R_X cannot be easily calculated directly from the voltage on the crystal. It is necessary to measure the current through the crystal using a current probe or to indirectly measure the current by measuring the voltage across a small resistor in series with the crystal. You can then calculate the power by using I^2R.

Once the drive level is known, if it is necessary to limit the drive level to the crystal, one of the simplest ways to do so is shown in Figure 9–14. A resistor (R_d) is added in series between X_1 and the external components. This resistor drops part of the voltage driven by the 'C3x and consequently lowers the drive voltage on the crystal. The disadvantage to this method is that the voltage drop reduces the overall loop gain of the oscillator circuit. The value of R_d must be large enough to bring the power dissipation of the crystal within the manufacturer's specification, but R_d must not be so large that the loop gain drops below 1 or the circuit no longer oscillates. Using crystals with minimum power dissipation ratings of 1 mW is recommended.

The oscillator circuit solutions in Table 9–2, when operated without R_d, have yielded crystal-power dissipation measurements near 1 mW. Differences in circuit and crystal parameters can cause the power dissipation in the crystal to slightly exceed 1 mW. If crystal-power dissipation is critical, adding a resistor (R_d) with a value of 33 Ω to limit the crystal-power dissipation or obtaining crystals with power dissipation ratings higher than 1 mW, is recommended. When operated with R_d = 33 Ω, each of the circuit solutions shown in Table 9–2 have exhibited less than 1 mW crystal power dissipation.

Figure 9–14. Addition of R_d to Limit Drive Level of the Crystal

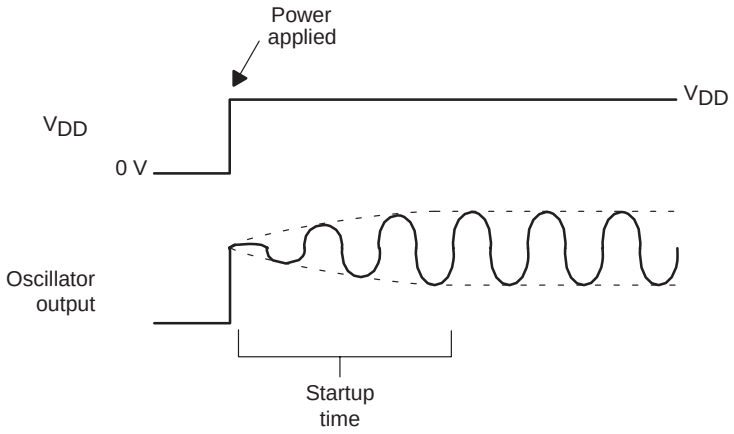


9.4.5 Startup Time

Figure 9–15 shows that when the oscillator starts, low-amplitude oscillations gradually build until the linearity limit of the amplifier is reached. You experience this startup time at power-up. Maximizing loop gain minimizes the startup time for the oscillator.

Startup time depends on the external components used, but generally requires at least 100 ms after power up for the oscillator to stabilize. For this reason, a reset delay of 150–200 ms is recommended following power up.

Figure 9–15. Oscillator Startup

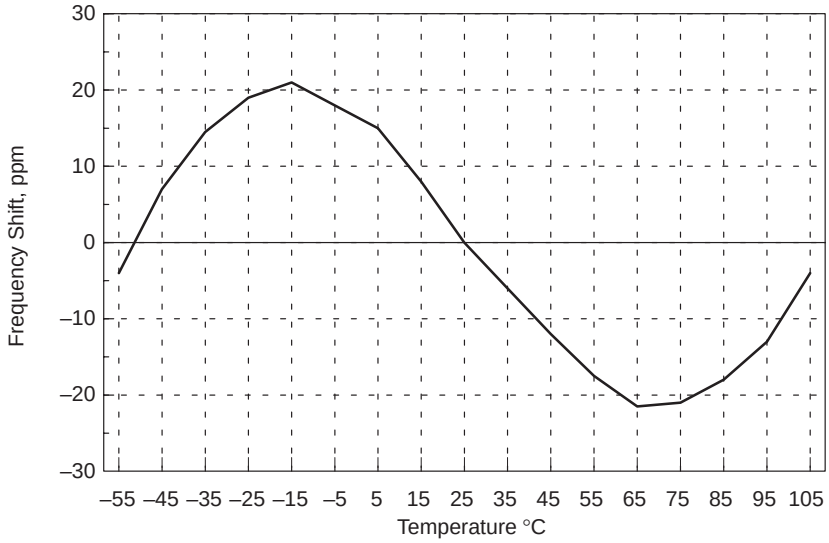


9.4.6 Frequency-Temperature Characteristics of Crystals

The actual operating frequency of a crystal depends on temperature. The extent to which frequency changes with respect to temperature strongly relates to the cut of the crystal. AT- and SC-cut crystals behave differently from DT-, CT-, and BT-cut crystals. Even slight changes in the cut angle of the crystal can strongly affect the frequency-temperature characteristics.

Most crystals available in the frequency range of interest for DSPs are AT-cut crystals. The frequency-temperature characteristic for AT-cut crystals is a third-order function, similar to that shown in Figure 9–16. This graph shows the general temperature-frequency behavior of AT-cut crystals. Similar information is readily available from crystal manufacturers.

Figure 9–16. Example Frequency-Temperature Characteristic of AT-Cut Crystals



9.4.7 Crystal Aging

Crystal aging is the gradual change in the frequency of a crystal over time. This change occurs due to stress relief between the mounting structure and the electrodes and absorption (or deabsorption) of contaminants from the resonator surfaces. Changes in temperature accelerate both of these mechanisms. The major mechanism for aging in crystals above 1 MHz is mass transfer to and from the resonator surfaces. The most rapid aging occurs early in the crystal's lifetime, and then aging tends to stabilize. For example, a crystal that ages 10–60 parts per million (ppm) in a year experiences 5 ppm of that aging in the first month. Crystals are available (at additional expense) that have very low aging rates, due to cleaner fabrication and packaging processes. These crystals have aging characteristics as low as 1×10^{-8} ppm per year. Complete information on aging characteristics is available from crystal manufacturers.

9.5 Oscillator Solutions for Common Frequencies

The oscillator solutions in this section were built and tested with samples from the manufacturers listed in Table 9–2. These circuits were tested at room temperature and verified to operate correctly within the recommended range of V_{DD} (4.75–5.25 V).

Table 9–2. Oscillator Solutions by Frequency

Frequency	Mode	Type	Supplier	Part Number	C_1, C_2	R_d	L_1
40 MHz	Fundamental	Crystal	SaRonix	HFX series crystals	10 pF	0/33†	–
40 MHz	Third overtone	Crystal	Anderson	011-668-04663	10 pF	0/33†	3.3 µH
50 MHz	Fundamental	Crystal	SaRonix	HFX series crystals	10 pF	0/33†	–
50 MHz	Third overtone	Crystal	SaRonix	SRX5223	10 pF	0/33†	3.3 µH
60 MHz	Third overtone	Crystal	Anderson	011-668-04725	10 pF	0/33†	3.3 µH

† When these circuits are operated without R_d, they yield crystal power dissipation measurements near 1 mW. Differences in circuit and crystal parameters can cause the power dissipation in the crystal to slightly exceed 1 mW. If crystal power dissipation is critical, it is recommended that 33 Ω of R_d be added to limit the crystal power dissipation or obtain crystals with power dissipation ratings higher than 1 mW. When operated with R_d = 33 Ω, each of the circuits shown exhibited less than 1 mW crystal power dissipation.

The following circuits are used for ceramic resonators and fundamental-mode crystal resonators. The circuit in Figure 9–17 is used for all circuits marked fundamental mode in Table 9–2. The circuit in Figure 9–18 is used for all circuits marked third-overtone mode in Table 9–2. Crystals used in these circuits must be parallel resonant with a series resistance of 40 ohms or less and must have a power dissipation rating of 1 mW or greater.

Figure 9–17. Fundamental-Mode Circuit

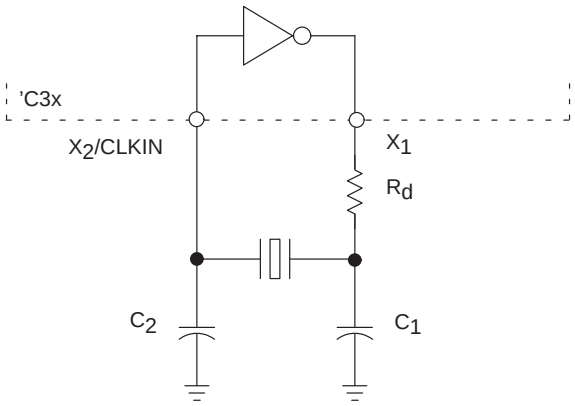
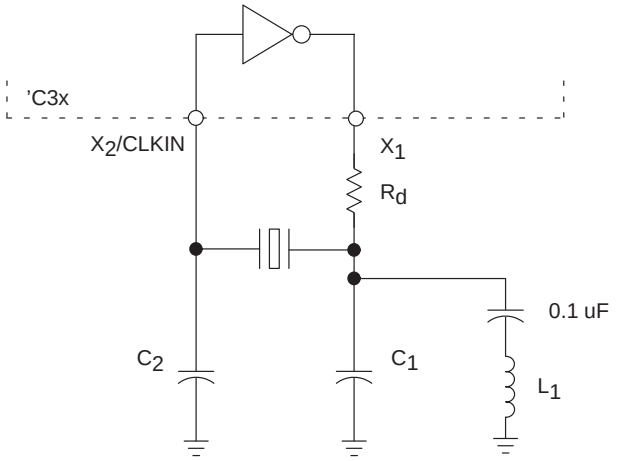


Figure 9–18. Third-Overtone Circuit



Chapter 10

XDS510 Emulator Design Considerations

This chapter explains the design requirements of the XDS510™ emulator and discusses the Extended Development System (XDS) cable (manufacturing part number 2617698–0001). This cable is identified by a label on the cable pod marked JTAG3/5V and supports both standard 3-V and 5-V target system power inputs.

The term JTAG emulation, as used in this book, refers to TI scan-based emulation, which is based on the IEEE 1149.1 standard.

Topic **Page**

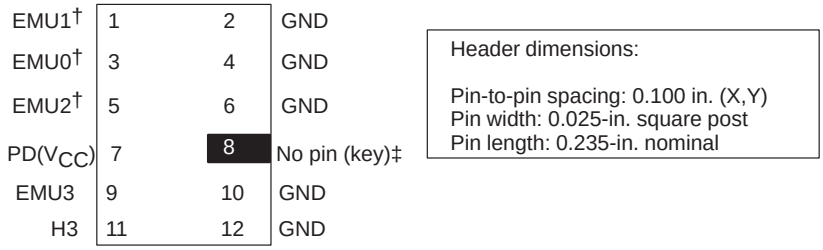
10.1 Designing the MPSD Emulator Connector (12-Pin Header)

The ’C3x uses modular port scan device (MPSD) technology to allow complete emulation through a serial scan path of the ’C3x. To communicate with the emulator, *your target system must have a 12-pin header* (2 rows of 6 pins) with the connections that are shown in Figure 10–1.To use the target cable, supply the signals shown in Table 10–1 to a 12-pin header with pin 8 cut out to provide keying. For the latest information, see the *JTAG/MPSD Emulation Technical Reference*.

Although you can use other headers, the recommended header is the unshrouded, straight header having the following DuPont connector systems part numbers:

- 65610–112
- 65611–112
- 37996–112
- 67997–112

Figure 10–1. 12-Pin Header Signals and Header Dimensions



† These signals must be pulled up with separate 20-kΩ resistors to V_{CC}.
‡ While the corresponding female position on the cable connector is plugged to prevent improper connection, the cable lead for pin 8 is present in the cable and is grounded as shown in the schematics and wiring diagrams in this document.

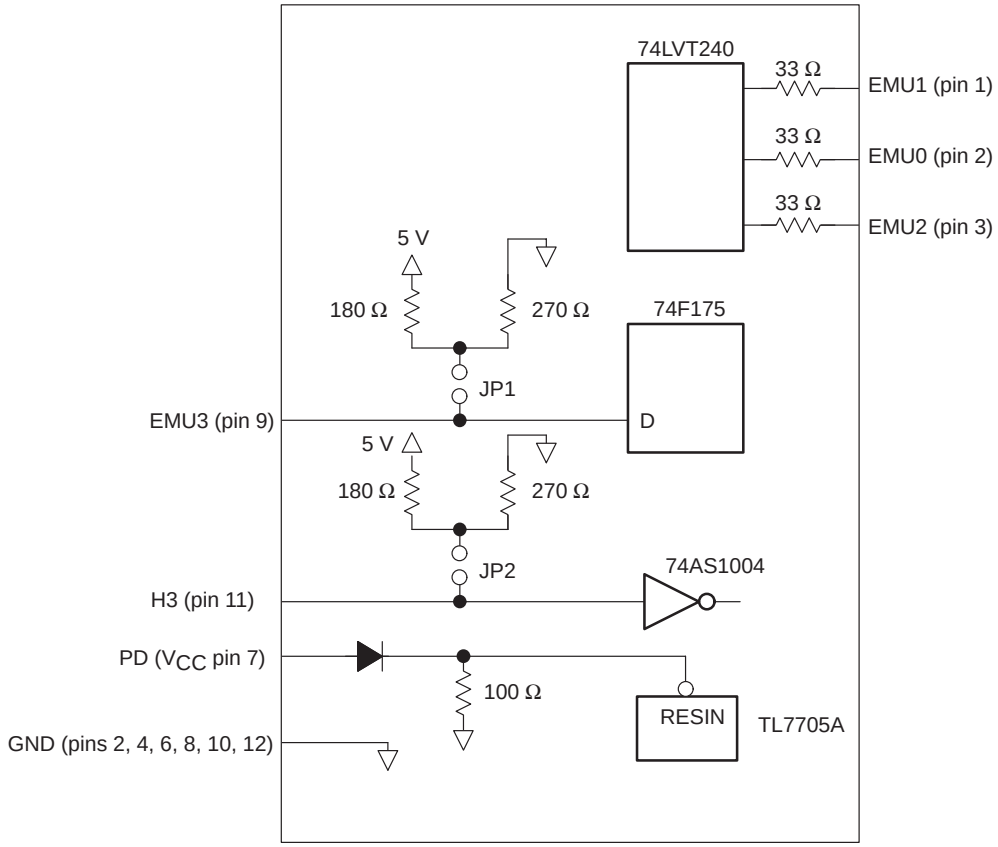
Table 10–1. 12-Pin Header Signal Descriptions and Pin Numbers

XDS510 Signal	Description	’C30 Pin Number	’C31 Pin Number
EMU0	Emulation pin 0	F14	124
EMU1	Emulation pin 1	E15	125
EMU2	Emulation pin 2	F13	126
EMU3	Emulation pin 3	E14	123
H3	’C3x H3	A1	82
PD	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD must be tied to V_{CC} in the target system.		

10.2 Emulator Cable Pod Logic

Figure 10–2 shows a portion of logic in the emulator cable pod. The 33-Ω resistors have been added to the EMU0, EMU1, and EMU2 lines to minimize cable reflections.

Figure 10–2. Emulator Cable Pod Interface



10.3 MPSD Emulator Cable Signal Timing

Figure 10–3 shows the signal timings for the emulator cable pod. Table 10–2 defines the timing parameters. The 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.

Figure 10–3. Emulator Cable Pod Timings

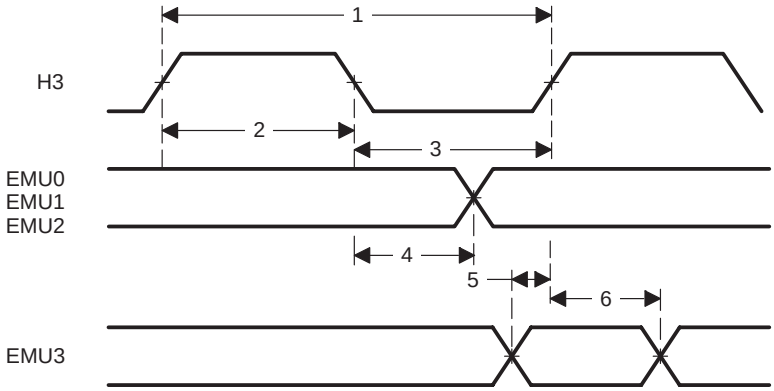


Table 10–2. Emulator Cable Pod Timing Parameters

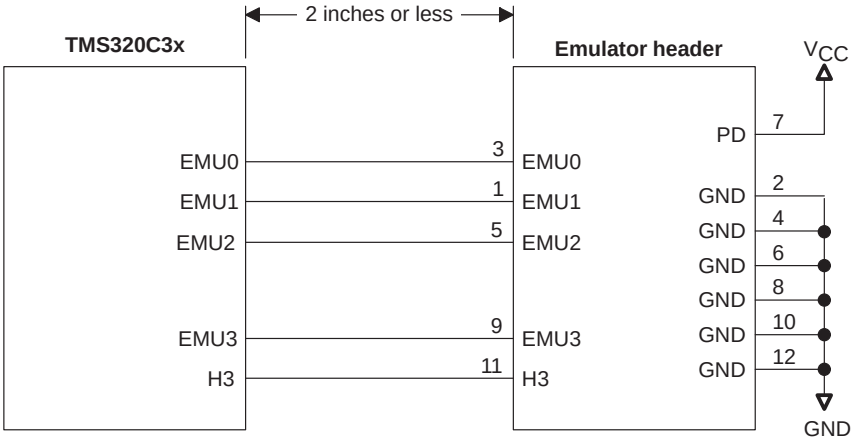
No.	Reference	Description	Min	Max	Unit
1	$t_{H3\ min}$ $t_{H3\ max}$	H3 period	35	200	ns
2	$t_{H3\ high\ min}$	H3 high pulse duration	15		ns
3	$t_{H3\ low\ min}$	H3 low pulse duration	15		ns
4	t_d (EMU0, 1, 2)	EMU0, 1, 2 valid from H3 low	7	23	ns
5	t_{su} (EMU3)	EMU3 setup time to H3 high	3		ns
6	t_{hd} (EMU3)	EMU3 hold time from H3 high	11		ns

10.4 Connections Between the Emulator and the Target System

It is extremely important to provide high-quality signals between the emulator and the ’C3x on the target system. In many cases, the signal must be buffered to produce high quality. The need for signal buffering can be divided into three categories, depending on the placement of the emulation header:

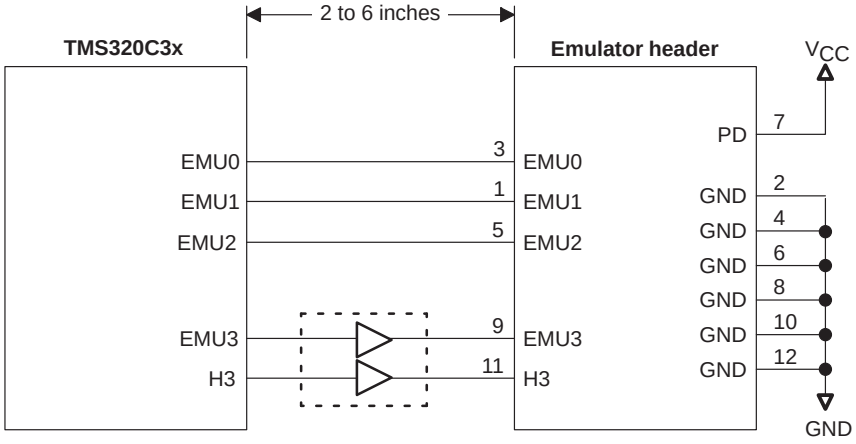
- **No signals buffered.** In this situation, the distance between the emulation header and the ’C3x should be no more than 2 inches (see Figure 10–4).

Figure 10–4. Connections Between the Emulator and the TMS320C3x With No Signals Buffered



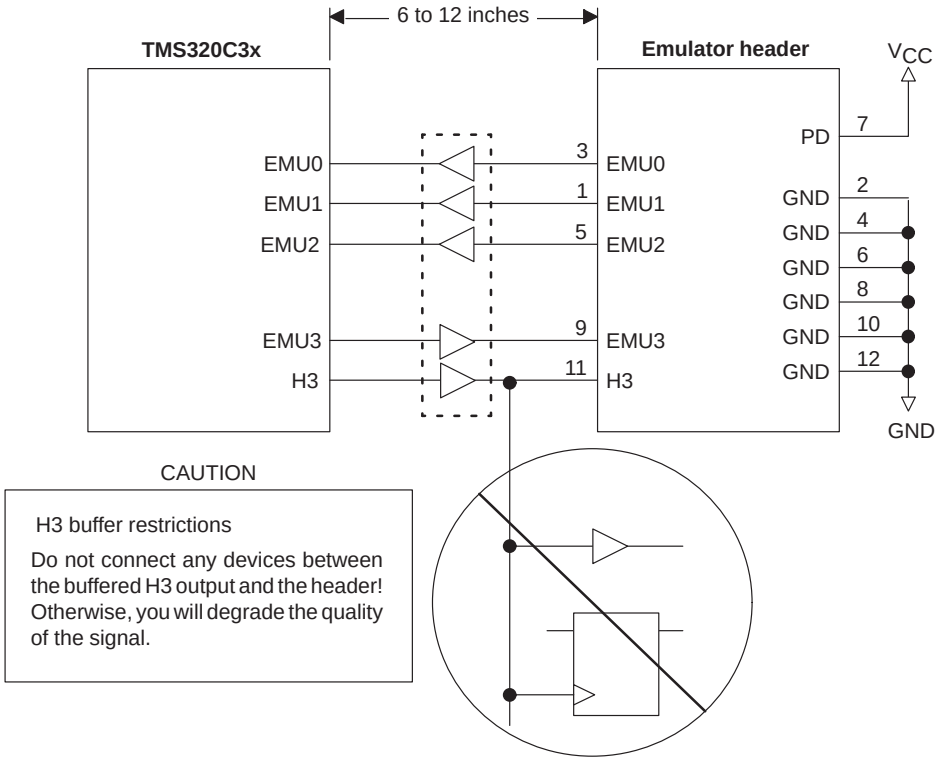
- **Transmission signals buffered.** In this situation, the distance between the emulation header and the ’C3x is greater than 2 inches but less than 6 inches. The transmission signals, H3 and EMU3, are buffered through the same package (see Figure 10–5).

Figure 10–5. Connections Between the Emulator and the TMS320C3x With Transmission Signals Buffered



- **All signals buffered.** The distance between the emulation header and the ’C3x is greater than 6 inches but less than 12 inches. All ’C3x emulation signals, EMU0, EMU1, EMU2, EMU3, and H3, are buffered through the same package (see Figure 10–6).

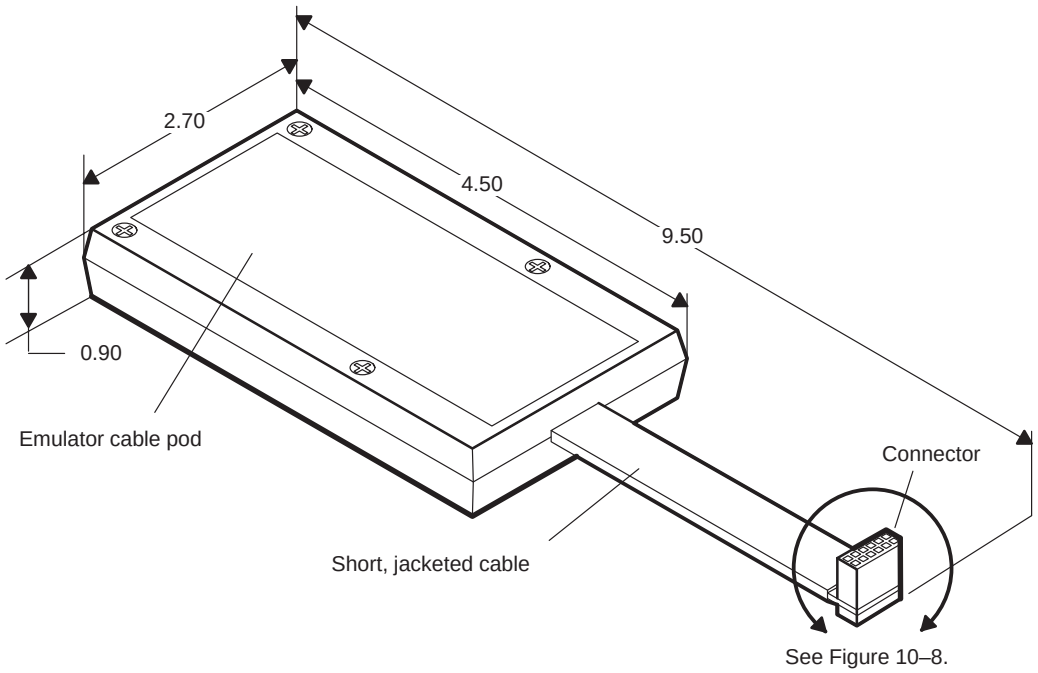
Figure 10–6. Connections Between the Emulator and the TMS320C3x With All Signals Buffered



10.5 Mechanical Dimensions for the 12-Pin Emulator Connector

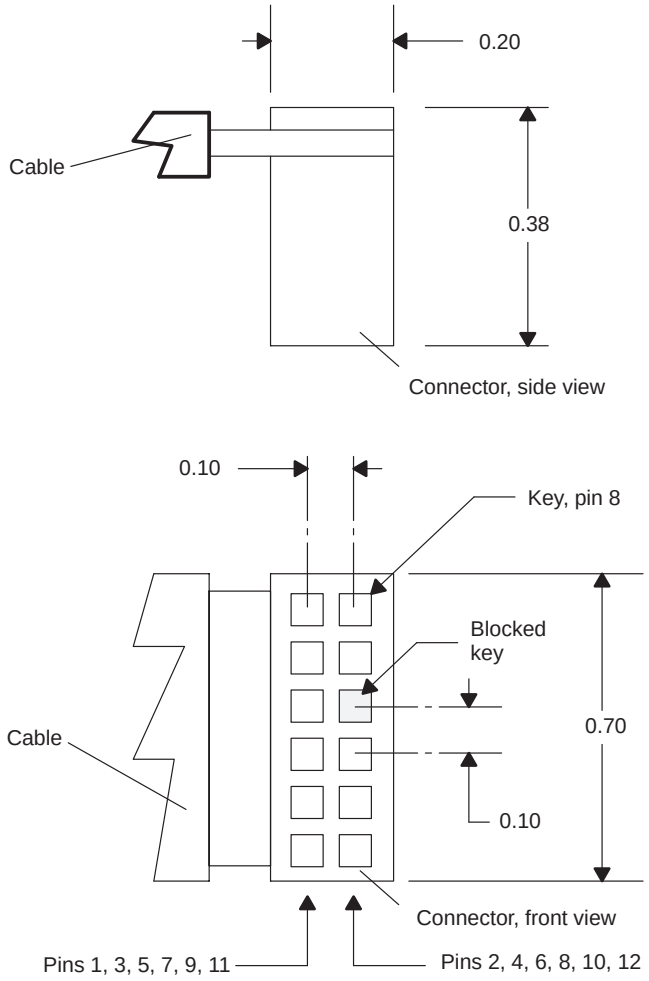
The ’C3x emulator target cable consists of a 3 foot section of jacketed cable, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet, 10 inches. Figure 10–7 and Figure 10–8 show the mechanical dimensions for the target cable pod and short cable. Note that the pin-to-pin spacing on the connector is 0.10 inches in both the X and Y planes. The cable pod box is nonconductive plastic with four recessed metal screws.

Figure 10–7. Pod/Connector Dimensions



Note: All dimensions are in inches and are nominal unless otherwise specified.

Figure 10–8. 12-Pin Connector Dimensions

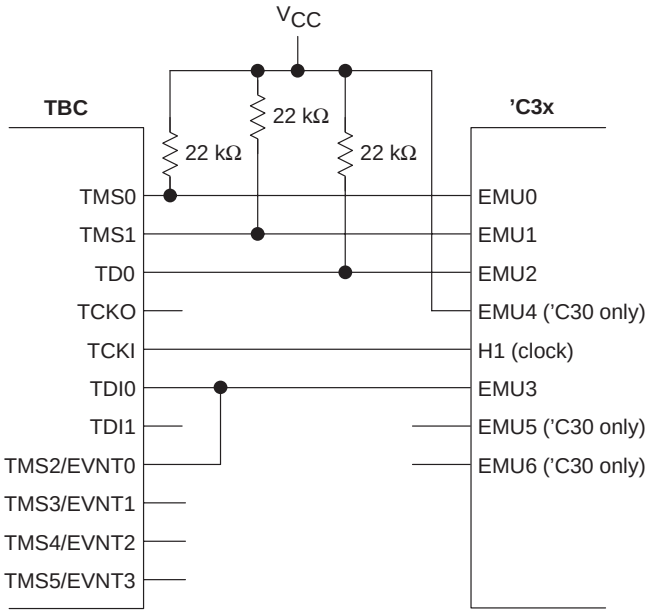


Note: All dimensions are in inches and are nominal unless otherwise specified.

10.6 Diagnostic Applications

For system diagnostic applications or to embed emulation compatibility on your target system, connect a ’C3x device directly to a TI ACT8990 test bus controller (TBC) as shown in Figure 10–9. The TBC is described in the Texas Instruments *Advanced Logic and Bus Interface Logic Data Book*. A TBC can connect to only one ’C3x device.

Figure 10–9. TBC Emulation Connections for TMS320C3x Scan Paths



Notes: 1) In a ’C3x design, the TBC can connect to only one ’C3x device.

2) The ’C3x device’s H1 clock drives TCKI on the TBC. This is different from the emulation header connections where H3 is used.

Chapter 11

Development Support and Part Ordering Information

This chapter provides development support information, device part numbers, and support tool ordering information for the ’C3x.

Each ’C3x support product is described in the *TMS320 Family Development Support Reference Guide*. 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*.

For information on pricing and availability, contact the nearest TI field sales office or authorized distributor.

Topic **Page**

11.1 Development Support

This section describes the development support provided by Texas Instruments.

11.1.1 Development Tools

Texas Instruments offers an extensive line of development tools for the ’C3x 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. These tools are described below.

Code Generation Tools

There are two types of code generation tools:

- *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 the PC (DOS, DOS extended memory, and OS/2), VAX/VMS, and SPARC workstations. See the *TMS320 Floating-Point DSP Optimizing C Compiler User’s Guide* for detailed information.

- *Assembler/linker.* Converts source mnemonics to executable object code. It supports both ’C3x and ’C4x assembly code. This product is currently available for the PC (DOS, DOS extended memory, and 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* for detailed information.

System Integration and Debug Tools

There are four types of system integration and debug tools:

- *Simulator*. Simulates through software the operation of the ’C3x and can be used in C and assembly software development. This product is currently available for the PC (DOS and Windows) and SPARC workstations. See the *TMS320C3x C Source Debugger User’s Guide* for detailed information.

- *XDS510 emulator*. Performs full-speed in-circuit emulation with the ’C3x, providing access to all registers as well as to internal and external memory. It can be used in C and assembly software development and has the capability of debugging multiple processors. This product is currently available for the PC (DOS, Windows, and OS/2) and SPARC workstations. This product includes the emulator board (emulator box, power supply, and small computer system interface (SCSI) connector cables in the SPARC version), the ’C3x C source debugger software, and the JTAG cable.

  Because ’C3x and ’C5x XDS510™ emulators also come with the same emulator board (or box), you can buy the ’C3x C source debugger software as a separate product called the ’C3x C Source Debugger Conversion Software. This enables you to debug ’C3x/’C4x/’C5x applications with the same emulator board. The emulator cable that comes with the ’C5x XDS510 emulator is not compatible with the ’C3x. You need a JTAG emulation conversion cable. See the *TMS320C3x C Source Debugger User’s Guide* for detailed information on the ’C3x emulator.

- *Evaluation module (EVM)*. Each EVM comes complete with a PC halfcard and software package. The EVM board contains the following:

  - A ’C30 and a 33-MFLOPS, 32-bit floating-point DSP
  - A 16K-word, zero-state SRAM, allowing coding of most algorithms directly on the board
  - A speaker/microphone-ready analog interface for multimedia, speech, and audio applications development
  - A multiprocessor serial port interface for connecting to multiple EVMs
  - A host port for PC communications

  The system also comes with all the software required to begin applications development on a PC host. Equipped with a C and assembly language source-level debugger for the DSP, the EVM has a window-oriented, mouse-driven interface that enables the downloading, executing, and debugging of assembly code or C code.

The ’C3x assembler/linker is also included with the EVM. For users who prefer programming in a high-level language, an optimizing ANSI C compiler and an Ada compiler are offered separately.

- *Emulation porting kit (EPK)*. Enables you to integrate emulation technology directly into your system without the need of an XDS510 board. The EPK is intended to be used by third parties and high-volume board manufacturers and requires a licensing agreement with Texas Instruments. The kit contains host (or PC) source and object code, which lets you tailor ’C30 EVM-like capabilities to your ’C3x system through the SM74ACT8990 test bus controller (TBC). The EPK can be used in such applications as program download for system self test and initialization or system emulation and debug to feature resident emulation support. EPK software includes the TI high-level language (HLL) debugger in object as well as source code for the TBC communication interface. The HLL code is the windowed debugger found with many TI DSP simulators, EVMs, and emulators. With the EPK, the HLL user interface can be ported directly to the system board. The source code for the TBC communication interface consists of such commands as read/write, memory run, stop, and reset that communicate with the ’C3x device. Using the EPK reduces system and development cost and speeds time to market. For more information on the kit, call the DSP hotline at (281)274–2320.

11.1.2 TMS320 Third Parties

The TMS320 family is supported by product and service offerings from more than 100 independent vendors and consultants, known as third parties. These support products take various forms (both software and hardware) from cross-assemblers, simulators, and DSP utility packages to logic analyzers and emulators. Additionally, TI third parties offer more than 150 algorithms that are available for license through the TMS320 software cooperative. These algorithms can greatly reduce development time and decrease time to market. The expertise of those involved in support services ranges from speech encoding and vector quantization to software/hardware design and system analysis.

For a more detailed description of services and products offered by third parties, See the *TMS320 Third Party Support Reference Guide* and the *TMS320 Software Cooperative Data Sheet Packet*. Call the Literature Response Center at (800) 477–8924 to request a copy.

11.1.3 Technical Training Organization (TTO) TMS320 Workshop

The ’C3x DSP design workshop is tailored for hardware and software design engineers and decision-makers who design and use the ’C3x generation of DSP devices. Hands-on exercises throughout the course give participants a rapid start in using ’C3x design skills. Microprocessor/assembly language experience is required. Experience with digital design techniques and C language programming experience is desirable. The following topics are covered in the ’C3x workshop:

- ’C3x architecture/instruction set
- Use of the PC-based ’C3x software simulator and EVM
- Floating-point and parallel operations
- Use of the ’C3x assembler/linker
- C programming environment
- System architecture considerations
- Memory and I/O interfacing
- ’C3x development support

For registration, pricing, or enrollment information on this and other TTO TMS320 workshops, call (800) 336–5236, ext. 3904.

11.1.4 TMS320 Literature

Extensive DSP documentation is available, including data sheets, user’s guides, and application reports. In addition, DSP textbooks that aid research and education have been published by Prentice-Hall, John Wiley and Sons, and Computer Science Press. To order literature or to subscribe to the DSP newsletter *Details on Signal Processing* (for up-to-date information on new products and services), call the Literature Response Center at (800)477–8924 or log on to the DSP Solutions web site at http://www.ti.com/dsps.

11.1.5 DSP Hotline

For answers to TMS320 technical questions on device problems, development tools, documentation, upgrades, and new products, you can contact the DSP hotline by:

- Phone at (281) 274–2320 Monday through Friday from 8:30 a.m. to 5:00 p.m. Central Time
- Fax at (281) 274–2324
- Electronic mail at dsph@ti.com
- European fax at 33–1–3070–1032
- Semiconductor Product Information Center (PIC) at (214) 644–5580

To ask about third-party applications and algorithm development packages, contact the third party directly. See the *TMS320 Third-Party Support Reference Guide* for addresses and phone numbers.

The DSP hotline does not provide pricing information. Contact the nearest TI field sales office or the TI PIC for prices and availability of TMS320 devices and support tools.

11.1.6 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. The BBS also gives information about 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 by:

- Modem: (300-, 1200-, or 2400-bps) dial (713)274–2323. Set your modem to 8 data bits,1 stop bit, no parity.
- Internet: Use anonymous *ftp* to *stp.ti.com* (Internet port address 192.94.94.1). The BBS content is located in the subdirectory called *mirrors*.

To find out more about the BBS, see the *TMS320 Family Development Support Reference Guide*.

11.2 TMS320C3x Part Ordering Information

This section provides device and support tool part numbers. Table 11–1 lists the part numbers for the ’C30 and ’C31; Table 11–2 gives ordering information for ’C3x hardware and software support tools. An explanation of the TMS320 family device and development support tool prefix and suffix designators follows the two tables to assist in understanding the TMS320 product numbering system.

Table 11–1. TMS320C3x Digital Signal Processor Part Numbers

Device	Technology	Operating Frequency	Package Type	Typical Power Dissipation
TMS320C30GEL	0.8-µm CMOS	33 MHz	Ceramic 181-pin PGA	1.00 W
TMS320C30GEL40	0.8-µm CMOS	40 MHz	Ceramic 181-pin PGA	1.25 W
TMS320C31PQL/PQA	0.8-µm CMOS	33 MHz	Plastic 132-pin QFP	0.75 W
TMS320C31PQL40	0.8-µm CMOS	40 MHz	Plastic 132-pin QFP	0.90 W
TMS320LC31PQL	0.8-µm CMOS	33 MHz	Plastic 132-pin QFP	0.50 W
TMS320C31PQL50	0.8-µm CMOS	50 MHz	Plastic 132-pin QFP	1.00 W
SMJ320C316FA27	0.8-µm CMOS	28 MHz	Ceramic 141-pin PGA	0.60 W
SMJ320C31HF627			Ceramic 132-pin QFP	0.60 W
SMJ320C316FA33			Ceramic 141-pin PGA	0.75 W
SMJ320C316HF633			Ceramic 132-pin PGA	0.75 W
SMJ320C306BM33	0.8-µm CMOS	33 MHz	Ceramic 181-pin PGA	1.10 W
SMJ320C30HF633			Ceramic 196-pin QFP	
SMJ320C30GBM28	0.8-µm CMOS	28 MHz	Ceramic 181-pin PGA	1.00 W
SMJ320C30HF628			Ceramic 196-pin QFP	1.00 W
SMJ320C30HTM28				
SMJ320C30GBM25	0.8-µm CMOS	25 MHz	Ceramic 181-pin PGA	1.00 W
SMJ320C30HF625			Ceramic 196-pin QFP	1.00 W
SMJ320C30HTM25				

Table 11–2. TMS320C3x Support Tool Part Numbers

(a) Software

Tool Description	Operating System	Part Number
C Compiler & Macro Assembler/ Linker	VAX/VMS	TMDS3243255-08
	PC-DOS/MS-DOS	TMDS3243855-02
	SPARC (Sun OS)†	TMDS3243555-08
Assembler/Linker	PC-DOS/MS-DOS; OS/2	TMDS3243850-02
Simulator	VAX VMS	TMDS3243251-08
	PC-DOS/MS-DOS	TMDS3243851-02
	SPARC (SUN OS) †	TMDS3243551-09
Digital Filter Design Package	PC-DOS	DFDP
TMS320C3x Emulation Porting Kit	PC; SPARC	TMDX3240030

(b) Hardware

Tool Description	Operating System	Part Number
XDS510 Emulator	PC/MS-DOS	TMDS3240130
Evaluation Module (EVM)	PC/MS-DOS	TMDS3260030

† Note that SUN UNIX supports 'C3x software tools on the 68 000 family-based SUN-3 series workstations and on the SUN-4 series machines that use the SPARC processor, but not on the SUN-386i series of workstations.

11.2.1 Device and Development Support Tool Prefix Designators

Prefixes to TI part numbers designate phases in the product's development stage for both devices and support tools, as shown in the following definitions:

Device Development Evolutionary Flow

- **TMX:** Experimental device that is not necessarily representative of the final device's electrical specifications
- **TMP:** Final silicon device that conforms to the device's electrical specifications but has not completed quality and reliability verification
- **TMS:** Fully qualified production device

Support Tool Development Evolutionary Flow

- **TMDX:** Development support product that has not yet completed TI's internal qualification testing for development systems
- **TMDS:** 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."

Note: Prototype Devices

TI recommends that prototype devices (TMX or TMP) not be used in production systems. Their expected end-use failure rate is undefined but predicted to be greater than standard qualified production devices.

TMS devices and TMDS development support tools have been fully characterized, and their quality and reliability have been fully demonstrated. TI's standard warranty applies to TMS devices and TMDS development support tools.

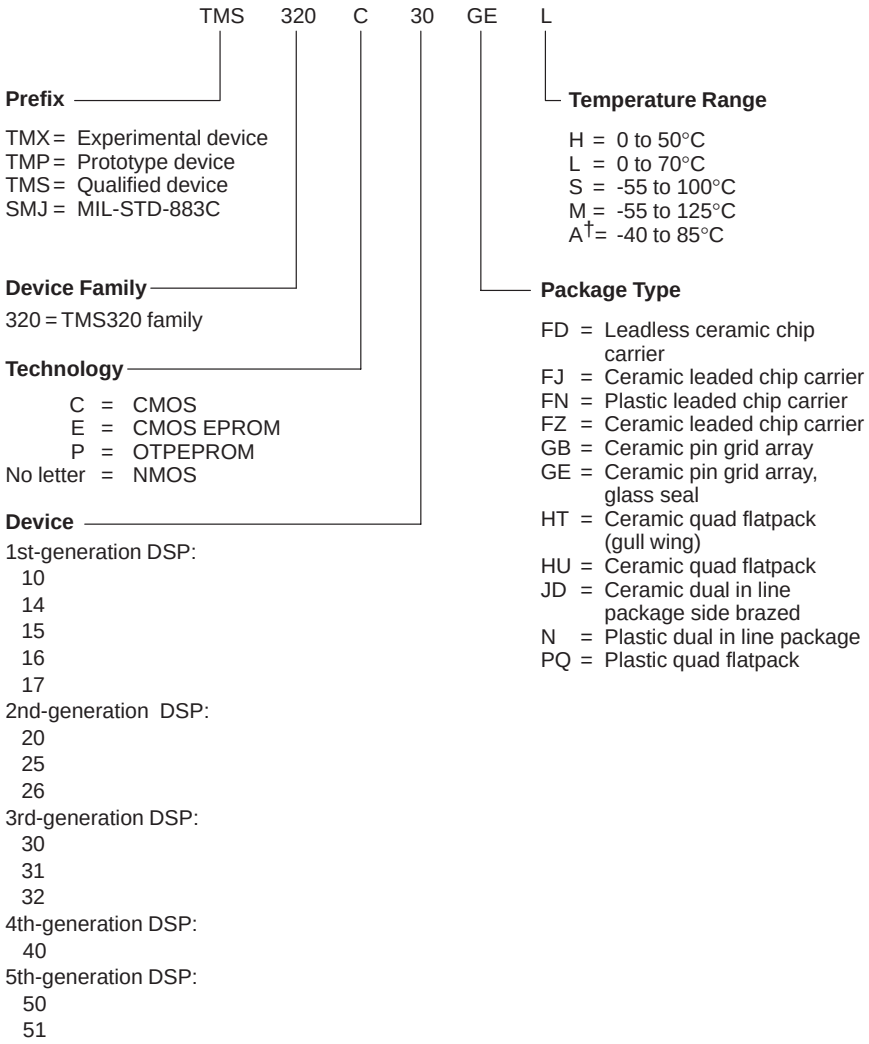
TMDX development support products are intended for internal evaluation purposes only. They are covered by TI's warranty and update policy for microprocessor development systems products; however, they should be used by customers only with the understanding that they are developmental in nature.

11.2.2 Device Suffixes

The suffix indicates the package type (for example, N, FN, or GE) and temperature range (for example, L).

Figure 11–1 presents a legend for reading the complete device name for any TMS320 family member.

Figure 11–1. TMS320 Device Nomenclature



† See electrical specifications for 'C31 PQA case temperature ratings

Chapter 12

TMS320C30 Power Dissipation

This chapter presents the information necessary to determine the requirements for the power supply current for the ’C30 under different operating conditions.

As device sophistication and levels of integration increase with evolving semiconductor technologies, actual levels of power dissipation vary widely. These levels depend heavily on the particular application in which the device is used and the nature of the program being executed. In addition, due to the characteristics of CMOS technology, power requirements vary according to clock rates and data values being processed. Using this information, you can determine the device’s power dissipation and, in turn, calculate thermal management requirements.

Topic **Page**

12.1 Power Dissipation Characteristics

Generally, power supply current requirements are related to the system, for example, operating frequency, supply voltage, temperature, and output load. As devices become more complex, the specification must also be based on what the device does. CMOS devices inherently draw current only during switching through the linear region. Therefore, the power supply current is related to the rate of switching. Furthermore, since the output drivers of the 'C30 are specified to drive direct current (dc) loads, the power supply current resulting from external writes depends not only on switching rate but also on the value of data written.

12.1.1 Power Supply Factors

The power-supply current consists of four basic factors:

- Quiescent current
- Internal operations
- Internal bus operations
- External bus operations

12.1.2 Power Supply Consumption Dependencies

The power-supply current consumption depends on many factors. Four are system-related:

- Operating frequency
- Supply voltage
- Operating temperature
- Output load

Several other factors are related to 'C30 operation. They include:

- Duty cycle of operations
- Number of buses used
- Wait states
- Cache usage
- Data value of internal and external bus

The total power supply current for the device is described in the following equation, which applies the four basic power supply current factors and the dependencies described above:

$$I = (I_q + I_{iops} + I_{ibus} + I_{xbus}) \times FV \times T$$

where:

I_q = quiescent current

I_{iops} = current from internal operations

I_{ibus} = current from internal bus usage, including data value and cycle time dependencies

I_{xbus} = current from external bus usage, including data value, wait state, cycle time, and capacitive load dependencies

FV = scale factor for frequency and supply voltage

T = scale factor for operating temperature

The application of this equation and the determination of all of the dependencies are described in detail in this chapter.

If a less detailed analysis is sufficient, use the minimum, typical, and maximum values to determine a rough estimate of the power supply current requirements:

- The minimum power supply current requirement is 110 mA.
- The typical and average current consumption is 200 mA, as described in the *TMS320C30 Digital Signal Processor* data sheet. These are associated with most algorithms running on the device unless data output is excessive.
- If an extremely conservative approach is desired, use the maximum value.

Maximum Current Requirement

The maximum current requirement is 600 mA and occurs only under worst case conditions. These include writing alternating data (AAAAAAAAh to 55555555h) out of both external buses simultaneously, every cycle, with 80 pF loads, and running at 33 MHz.

12.1.3 Determining Algorithm Partitioning

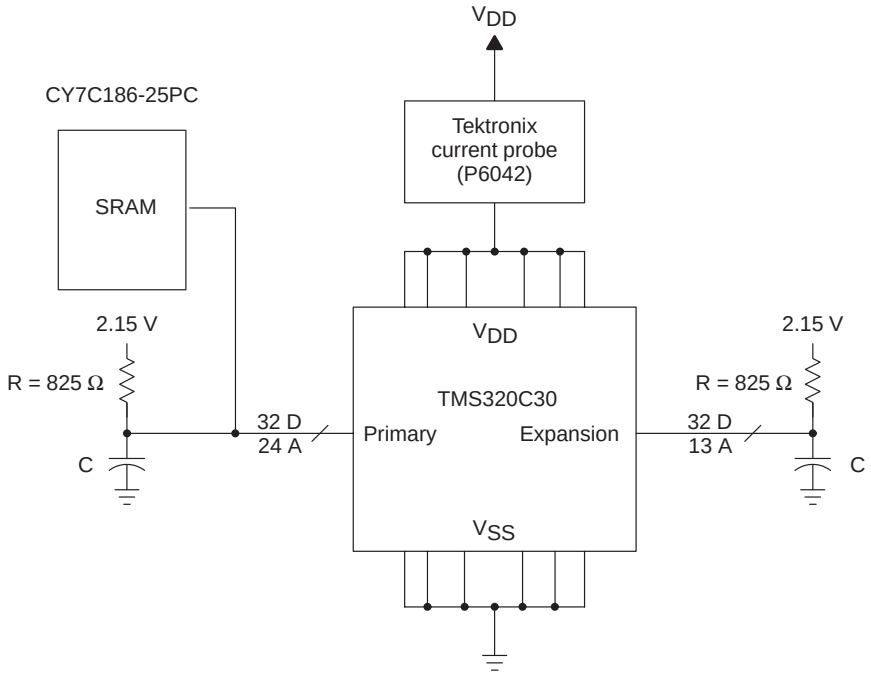
Each part of an algorithm has its own pattern with respect to internal and external bus usage. To analyze the power supply current requirement, you must partition an algorithm into segments with distinct concentrations of internal or external bus usage. Analyze each program segment to determine its power supply current requirement. You can then calculate the average power supply current from the requirements of each segment of the algorithm.

12.1.4 Test Setup Description

All ’C30 supply current measurements were performed on the test setup shown in Figure 12–1. The test setup consists of a ’C30, 8K words of zero-wait-state Cypress Semiconductor SRAMs (CY7C186–25PC), and resistor/capacitor (RC) loads on all data and address lines. A Tektronix™ current probe (P6042) measures the power supply current in all V_{DD} lines of the device. The supply voltage on the output load is 2.15 V. Unless otherwise specified, all measurements are made at a:

- Supply voltage of 5.0 V
- Input clock frequency of 33 MHz
- Capacitive load of 80 pF
- Operating temperature of 25°C

Figure 12–1. Current Measurement Test Setup for the TMS320C30



12.2 Current Requirements for Internal Circuitry

The power supply current requirement for internal circuitry consists of the following factors: quiescent current, internal operations, and internal bus operations. Quiescent current and internal operations are constants, but the internal bus operations vary with the rate of internal bus usage and the data values being transferred.

12.2.1 Quiescent Current

Quiescent current refers to the baseline supply current drawn by the ’C30 during minimal internal activity. It includes the current required to fetch an instruction from on- or off-chip memory. Examples of quiescent current include:

- Maintaining timers and serial ports
- Executing the IDLE instruction
- ’C30 in HOLD mode pending external bus access
- ’C30 in reset
- Branching to self

The quiescent requirement for the ’C30 equals 110 mA.

12.2.2 Internal Operations

Internal operations include register-to-register multiplication, ALU operations, and branches. It does not include external bus usage or significant internal bus usage. Internal operations add a constant 55 mA above the quiescent current. Therefore, the total contribution of quiescent current (110 mA) and internal operations (55 mA) is 165 mA. During an RPTS instruction (repeat single instruction), activity other than the instruction being repeated is suspended; therefore, internal power supply current is related only to the operation performed by the instruction being executed.

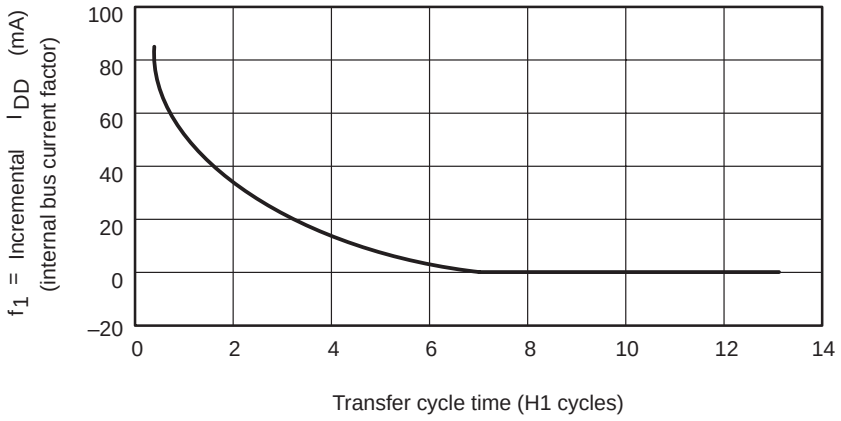
12.2.3 Internal Bus Operations

Internal bus operations include all operations that use the internal buses extensively, such as internal RAM access every cycle. No distinction is made between internal reads (such as instruction or operand fetches from internal ROM or internal RAM banks) and internal writes (such as operand stores to internal RAM banks); internally they are equal. Since power consumption depends on the data value in the internal bus, significant use of internal buses adds a data-dependent factor to the 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 ’C30. Therefore, you must determine the algorithm’s internal usage in order to accurately calculate the power supply current requirements. The ’C30 software simulator and XDS™ emulator both provide benchmarking and timing capabilities that help you determine bus usage.

The current resulting from internal bus usage varies exponentially with transfer rates. Figure 12–2 shows the internal bus current requirements for transferring alternating data (AAAAAAAAh to 55555555h). A transfer rate less than 1 implies multiple accesses per single H1 cycle (that is, using direct memory access (DMA), etc.). Transfer cycle times greater than 1 refer to single-cycle transfers with one or more cycles between them. The minimum transfer cycle time is one third, which corresponds to three accesses in a single H1 cycle.

Figure 12–2. Internal Bus Current Versus Transfer Rate (AAAAAAAAh to 55555555h)



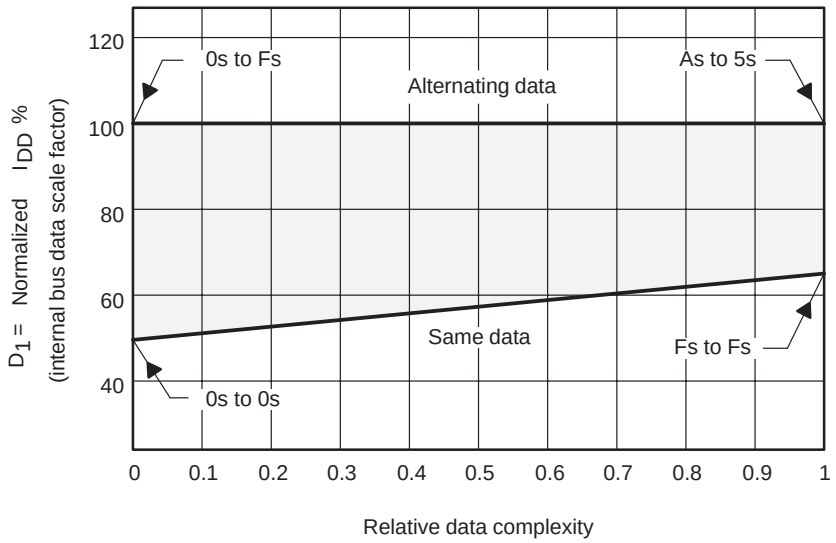
The data set AAAAAAAAh to 55555555h exhibits the maximum current for these types of operations. Less current is required for transferring other data patterns, and current values can be derated accordingly.

As the transfer rate decreases (transfer cycle time increases), the incremental I_{DD} approaches 0 mA. Transfer rates corresponding to more than seven H1 cycles do not add any current and are considered insignificant. This figure represents the incremental I_{DD} from 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 or one-third H1 transfer cycle time. At this rate, 85 mA is added to the quiescent (110 mA) and internal operation (55 mA) current values for a total of 250 mA.

Figure 12–3 shows the data dependence of the internal bus current requirement when the data is other than As followed by 5s. The shaded trapezoidal region represents the internal bus current consumed for all possible data values transferred. The lower line represents the scale factor for transferring the same data (all 0s or all Fs). The upper line represents the scale factor for transferring alternating data (all 0s to all Fs or all As to all 5s).

Figure 12–3. Internal Bus Current Versus Data Complexity Derating Curve



The number of possible permutations of data values is quite large. The extent to which data varies is referred to as relative data complexity. This term 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. Relative data complexity ranges from 0, signifying minimal variation of data, to a normalized value of 1, signifying greatest data variation.

If a statistical knowledge of the data exists, Figure 12–3 can be used to determine the exact power supply requirement according to internal bus usage. For example, Figure 12–3 indicates a 63% scale factor when all Fs are moved internally every cycle with two accesses per cycle. This scale factor is multiplied by 55 mA (from Figure 12–2, at one-half H1 cycle transfer time), yielding 34.65 mA because of internal bus usage. Therefore, an algorithm running under these conditions requires about 200 mA of power supply current (110 + 55 + 34.65).

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 0.80 and can be used as an estimate of a nominal scale factor. You can use this nominal data scale factor of 80% for internal bus data dependency, adding 44 mA to 110 mA (quiescent current) and 55 mA (internal operations) to yield 210 mA. As an upper bound, assume worst case conditions of three accesses of alternating data every cycle, adding 85 mA (from Figure 12–2) to 110 mA (quiescent current) and 55 mA (internal operations) to yield 250 mA.

12.3 Current Requirement for Output Driver Circuitry

The output driver circuits on the ’C30 are required to drive significantly higher dc and capacitive loads than internal device logic. Therefore, they are designed to drive larger currents than internal devices. Because of this, output drivers impose higher supply current requirements than other sections of circuitry on the device.

Accordingly, the highest values of supply current are required when external writes are performed at high speed. During reads, or when the external buses are not in use, the ’C30 does not drive the data bus; this eliminates the most significant factor of output buffer current. Furthermore, in typical cases, only a few address lines change, or the whole address bus is static. Under these conditions, an insignificant amount of supply current is consumed. When no external writes are performed or when writes are performed infrequently, current from output buffer circuitry can be ignored.

When external writes are performed, the current required to supply the output buffers depends on several factors:

- Data pattern transferred
- Rate at which transfers are made
- Number of wait states implemented (because wait states affect rates at which bus signals switch)
- External bus dc and capacitive loading

External operations involve writes external to the device and constitute the major power supply current factor. The power supply current for the external buses is made up of three factors and is summarized in the following equation:

$I_{base} + I_{prim} + I_{exp}$ = power supply current for the external buses

where:

I_{base} = 60-mA baseline current

I_{prim} = primary bus current

I_{exp} = expansion bus current

The remainder of this section describes in detail the calculation of external bus current factors.

12.3.1 Primary Bus Current

The current from primary bus writes varies with both wait states and write cycle time. Current factors from output driver circuitry are represented as offsets from the previously computed value (quiescent + internal operations + internal bus). Since the baseline value is related to internal current factors, negative values for current offset are obtained under some circumstances. However, negative current does not occur.

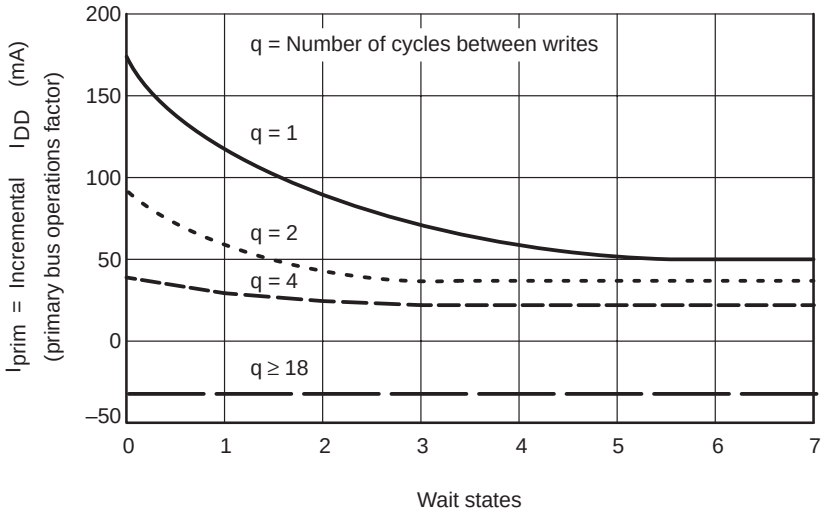
To obtain accurate current values, you must first establish the timing of write cycles of the buses. To determine the rate and timings at which write cycles to the external buses occur, you must analyze program activity, including any pipeline conflicts that may exist. Information from this manual and the 'C30 emulator or simulator is useful in making these determinations. You must account for the effects of cache use in these analyses because the cache can affect whether instructions are fetched from external memory.

When evaluating external write activity in a given program segment, you must consider whether a particular level of external write activity is significant. If writes are performed at very slow rates on both the primary and the expansion buses, the current from external writes can be ignored . If writes are performed at high speed on only one of the two external buses, you should calculate current requirements.

Although you can obtain negative incremental current values under some circumstances, the total contribution for external buses, including baseline current, is always positive. When external buses are not used much, the total current requirements approach the current contribution from the internal factors, which is solely a function of internal activity. This places a lower limit on current contributions from the primary and expansion buses, because the total current from external buses is the sum of the 60-mA baseline value and the primary and expansion bus factors. This effect is discussed in further detail in the rest of this section.

Once you establish bus-write cycle timing, use Figure 12–4 to determine the contribution to supply current from this bus activity. Figure 12–4 shows current contributions from the primary bus for various numbers of wait states and H1 cycles between writes. This current contribution is exhibited when writes of alternating 55555555h and AAAAAAAAh are performed at a capacitive load of 80 pF per output signal line. This condition exhibits the highest current values on the device. The curve in the figure represents incremental or additional current contributed by the primary bus output driver circuitry while writing alternating 55555555h and AAAAAAAAh. Current values obtained from this graph are scaled and added to several other current values to calculate the total current for the device. As indicated in the figure, the lower curve represents the current contribution for 18 or more cycles between writes.

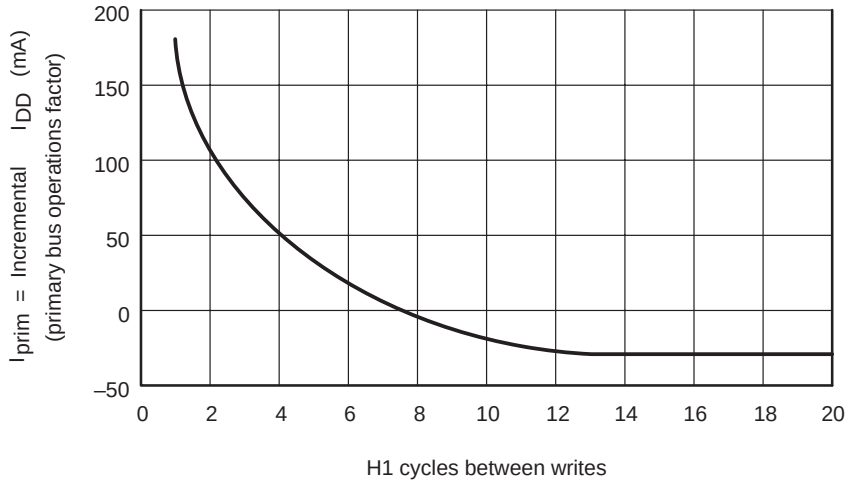
Figure 12–4. Primary Bus Current Versus Transfer Rate and Wait States



The number of cycles between writes refers to the number of H1 cycles between the active portion of the write cycles (as defined in the *TMS320C30 Digital Signal Processor* data sheet), that is, when $\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$, or $\overline{\text{IOSTRB}}$ and R/$\overline{\text{W}}$ (or XR/$\overline{\text{W}}$, as the case may be) are low between H1 cycles. As shown in Figure 12–4, the minimum number of cycles between writes is 1, because with back-to-back writes there is one H1 cycle between active portions of the writes.

To further illustrate the relationship between current and write cycle time, Figure 12–5 shows the characteristics of current for various numbers of cycles between writes for zero wait states. You can use the information on this curve to obtain more precise values of current if zero wait states are used and the number of cycles between writes does not fall on one of the curves in Figure 12–4.

Figure 12–5. Primary Bus Current Versus Transfer Rate at Zero Wait States



Although these graphs contain negative current values, negative current has not necessarily actually occurred. The negative values exist because the graphs represent a current offset from the previously computed current value. Using this approach to depict current contributions from different factors breaks down the current calculations to allow you to make calculations independently.

Figure 12–4 and Figure 12–5 show that the current consumption during external bus writes is negative if writes are performed at intervals of more than 18 cycles. Under these conditions, use the incremental value of –30-mA current contribution from the primary bus. You should use a value of –30 mA only if the expansion bus is used extensively because the total contribution for external buses, including baseline current, must always be positive. If the expansion bus is not used and the primary bus is not used much, the current contribution from the primary bus is always greater than or equal to 20 mA. This ensures that the correct total current value is obtained when summing external bus factors. Once a current value has been obtained from Figure 12–4 or Figure 12–5, this value can, if necessary, be scaled by a data dependency factor, as described in section 12.3.3 on page 12-14. This scaled value is then summed along with several other current values to determine the total supply current.

12.3.2 Expansion Bus Current

Currents from the primary and expansion buses differ slightly for several reasons, including the fact that the expansion bus has 11 fewer address outputs than the primary bus (13 rather than 24). This overall current contribution is slightly lower from the expansion bus than from the primary bus.

Determining the expansion bus current uses the same premise as determining the primary bus current. Figure 12–6 and Figure 12–7 show the same current relationships for the expansion bus as Figure 12–4 and Figure 12–5 show for the primary bus. The total external buses' current contributions must be positive; if the primary bus is not used and the expansion bus is not used much, the minimum current contribution from the expansion bus is –30 mA. The current values obtained from these figures must be scaled by a data dependency factor, as described in section 12.3.3 on page 12-14.

Figure 12–6. Expansion Bus Current Versus Transfer Rate and Wait States

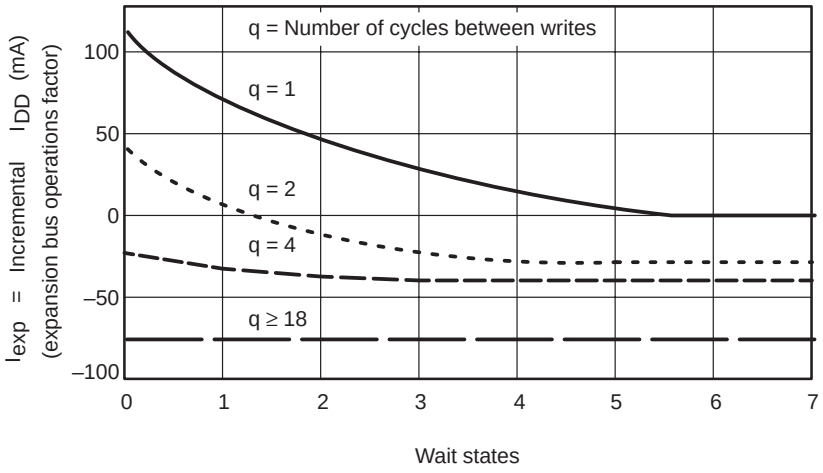
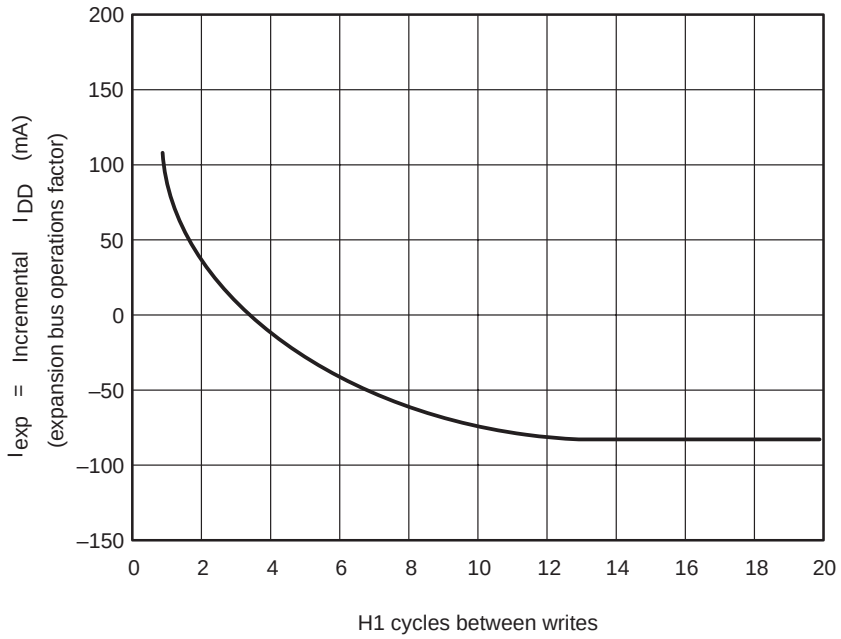


Figure 12–7. Expansion Bus Current Versus Transfer Rate at Zero Wait States



12.3.3 Data Dependency Factors

Data dependency of current for the primary and expansion buses is expressed as a scale factor that is a percentage of the maximum current of either of the two buses. Data dependencies are shown in Figure 12–8 for the primary bus and in Figure 12–9 for the expansion bus.

These two figures show normalized weighting factors that you can use 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 can be seen from Figure 12–8 and Figure 12–9, the minimum current is exhibited by writing all 0s, while the maximum current occurs when writing alternating 55555555h and AAAAAAAAh. This condition results in a weighting factor of 1, which corresponds to using the values from Figure 12–4 and/or Figure 12–5 directly.

As with internal bus operations, data dependencies for the external buses are well defined, but accurate prediction of data patterns is often impractical. Unless you have precise knowledge of data patterns, you should use an estimate of a median or average value for scale factor. If you assume that data is neither 5s and As, nor all 0s, and varies randomly, a value of 0.85 is appropriate. Otherwise, if you prefer a conservative approach, you can use a value of 1.0 as an upper bound.

Figure 12–8. Primary Bus Current Versus Data Complexity Derating Curve

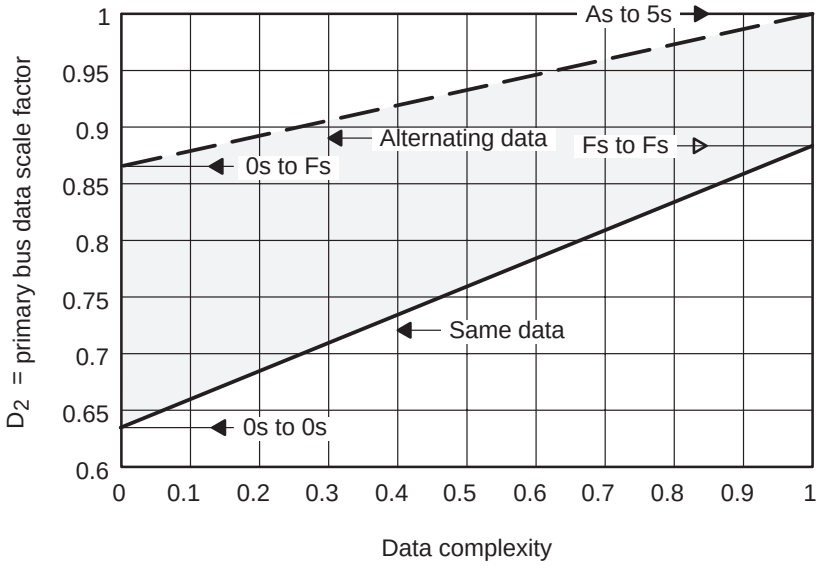
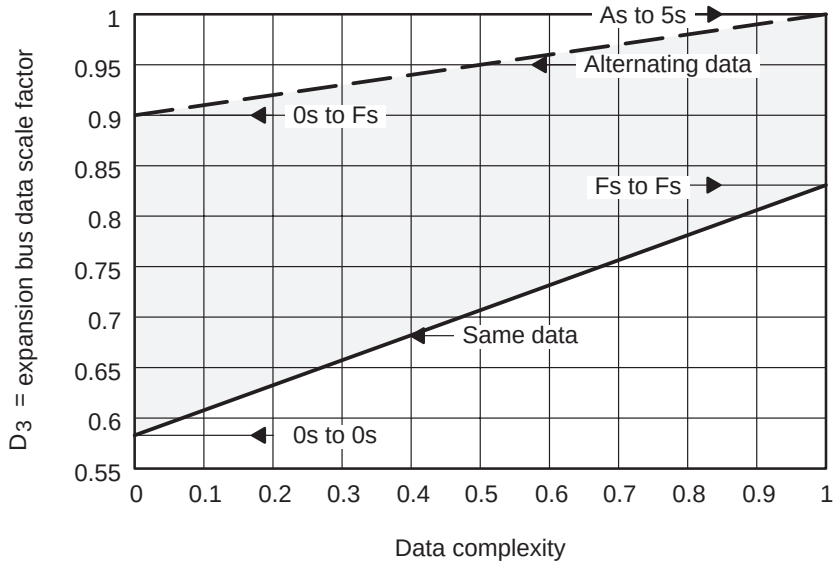


Figure 12–9. Expansion Bus Current Versus Data Complexity Derating Curve



Regardless of the approach you take for scaling, once you determine the scale factors for primary and expansion buses, apply these scale factors to the current values found by using the graphs in the previous two sections. For example, if a nominal scale factor of 0.85 is used and the system uses zero wait states with two cycles between accesses on both the primary and expansion buses, the current contribution from the two buses is as follows:

Primary: 0.85×80 mA = 68 mA
Expansion: 0.85×40 mA = 34 mA

12.3.4 Capacitive Load Dependence

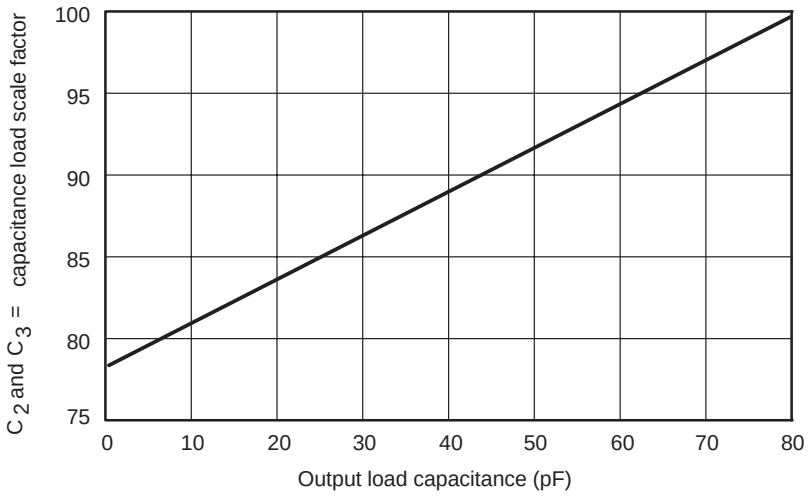
Once you account for cycle timing and data dependencies, calculate and apply the capacitive loading effects. Figure 12–10 shows the scale factor to apply to the current values obtained above as a function of actual load capacitance if the load capacitance presented to the buses is less than 80 pF.

In the previous example, if the load capacitance is 20 pF instead of 80 pF, a scale factor of 0.84 is used, yielding:

Primary: 0.84×68 mA = 57.12 mA
Expansion: 0.84×34 mA = 28.56 mA

The slope of the load capacitance line in Figure 12–10 is 26% normalized I_{DD} per pF. While this slope may be used to interpolate scale factors for loads greater than 80 pF, the ’C30 is specified to drive output loads of less than 80 pF. Interface timings cannot be ensured at higher loads.

Figure 12–10. Current Versus Output Load Capacitance



12.4 Calculation of Total Supply Current

The previous sections discuss currents contributed by several sources on the ’C30. Because actual current values are unique and independent for each source, each current source is discussed separately. In an actual application, however, the sum of the independent contributions from each current determines the total current requirement for the device. This current value is 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 must be connected in parallel to a 5-volt source and use ground planes with as little impedance as possible.

12.4.1 Combining Supply Current from All Factors

To determine the total supply current requirements for any given program activity, calculate each of the appropriate factors and combine them in the following sequence:

1) Start with 110-mA quiescent current.

2) Add 55 mA for internal operations unless the device is dormant. Dormant periods occur during the execution of IDLE, NOPs, branches to self, or performance of internal and/or external bus operations using an RPTS instruction (see section 12.2.2 on page 12-5). Internal or external bus operations executed through RPTS do not contribute an internal operations power supply current factor. However, current factors in the next two steps may still be required, even though the 55 mA is omitted.

3) If significant internal bus operations are performed, add the calculated current value. (See section 12.2.3 on page 12-5.)

4) If external writes are performed at high speed, add 60 mA and then add the values for primary and expansion bus current factors. (See section 12.3 on page 12-9.) If only one external bus is used, the appropriate incremental current for the unused bus must still be included because the current offsets include factors required for operating both buses. The total current contribution for external buses, including baseline, is always positive.

The current value obtained from summing these factors is the total device current requirement for a given program activity.

12.4.2 Supply Voltage, Operating Frequency, and Temperature Dependencies

Current dependencies specific to each supply current factor (such as internal or external bus operations) are discussed in section 12.1.2 on page 12-2. Supply voltage level, operating temperature, and operating frequency affect the requirements for the total supply current and must be maintained within the required device specifications.

Once you determine the total current for a particular program segment, the dependencies that affect the total current requirements are applied as a scale factor in the same manner as data dependencies discussed in other sections. Figure 12–11 shows the relative scale factors for the supply current values as a function of both V_{DD} and operating frequency.

Power supply current consumption does not vary significantly with operating temperature. However, a scale factor of 2% normalized I_{DD} per 50°C change in operating temperature may be used to derate current within the specified range noted in the *TMS320C30 Digital Signal Processor* data sheet. This temperature dependence is shown graphically in Figure 12–12. A temperature scale factor of 1.0 corresponds to current values at 25°C, which is the temperature for all references in the document.

Figure 12–11. Current Versus Frequency and Supply Voltage

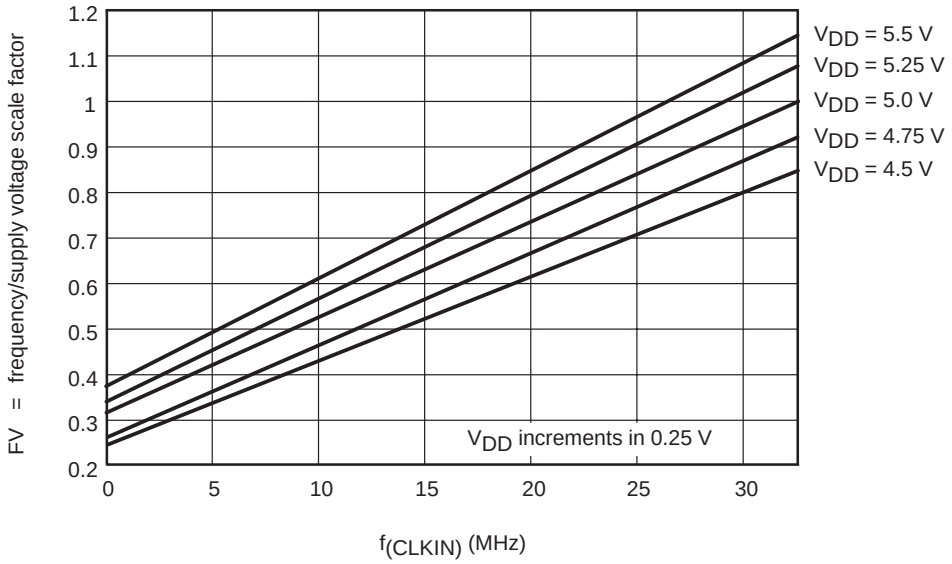
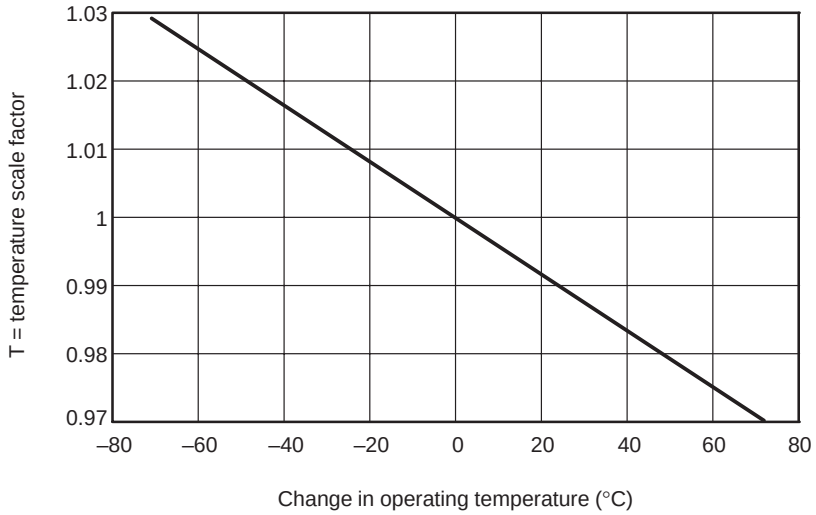


Figure 12–12. Current Versus Operating Temperature Change



12.4.3 Total Current Equation Example

The procedure for determining the power supply current requirement is summarized in the following equation:

$$I = (I_q + I_{iops} + I_{ibus} + I_{xbus}) \times FV \times T$$

where:

I_q = 110 mA

I_{iops} = 55 mA

$I_{ibus} = D_1 \times f_1$ (see Table 12–1 on page 12-20)

$I_{xbus} = I_{base} + I_{prim} + I_{exp}$

with

I_{base} = 60 mA

$I_{prim} = D_2 \times C_2 \times F_2$ (see Table 12–1)

$I_{exp} = D_3 \times C_3 \times F_3$ (see Table 12–1)

FV = scale factor for frequency and supply voltage

T = scale factor for operating temperature

Table 12–1 describes the variables used in the power supply current equation. The table displays figure numbers from which the value can be obtained.

Table 12–1. Current Equation Variables

Variable	Description	Graph/Value
I_q	Quiescent current	110 mA
I_{iops}	Internal operations current	55 mA
I_{ibus}	Internal bus operations current	†
D_1	Internal bus data scale factor	Figure 12–3
f_1	Internal bus current requirement	Figure 12–2
I_{xbus}	External bus operations current	†
I_{base}	External bus base current	60 mA
I_{prim}	Primary bus operations current	†
D_2	Primary bus data scale factor	Figure 12–8
C_2	Primary bus capacitance load scale factor	Figure 12–10
f_2	Primary bus current requirement	Figure 12–4 or Figure 12–5
I_{exp}	Expansion bus operations current	†
D_3	Expansion bus data scale factor	Figure 12–9
C_3	Expansion bus capacitance load scale factor	Figure 12–10
f_3	Expansion bus current requirement	Figure 12–6 or Figure 12–7
FV	Frequency/supply voltage scale factor	Figure 12–11
T	Temperature scale factor	Figure 12–12

† See power supply current equation on page 12-19.

12.4.4 Peak Versus Average Current

If current is observed over the course of an entire program, some segments usually exhibit significantly different levels of current required for different durations of time. For example, a program may spend 80% of its time performing internal operations, drawing a current of 250 mA; it may spend the remaining 20% of its time performing writes at full speed to the expansion bus, drawing 300 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 of high peak current are short in duration. Average current can be obtained by performing a weighted sum of the currents from the various independent program segments over time. In the example above, the average current can be calculated as follows:

$$I = 0.8 \times 250\ mA + 0.2 \times 300\ mA = 260\ mA$$

Using this approach, you can calculate average current for any number of program segments.

12.4.5 Thermal Management Considerations

Heating characteristics of the ’C30 depend on power dissipation, which in turn depends on power supply current. When you make thermal management calculations, you must consider how power supply current contributes to power dissipation and to the time constant of the ’C30 package thermal characteristics.

Depending on sources and destinations of current on the device, some current contributions to I_{DD} do not constitute a factor of power dissipation at 5 V. Accordingly, if you use the total current flowing into V_{DD} to calculate power dissipation at 5 V, you obtain erroneously large values for power dissipation. Power dissipation is defined as:

$$P = I \times V$$

where:

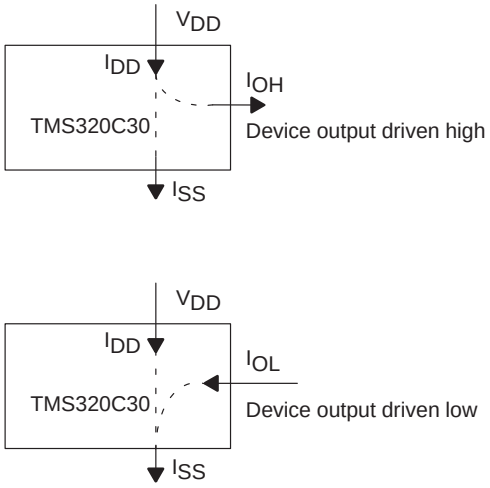
P = power

I = current

V = voltage

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 factors out of the total supply current value; then calculate their contribution to power dissipation separately and add it to the total power dissipation (see Figure 12–13). If this is not done, these currents resulting from driving a logic high level into a dc load cause unrealistically high power dissipation values. The error occurs because the currents resulting from driving a logic high level into a dc load appears as a portion of the current used to calculate power dissipation from V_{DD} at 5 volts.

Figure 12–13. Load Currents



Furthermore, external loads draw supply-only current when outputs are driven high because, when outputs are in the logic 0 state, the device is sinking current that is supplied from an external source. The power dissipation from this current factor does not have a contribution through I_{DD} but contributes to power dissipation with a magnitude of:

$$P = V_{OL} \times I_{OL}$$

where:

V_{OL} = low-level output voltage

I_{OL} = current being sunk by the output (as shown in Figure 12–13)

The power dissipation factor from outputs that are driven low must be calculated and added to the total power dissipation.

When outputs with dc loads are switched, the power dissipation factors from outputs being driven high and outputs being driven low are averaged and added to the total device power dissipation. You should calculate power factors from dc loading of the outputs separately for each program segment before you calculate average power.

Any unused inputs that are left disconnected may float to a voltage level that causes input buffer circuits to remain in the linear region and therefore contribute a significant factor to power supply current. Accordingly, you should deactivate any unused inputs by grounding them or pulling them high if you desire absolute minimum power dissipation. If you must pull several unused inputs high, pull them high together using one resistor to minimize component count and board space.

When you use power dissipation values to determine thermal requirements, you should use the average power unless the time duration of individual program segments is long. The thermal characteristics of the 'C30 in the 181-pin grid array (PGA) package are exponential in nature, with a time constant of t = 4.5 minutes. When subjected to a change in power, the temperature of the device package will, after 4.5 minutes, reach approximately 63% of the total temperature change. Accordingly, if the time duration of program segments exhibiting high power dissipation values is short (on the order of a few seconds), you can use average power, calculated in the same manner as average current (as described in section 12.4.4 on page 12-20).

Otherwise, you should calculate maximum device temperature on the basis of the actual time duration of the program segments involved. For example, if a particular program segment lasts for seven minutes, you can calculate that a device will reach approximately 80% of the temperature change from the total power dissipation during the program segment.

You can determine average power by calculating the power for each program segment (including the previous considerations) and performing a time average of these values, rather than simply multiplying the average current as determined in the previous section by V_{DD}.

Specific device temperature calculations are made using the 'C30 thermal impedance characteristics in the *TMS320C30 Digital Signal Processor* data sheet.

12.5 Example Supply Current Calculations

A fast Fourier transform (FFT) is a typical DSP algorithm. The FFT code in the example calculation processes data in the RAM blocks and writes the result out to zero-wait-state external SRAM on the primary bus. The program executes out of zero-wait-state external SRAM on the primary bus, and enables the ’C30’s cache. The entire algorithm consists mainly of internal bus operations and includes quiescent current and internal operations. At the end of processing, the 1024 results are written to the primary bus. Therefore, the algorithm exhibits a higher current requirement during the write portion, where the external bus is used significantly.

12.5.1 Processing

The processing portion of the algorithm is 95% of the FFT execution. During this portion, the power supply current is required only for the internal circuitry. Data is processed in several loops. During these loops, two operands are transferred on every cycle. The current required for internal bus operations is 55 mA, (see section 12.2.2 on page 12-5). The data is assumed to be random. A data value scale factor of 0.8 is used from Figure 12–3 on page 12-7. This value scales 55 mA, yielding 44 mA for internal bus operations. Adding 44 mA to the quiescent current requirement and internal operations current requirement yields a current requirement of 209 mA for the major portion of the algorithm.

$I = I_q + I_{iops} + I_{ibus}$

$I = 110\ mA + 55\ mA + (55mA)(0.8) = 209\ mA$

12.5.2 Data Output

The portion of the FFT corresponding to writing out data is approximately 5% of the total processing time. Again, the data being written is assumed to be random. From Figure 12–3 on page 12-7 and Figure 12–8 on page 12-15, scale factors of 0.80 and 0.85 are used for derating from data value dependency for internal and primary buses, respectively. During the data dump portion of the code, a load and store are performed every cycle. The parallel load/store instruction is in an RPTS loop, so there is no contribution from internal operations because the instruction is fetched only once. The only internal contributions are from quiescent current and internal bus operations. Figure 12–5 on page 12-12 indicates a 170-mA current contribution from back-to-back zero-wait-state writes, and Figure 12–7 on page 12-14 indicates a –80-mA contribution when the expansion bus is idle (that is, with more than 18 H1 cycles between writes). The total contribution from this portion of the code is:

$I = I_q + I_{ibus} + I_{xbus}$

or

$I = 110 + (55\ mA)(0.8) + 60\ mA - 80\ mA + (170\ mA)(0.85) = 278.5\ mA$

12.5.3 Average Current

The average current is derived from the two portions of the FFT. The processing portion takes 95% of the time and requires about 210 mA, and the data dump portion takes the other 5% and requires about 280 mA. The average is calculated as:

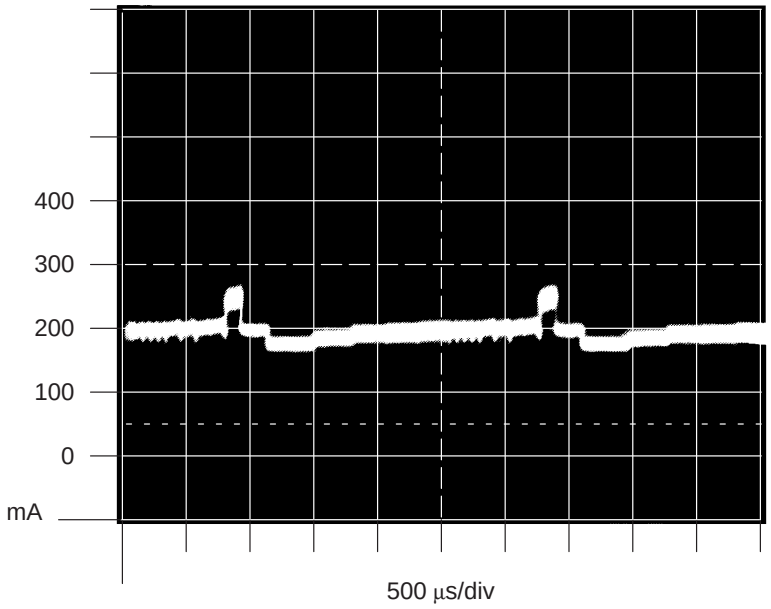
$$I_{avg} = (0.95)(210\ mA) + (0.05)(280\ mA) = 213.5\ mA$$

From the thermal characteristics specified in the ’C30 data sheet, it can be shown that this current level corresponds to a case temperature of 43°C. This temperature meets the maximum device specification of 85°C and, hence, requires no forced air cooling.

12.5.4 Experimental Results

A photograph of the power supply current for the FFT is shown Figure 12–14. During the FFT processing, the measured current varies between 180 and 220 mA. The peak of the current during external writes is 270 mA, and the average current requirement, as measured on a digital multimeter, is 200 mA. The calculations yield results that are extremely close to the actual measured power supply current.

Figure 12–14. Photo of I_{DD} for FFT



Note: Input clock frequency = 33 MHz, voltage level = 5.0 V_{DD}

Appendix A

TMS320C32 Boot Table Examples

The ’C32 boot loader loads programs received from standard memory devices or through the serial port. These programs have a particular data stream structure called a boot table. This appendix shows examples of different ’C32 boot tables in 32-, 16-, and 8-bit-wide ROM that are transmitted through the serial port.

Figure A–1 through Figure A–4 show four instances of the boot table, each containing four blocks. The destination for the first and third block of each boot table is 16-bit $\overline{\text{STRB0}}$ memory. The second block is booted to the 32-bit $\overline{\text{IOSTRB}}$ memory. Block 4 is destined for the 8-bit memory in the $\overline{\text{STRB1}}$ portion of the memory map.

Each figure represents a boot from a different source medium. In Figure A–1, the boot table resides in the 32-bit $\overline{\text{IOSTRB}}$ memory. It is pointed to by the $\overline{\text{INT1}}$ pin low after reset in the microcontroller/boot-loader mode. The boot table in Figure A–2 is stored in the 16-bit $\overline{\text{STRB0}}$ memory (pointed to by $\overline{\text{INT0}}$). The boot table in Figure A–3 resides in the 8-bit $\overline{\text{STRB1}}$ memory (pointed to by $\overline{\text{INT2}}$). The final example, shown in Figure A–4, represents the boot table stored in the host memory before being sent to the ’C32 over the serial port. Unlike the boot from memory, the serial port boot table omits the memory width control word from the beginning of the table.

The shaded areas of the boot table examples represent the contents of the individual blocks of code or data. The unshaded portions are the control words that instruct the boot loader program to transfer the blocks to the memory map.

Figure A–1. Boot From a 32-Bit-Wide ROM to 8-, 16-, and 32-Bit-Wide RAM

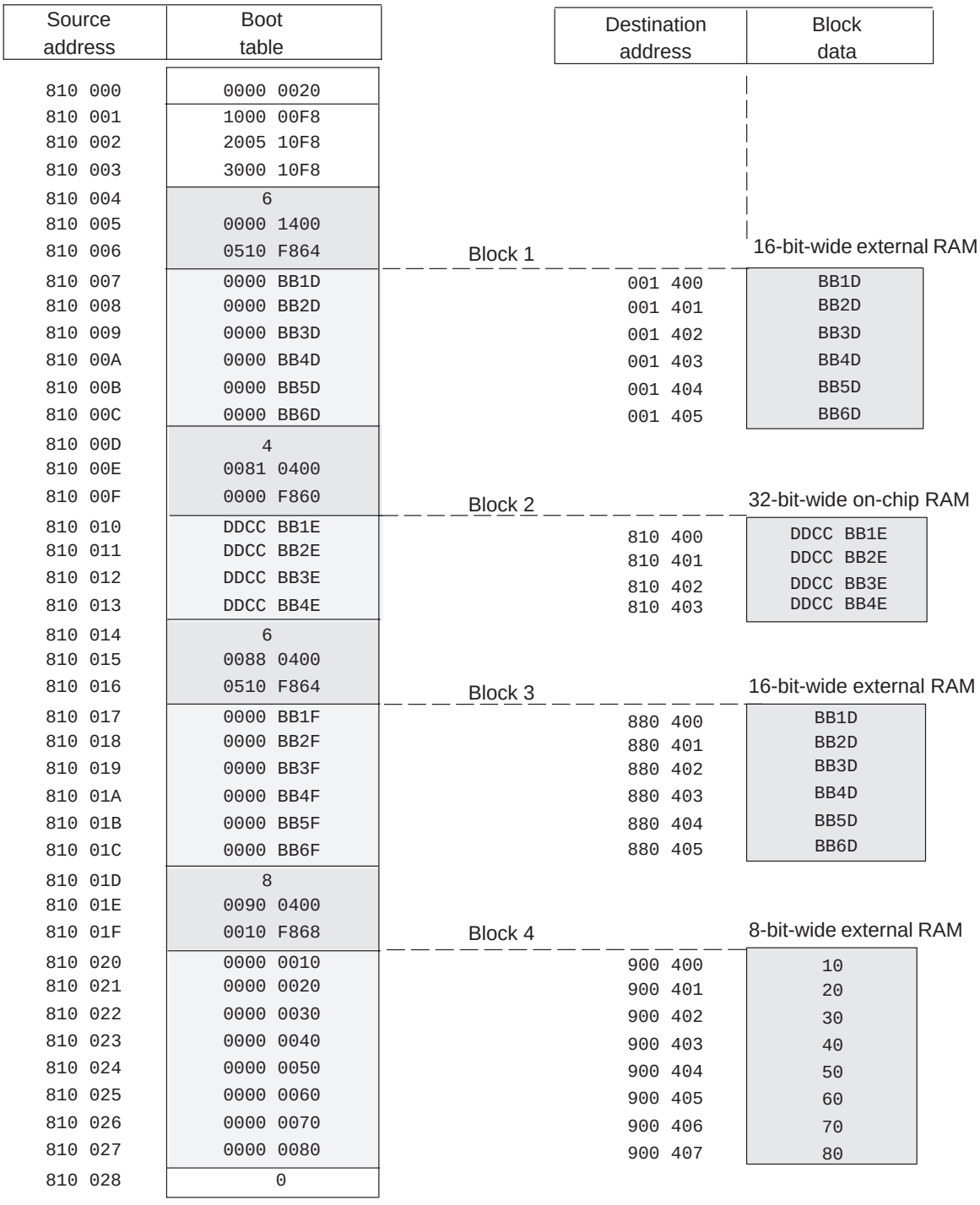


Figure A–2. Boot From a 16-Bit-Wide ROM to 8-, 16-, and 32-Bit-Wide RAM

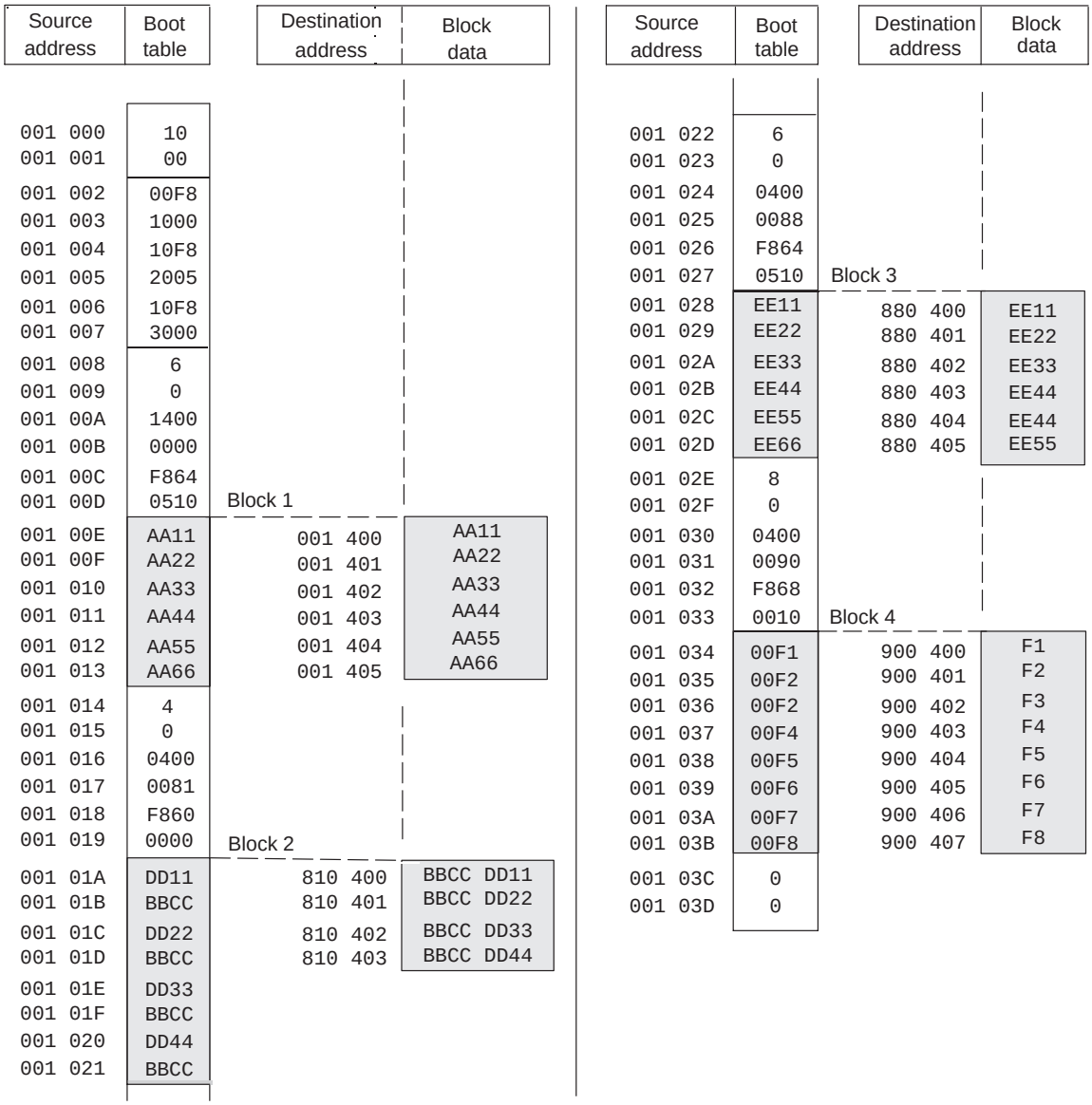


Figure A–3. Boot From a Byte-Wide ROM to 8-, 16-, and 32-Bit-Wide RAM

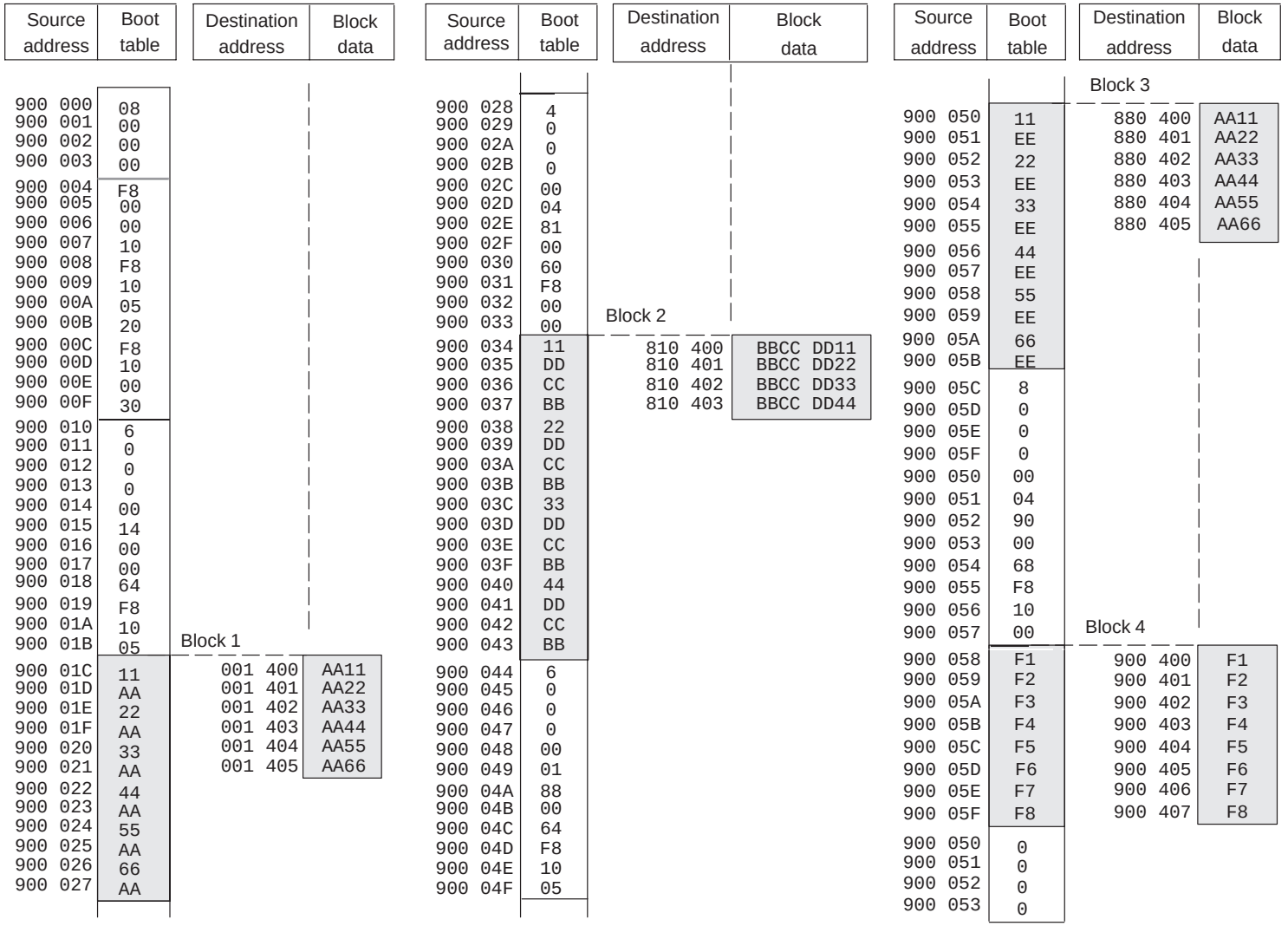
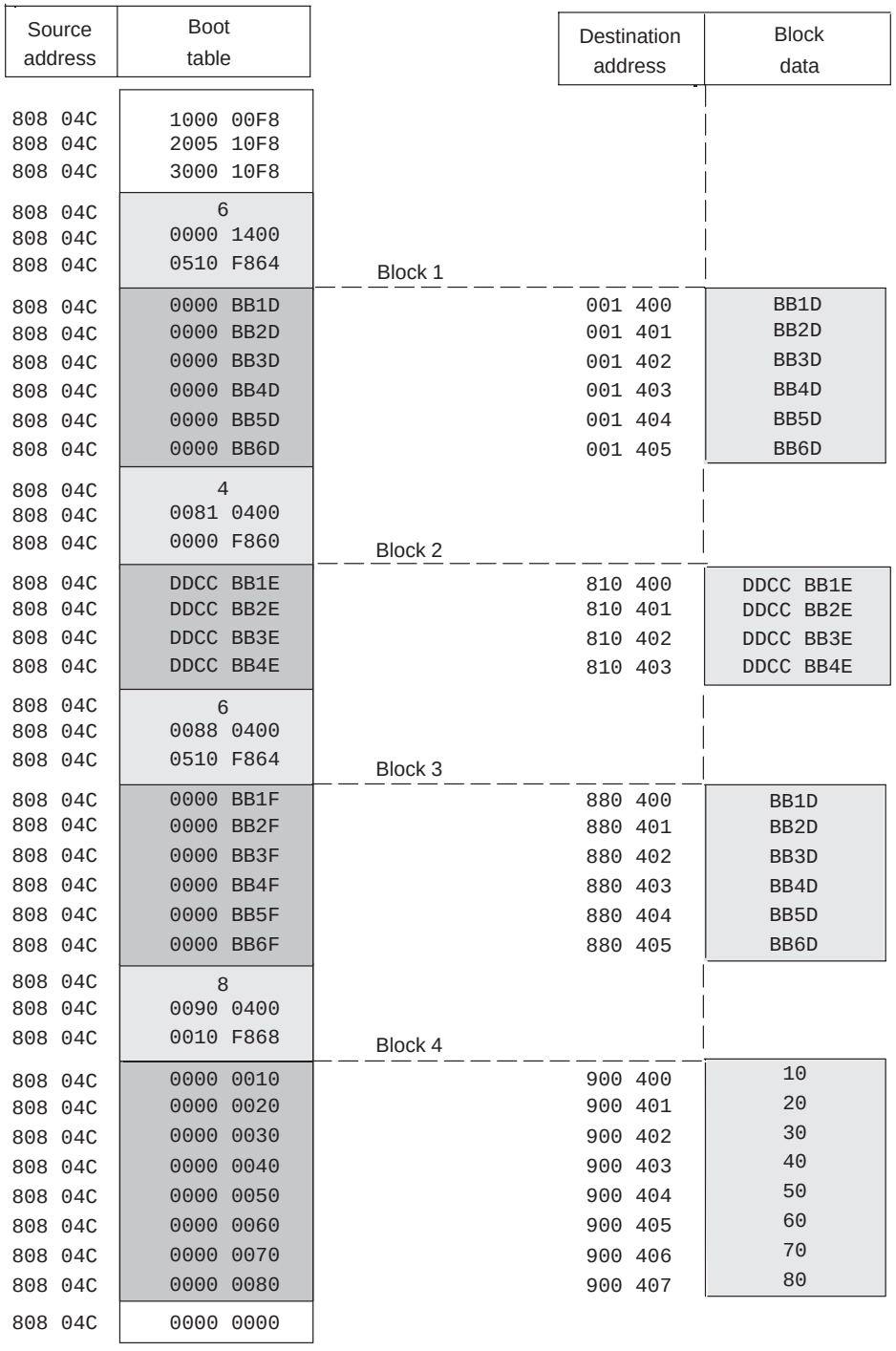


Figure A–4. Boot From Serial Port to 8-, 16-, and 32-Bit-Wide RAM



Appendix B

TMS320C32 Boot Loader Operations

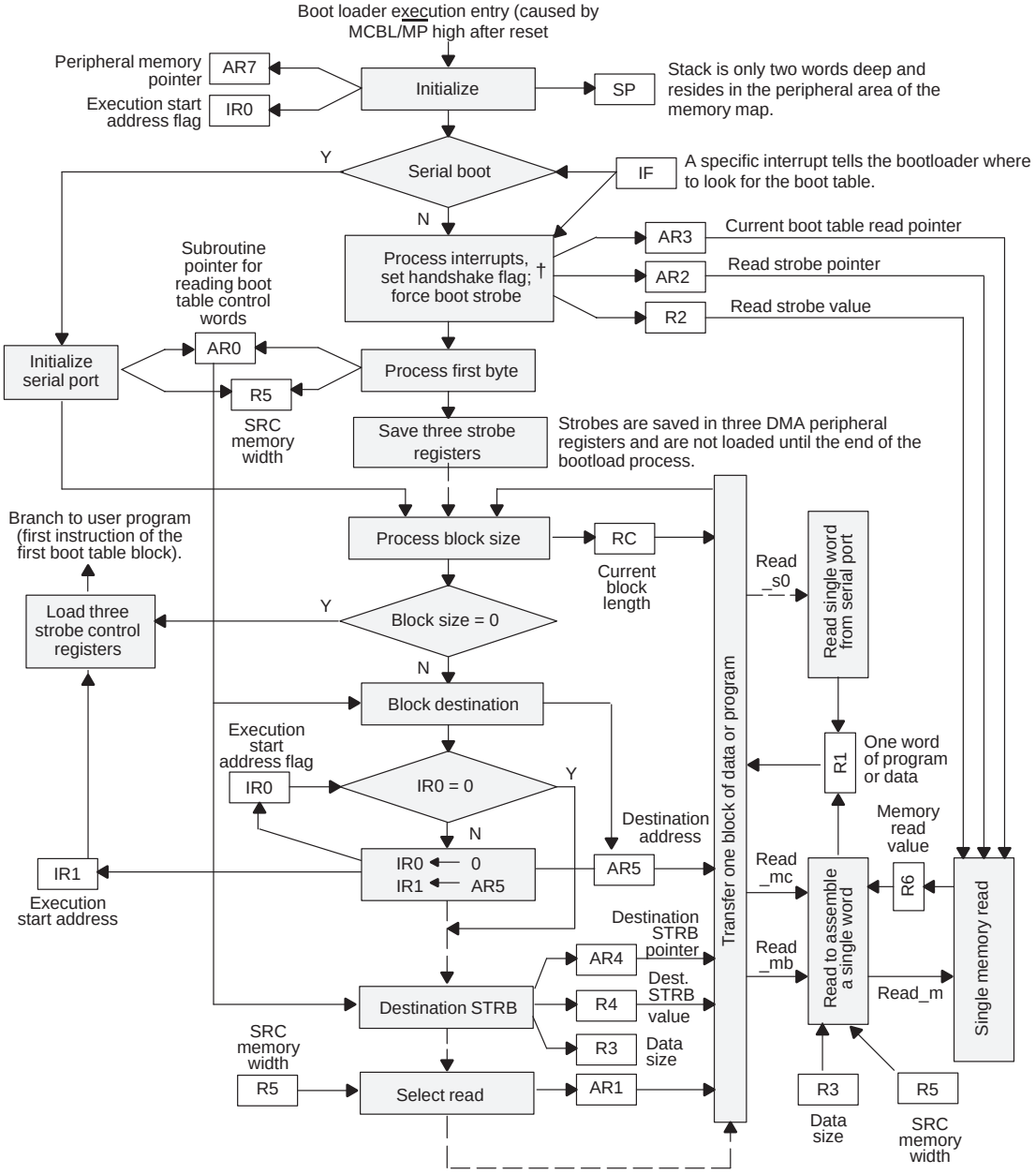
This appendix contains the source code and boot loader opcodes for the ’C32. It also describes the on-chip boot loader program that initializes the DSP system following power up or reset.

Topic **Page**

B.1 TMS320C32 Boot Loader Source Code Description

Figure B–1 shows the boot loader program flowchart. The shaded areas represent portions of code; the square shapes depict registers containing data. The boot loader reads the boot table from one of three memory locations (1000h, 810000h, 900000h) or from the serial port. The boot loader processes each block of the boot table separately. First, the words of the program or data are assembled from bytes (or half-words). The assembled words are then written to their destinations one at a time. Each block can be transferred to any memory address range within the memory map. The blocks in the boot table are preceded by three control words: block size, destination address, and strobe control register value. The boot loader ends execution when it finds a 0 for the size of the next block. At that point, it initializes the three strobe control registers and branches to the first instruction of the first block. For that reason, the first boot table block always contains program information and not data. For information about the boot loader operation, see section B.3, *Boot Loader Source Code Listing*, on page B-6 and the *TMS320C3x User's Guide*.

Figure B–1. TMS320C32 Boot Loader Program Flowchart



† Handshake mode is enabled by setting the IOXF0 bit of IOF register to 1 when $\overline{\text{INT3}}$ *and* any of $\overline{\text{INT2}}$, $\overline{\text{INT1}}$, or $\overline{\text{INT0}}$ signals are asserted following reset.

Note: Shaded boxes indicate operations; white boxes indicate registers.

B.2 TMS320C32 Boot Loader Opcodes

Table B–1 lists the ’C32 boot loader opcodes (shown in boldface type). In most cases, an opcode is the first byte of the machine code that describes the type of operation and combination of operands interpreted by the central processing unit (CPU).

Table B–1. TMS320C32 Boot Loader Opcodes

ADDRESS	OPCODE	ADDRESS	OPCODE	ADDRESS	OPCODE	ADDRESS	OPCODE
00000000	**00000045**	00000034	00000000	00000068	**1a660001**	0000009d	**086800a7**
00000001	00000000	00000035	00000000	00000069	**6a060004**	0000009e	**08650000**
00000002	00000000	00000036	00000000	0000006a	**09e6ffff**	0000009f	**08620000**
00000003	00000000	00000037	00000000	0000006b	**09eeffff**	000000a0	**080a000f**
00000004	00000000	00000038	00000000	0000006c	**09e50001**	000000a1	**08600111**
00000005	00000000	00000039	00000000	0000006d	**6a00fffa**	000000a2	**15400743**
00000006	00000000	0000003A	00000000	0000006e	**186e0002**	000000a3	**08670a30**
00000007	00000000	0000003B	00000000	0000006f	**04ee0000**	000000a4	**09e70010**
00000008	00000000	0000003C	00000000	00000070	**6a070002**	000000a5	**15470740**
00000009	00000000	0000003D	00000000	00000071	**72000053**	000000a6	**6a00ffcc**
0000000A	00000000	0000003E	00000000	00000072	**6f80fffe**	000000a7	**1a770020**
0000000B	00000000	0000003F	00000000	00000073	**70000008**	000000a8	**6a05fffe**
0000000C	00000000	00000040	00000000	00000074	**15410704**	000000a9	**02f70fdf**
0000000D	00000000	00000041	00000000	00000075	**70000008**	000000aa	**0841074c**
0000000E	00000000	00000042	00000000	00000076	**15410706**	000000ab	**78800000**
0000000F	00000000	00000043	00000000	00000077	**70000008**	000000ac	**08630003**
00000010	00000000	00000044	00000000	00000078	**15410708**	000000ad	**08730001**
00000011	00000000	00000045	**086f4040**	00000079	**70000008**	000000ae	**09930005**
00000012	00000000	00000046	**09ef0009**	0000007a	**08010001**	000000af	**18730001**
00000013	00000000	00000047	**08740023**	0000007b	**6a060007**	000000b0	**080e0003**
00000014	00000000	00000048	**1014000f**	0000007c	**08400704**	000000b1	**026e0001**
00000015	00000000	00000049	**0871ffff**	0000007d	**15400760**	000000b2	**09ee0003**
00000016	00000000	0000004a	**08000017**	0000007e	**08400706**	000000b3	**08000005**
00000017	00000000	0000004b	**02e0000f**	0000007f	**15400764**	000000b4	**04e00001**
00000018	00000000	0000004c	**04e00008**	00000080	**08400708**	000000b5	**6a050003**
00000019	00000000	0000004d	**6a05004f**	00000081	**15400768**	000000b6	**09e0ffff**
0000001A	00000000	0000004e	**080a000f**	00000082	**68000012**	000000b7	**09eeffff**
0000001B	00000000	0000004f	**026a0060**	00000083	**081b0001**	000000b8	**6a00fffb**
0000001C	00000000	00000050	**1a600004**	00000084	**187b0001**	000000b9	**186e0001**
0000001D	00000000	00000051	**536b4080**	00000085	**70000008**	000000ba	**08600000**
0000001E	00000000	00000052	**6a060008**	00000086	**080d0001**	000000bb	**08610000**
0000001F	00000000	00000053	**026a0004**	00000087	**4f100000**	000000bc	**02740003**
00000020	00000000	00000054	**1a600001**	00000088	**5312000d**	000000bd	**72000007**
00000021	00000000	00000055	**536b0008**	00000089	**53710000**	000000be	**18740003**
00000022	00000000	00000056	**6a060004**	0000008a	**70000008**	000000bf	**21871306**
00000023	00000000	00000057	**026a0004**	0000008b	**08040001**	000000c0	**09870000**
00000024	00000000	00000058	**1a600004**	0000008c	**02e1006c**	000000c1	**10010007**
00000025	00000000	00000059	**536b4800**	0000008d	**258c010f**	000000c2	**02000005**
00000026	00000000	0000005a	**6a05ffef**	0000008e	**09e4fff8**	000000c3	**6f80fff8**
00000027	00000000	0000005b	**1a600008**	0000008f	**08030004**	000000c4	**78800000**
00000028	00000000	0000005c	**6a050002**	00000090	**09e3fff0**	000000c5	**1a780002**
00000029	00000000	0000005d	**1a780080**	00000091	**02e30003**	000000c6	**1542c200**
0000002A	00000000	0000005e	**08780006**	00000092	**1a61000c**	000000c7	**6a060002**
0000002B	00000000	0000005f	**0862000f**	00000093	**52e30003**	000000c8	**08462301**
0000002C	00000000	00000060	**09e20010**	00000094	**04e50000**	000000c9	**78800000**
0000002D	00000000	00000061	**1042c200**	00000095	**52e900a7**	000000ca	**1b40c700**
0000002E	00000000	00000062	**1542c200**	00000096	**536900ad**	000000cb	**1a780080**
0000002F	00000000	00000063	**09eb0009**	00000097	**6400009b**	000000cc	**6a06fffd**
00000030	00000000	00000064	**086800ac**	00000098	**70000009**	000000cd	**08462301**
00000031	00000000	00000065	**08650001**	00000099	**1544c400**	000000ce	**08780002**
00000032	00000000	00000066	**086e0020**	0000009a	**0c800000**	000000cf	**1a780080**
00000033	00000000	00000067	**7200005d**	0000009b	**15412501**	000000d0	**6a05fffe**
				0000009c	**6a00ffdc**	000000d1	**08780006**
						000000d2	**78800000**

B.3 Boot Loader Source Code Listing

```
*************************************************************************
* C32BOOT – TMS320C32 BOOT LOADER PROGRAM    (143 words)     March–96
*     (C) COPYRIGHT TEXAS INSTRUMENTS INCORPORATED, 1994    v.27
*=======================================================================*
*
* NOTE:
*
*  1. Following device reset, the program waits for an external
*     interrupt. The interrupt type determines the initial address
*     from which the boot loader starts loading the boot table to the
*     destination memory:
```

INTERRUPT PIN	BOOT TABLE START ADDRESS	BOOT SOURCE
INTR0	1000h (STRB0	P_PORT
INTR1	810000h (IOSTRB)	P_PORT
INTR2	900000h (STRB1)	P_PORT
INTR3	80804Ch (sport0 Rx)	SERIAL
INTR0 and INT3	1000h (STRB0) ASYNC	PPORT,XF0/XF1
INTR1 and INT3	810000h (IOSTRB) ASYNC	PPORT,XF0/XF1
INTR2 and INT3	900000h (STRB1) ASYNC	PPORT,XF0/XF1

```
*     If INT3 is asserted together with INT2, or INT1, or INT0 following
*     reset, that indicates that the boot table is to be read
*     asynchronously from EPROM using pins XF0 and XF1 for handshaking.
*     The handshaking protocol assumes that the data ready signal
*     generated by the host arrives through pin XF1. The data
*     acknowledge signal is output from the C32 on pin XF0. Both
*     signals are active low. The C32 continuously toggles the IACK
*     signal while waiting for the host to assert data ready signal
*     (pin XF1).
*
*  2. The boot operation involves transfer of one or more source
*     blocks from the boot media to the destination memory. The block
*     structure of the boot table serves the purpose of distributing
*     the source data/program among different memory spaces. Each
*     block is preceded by several 32-bit control words describing
*     the block contents to the boot loader program.
*
*  3. When loading from the serial port, the boot loader reads the source
*     data/program and writes it to the destination memory. There is
*     only one way to read the serial port. When loading from EPROM,
*     however, there are 4 ways to read and assemble the
*     source contents, depending on the width of boot memory and the
```

```
*     size of the program/data being transferred. Because there is a
*     possibility that reads and writes can span the same STRB space,
*     the boot loader loads the appropriate STRB control registers
*     before each read and write.
*
*  4. If the boot source is an EPROM whose physical width is less than
*     32 bits, the physical interface of the EPROM device(s) to the
*     processor must be the same as that of the 32-bit interface.
*     (This involves a specific connection to the C32's strobe and
*     address signals). The reason for such an arrangement is that
*     to function properly, the boot loader program always expects
*     32-bit data from 32-bit wide memory during the boot load
*     operation. Valid boot EPROM widths are : 1, 2, 4, 8, 16
*     and 32 bits.

*  5. A single source block cannot cross STRB boundaries. For
*     example, its destination cannot overlap STRB0 space and IOSTRB
*     space. Additionally, all of the destination addresses of a
*     single source block must reside in physical memory of the
*     same width. It is not permitted to mix program and data in the
*     same source block.
*
*  6. The boot loader stops boot operation when it finds a 0 in the
*     block size control word. Therefore, each boot table must
*     end with a 0, prompting the boot loader to branch to the
*     first address of the first block and start program execution
*     from that location.
*
*=======================================================================*
* 'C32 boot loader program register assignments, and altered memory
* locations
*=======================================================================*
*
* AR7 – peripheral memory map    IOF – XF0 (handshake – data acknowledge)
* AR0 – read cntrl data subr pointer   IOF – XF1 (handshake – data ready)
* AR1 – read block data/prg subr pointer
*
*  R2 – read STRB value            R4 – write STRB value
* AR2 – read STRB pointer         AR4 – write STRB pointer
* AR3 – read data/prg pointer     AR5 – write data/prg pointer
*
*                        read –-> R1 –-> write
*
* IR0 – EXEC start flag          stack – 808024h – TIM0 cnt reg
* IR1 – EXEC start address               808028h – TIM0 per reg
*                               IOSTRB – 808004h – DMA0 dst reg
*  R3 – data size                STRB0 – 808006h – DMA0 dst reg
*  R5 – mem width                STRB1 – 808008h – DMA0 cnt reg
```

```
*
*  R6 – memory read value       AR6,R7,R0,BK – scratch registers
*
*=====================================================================*

reset      .word    start        ; reset vector
           .space   44h          ; program starts @45h

*=====================================================================*


* Initialize registers : 808000h –-> AR7,  808023h –-> SP,  –1 –-> IR0
*=====================================================================*

start      LDI      4040h,AR7     ; load peripheral memory map
           LSH      9,AR7         ; base address = 808000h
           LDI      23h,SP        ; initialize stack pointer to
           OR       AR7,SP        ; 808023h (timer counter – 1)
           LDI      –1,IR0        ; reset exec start addr flag

*=====================================================================*
* Test for INT3 and, if set exclusively, proceed with serial
* boot load. Else, load AR3 with 1000h if INT0, 810000h if INT1,
* 900000h if INT2. Also load the appropriate boot strobe pointer  –-> AR2
* and force the boot strobe value to reflect 32-bit memory width.
* If (INT0 or INT1 or INT2) and INT3, turn on the handshake mode.
*=====================================================================*
wait1      LDI      IF,R0
           AND      0Fh,R0           ; clean
           CMPI     8,R0             ; test for INT3
           BEQ      serial   ;*******; serial boot load mode
           LDI      AR7,AR2

           ADDI     60h,AR2          ; 808060h (IOSTRB)   –->  AR2
           TSTB     2,R0             ; test for  INT1
           LDINZ    4080h,AR3        ; 810000h / 2**9
           BNZ      exit3    ;*******;

          ADDI      4,AR2            ; 808064h  (STRB0)   –->  AR2
          TSTB      1,R0             ; test for  INT0
          LDINZ     8,AR3            ; 001000h / 2**9
          BNZ       exit3    ;*******;

          ADDI      4,AR2            ; 808068h  (STRB1)   –->  AR2
          TSTB      4,R0             ; test for  INT2
          LDINZ     4800h,AR3        ; 900000h / 2**9
          BZ        wait1    ;*******;
```

```
exit3    TSTB      8,R0           ;*; test#1 – INT3 asserted
         BZ        exit2          ;*; test#2 – INXF1 low (not used)
         TSTB      80h,IOF        ;*; enable handshake mode if
         LDI       6,IOF          ;*; test#1 passed

exit2    LDI       0Fh,R2
         LSH       16,R2            ; force boot data size to 32
         OR        *AR2,R2          ; force boot mem width to 32
         STI       R2,*AR2
         LSH       9,AR3            ; boot mem start addr --> AR3
*                                                    xx000001 –  1 bit
*================================================== xx000010 –  2 bit
* Process MEMORY WIDTH control word (32 bits long)  xx000100 –  4 bit
*================================================== xx001000 –  8 bit
*                                                    xx010000 – 16 bit
*                                                    xx100000 – 32 bit
         LDI       read_mc,AR0      ; use memory to read cntrl words
                                    ;             read_mc -->  AR0
         LDI       1,R5             ; mem width = 1        (init)
         LDI       32,AR6           ; mem reads = 32       (init)
        CALLU      read_m           ; read memory once     (1st read)

loop2    TSTB      1,R6
         BNZ       label4
         LSH       –1,R6            ; look at next bit
         LSH       –1,AR6           ; decr mem reads
         LSH       1,R5             ; incr mem width   -->  R5
         BU        loop2    ;*******;

label4   SUBI      2,AR6
         CMPI      0,AR6            ; set flags
         BN        strobes  ;*******; total # of mem reads = 32/R5
label5  CALLU      read_m           ; read memory once
        DBU        AR6,label5  ;****;

*=====================================================================*
* Read and save IOSTRB, STRB0 & STRB1 (to be loaded at end of
* boot load)
*=====================================================================*

strobes  CALLU      AR0
          STI       R1,*+AR7(4)      ; IOSTRB    -->     (DMA src)
         CALLU      AR0
          STI       R1,*+AR7(6)      ; STRB0     -->     (DMA dst)
         CALLU      AR0
          STI       R1,*+AR7(8)      ; STRB1     -->     (DMA cnt)

*=====================================================================*
```

```
* Process block size (# of bytes, half-words, or words after STRB
* cntrl)
*=================================================================*

block   CALLU      AR0               ; read boot memory cntrl word
         LDI       R1,R1             ; is this the last block ?
         BNZ       label2    ;*******; no, go around

         LDI      *+AR7(4),R0        ;                    (DMA src)
         STI       R0,*+AR7(60h)     ; restore IOSTRB
         LDI      *+AR7(6),R0        ;                    (DMA dst)
         STI      R0,*+AR7(64h)      ; restore  STRB0
         LDI      *+AR7(8),R0        ;                    (DMA cnt)
         STI      R0,*+AR7(68h)      ; restore  STRB1
         BU       IR1        ;*******; branch to start of program

label2   LDI      R1,RC              ; setup transfer loop
         SUBI     1,RC               ; RC – 1 –-> RC



*=================================================================*
* Process block destination address, save start address of first
* block
*=================================================================*

      CALLU      AR0                ; read boot memory cntrl word
       LDI       R1,AR5             ; set dest addr           –-> AR5
       CMPI      0,IR0              ; look at EXEC start addr flag
       LDINZ     AR5,IR1            ; if –1, EXEC start addr  –-> IR1
       LDINZ     0,IR0              ; set EXEC start addr flag

*=================================================================*
* (For internal destination, this word must be 0 or 60h. The first
* case results in 0 –-> DMA control register, in second case 0 –->
* IOSTRB register).
* Process block destination strobe control (sss...sss 0110 xx00)
*======================================= strb value ==== 00 – IOSTRB
*                                                       01 –  STRB0
     CALLU      AR0                ;                    10 –  STRB1
      LDI       R1,R4
      AND       6Ch,R1             ; dest mem strb pntr –-> AR4
      OR3       AR7,R1,AR4

      LSH       –8,R4              ; dest memory strobe –-> R4

      LDI       R4,R3
      LSH       –16,R3
```

```
      AND       3,R3                ; dest data size     --> R3
      TSTB      0Ch,R1              ; (IOSTRB case)
      LDIZ      3,R3

*=====================================================================*
* Look at R5 and choose serial or memory read for block data/program
*=====================================================================*

        CMPI      0,R5
        LDIEQ    read_s0,AR1                ; read serial port0
        LDINE    read_mb,AR1                ; read memory

*=====================================================================*
* Transfer one block of data or program
*=====================================================================*

         RPTB     loop4
        CALLU     AR1                      ; read data/prg
         STI      R4,*AR4                  ; set write strobe
         NOP                               ; pipeline
loop4    STI      R1,*AR5++                ; write data/prg!!!!!!!!!!
          BU      block           ;*******; process next block


*=====================================================================*
* Load R5 with 0, load read_s0 to AR0 and initialize serial port_0
*=====================================================================*

serial    LDI     read_s0,AR0         ; use serial to read cntrl words
          LDI     0,R5                   ; memory WIDTH = serial
          LDI     0,R                    ; dummy
          LDI     AR7,AR2                ; dummy

          LDI     111h,R0                ; 0000111h --> R0
          STI     R0,*+AR7(43h)          ; set CLKR,DR,FSR as serial
          LDI     0A30h,R7               ;                port pins
          LSH     16,R7                  ; A300000h --> R7
          STI     R7,*+AR7(40h)          ; set serial global cntrl reg
          BU      strobes             ;*******; process first block

*=====================================================================*
* Read a single value from serial or boot memory. The number of
* memory reads depends on memory width and data size. R1 returns the
* read value. (Serial sim: NOP --> BZ read_s0 & LDI @4000H,R1 --> LDI
*  *+AR7(4Ch),R1)
*=====================================================================*

read_s0  TSTB     20h,IF              ; look at RINT0 flag
```

```
         BZ       read_s0            ; wait for receive buffer full
         AND      0FDFh,IF           ; reset interrupt flag
         LDI      *+AR7(4Ch),R1      ; read data            -->  R1
         RETSU
*-------------------------------------------------------------------
read_mc  LDI      3,R3                 ; data size = 32, 3 --> R3

read_mb  LDI      1,BK                 ; 00000001  (ex: mem width=8)
         LSH      R5,BK                ; 00000100
         SUBI     1,BK                 ; 000000FF  =  mask --> BK

         LDI      R3,AR6               ;  0 -    1 000  EXPAND
         ADDI     1,AR6                ;  1 -   10 000  DATA --> AR6
         LSH     3,AR6                 ; 11 -  100 000  SIZE
         LDI        R5,R0
loop3    CMPI       1,R0
         BEQ        exit1              ; DATA SIZE
         LSH        -1,R0              ; --------- - 1       --> AR6
         LSH        -1,AR6             ; MEM WIDTH
         BU         loop3       ;*******;
exit1    SUBI       1,AR6

         LDI        0,R0               ; init shift value
         LDI        0,R1               ; init accumulator
loop1    ADDI       3,SP               ; 808027h --> SP
        CALLU       read_m             ; read memory once    --> R6
         SUBI       3,SP               ; 808024h --> SP
         AND3       R6,BK,R7           ; apply mask
         LSH        R0,R7              ; shift
         OR         R7,R1             ; accumulate        -->  R1
         ADDI       R5,R0              ; increment shift value
         DBU         AR6,loop1   ;*****; decrement #of chunks --> AR6
         RETSU

*===================================================================*
* Perform a single memory read from the source boot table.
* Handshake enabled if IOXF0 bit of IOF reg is set, disabled when
* reset. IACK will pulse continuously if handshake enabled and data
* not ready (to achieve zero-glue interface when connecting to a C40
* comm-port)
*===================================================================*

read_m   TSTB       2,IOF              ; handshake mode enabled ?
         STI        R2,*AR2            ; set read strobe !!!!!!!!!!!!
         BNZ        loop5              ; yes, jump over
         LDI        *AR3++,R6          ; no,  just read memory & return
         RETSU
*-------------------------------------------------------------   (C40)
```

```
loop5   IACK      *AR7              ;*; intrnl dummy read pulses IACK
        TSTB       80h,IOF          ;*; wait for data ready
        BNZ        loop5            ;*; (XF1 low from host)

        LDI        *AR3++,R6        ;*; read memory once   -->  R6

        LDI        2,IOF            ;*; assert data acknowledge
                                    ;*; (XF0 low to host)

loop6   TSTB       80h,IOF          ;*; wait for data not ready
        BZ         loop6            ;*; (XF1 high from host)

        LDI        6,IOF            ;*; deassert data acknowledge
                                    ;*; (XF0 high to host)
        RETSU
*===============================================================*
```

Appendix C

Memory Access for C Programs

This appendix describes the two memory models that can be used to access data when programming in C.

Two memory models can be used to access data when programming in C. In the small model (default), the external bus cycles use direct addressing to access data from memory. Direct addressing uses 16 bits of address in the instruction opcode. The address is combined with the 8-bit data page (defined beforehand) to access the data from memory. The 16-bit address limits the number of words that the small model can access to 64K words. However, this mode produces fast and compact code because each data access uses only a single instruction (see Figure C–1).

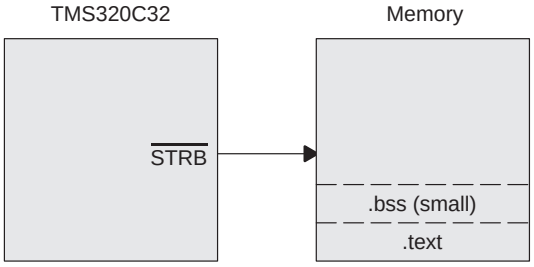
The big model is not limited to 64K words because each data access in C explicitly sets the data page pointer (DP register). The 8-bit data page and 16-bit direct address are combined for a total address reach of 16M words, but at a price of two instructions per data access (see Figure C–1).

Dynamically allocated memory can be used if the application needs a large address reach, compact code size, and fast execution. The MALLOC function from the runtime support library (RTS) can be called at run time to reserve a block of memory in the .SYSMEM section. Upon return, MALLOC returns a pointer to the newly allocated block. Any reference to that block of memory results in assembled code using indirect addressing, in which the opcode contains a pointer to the auxiliary register that holds the address of the operand (see Figure C–1). Code referring to the dynamically allocated memory is fast and has a 16M-word address reach (24 bits). The price is a one-time call to MALLOC for each dynamically allocated array. For that reason, MALLOC is most efficient with large data arrays where the overhead associated with the call is insignificant when compared to a large number of data accesses that use the big arrays.

Figure C–1. Memory Allocation in C Programs

(a) Small model (default)

- Static memory – assigned at compile time
- Maximum size – 64K words
- Fast execution

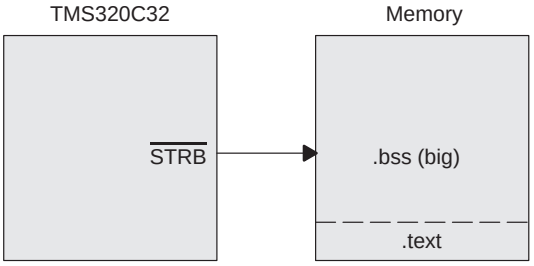


C statement	Equivalent assembly code

C = A + B	LDI @ 0FFFDh, R0
	LDI @0FFFEh, R1
	ADDI R0, R1
	STI R1, @ 0FFFh

(b) Big model (-mb option)

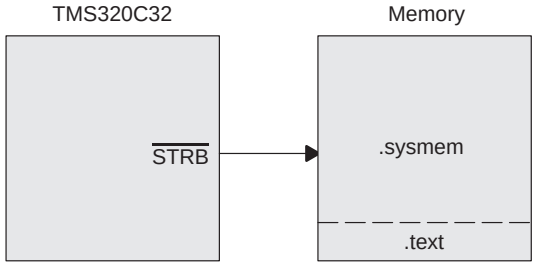
- Static memory – assigned at compile time
- Maximum size – 64M words
- Slow execution



C statement	Equivalent assembly code
C = A + B	LDP @ 880001h, DP LDI @ 880001h, R0 LDP @ 1002h, DP LDI @ 1002h, R1 LDP @ 8A0003, DP STI R1, @ 8A0003

(c) RTS library (MALLOC)

- Dynamic memory – assigned at execution time
- Maximum size – 64M words
- Fast execution
- Best for big arrays (one time overhead – MALLOC call)



C statement	Equivalent assembly code

C = A + B	LDI *AR0, R0
	LDI *AR1, R1
	ADDI R0, R1
	STI R1, *AR2

Figure C–2 shows how to use MALLOC to allocate a block of 32-bit memory at run time. In this example, MALLOC is called three times to allocate memory from the heap.

After each MALLOC call, the newly allocated block of memory can be used by other program functions by using the pointer BUFFER_32. The size of the heap (representing all of dynamically allocated memory) is defined in the linker command file by using the HEAP keyword followed by the size of the block. Any portion of the heap allocated with the MALLOC call is added to the .SYSMEM section. The SECTIONS directive can then be used to map the dynamically allocated sections to an address range in the physical memory. (For more information, see the *TMS320C3x/C4x Assembly Language Tools User's Guide* or *TMS320C3x/C4x Optimizing C Compiler User's Guide* .)

Dynamically allocated memory provides the only method for a C program to access 8- or 16-bit wide memory. This means that physical memory that is less than 32 bits wide cannot be accessed using small or big model addressing. Instead, the MALLOC8 and MALLOC16 RTS library functions can allocate blocks of 8- and 16-bit wide memory. These routines work like the 32-bit MALLOC by returning pointers to 8- or 16-bit memory blocks. These can be used by code that follows the MALLOC call to access that memory (see Figure C–3 and Figure C–4). The 8-bit data allocated by MALLOC8 is placed in the .SYSM8 section by the linker, while the 16-bit data is deposited in the .SYSM16 section. HEAP8 and HEAP16 linker keywords limit the total amount of 8- or 16-bit memory that the C compiler can allocate into those sections. (For more information, see the *TMS320C3x/C4x Optimizing C Compiler User's Guide* .)

Figure C–2. Dynamic Memory Allocation for TMS320C32 (One Block of 32-Bit Memory)

(a) C code

```
        .
        .
        .
int    *BUFFER_32                  /* declare a pointer to a pool of 32-bit memory */
        .
        .
        .
        .
        .
        .
        .

BUFFER_32 = MALLOC (2048 * sizeof (int))      /* allocate 2K words of memory */
dsp_func4 ( BUFFER_32)                        /* use the above memory */
        .
        .
        .
BUFFER_32 = MALLOC  (512 * sizeof (int))      /* allocate 0.5K words of memory */
dsp_func5 ( BUFFER_32)                        /* use the above memory */
        .
        .
        .
BUFFER_32  = MALLOC (1024 * sizeof (int))     /* allocate 1K words of memory */
dsp_func6 (BUFFER_32)                         /* use the above memory */
        .
        .
        .
```

(b) LINKER command file

```
        .
        .
        .
–heap   0x4000                /* set the size of the dynamic 32-bit memory section */
        .
        .
        .
STRB_RAM     org = 0x1000, len = 0x8000       /* define physical 32-bit memory */
        .
        .
        .
.sysmem  >   STRB_RAM            /* assign logical section to physical memory */
        .
        .
        .
```

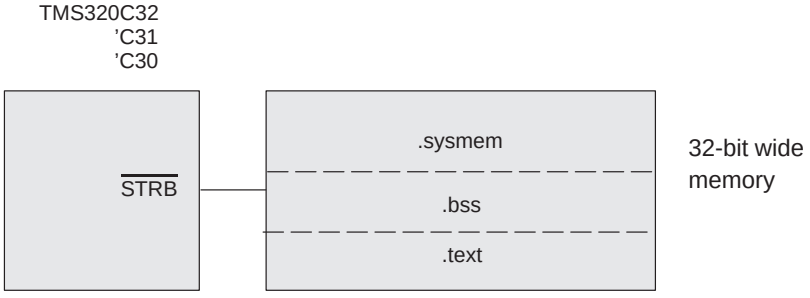


Figure C–3. Dynamic Memory Allocation for TMS320C32 (One Block of 16-Bit Memory)

(a) C code

```
       •
       •
       •
int    *BUFFER_16                    /* declare a pointer to a pool of 16-bit memory */
       •
       •
       •
*0x808064 = 0x5000      /* STRB0 control register : data size = 16, memory width = 16 */
       •
       •
       •
BUFFER_16 = MALLOC16(1024 * sizeof (int))      /* allocate 2K half-words of memory */
dsp_func4 ( BUFFER_16)                         /* use the above memory */
       •
       •
       •
BUFFER_16 = MALLOC16  (512 * sizeof (int))     /* allocate 1K half-words of memory */
dsp_func5 ( BUFFER_16)                         /* use the above memory */
       •
       •
       •
BUFFER_16  = MALLOC8 (2048 * sizeof (int))     /* allocate 4K half-words of memory */
dsp_func6 (BUFFER_16)                          /* use the above memory */
       •
       •
       •
```

(b) LINKER command file

```
       •
       •
       •
–heap 16  0x4000           /* set the size of the dynamic 16-bit memory section */
       •
       •
       •
STRB0_RAM       org = 0x880000,   len = 0x8000  /* define physical 16-bit memory */
       •
       •
       •
.sysm16    >   STRB0_RAM          /* assign logical section to physical memory */
       •
       •
       •
```

(c) 'C32 external memory contents

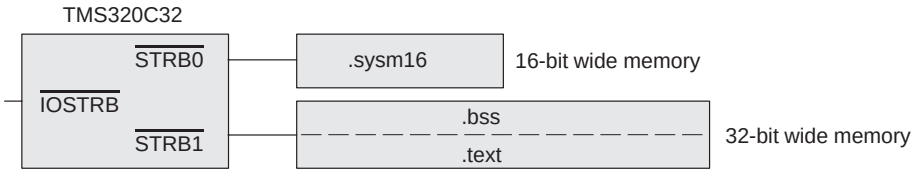


Figure C–4. Dynamic Memory Allocation for TMS320C32 (One Block Each of 32-, 16-, and 8-Bit Memory)

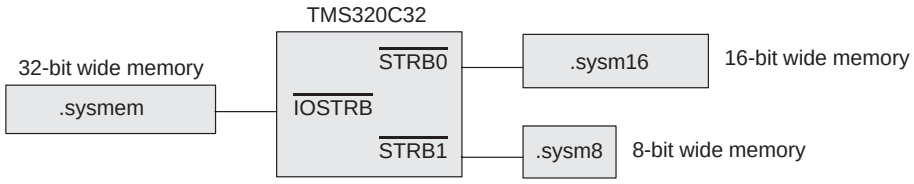
(a) C code

```
        •
        •
int    *BUFFER_32      /* declare a pointer to a pool of 32-bit memory */
int    *BUFFER_16      /* declare a pointer to a pool of 16-bit memory */
int    *BUFFER_08      /* declare a pointer to a pool of 8-bit memory */
        •
        •
*0x808064 = 0x5000     /* STRB0 control register : data size = 16, memory width = 16 */
*0x808068 = 0x0000     /* STRB1 control register : data size =  8 , memory width =  8  */
        •
        •
BUFFER_32 = MALLOC    (1024 * sizeof (int))   /* allocate 1K words of memory */
BUFFER_16 = MALLOC16(1024 * sizeof (int))     /* allocate 2K halfwords of memory */
BUFFER_08 = MALLOC8  (1024 * sizeof (int))    /* allocate 4K bytes of memory */
dsp_func1 (BUFFER_32, BUFFER_16, BUFFER_08)   /* use the above memory */
        •
        •
BUFFER_32 = MALLOC    (2048 * sizeof (int))   /* allocate 2K words of memory */
BUFFER_16 = MALLOC16  (512 * sizeof (int))    /* allocate 1K half-words of memory */
dsp_func2 (BUFFER_32, BUFFER_16)              /* use the above memory */
        •
        •
BUFFER_08  = MALLOC8 (4096 * sizeof (int))    /* allocate 16K bytes of memory */
dsp_func3 (BUFFER_08)                         /* use the above memory */
        •
        •
```

(b) LINKER command file

```
        •
        •
–heap     0x4000                /* set the size of the dynamic 32-bit memory section */
–heap 16  0x4000                /* set the size of the dynamic 16-bit memory section */
–heap  8  0x4000                /* set the size of the  dynamic  8-bit memory section */
        •
        •
IOSTRB_RAM      org = 0x810000,   len = 0x8000     /* define physical 32-bit memory */
STRB0_RAM       org = 0x880000,   len = 0x8000     /* define physical 16-bit memory */
STRB1_RAM       org = 0x900000,   len = 0x8000     /* define physical  8-bit  memory */
        •
        •
.sysmem    >   IOSTRB_RAM              /* assign logical section to physical memory */
.sysm16    >   STRB0_RAM               /* assign logical section to physical memory */
.sysm8     >   STRB1_RAM               /* assign logical section to physical memory */
        •
        •
```

(c) 'C32 external memory contents



Appendix D

Memory Interface and Address Translation

This appendix describes how to use the ’C32’s memory interfaces to connect to various external devices.

The ’C32 memory interface supports variable-width memory and variable-size data. The physical width of a memory bank connected to the ’C32 can be 8, 16, or 32 bits wide. When connecting 16-bit external memory, the A_{-1} address pin must be connected to the A_0 pin of the memory device, causing a 1-bit shift in the connection of the remaining address lines. For 8-bit memory, two extra address pins are used (A_{-1} and A_{-2}), effectively shifting the external address by two bits. No external address shift is needed for connecting 32-bit wide memory (or boot table memory, regardless of its width).

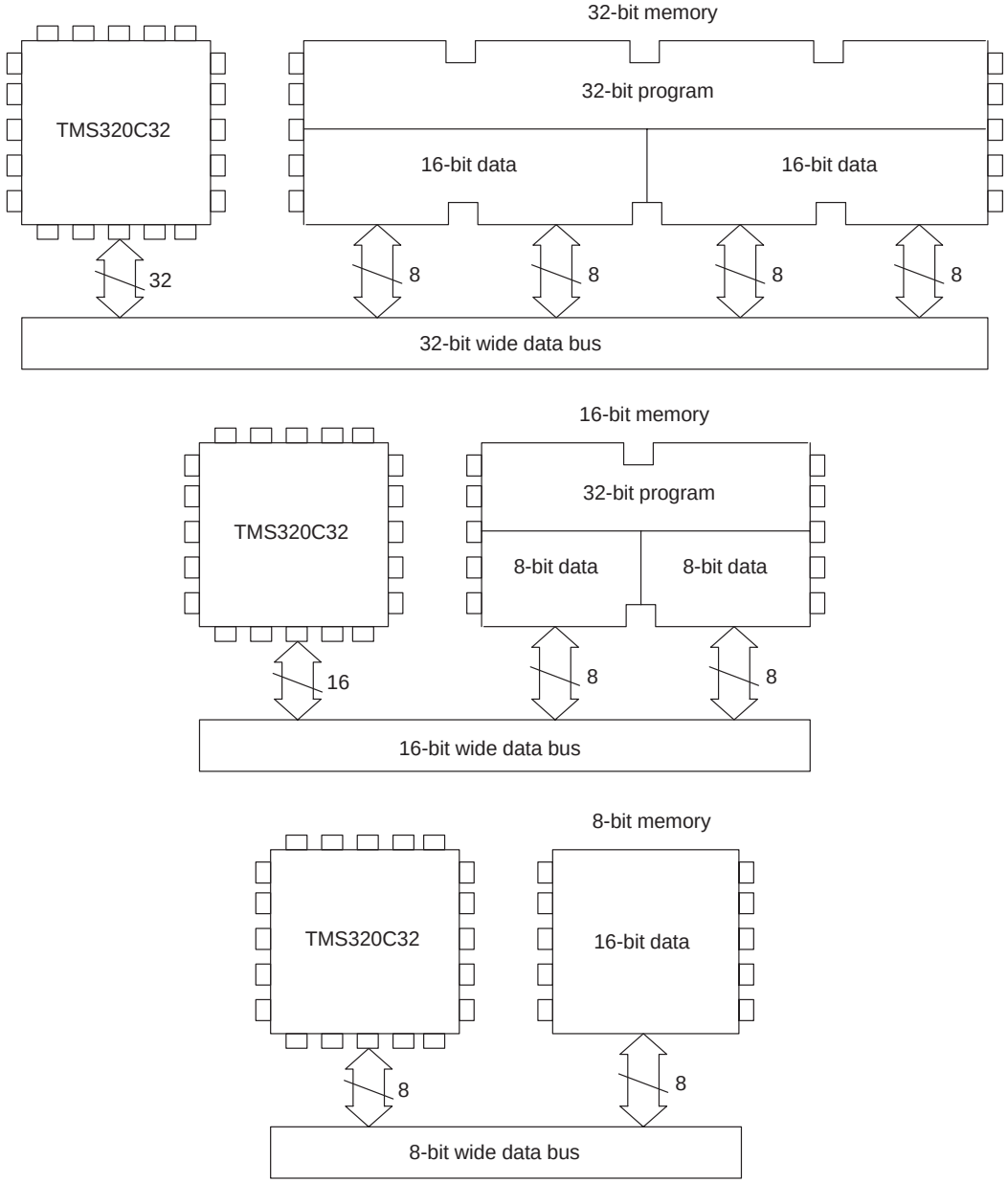
The ’C32 can access data of any size, regardless of the physical width of an external memory bank. For example, byte-wide data can be packed in 16-bit memory, or 32-bit data can be accessed from 8-bit wide memory. The latter takes four cycles. The variable-data size feature is made possible by dividing the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ controls into four signals each. The four control signals, in addition to being strobes, serve a byte-enable function.

Figure D–1 shows examples of three ’C32 systems, each connected to a memory bank of a different width.

Regardless of memory width, the data inside each bank can be 8, 16, or 32 bits wide. Before data of a particular size can be accessed, the respective strobe control register must be programmed for that size. While the data size can vary, the program is always 32 bits wide. Even if they are different sizes, program and data can reside within the same physical bank of memory.

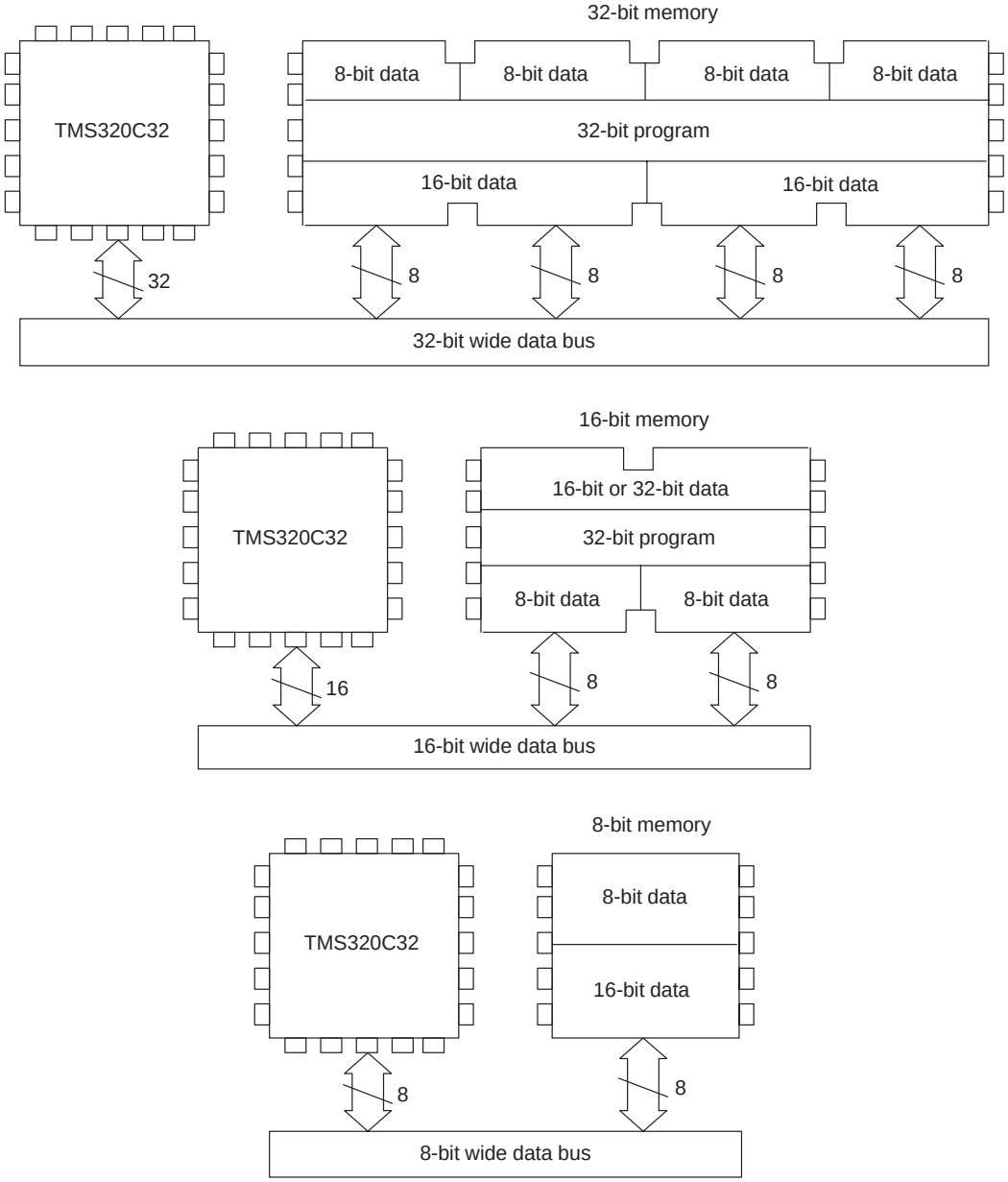
Up to two data sizes can reside simultaneously alongside the 32-bit program in a single bank (see Figure D–2 on page D-3).

Figure D–1. Data and Program Packing (Program and a Single Data Size)



NOTE: 8-bit programs are not supported.

Figure D–2. Data and Program Packing (Program and Two Different Data Sizes)



NOTE: 8-bit programs are not supported.

Since there are two strobes that support flexible memory ($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$), they each can be programmed for a different data size using the respective strobe control registers. By setting the strobe configuration bit in one control register, both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ strobes can be mapped to $\overline{\text{STRB0}}$ control signals. This creates a section of physical memory that is mapped into the same address range as another section of memory with a hardware switch to determine which range is active. In this overlay mode, data accesses to and from the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ portions of the memory map drive the $\overline{\text{STRB0}}$ signals to control a single memory bank. The access to the program and to two different data sizes from a single memory bank with no additional logic devices is a powerful ’C32 feature that minimizes system cost with no performance penalty. See the *TMS320C3x User’s Guide* for more information on the ’C32 enhanced external memory interface.

The translation starts when an instruction requests a data read from a certain external address. Address locations referenced by program instructions are logical addresses. Before the logical address shows up on the external pins of the ’C32, it may undergo a 1- or 2-bit shift to the right that depends only on the size of the data being accessed. The address at the pins is a physical address. Before it is presented at the pins of the memory device, the physical address may again be shifted (this time to the left) if the memory is other than 32 bits wide. The physical-to-memory address shift is one bit for 16-bit wide memory and two bits for 32-bit memory. The Table D–1 and Table D–2 summarize the rules that apply to the variable data size and memory width for any ’C32 system.

Table D–1. Variable Memory Width

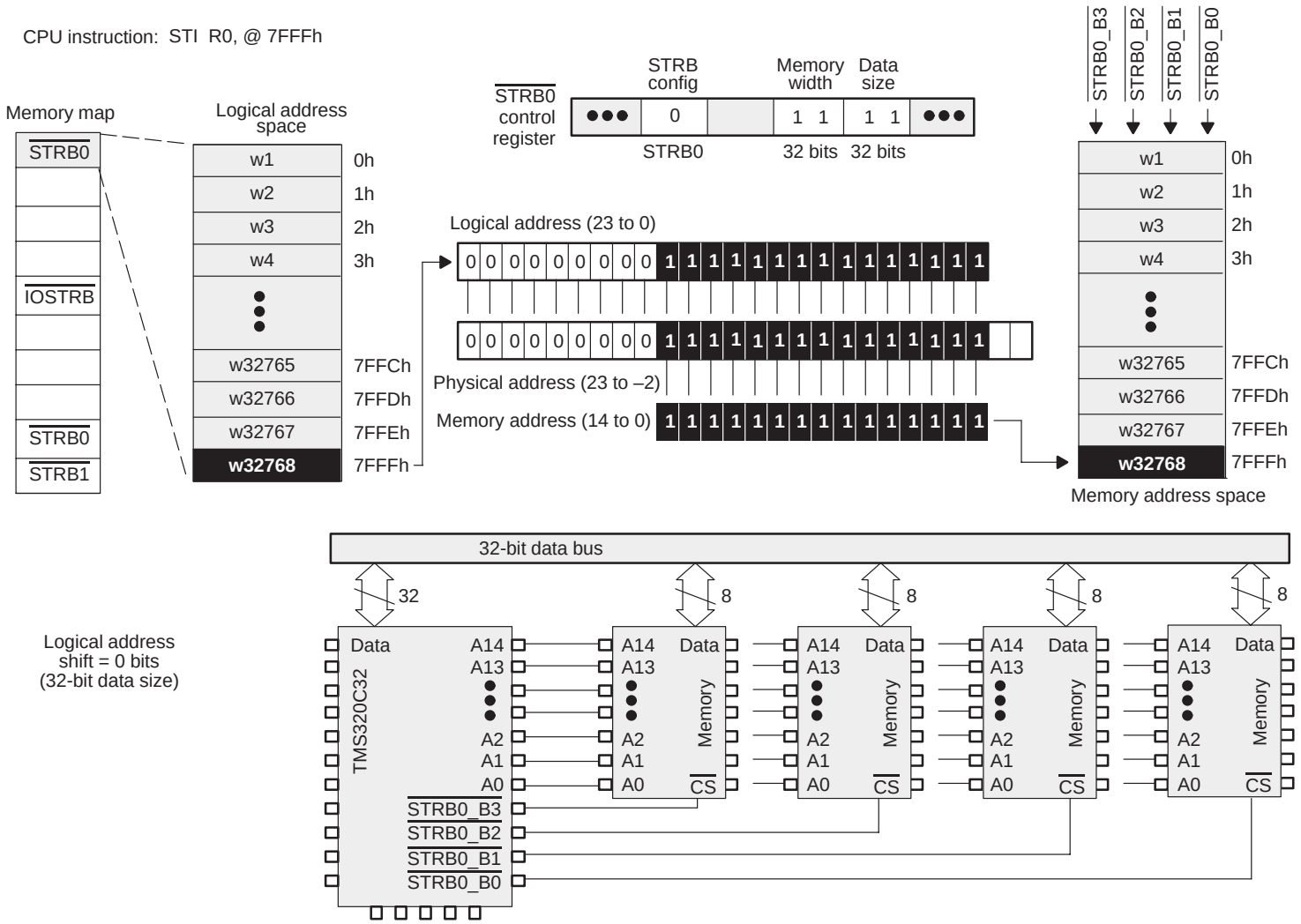
Memory Width	Strobes Valid	Physical Address Lines Valid	Physical Address to Memory Address Shift (bits)
32	$\overline{\text{STRBx\_B3}}$ $\overline{\text{STRBx\_B2}}$ $\overline{\text{STRBx\_B1}}$ $\overline{\text{STRBx\_B0}}$	A23–A0	0
16	$\overline{\text{STRBx\_B1}}$ $\overline{\text{STRBx\_B0}}$	A23–A0 A–1	1
8	$\overline{\text{STRBx\_B0}}$	A23–A0 A–1 A–2	2

Table D–2. Variable Data Size

Data Size	Logical to Physical Address Shift (bits)
32	0
16	1
8	2

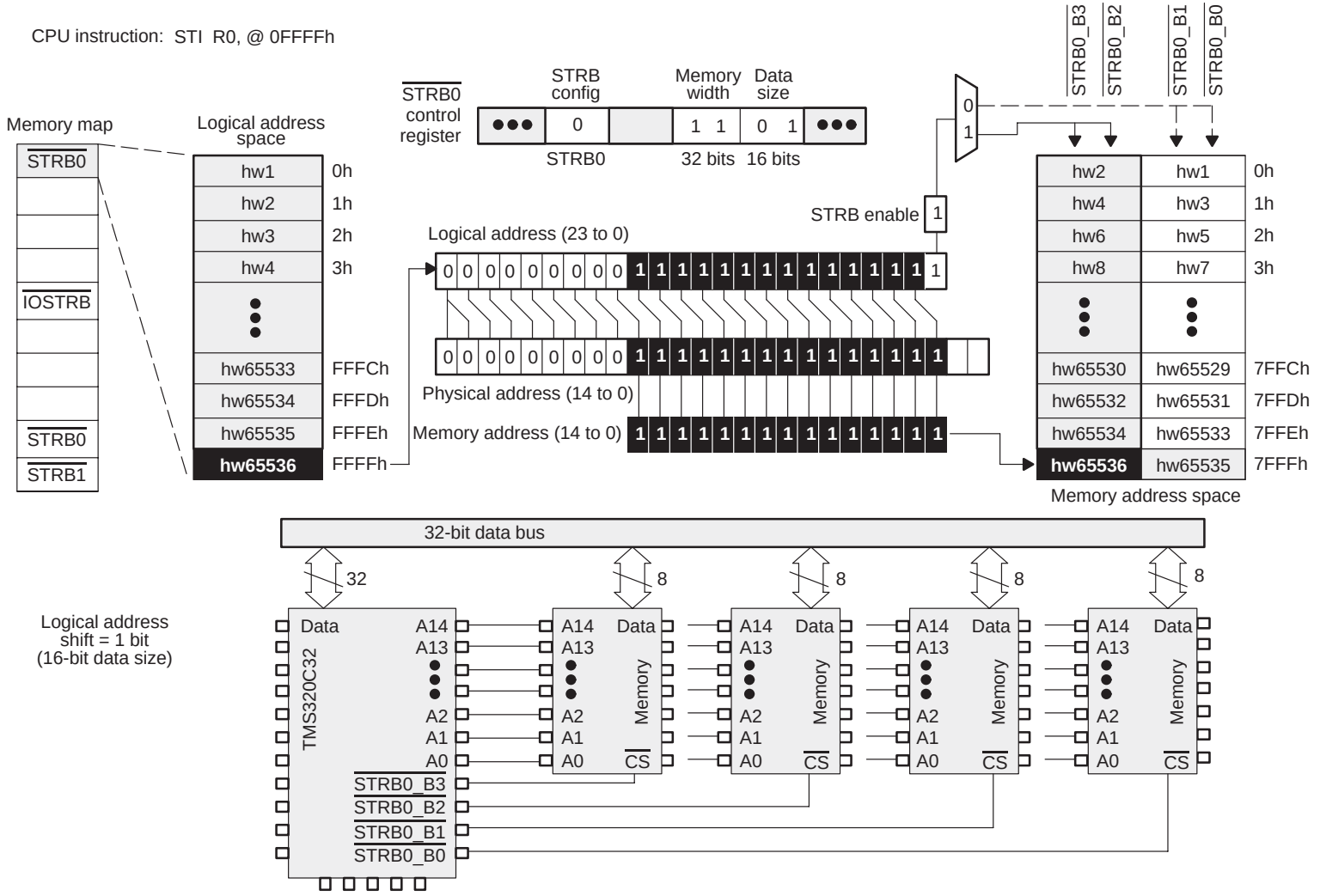
Figure D–3 through Figure D–11 show how the address changes when accessing data of varying size from memory that is 32, 16, and 8 bits wide. The three data sizes and three memory widths comprise the nine cases that cover all possible combinations.

Figure D–3. Address Translation for 32-Bit Data Stored in 32-Bit-Wide Memory



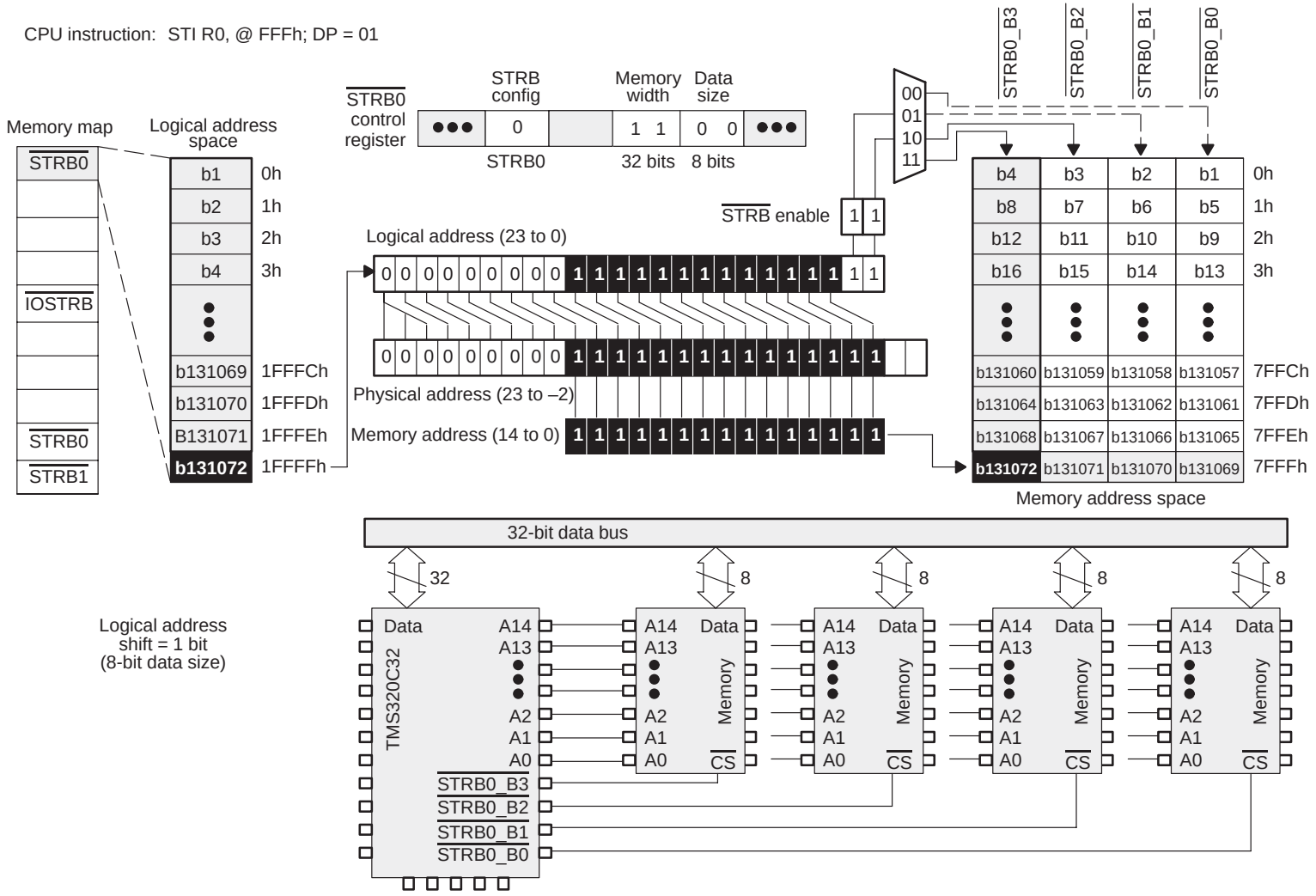
Note: The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Figure D–4. Address Translation for 16-Bit Data Stored in 32-Bit-Wide Memory



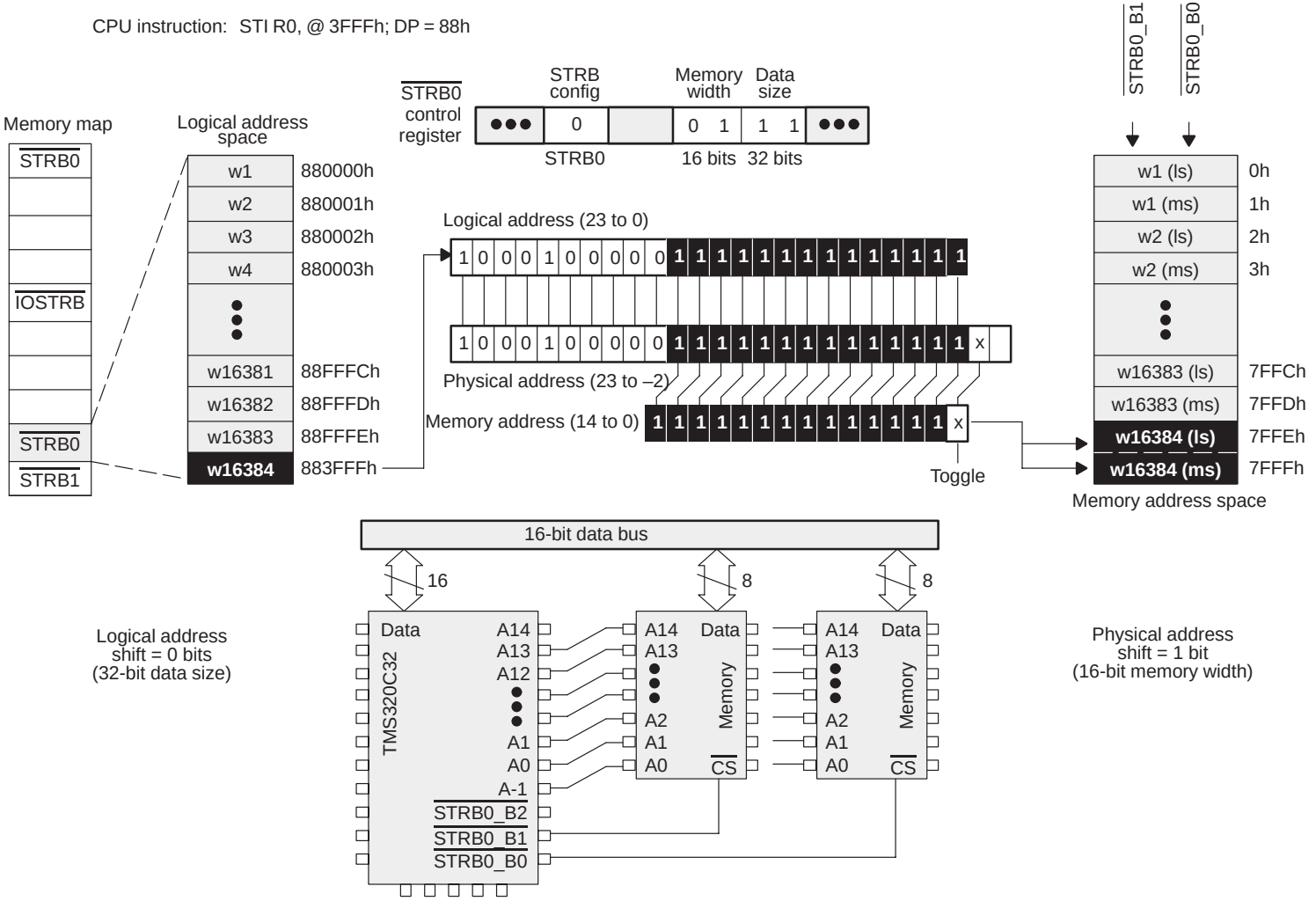
Note: The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Figure D–5. Address Translation for 8-Bit Data Stored in 32-Bit-Wide Memory



Note: The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Figure D–6. Address Translation for 32-Bit Data Stored in 16-Bit-Wide Memory

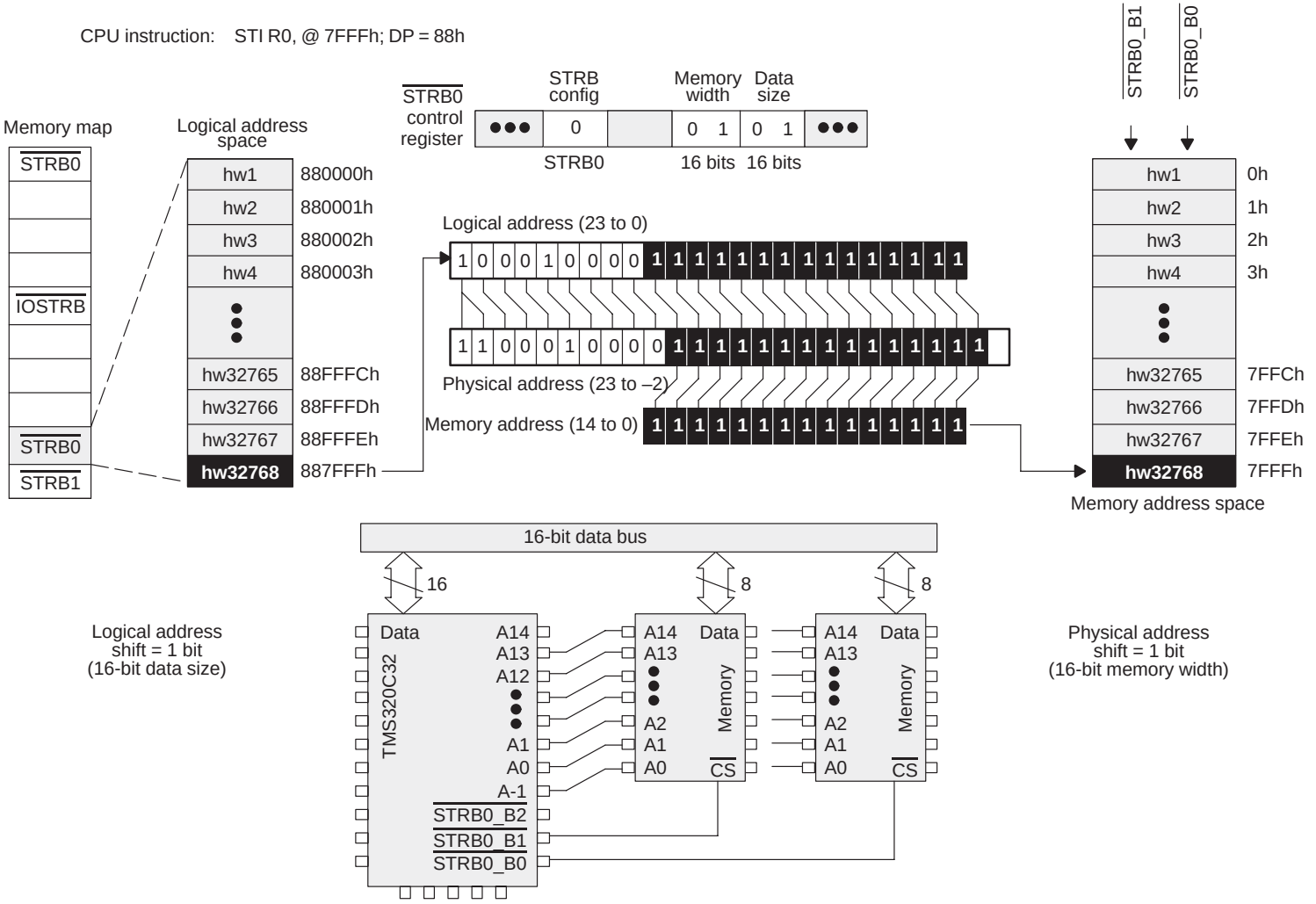


Notes: 1) The amount of shift between logical and physical addresses depends only on the size of data being transferred.

2) The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

Memory Interface and Address Translation

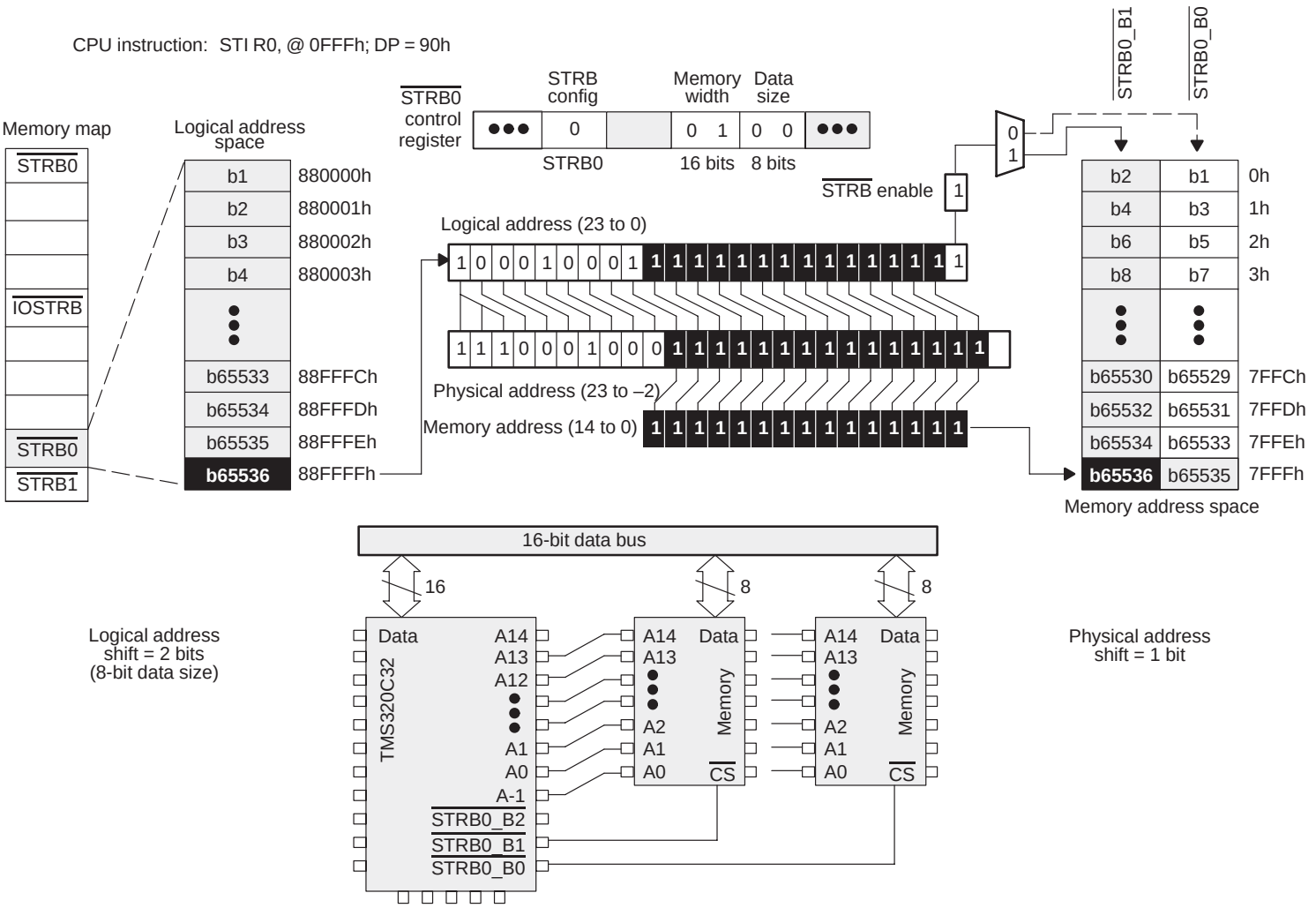
Figure D–7. Address Translation for 16-Bit Data Stored in 16-Bit-Wide Memory



Notes: 1) The amount of shift between logical and physical addresses depends only on the size of data being transferred.

2) The amount of shift in the physical connection between the 'C32 and the external memory depends only on the width of the memory bank.

Figure D–8. Address Translation for 8-Bit Data Stored in 16-Bit-Wide Memory

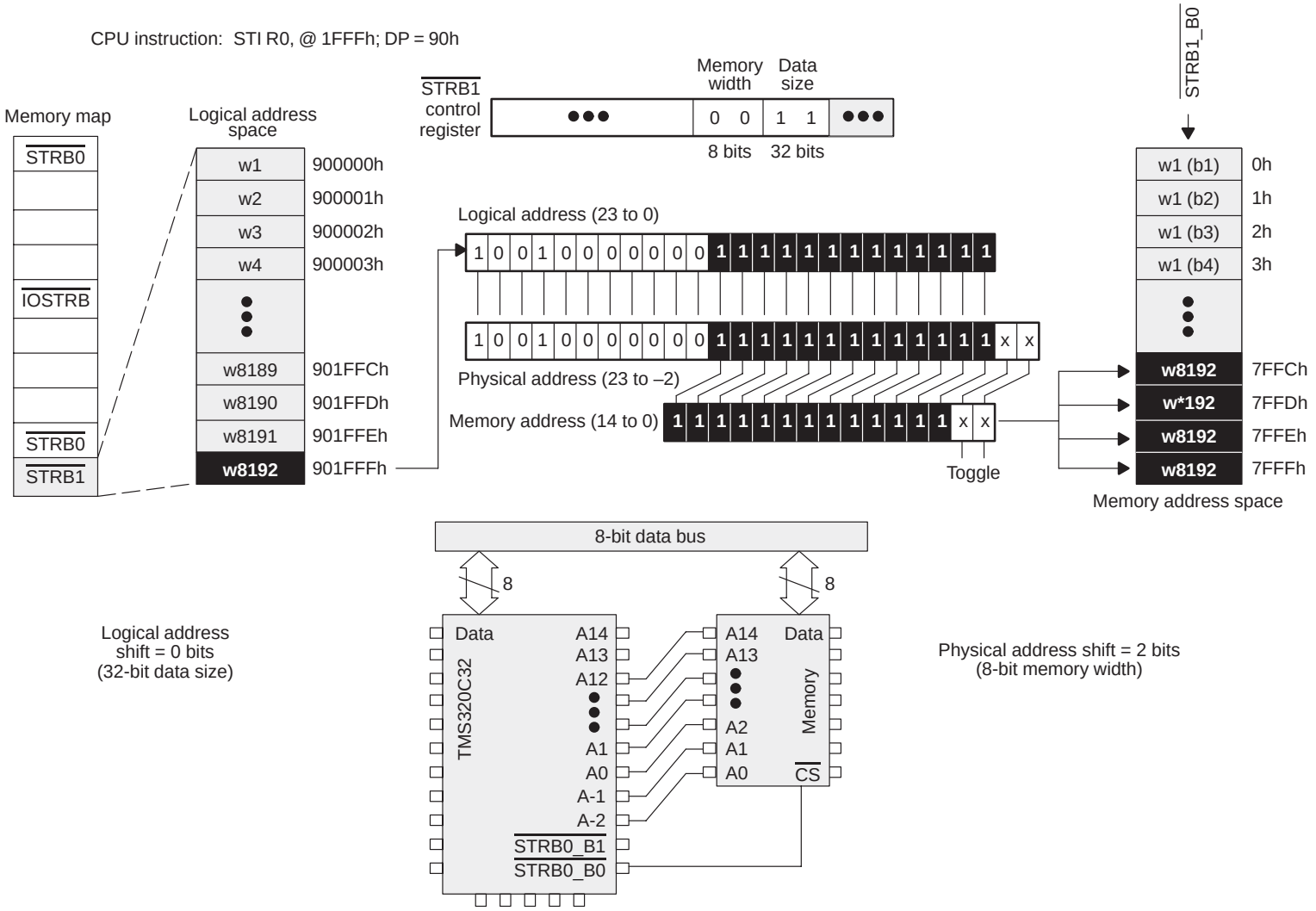


Notes: 1) The amount of shift between logical and physical addresses depends only on the size of data being transferred.

2) The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

Memory Interface and Address Translation

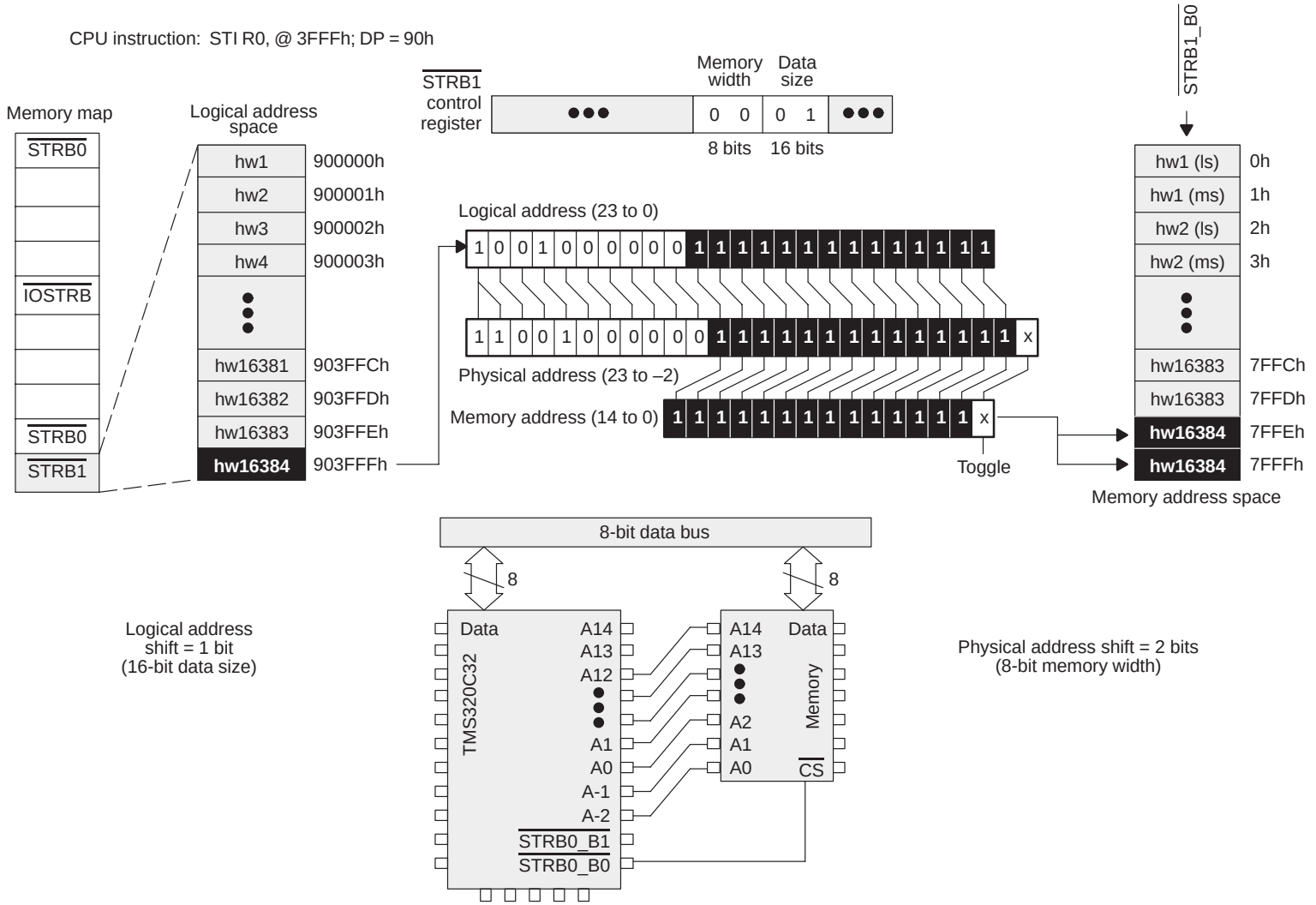
Figure D–9. Address Translation for 32-Bit Data Stored in 8-Bit-Wide Memory



Notes: 1) The amount of shift between logical and physical addresses depends only on the size of data being transferred.

2) The amount of shift in the physical connection between the 'C32 and the external memory depends only on the width of the memory bank.

Figure D–10. Address Translation for 16-Bit Data Stored in 8-Bit-Wide Memory

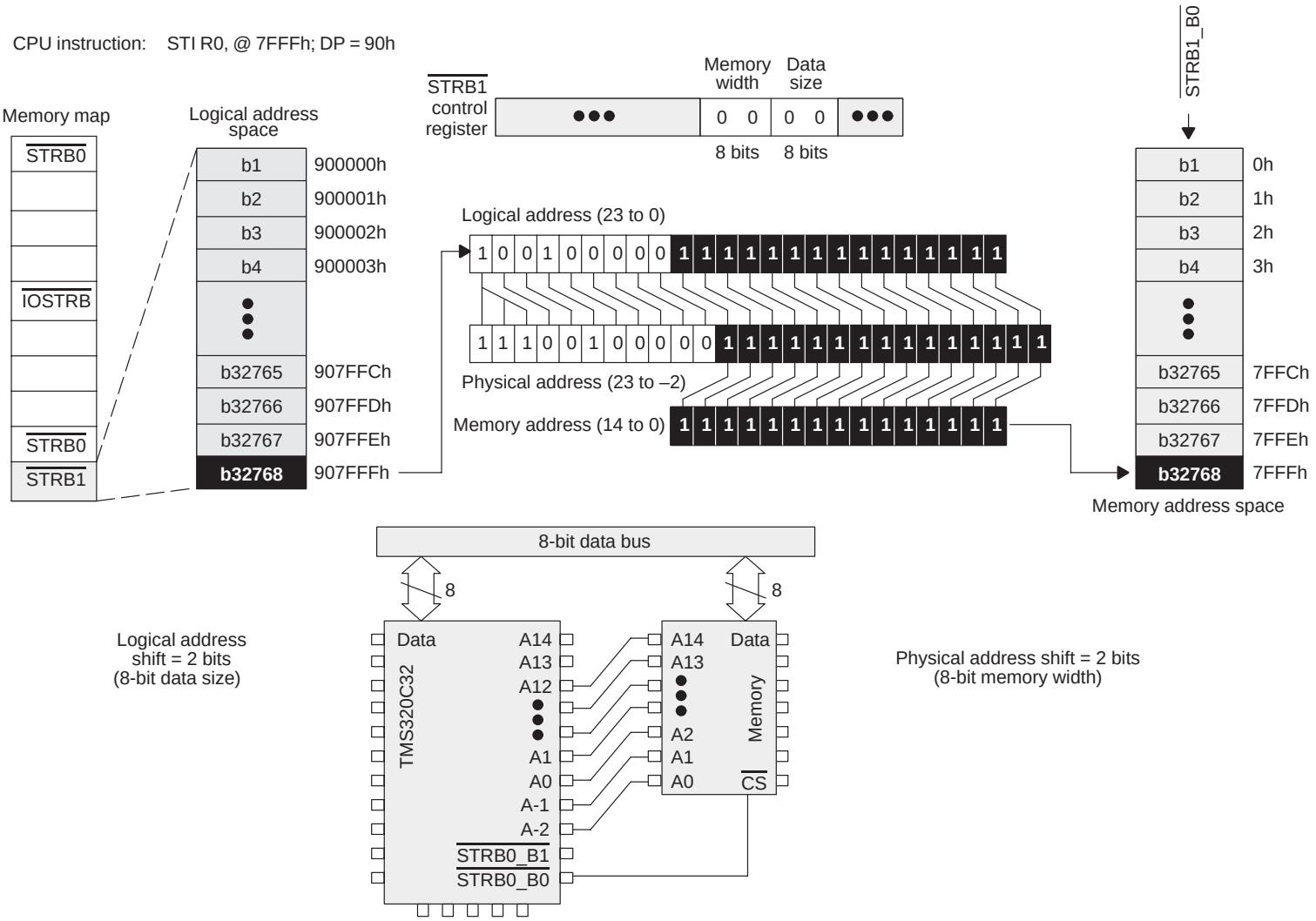


Notes: 1) The amount of shift between logical and physical addresses depends only on the size of data being transferred.

2) The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

Memory Interface and Address Translation

Figure D–11. Address Translation for 8-Bit Data Stored in 8-Bit-Wide Memory



Notes: 1) The amount of shift between logical and physical addresses depends only on the size of data being transferred.

2) The amount of shift in the physical connection between the 'C32 and the external memory depends only on the width of the memory bank.

Index

A

B

C

D

E

F

G

H

I

L

M

O

P

Q

R

S

T

U

W

X

Z