TMS320C3x User's Guide

Literature Number: SPRU031E
2558539-9761 revision L
July 1997

PRINTED WITH SOY INK™

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 an applications reference book for the TMS320C3x generation of digital signal processors (DSPs). These include the TMS320C30, TMS320C31, TMS320LC31, and TMS320C32. Throughout the book, all references to ’C3x refer collectively to the ’C30, ’C31, ’LC31 and ’C32.

This book provides information to assist managers and hardware/software engineers in application development. It includes example code and hardware connections for various applications.

The guide shows how to use the instructions set, the architecture, and the ’C3x interface. It presents examples for frequently used applications and discusses more involved examples and applications. It also defines the principles involved in many applications and gives the corresponding assembly language code for instructional purposes and for immediate use. Whenever the detailed explanation of the underlying theory is too extensive to be included in this manual, appropriate references are given for further information.

Notational Conventions

This document uses the following conventions.

❑ Program listings, program examples, and interactive displays are shown in a `special typeface`. 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.).

  Here 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
```

  Here is an example of a system prompt and a command that you might enter:

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

- In syntax descriptions, the instruction, command, or directive is in **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** must be entered as shown; portions of a syntax that are in *italics* describe the type of information that must be entered. Here is an example of a directive syntax:

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

  The directive .asect has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter is an actual section name, enclosed in double quotes; the second parameter is an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here 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).

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here 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$_n$]*

  This syntax shows that .byte has at least one value parameter, but you may supply additional value parameters, separated by commas.

Information About Cautions

This book contains cautions.

This is an example of a caution statement.

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

The information in a caution is provided for your protection. Please read each caution 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 as indicated in the section *If You Need Assistance…* on page vi. When ordering, please identify the book by its title and literature number.

TMS320C3x General Purpose Applications User's Guide (literature number SPRU194) provides information to assist you in application development for the TMS320C3x generation of digital signal processors (DSPs). It includes example code and hardware connections for various appliances. It also defines the principles involved in many applications and gives the corresponding assembly language code for instructional purposes and for immediate use.

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 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 User's Guide (literature number SPRU053) tells you how to invoke the 'C3x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

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 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the family of TMS320 digital signal processors. A myriad of products and applications are offered—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. 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.

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.

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
Microcontroller Home Page	http://www.ti.com/sc/micro
Networking Home Page	http://www.ti.com/sc/docs/network/nbuhomex.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		
Microcontroller Hotline	(281) 274-2370	Fax: (281) 274-4203	Email: micro@ti.com
Microcontroller Modem BBS	(281) 274-3700 8-N-1		
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			
Networking Hotline		Fax: (281) 274-4027 Email: TLANHOT@micro.ti.com	

❑ 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: comments@books.sc.ti.com

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

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 trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.

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

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C3x generation of digital signal processors (DSPs) are high-performance CMOS 32-bit floating-point devices in the TMS320 family of single-chip DSPs.

The ’C3x generation integrates both system control and math-intensive functions on a single controller. This system integration allows fast, easy data movement and high-speed numeric processing performance. Extensive internal busing and a powerful DSP instruction set provide the devices with the speed and flexibility to execute at up to 60 million floating-point operations per second (MFLOPS). The devices also feature a high degree of on-chip parallelism that allows users to perform up to 11 operations in a single instruction.

Topic **Page**

1.1 TMS320C3x Devices

The ’C3x family consists of three members: the ’C30, ’C31, and ’C32. The ’C30, ’C31, and ’C32 can perform parallel multiply and arithmetic logic unit (ALU) operations on integer or floating-point data in a single cycle.

The processors also possess the following features for high performance and ease of use:

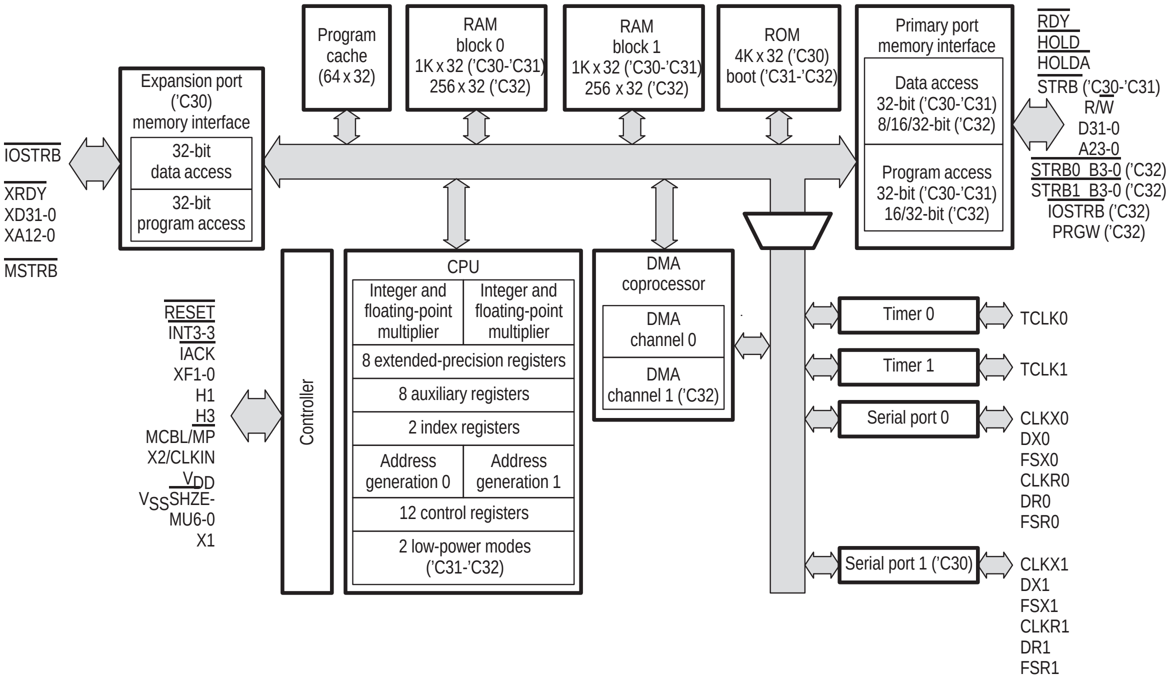
- General-purpose register file
- Program cache
- Dedicated auxiliary register arithmetic units (ARAU)
- Internal dual-access memories
- One direct memory access (DMA) channel (a two-channel DMA on the TMS320C32) supporting concurrent I/O
- Short machine-cycle time

General-purpose applications are greatly enhanced by the large address space, multiprocessor interface, internally and externally generated wait states, two external interface ports (one on the ’C31 and the ’C32) two timers, two serial ports (one on the ’C31 and the ’C32), and multiple-interrupt structure. The ’C3x supports a wide variety of system applications from host processor to dedicated coprocessor.

High-level language is implemented more easily through a register-based architecture, large address space, powerful addressing modes, flexible instruction set, and well-supported floating-point arithmetic.

Figure 1–1 shows a block diagram of ’C3x devices.

Figure 1–1. TMS320C3x Devices Block Diagram



1.1.1 TMS320C3x Key Specifications

The key specifications of the 'C3x devices include the following:

- Performance up to 60 MFLOPS
- Highly efficient C language engine
- Large address space: 16M words × 32 bits
- Fast memory management with on-chip DMA
- Industry-exclusive 3-V versions available on some devices

1.1.2 TMS320C30

The 'C30 is the first member of the 'C3x generation. It differs from the 'C31 and 'C32 by offering 4K ROM, 2K RAM, a second serial port, and a second external bus.

1.1.3 TMS320C31 and TMS320LC31

The 'C31 and 'LC31 are the second members of the 'C3x generation. They are low-cost 32-bit floating-point DSPs which have a boot-loader program, 2K RAM, single external port, single serial port, and are available in 3.3-V operation ('LC31).

1.1.4 TMS320C32

The ’C32 is the newest member of the ’C3x generation. They are enhanced versions of the ’C3x family and the lowest cost floating-point processors on the market today. These enhancements include a variable-width memory interface, two-channel DMA coprocessor with configurable priorities, flexible boot loader, and a relocatable interrupt vector table.

Table 1–1. *TMS320C30, TMS320C31, TMS320LC31, and TMS320C32 Comparison*

			Memory (words)								
			On-Chip			**Off-Chip**	**Peripherals**				
Device Name	**Freq (MHz)**	**Cycle Time (ns)**	**RAM**	**ROM**	**Cache**	**Parallel**	**Serial**	**DMA Channels**	**Timers**	**Package Type**	**Temperature**
’C30 (5 V)	27	75	2K	4K	64	16M × 32 8K × 32	2	1	2	181 PGA	0° to 85° (commercial)
	33	60	2K	4K	64	16M × 32 8K × 32	2	1	2	181 PGA	0° to 85° (commercial) –55° to 125° (military)
	40	50	2K	4K	64	16M × 32 8K × 32	2	1	2	181 PGA 208 PQFP	0° to 85° (commercial)
	50	40	2K	4K	64	16M × 32 8K × 32	2	1	2	181 PGA 208 PQFP	0° to 85° (commercial)
’C31 (5 V)	27	75	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial) –55° to 125° (military)
	33	60	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial) –40° to 125° (extended) –55° to 125° (military)
	40	50	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial) –40° to 125° (extended) –55° to 125° (military)
	50	40	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial) –40° to 125° (extended)
	60	33	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial) –40° to 125° (extended)
’LC31	33	60	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial)
(3.3 V)	40	50	2K	Boot loader	64	16M × 32	1	1	2	132 PQFP	0° to 85° (commercial)

Table 1–1. TMS320C30, TMS320C31, TMS320LC31, and TMS320C32 Comparison (Continued)

			Memory (words)								
			On-Chip			**Off-Chip**	**Peripherals**				
Device Name	**Freq (MHz)**	**Cycle Time (ns)**	**RAM**	**ROM**	**Cache**	**Parallel**	**Serial**	**DMA Channels**	**Timers**	**Package Type**	**Temperature**
’C32 (5 V)	40	50	512	Boot loader	64	16M × 32/16/8	1	2	2	144 PQFP	0° to 85° (commercial) –40° to 125° (extended)
	50	40	512	Boot loader	64	16M × 32/16/8	1	2	2	144 PQFP	0° to 85° (commercial) –40° to 125° (extended) –55° to 125° (military)
	60	33	512	Boot loader	64	16M × 32/16/8	1	2	2	144 PQFP	0° to 85° (commercial)

1.2 Typical Applications

The TMS320 family’s versatility, realtime performance, and multiple functions offer flexible design approaches in a variety of applications, which are shown in Table 1–2.

Table 1–2. Typical Applications of the TMS320 Family

General-Purpose DSP	**Graphics/Imaging**	**Instrumentation**
Digital filtering	3-D transformations rendering	Spectrum analysis
Convolution	Robot vision	Function generation
Correlation	Image transmission/compression	Pattern matching
Hilbert transforms	Pattern recognition	Seismic processing
Fast Fourier transforms	Image enhancement	Transient analysis
Adaptive filtering	Homomorphic processing	Digital filtering
Windowing	Workstations	Phase-locked loops
Waveform generation	Animation/digital map	
	Bar-code scanners	
Voice/Speech	**Control**	**Military**
Voice mail	Disk control	Secure communications
Speech vocoding	Servo control	Radar processing
Speech recognition	Robot control	Sonar processing
Speaker verification	Laser printer control	Image processing
Speech enhancement	Engine control	Navigation
Speech synthesis	Motor control	Missile guidance
Text-to-speech	Kalman filtering	Radio frequency modems
Neural networks		Sensor fusion
Telecommunications		**Automotive**
Echo cancellation	FAX	Engine control
ADPCM transcoders	Cellular telephones	Vibration analysis
Digital PBXs	Speaker phones	Antiskid brakes
Line repeaters	Digital speech	Anticollision
Channel multiplexing	Interpolation (DSI)	Adaptive ride control
Modems	X.25 packet switching	Global positioning
Adaptive equalizers	Video conferencing	Navigation
DTMF encoding/decoding	Spread spectrum	Voice commands
Data encryption	Communications	Digital radio
		Cellular telephones
Consumer	**Industrial**	**Medical**
Radar detectors	Robotics	Hearing aids
Power tools	Numeric control	Patient monitoring
Digital audio/TV	Security access	Ultrasound equipment
Music synthesizer	Power line monitors	Diagnostic tools
Toys and games	Visual inspection	Prosthetics
Solid-state answering	Lathe control	Fetal monitors
Machines	CAM	MR imaging

Chapter 2

Architectural Overview

This chapter provides an architectural overview of the ’C3x processor. It includes a discussion of the CPU, memory interface, boot loader, peripherals, and direct memory access (DMA) of the ’C3x processor.

Topic **Page**

2.1 Overview

The ’C3x architecture responds to system demands that are based on sophisticated arithmetic algorithms that emphasize both hardware and software solutions. High performance is achieved through the precision and wide dynamic range of the floating-point units, large on-chip memory, a high degree of parallelism, and the DMA controller.

Figure 2–1 through Figure 2–3 show functional block diagrams of the ’C30, ’C31, and ’C32 architectures, respectively.

Figure 2–1. TMS320C30 Block Diagram

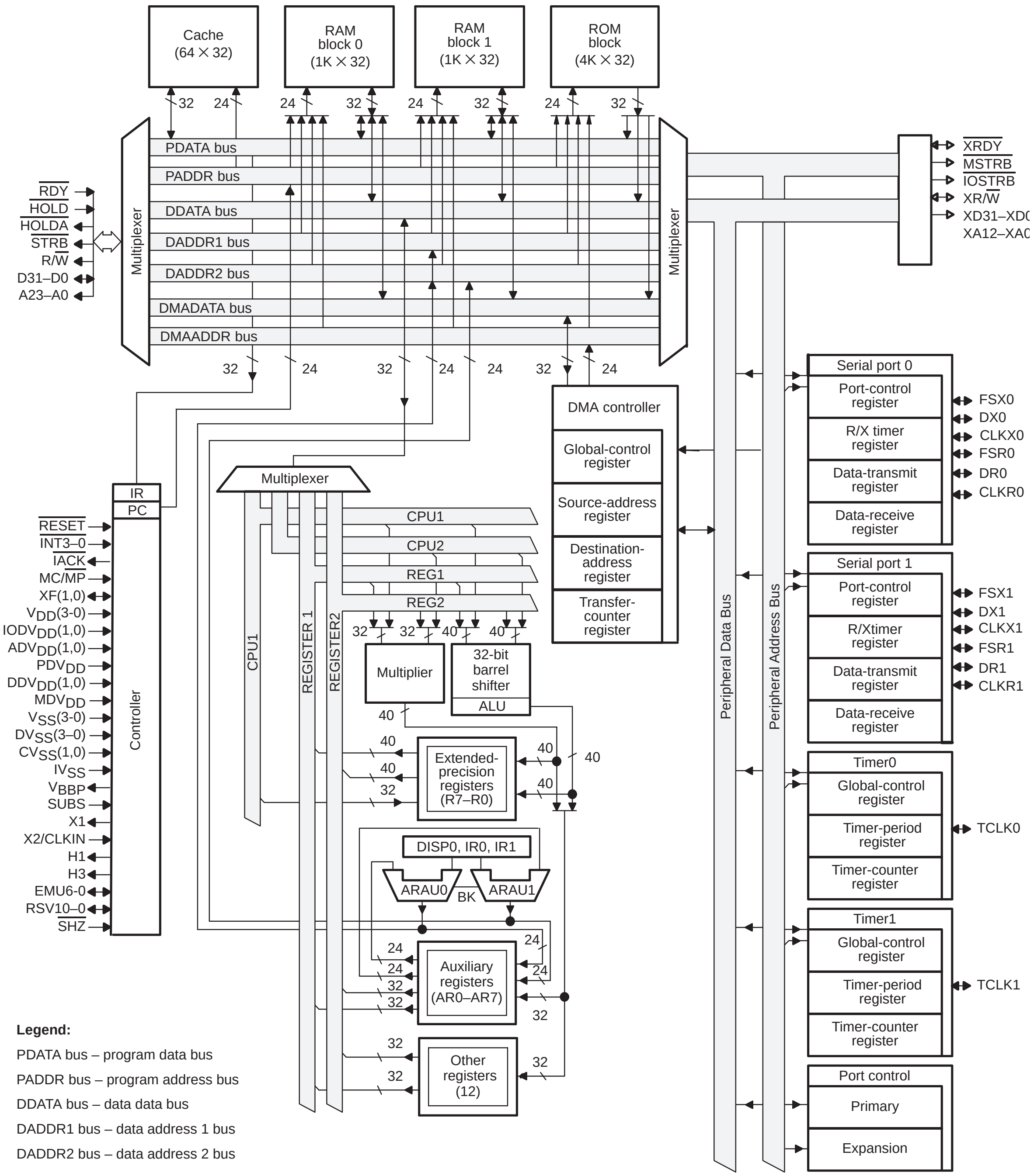
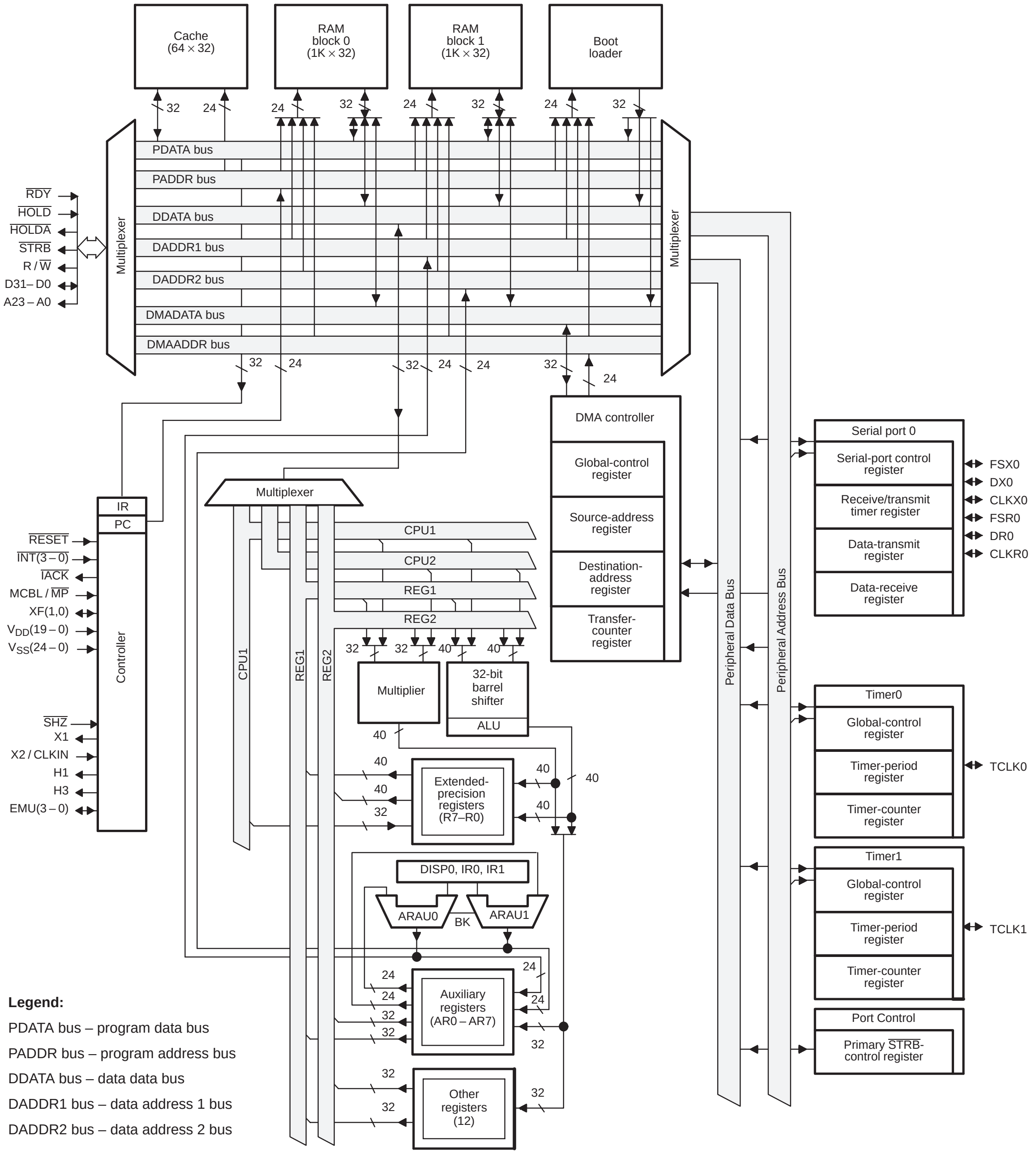


Figure 2–2. TMS320C31 Block Diagram



Legend:

PDATA bus – program data bus

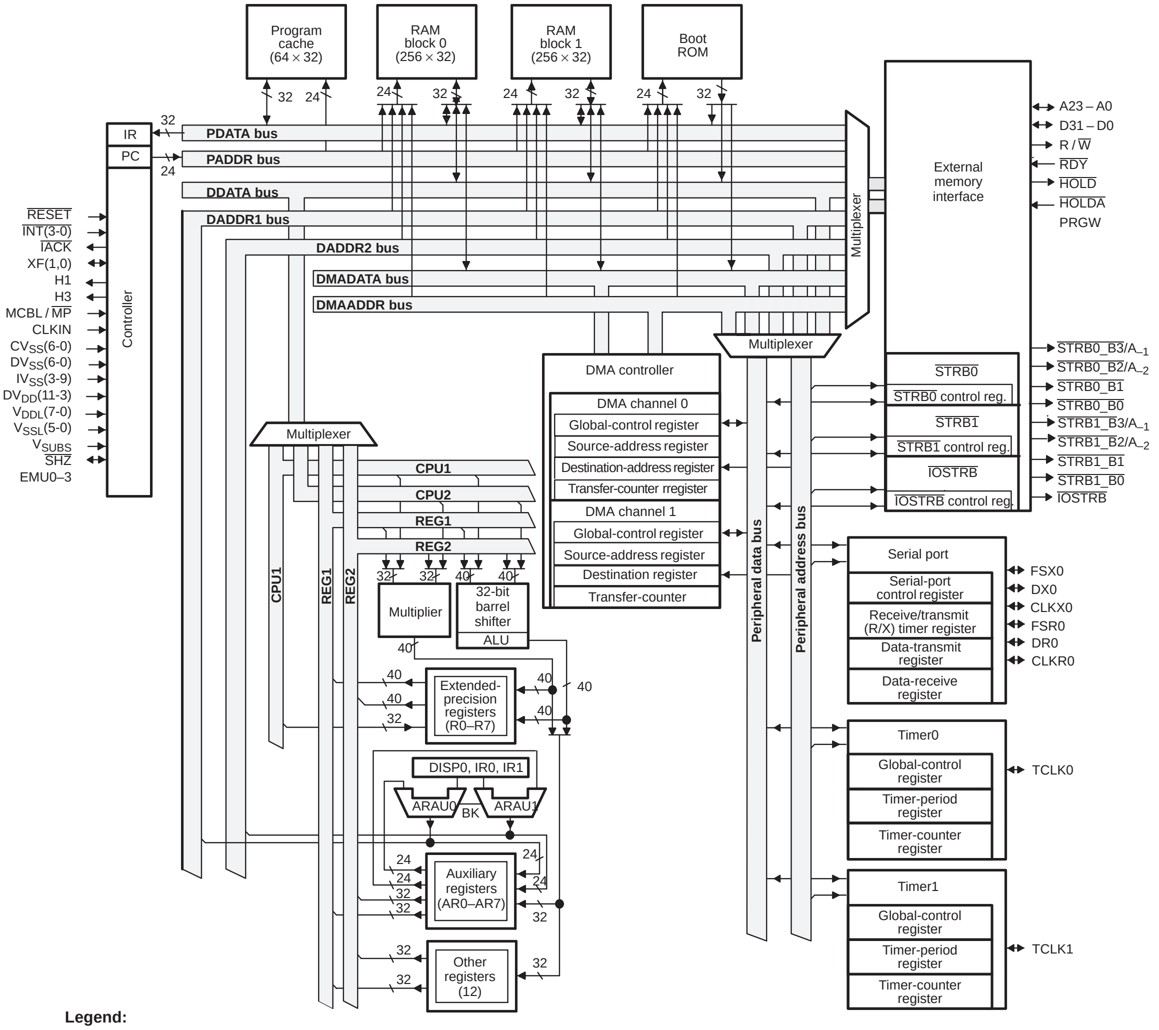
PADDR bus – program address bus

DDATA bus – data data bus

DADDR1 bus – data address 1 bus

DADDR2 bus – data address 2 bus

Figure 2–3. TMS320C32 Block Diagram



Legend:

PDATA bus – program data bus

PADDR bus – program address bus

DDATA bus – data data bus

DADDR1 bus – data address 1 bus

DADDR2 bus – data address 2 bus

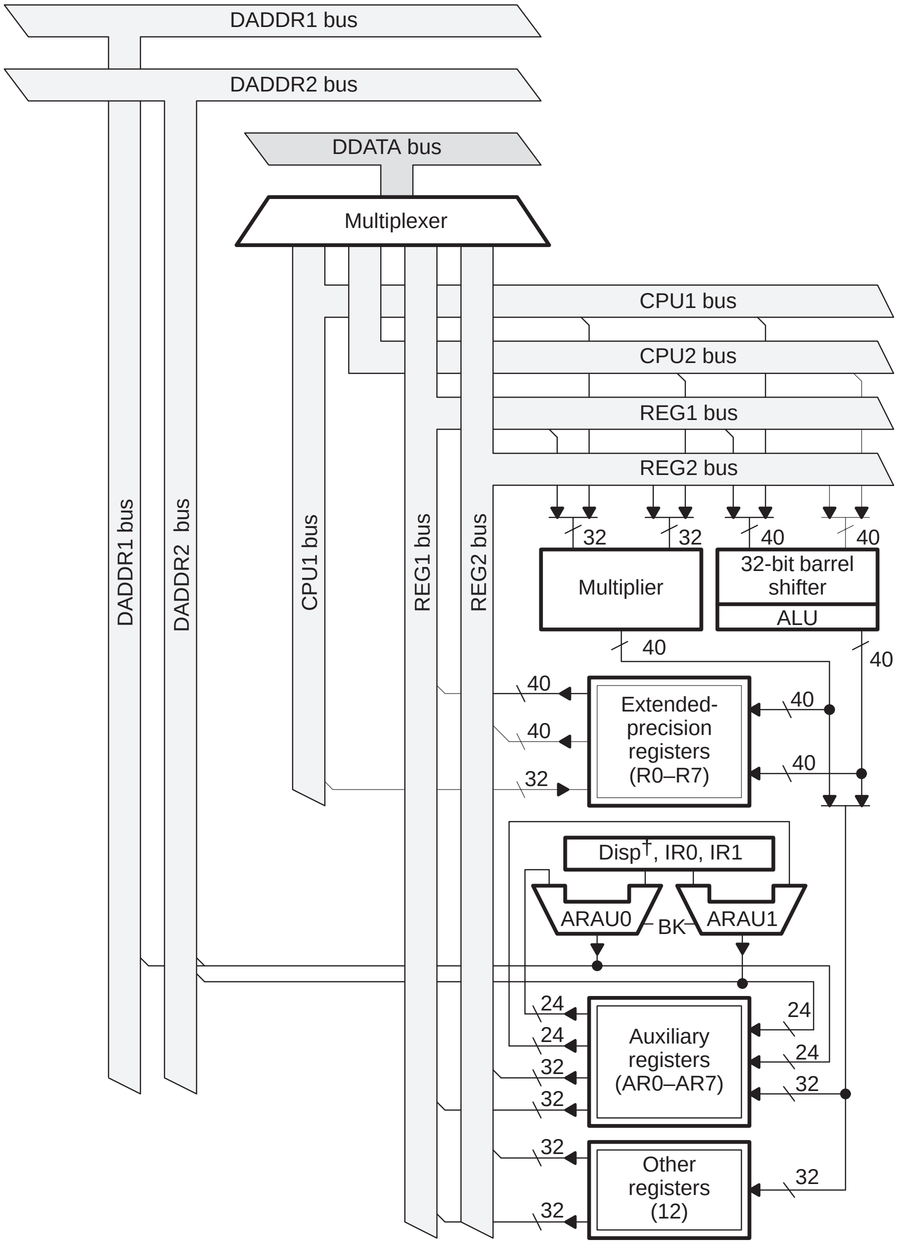
2.2 Central Processing Unit (CPU)

The ’C3x devices (’C30, ’C31, and ’C32) have a register-based CPU architecture. The CPU consists of the following components:

❑ Floating-point/integer multiplier
❑ Arithmetic logic unit (ALU)
❑ 32-bit barrel shifter
❑ Internal buses (CPU1/CPU2 and REG1/REG2)
❑ Auxiliary register arithmetic units (ARAUs)
❑ CPU register file

Figure 2–4 shows a diagram of the various CPU components.

Figure 2–4. Central Processing Unit (CPU)



† Disp = an 8-bit integer displacement carried in a program-control instruction

2.2.1 Floating-Point/Integer Multiplier

The multiplier performs single-cycle multiplications on 24-bit integer and 32-bit floating-point values. The ’C3x implementation of floating-point arithmetic allows for floating-point or fixed-point operations at speeds up to 33-ns per instruction cycle. To gain even higher throughput, you can use parallel instructions to perform a multiply and an ALU operation in a single cycle.

When the multiplier performs floating-point multiplication, the inputs are 32-bit floating-point numbers, and the result is a 40-bit floating-point number. When the multiplier performs integer multiplication, the input data is 24 bits and yields a 32-bit result. See Chapter 5, *Data Formats and Floating-Point Operation,* for detailed information.

2.2.2 Arithmetic Logic Unit (ALU) and Internal Buses

The ALU performs single-cycle operations on 32-bit integer, 32-bit logical, and 40-bit floating-point data, including single-cycle integer and floating-point conversions. Results of the ALU are always maintained in 32-bit integer or 40-bit floating-point formats. The barrel shifter is used to shift up to 32 bits left or right in a single cycle. See Chapter 5, *Data Formats and Floating-Point Operation,* for detailed information.

Four internal buses, CPU1, CPU2, REG1, and REG2 carry two operands from memory and two operands from the register file, allowing parallel multiplies and adds/subtracts on four integer or floating-point operands in a single cycle.

2.2.3 Auxiliary Register Arithmetic Units (ARAUs)

Two auxiliary register arithmetic units (ARAU0 and ARAU1) can generate two addresses in a single cycle. The ARAUs operate in parallel with the multiplier and ALU. They support addressing with displacements, index registers (IR0 and IR1), and circular and bit-reversed addressing. See Chapter 6, *Addressing Modes,* for more information.

2.3 CPU Primary Register File

The ’C3x provides 28 registers in a multiport register file that is tightly coupled to the CPU. Table 2–1 lists the register names and functions.

All of the primary registers can be operated upon by the multiplier and ALU and can be used as general-purpose registers. The registers also have some special functions. For example, the eight extended-precision registers are especially suited for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide such system functions as addressing, stack management, processor status, interrupts, and block repeat. See Chapter 3, *CPU Registers,* for more information.

Table 2–1. Primary CPU Registers

Register Name	Assigned Function	Section	Page
R0	Extended-precision register 0	3.1.1	3-3
R1	Extended-precision register 1	3.1.1	3-3
R2	Extended-precision register 2	3.1.1	3-3
R3	Extended-precision register 3	3.1.1	3-3
R4	Extended-precision register 4	3.1.1	3-3
R5	Extended-precision register 5	3.1.1	3-3
R6	Extended-precision register 6	3.1.1	3-3
R7	Extended-precision register 7	3.1.1	3-3
AR0	Auxiliary register 0	3.1.2	3-4
AR1	Auxiliary register 1	3.1.2	3-4
AR2	Auxiliary register 2	3.1.2	3-4
AR3	Auxiliary register 3	3.1.2	3-4
AR4	Auxiliary register 4	3.1.2	3-4
AR5	Auxiliary register 5	3.1.2	3-4
AR6	Auxiliary register 6	3.1.2	3-4
AR7	Auxiliary register 7	3.1.2	3-4
DP	Data-page pointer	3.1.3	3-4
IR0	Index register 0	3.1.4	3-4

Table 2–1. Primary CPU Registers (Continued)

Register Name	Assigned Function	Section	Page
IR1	Index register 1	3.1.4	3-4
BK	Block-size register	3.1.5	3-4
SP	System-stack pointer	3.1.6	3-4
ST	Status register	3.1.7	3-5
IE	CPU/DMA interrupt-enable register	3.1.8	3-9
IF	CPU interrupt flag	3.1.9	3-11
IOF	I/O flag	3.1.10	3-16
RS	Repeat start-address	3.1.11	3-17
RE	Repeat end-address	3.1.11	3-17
RC	Repeat counter	3.1.11	3-17

The **extended-precision registers (R7–R0)** can store and support operations on 32-bit integers and 40-bit floating-point numbers. Any instruction that assumes the operands are floating-point numbers uses bits 39–0. If the operands are either signed or unsigned integers, only bits 31–0 are used; bits 39–32 remain unchanged. This is true for all shift operations. See Chapter 5, *Data Formats and Floating-Point Operation,* for extended-precision register formats for floating-point and integer numbers.

The 32-bit **auxiliary registers (AR7–AR0)** are accessed by the CPU and modified by the two ARAUs. The primary function of the auxiliary registers is the generation of 24-bit addresses. They also can be used as loop counters or as 32-bit general-purpose registers that are modified by the multiplier and ALU. See Chapter 6, *Addressing Modes*, for detailed information and examples of the use of auxiliary registers in addressing.

The **data-page pointer (DP)** is a 32-bit register. The eight least significant bits (LSBs) of the data-page pointer are used by the direct addressing mode as a pointer to the page of data being addressed. Data pages are 64K words long, with a total of 256 pages.

The 32-bit **index registers (IR0, IR1)** contain the value used by the ARAU to compute an indexed address. See Chapter 6, *Addressing Modes*, for examples of the use of index registers in addressing.

The ARAU uses the 32-bit **block size register (BK)** in circular addressing to specify the data block size.

The **system-stack pointer (SP)** is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. A *push* performs a preincrement; a *pop* performs a postdecrement of the system-stack pointer. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH and POP instructions. See Section 6.10, *System and User Stack Management*, on page 6-29, for more information.

The **status register (ST)** contains global information relating to the state of the CPU. Operations usually set the condition flags of the status register according to whether the result is 0, negative, etc. These include register load and store operations as well as arithmetic and logical functions. When the status register is loaded, however, a bit-for-bit replacement is performed with the contents of the source operand, regardless of the state of any bits in the source operand. Following a load, the contents of the status register are identical to the contents of the source operand. This allows the status register to be easily saved and restored. See Table 3–2 on page 3-6 for a list and definitions of the status register bits.

The **CPU/DMA interrupt-enable register (IE)** is a 32-bit register. The CPU interrupt-enable bits are in locations 10– 0. The DMA interrupt-enable bits are in locations 26–16. A 1 in a CPU/DMA interrupt-enable register bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. See Section 3.1.8 on page 3-9 for more information.

The **CPU interrupt flag register (IF)** is also a 32-bit register. A 1 in a CPU interrupt flag register bit indicates that the corresponding interrupt is set. A 0 indicates that the corresponding interrupt is not set. See Section 3.1.9 on page 3-11 for more information.

The **I/O flag register (IOF)** controls the function of the dedicated external pins, XF0 and XF1. These pins may be configured for input or output and may also be read from and written to. See Section 3.1.10 on page 3-16 for more information.

The **repeat-counter (RC)** is a 32-bit register that specifies the number of times to repeat a block of code when performing a block repeat. When the processor is operating in the repeat mode, the 32-bit *repeat start-address register (RS)* contains the starting address of the block of program memory to repeat, and the 32-bit *repeat end-address register (RE)* contains the ending address of the block to repeat.

2.4 Other Registers

The **program-counter (PC)** is a 32-bit register containing the address of the next instruction to fetch. Although the PC is not part of the CPU register file, it is a register that can be modified by instructions that modify the program flow.

The **instruction register (IR)** is a 32-bit register that holds the instruction opcode during the decode phase of the instruction. This register is used by the instruction decode control circuitry and is not accessible to the CPU.

2.5 Memory Organization

The total memory space of the ’C3x is 16M (million) 32-bit words. Program, data, and I/O space are contained within this 16M-word address space, allowing the storage of tables, coefficients, program code, or data in either RAM or ROM. In this way, memory usage is maximized and memory space allocated as desired.

2.5.1 RAM, ROM, and Cache

Figure 2–5 shows how the memory is organized on the ’C30. RAM blocks 0 and 1 are each 1K × 32 bits. The ROM block, available only on the ’C30, is 4K × 32 bits. Each RAM and ROM block is capable of supporting two CPU accesses in a single cycle.

Figure 2–6 shows how the memory is organized on the ’C31. RAM blocks 0 and 1 are each 1K × 32 bits and support two accesses in a single cycle. A boot loader allows the loading of program and data at reset from 8-, 16-, 32-bit-wide memories or serial port.

Figure 2–7 shows how the memory is organized on the ’C32. RAM blocks 0 and 1 are each 256 × 32 bits and support two accesses in a single cycle. A boot loader allows the loading of program and data at reset from 1-, 2-, 4-, 8-, 16-, and 32-bit-wide memories or serial port. The ’C32 enhanced external memory interface provides the flexibility to address 8-, 16-, or 32-bit data independently of the external memory width. The external memory width can be 8-, 16-, or 32-bits wide.

The ’C3x’s separate program, data, and DMA buses allow for parallel program fetches, data reads and writes, and DMA operations. For example, the CPU can access two data values in one RAM block and perform an external program fetch in parallel with the DMA controller loading another RAM block, all within a single cycle.

Figure 2–5. Memory Organization of the TMS320C30

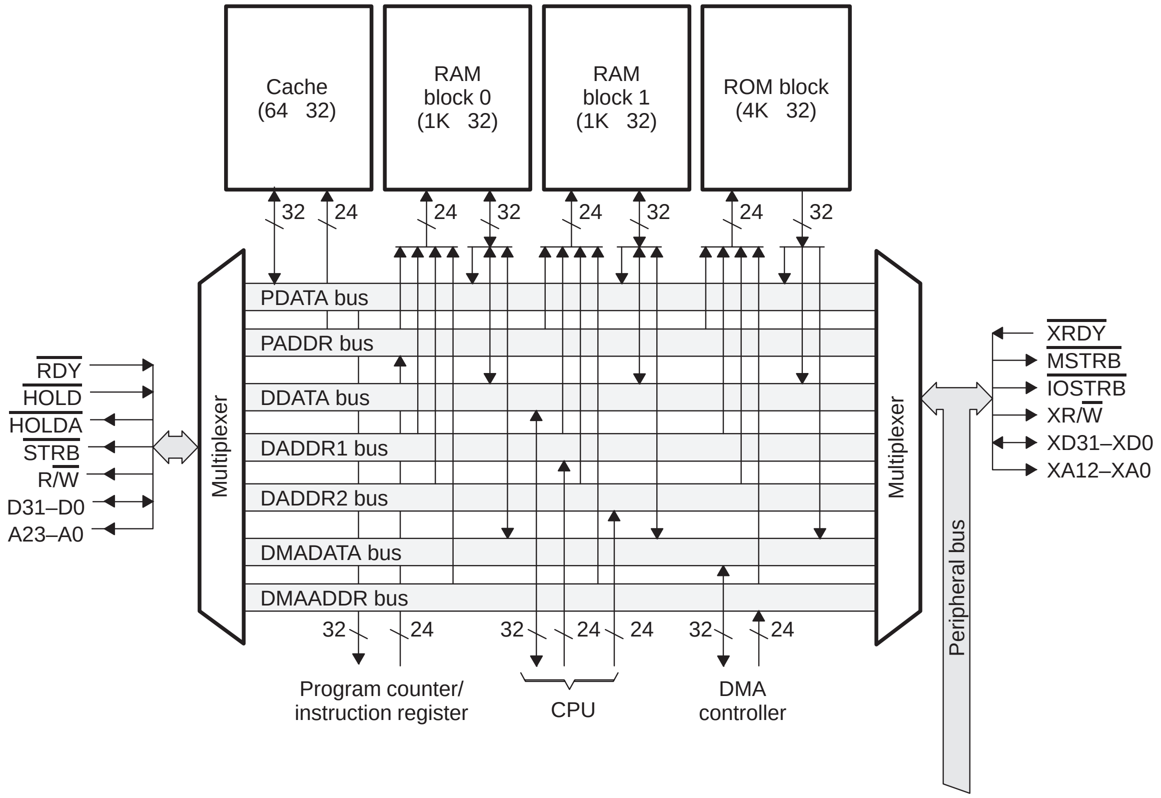


Figure 2–6. Memory Organization of the TMS320C31

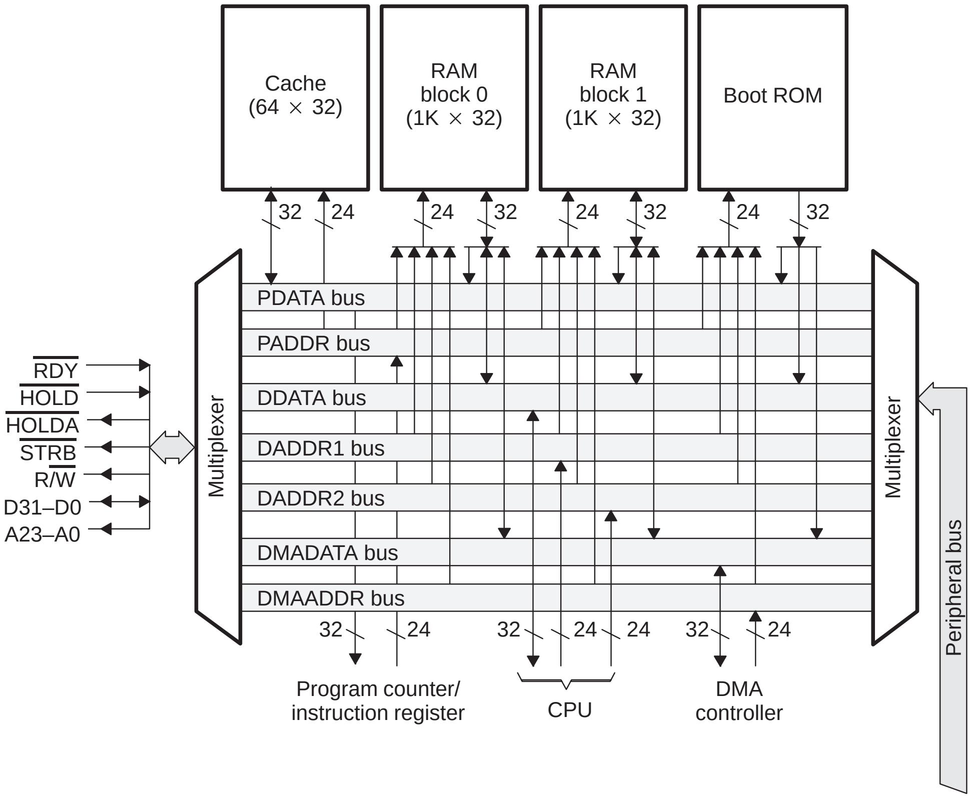
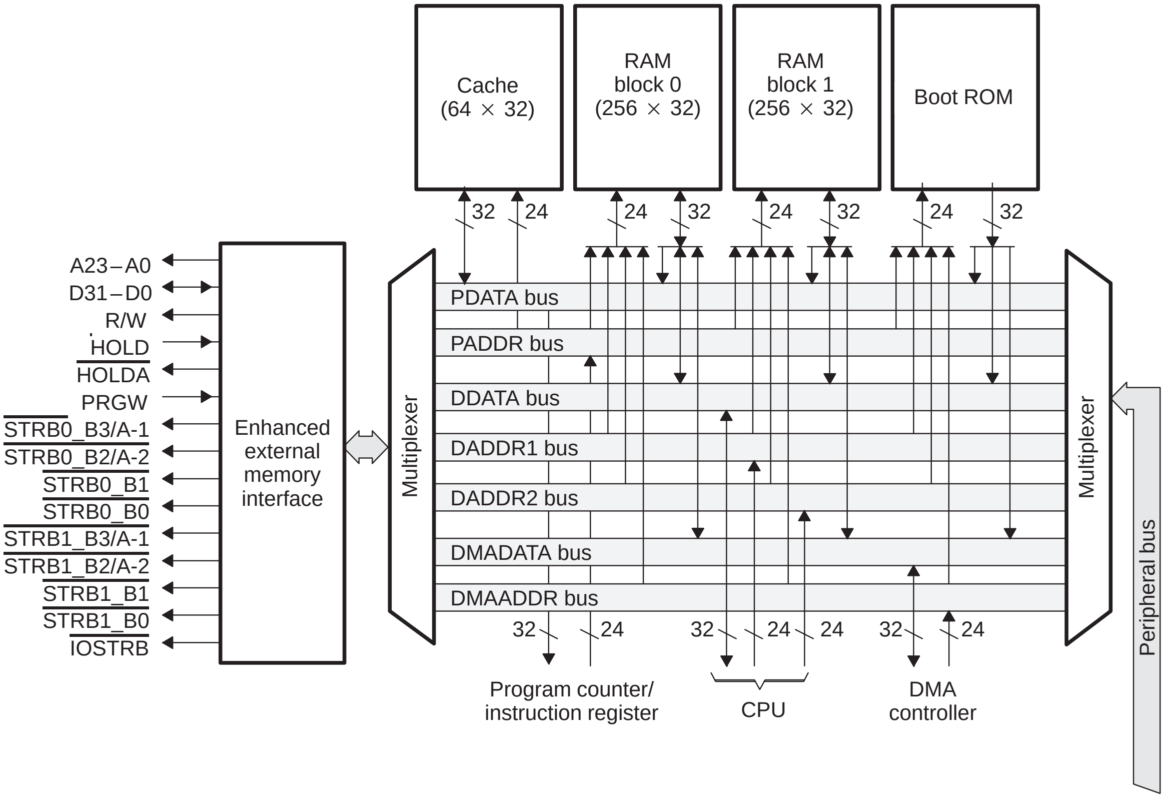


Figure 2–7. Memory Organization of the TMS320C32



A 64 × 32-bit instruction cache is provided to store often-repeated sections of code, which greatly reduces the number of off-chip accesses. This allows for code to be stored off chip in slower, lower-cost memories. The external buses are also freed for use by the DMA, external memory fetches, or other devices in the system.

See Chapter 4, *Memory and the Instruction Cache*, for more information.

2.5.2 Memory Addressing Modes

The ’C3x supports a base set of general-purpose instructions as well as arithmetic-intensive instructions that are particularly suited for digital signal processing and other numeric-intensive applications. See Chapter 6, *Addressing Modes*, for more information.

Four groups of addressing modes are provided on the ’C3x. Each group uses two or more of several different addressing types. The following list shows the addressing modes with their addressing types.

- ❑ General instruction addressing modes:
  - **Register.** The operand is a CPU register.
  - **Short immediate.** The operand is a 16-bit (short) or 24-bit (long) immediate value.
  - **Direct.** The operand is the contents of a 24-bit address formed by concatenating the 8 bits of data-page pointer and a 16-bit operand.
  - **Indirect.** An auxiliary register indicates the address of the operand.
- ❑ 3-operand instruction addressing modes:
  - **Register.** Same as for general addressing mode.
  - **Indirect.** Same as for general addressing mode.
- ❑ Parallel instruction addressing modes:
  - **Register.** The operand is an extended-precision register.
  - **Indirect.** Same as for general addressing mode.
- ❑ Branch instruction addressing modes:
  - **Register.** Same as for general addressing mode.
  - **PC-relative.** A signed 16-bit displacement or a 24-bit displacement is added to the PC.

2.6 Internal Bus Operation

Much of the ’C3x’s high performance is due to internal busing and parallelism. Separate buses allow for parallel program fetches, data accesses, and DMA accesses:

- ❑ **Program buses:** PADDR and PDATA
- ❑ **Data buses:** DADDR1, DADDR2, and DDATA
- ❑ **DMA buses:** DMAADDR and DMADATA

These buses connect all of the physical spaces (on-chip memory, off-chip memory, and on-chip peripherals) supported by the ’C3x. Figure 2–5, Figure 2–6, and Figure 2–7 show these internal buses and their connections to on-chip and off-chip memory blocks.

The program counter (PC) is connected to the 24-bit program address bus (PADDR). The instruction register (IR) is connected to the 32-bit program data bus (PDATA). These buses can fetch a single instruction word every machine cycle.

The 24-bit data address buses (DADDR1 and DADDR2) and the 32-bit data data bus (DDATA) support two data-memory accesses every machine cycle. The DDATA bus carries data to the CPU over the CPU1 and CPU2 buses. The CPU1 and CPU2 buses can carry two data-memory operands to the multiplier, ALU, and register file every machine cycle. Also internal to the CPU are register buses REG1 and REG2, which can carry two data values from the register file to the multiplier and ALU every machine cycle. Figure 2–4 shows the buses internal to the CPU section of the processor.

The DMA controller is supported with a 24-bit address bus (DMAADDR) and a 32-bit data bus (DMADATA). These buses allow the DMA to perform memory accesses in parallel with the memory accesses occurring from the data and program buses.

2.7 External Memory Interface

The ’C30 provides two external interfaces: the primary bus and the expansion bus. The ’C31 provides one external interface: the primary bus. The ’C32 provides one enhanced external interface with three independent multi-function strobes. These buses consist of a 32-bit data bus and a set of control signals. The primary and enhanced memory buses have a 24-bit address bus, whereas the expansion bus has a 13-bit address bus. These buses address external program/data memory or I/O space. The buses also have external $\overline{\text{RDY}}$ signals for wait-state generation. You can insert additional wait states under software control. Chapter 9, *External Memory Interface*, covers external bus operation.

The ’C3x family was designed for 32-bit instructions and 32-bit data operations. This architecture has many advantages, including a high degree of parallelism and provisions for a C compiler. However, the ’C30 and ’C31 require a 32-bit-wide external memory even when the data requires only 8- or 16-bit-wide memories. The ’C32 enhanced external memory interface overcomes this limitation by providing the flexibility to address 8-, 16-, or 32-bit data independently of the external memory width. In this way, the chip count and the size of external memory is reduced. The number of memory chips can be further reduced by the ’C32’s ability to allow code execution from 16- or 32-bit-wide memories. The ’C32 memory interface also reduces the total amount of RAM by allowing the physical data memory to be 8, 16, or 32 bits wide. Internally, the ’C32 has a 32-bit architecture. So you can treat the ’C32 as a 32-bit device regardless of the physical external memory width. The external memory interface handles the conversion between external memory width and ’C32 internal 32-bit architecture.

2.7.1 TMS320C32 16- and 32-Bit Program Memory

’C32

The ’C32 executes code from either 16- or 32-bit-wide memories. When connected to 32-bit memories, ‘C32 program execution is identical to that of the ’C31. When connected to 16-bit zero wait-state memory, the ’C32 takes two instruction cycles to fetch a single 32-bit instruction. During the first cycle, the ’C32 fetches the lower 16 bits. During the second cycle, the ’C32 fetches the upper 16 bits and concatenates them with the previously fetched lower 16 bits. This process occurs entirely within the memory interface and is transparent to you. An external pin, $\overline{\text{PRGW}}$, dictates the external program memory width.

2.7.2 TMS320C32 8-, 16-, and 32-Bit Data Memory

The ’C32 external memory interface can load and store 8-, 16-, or 32-bit quantities into external memory and convert them into an internally-equivalent 32-bit representation. The external memory interface accomplishes this without changing the CPU instruction set. Figure 2–8 shows the supported external memory widths, data types and sizes for zero wait-state memory and the associated cycle count.

Figure 2–8. TMS320C32-Supported Data Types and Sizes and External Memory Widths

		Memory Width		
		8	16	32
Data	8	1-cycle read	1-cycle read	1-cycle read
Type	16	2-cycle read	1-cycle read	1-cycle read
Size	32	4-cycle read	2-cycle read	1-cycle read

To access 8-, 16-, or 32-bit data quantities (types) from 8-, 16-, or 32-bit-wide memory, the memory interface uses either strobe $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$, depending on the address location within the memory map. Each strobe consists of four pins for byte enables and/or additional addresses. For a 32-bit memory interface, all four pins are used as strobe byte-enable pins. These strobe byte-enable pins select one or more bytes of the external memory. For a 16-bit memory interface, the ’C32 uses one of these pins as an additional address pin, while using two pins as strobe byte-enable pins. For an 8-bit memory interface, the ’C32 uses two of these pins as additional address pins while using one pin as strobe pin. The ’C32 manipulates the behavior of these pins according to the contents of the bus control registers (one control register per strobe). By setting a few bit fields in this register, you indicate the data-type size and external memory width.

2.8 Interrupts

The ’C3x supports four external interrupts ($\overline{\text{INT3}}$–$\overline{\text{INT0}}$), a number of internal interrupts, and a nonmaskable external $\overline{\text{RESET}}$ signal. These can be used to interrupt either the DMA or the CPU. When the CPU responds to the interrupt, the $\overline{\text{IACK}}$ pin can be used to signal an external interrupt acknowledge. Section 7.5, *Reset Operation*, on page 7-21 covers $\overline{\text{RESET}}$ and interrupt processing.

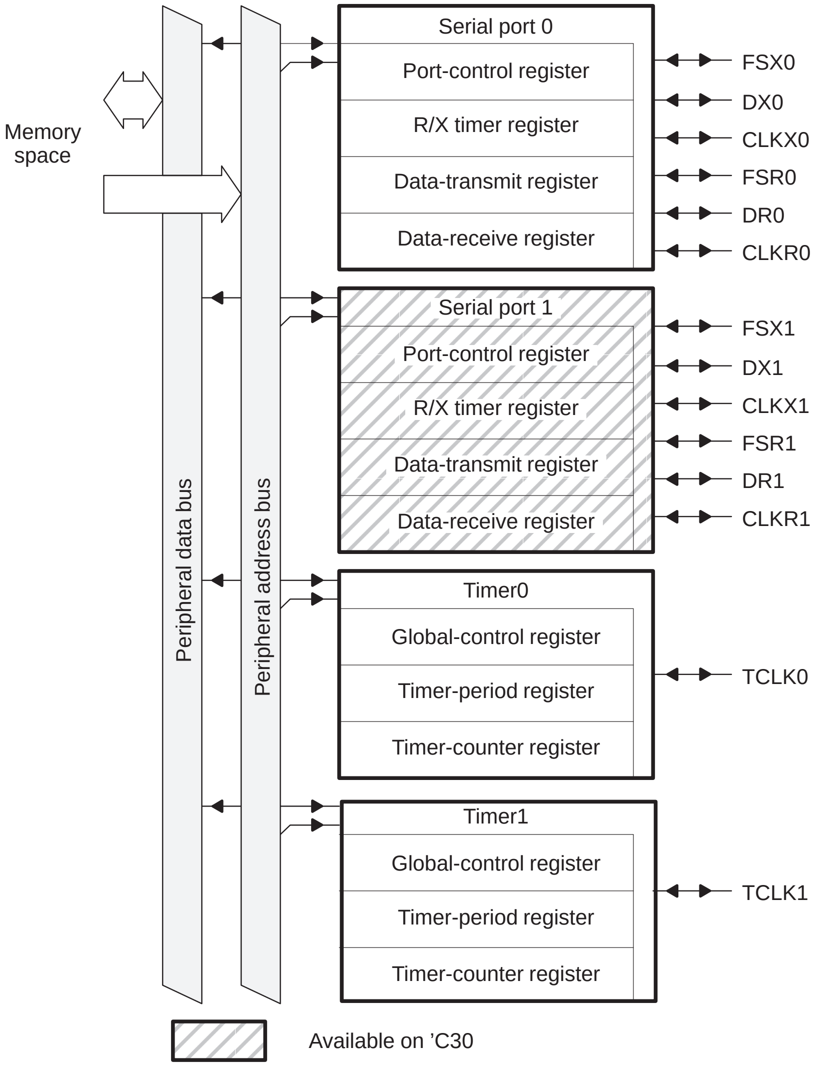
The ’C30 and ’C31 external interrupts are level-triggered. To reduce external logic and simplify the interface, the ’C32 external interrupts are edge- and level- or level-only triggered. The triggering is user-selectable through a bit in the status register. See Section 3.1.7, *Status Register (ST)*, for more information.

Two external I/O flags, $\overline{\text{XF0}}$ and $\overline{\text{XF1}}$, can be configured as input or output pins under software control. These pins are also used by the interlocked operations of the ’C3x. The interlocked-operations instruction group supports multiprocessor communication. See Section 7.4, *Interlocked Operations*, on page 7-13 for examples.

2.9 Peripherals

All ’C3x peripherals are controlled through memory-mapped registers on a dedicated peripheral bus. This peripheral bus is composed of a 32-bit data bus and a 24-bit address bus. This peripheral bus permits straightforward communication to the peripherals. The ’C3x peripherals include two timers and two serial ports (only one serial port and one DMA coprocessor are available on the ’C31 and one serial port and two DMA coprocessor channels on the ’C32). Figure 2–9 shows these peripherals with their associated buses and signals. See Chapter 12, *Peripherals*, for more information.

Figure 2–9. Peripheral Modules



2.9.1 Timers

The two timer modules are general-purpose 32-bit timer/event counters with two signaling modes and internal or external clocking. They can signal internally to the ’C3x or externally to the outside world at specified intervals or they can count external events. Each timer has an I/O pin that can be used as an input clock to the timer, as an output signal driven by the timer, or as a general-purpose I/O pin. See Chapter 12, *Peripherals*, for more information about timers.

2.9.2 Serial Ports

The bidirectional serial ports (two on ’C30, one each on the ’C31 and ’C32) are totally independent. They are identical to a complementary set of control registers that control each port. Each serial port can be configured to transfer 8, 16, 24, or 32 bits of data per word. The clock for each serial port can originate either internally or externally. An internally generated divide-down clock is provided. The pins are configurable as general-purpose I/O pins. The serial ports can also be configured as timers. A special handshake mode allows ’C3x devices to communicate over their serial ports with guaranteed synchronization.

2.10 Direct Memory Access (DMA)

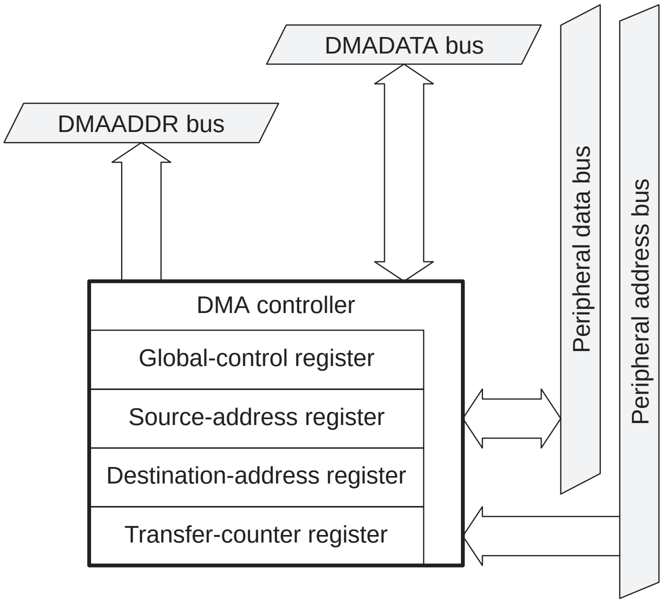
The on-chip DMA controller can read from or write to any location in the memory map without interfering with the CPU operation. The ’C3x can interface to slow, external memories and peripherals without reducing throughput to the CPU. The DMA controller contains its own address generators, source and destination registers, and transfer counter. Dedicated DMA address and data buses minimize conflicts between the CPU and the DMA controller. A DMA operation consists of a block or single-word transfer to or from memory. See Section 12.3, *DMA Controller*, on page 12-48 for more information. Figure 2–10 shows the DMA controller and its associated buses.

The ’C30 and ’C31 DMA coprocessors have one channel, while the ’C32 DMA coprocessor has two channels. Each channel of the ’C32 DMA coprocessor is equivalent to the ’C30/31 DMA with the addition of user-configurable priorities. Because the DMA and CPU have distinct buses on the ’C3x devices, they can operate independently of each other. However, when the CPU and DMA access the same on-chip or external resources, the bandwidth can be exceeded and priorities must be established. The ’C30 and ’C31 assign highest priority to the CPU. The ’C32 DMA coprocessor provides more flexibility by allowing you to choose one of the following priorities:

- **CPU:** For all resource conflicts, the CPU has priority over the DMA.
- **DMA:** For all resource conflicts, the DMA has priority over the CPU.
- **Rotating:** When the CPU and DMA have a resource conflict during consecutive instruction cycles, the CPU is granted priority. On the following cycle, the DMA is granted priority. Alternate access continues as long as the CPU and DMA requests conflict in consecutive instruction cycles.

The DMA/CPU priority is configured by the DMA PRI bit fields of the corresponding DMA global-control register. See Section 12.3, *DMA Controller*, on page 12-48 for a complete description.

Figure 2–10. DMA Controller



2.11 TMS320C30, TMS320C31, and TMS320C32 Differences

Table 2–2 shows the major differences between the ’C32, ’C31, and the ’C30 devices.

Table 2–2. Feature Set Comparison

Feature	**’C30**	**’C31**	**’C32**
External bus	Two buses: ❑ Primary bus: 32-bit data 24-bit address STRB active for 0h–7FFFFFh and 80A000h–FFFFFFh ❑ Expansion bus: 32-bit data 13-bit address MSTRB active for 800000h–801FFFh IOSTRB active for 804000h–805FFFh	One bus: 32-bit data 24-bit address STRB active 0h–7FFFFFh and 80A000h–FFFFFFh	One bus: ❑ 32-bit data 24-bit address STRB0 active for 0h–7FFFFFh and 880000h–8FFFFFh; ❑ 8-, 16-, 32-bit data in 8-, 16-, 32-bit-wide memory STRB1 active for 900000h–FFFFFFh; ❑ 8-, 16-, 32-bit data in 8-, 16-, 32- bit-wide memory IOSTRB active for 810000h–82FFFFh
ROM	4k	No	No
Boot loader	No	Yes	Yes
On-chip RAM	2k address: 809800h–809FFFh	2k address: 809800h–809FFFh	512 address: 87FE00h–87FFFFh
DMA	1 channel CPU greater priority than DMA	1 channel CPU greater priority than DMA	2 channels Configurable priorities
Serial ports	2	1	1
Timers	2	2	2
Interrupts	Level-triggered	Level-triggered	Level-triggered or combination of edge- and level-triggered
Interrupt vector table	Fixed 0–3Fh	Microprocessor: 0–3Fh fixed Boot loader: 809C1h–809FFFh fixed	Relocatable
Package	208 PQFP 181 PGA	132 PQFP	144 PQFP
Voltage	5 V	5 V and 3.3 V	5 V
Temperature	0° to 85°C (commercial) –40 to 125°C (extended) –55 125°C (military)	0° to 85°C (commercial) –40 to 125°C (extended) –55 125°C (military)	0° to 85°C (commercial) –40 to 125°C (extended) –55 125°C (military)

Chapter 3

CPU Registers

The central processing unit (CPU) register file contains 28 registers that can be operated on by the multiplier and arithmetic logic unit (ALU). Included in the register file are the auxiliary registers, extended-precision registers, and index registers.

Three registers in the ’C32 CPU register file have been modified to support new features (2-channel DMAs, program execution from 16-bit memory width, etc.) The registers modified in the ’C32 are: the status (ST) register, interrupt-enable (IE) register, and interrupt flag (IF) register.

Topic **Page**

3.1 CPU Multiport Register File

The ’C3x provides 28 registers in a multiport register file that is tightly coupled to the CPU. The program counter (PC) is not included in the 28 registers. All of these registers can be operated on by the multiplier and the ALU and can be used as general-purpose 32-bit registers.

Table 3–1 lists the registers’ names and assigned functions of the ’C3x.

Table 3–1. CPU Registers

Register Symbol	Register Machine Value (hex)	Assigned Function Name	Section	Page
R0	00	Extended-precision register 0	3.1.1	3-3
R1	01	Extended-precision register 1	3.1.1	3-3
R2	02	Extended-precision register 2	3.1.1	3-3
R3	03	Extended-precision register 3	3.1.1	3-3
R4	04	Extended-precision register 4	3.1.1	3-3
R5	05	Extended-precision register 5	3.1.1	3-3
R6	06	Extended-precision register 6	3.1.1	3-3
R7	07	Extended-precision register 7	3.1.1	3-3
AR0	08	Auxiliary register 0	3.1.2	3-4
AR1	09	Auxiliary register 1	3.1.2	3-4
AR2	0A	Auxiliary register 2	3.1.2	3-4
AR3	0B	Auxiliary register 3	3.1.2	3-4
AR4	0C	Auxiliary register 4	3.1.2	3-4
AR5	0D	Auxiliary register 5	3.1.2	3-4
AR6	0E	Auxiliary register 6	3.1.2	3-4
AR7	0F	Auxiliary register 7	3.1.2	3-4
DP	10	Data-page pointer	3.1.3	3-4
IR0	11	Index register 0	3.1.4	3-4
IR1	12	Index register 1	3.1.4	3-4
BK	13	Block-size register	3.1.5	3-4
SP	14	System-stack pointer	3.1.6	3-4
ST	15	Status register	3.1.7	3-5
IE	16	CPU/DMA interrupt-enable	3.1.8	3-9
IF	17	CPU interrupt flags	3.1.9	3-11
IOF	18	I/O flags	3.1.10	3-16
RS	19	Repeat start-address	3.1.11	3-17
RE	1A	Repeat end-address	3.1.11	3-17
RC	1B	Repeat counter	3.1.11	3-17

The registers also have some special functions for which they are particularly appropriate. For example, the eight extended-precision registers are especially suited for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions, such as addressing, stack management, processor status, interrupts, and block repeat. See Chapter 6, *Addressing Modes*, for more information.

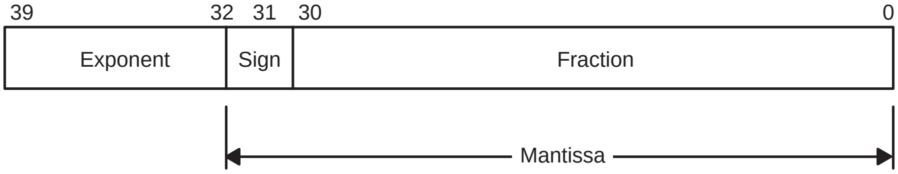
3.1.1 Extended-Precision Registers (R7–R0)

The eight extended-precision registers (R7–R0) can store and support operations on 32-bit integer and 40-bit floating-point numbers. These registers consist of two separate and distinct regions:

- Bits 39–32: dedicated to storage of the exponent (e) of the floating-point number.
- Bits 31–0: store the mantissa of the floating-point number:
  - Bit 31: sign bit (s)
  - Bits 30–0: the fraction (f)

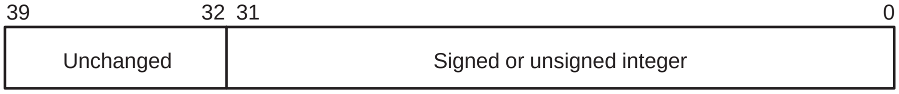
Any instruction that assumes the operands are floating-point numbers uses bits 39–0. Figure 3–1 illustrates the storage of 40-bit floating-point numbers in the extended-precision registers.

Figure 3–1. Extended-Precision Register Floating-Point Format



For integer operations, bits 31–0 of the extended-precision registers contain the integer (signed or unsigned). Any instruction that assumes the operands are either signed or unsigned integers uses only bits 31–0. Bits 39–32 remain unchanged. This is true for all shift operations. The storage of 32-bit integers in the extended-precision registers is shown in Figure 3–2.

Figure 3–2. Extended-Precision Register Integer Format



3.1.2 Auxiliary Registers (AR7–AR0)

The CPU can access the eight 32-bit auxiliary registers (AR7–AR0), and the two auxiliary register arithmetic units (ARAUs) can modify them. The primary function of the auxiliary registers is the generation of 24-bit addresses. However, they can also operate as loop counters in indirect addressing or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. See Chapter 6, *Addressing Modes*, for more information.

3.1.3 Data-Page Pointer (DP)

The data-page pointer (DP) is a 32-bit register that is loaded using the load data page (LDP) instruction (see Chapter 13, *Assembly Language Instructions*). The eight LSBs of the data-page pointer are used by the direct addressing mode as a pointer to the page of data being addressed (see Section 6.3, *Direct Addressing*, on page 6-4). Data pages are 64K-words long, with a total of 256 pages. Bits 31–8 are reserved; you must always keep these set to 0 (cleared).

3.1.4 Index Registers (IR0, IR1)

The 32-bit index registers (IR0 and IR1) are used by the ARAU for indexing the address. See Chapter 6, *Addressing Modes*, for more information.

3.1.5 Block Size (BK) Register

The 32-bit block size register (BK) is used by the ARAU in circular addressing to specify the data block size. See Section 6.7, *Circular Addressing*, on page 6-21 for more information.

3.1.6 System-Stack Pointer (SP)

The system-stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH, PUSHF, POP, and POPF instructions. Stack pushes and pops perform preincrements and postdecrements on all 32 bits of the SP. However, only the 24 LSBs are used as an address. See Section 6.10, *System and User Stack Management*, on page 6-29 for more information.

3.1.7 Status (ST) Register

The status (ST) register contains global information about the state of the CPU. Operations usually set the condition flags of the status register according to whether the result is 0, negative, etc. This includes register load and store operations as well as arithmetic and logical functions. However, when the status register is loaded, the contents of the source operand replace the ST’s contents bit for bit, regardless of the state of any bits in the source operand. Following an ST load, the contents of the status register are identical to the contents of the source operand. This allows the status register to be saved easily and restored. At system reset, a 0 is written to this register.

Figure 3–3 shows the format of the status register for the ’C30 and ’C31 devices. Figure 3–4 shows the format of the status register for the ’C32 device. Table 3–2 describes the status register bits, their names, and their functions.

Figure 3–3. Status Register (TMS320C30 andTMS320C31)

’C30 ’C31

31 – 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	GIE	CC	CE	CF	xx	RM	OVM	LUF	LV	UF	N	Z	V	C
			R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) R = read, W = write

Figure 3–4. Status Register (TMS320C32 Only)

’C32

31 – 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	PRGW status	INT config	GIE	CC	CE	CF	xx	RM	OVM	LUF	LV	UF	N	Z	V	C
	R	R/W	R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) R = read, W = write

Table 3–2. Status Register Bits

Bit Name	Reset Value	Name	Description
C	0	Carry flag	Carry condition flag
V	0	Overflow flag	Overflow condition flag
Z	0	Zero flag	Zero condition flag
N	0	Negative flag	Negative condition flag
UF	0	Floating-point underflow flag	Floating-point underflow condition flag
LV	0	Latched overflow flag	Latched overflow condition flag
LUF	0	Latched floating-point underflow flag	Latched floating-point underflow condition flag
OVM	0	Overflow mode flag	Overflow mode flag The overflow mode flag affects only integer operations. If OVM = 0, the overflow mode is turned off and integer results that overflow are treated in no special way. If OVM = 1, integer results overflowing in the positive direction are set to the most positive, 2s-complement number (7FFF FFFFh), and integer results overflowing in the negative direction are set to the most negative 32-bit, 2s-complement number (8000 0000h).
RM	0	Repeat mode flag	Repeat mode flag If RM = 1, the PC is modified in either the repeat-block or repeat-single mode.
CE	0	Cache enable	CE enables or disables the instruction cache. Set CE = 1 to enable the cache, allowing the cache to be used according to the least recently used (LRU) stack manipulation. Set CE = 0 to disable the cache, preventing cache updates or modifications (no cache fetches can be made). Cache clearing (CC = 1) is allowed when CE = 0.

Note: If a load of the status register occurs simultaneously with a CPU interrupt pulse trying to reset GIE, GIE is reset.

Table 3–2. Status Register Bits (Continued)

<table>
<tr><th>Bit Name</th><th>Reset Value</th><th>Name</th><th>Description</th></tr>
<tr><td>CF</td><td>0</td><td>Cache freeze</td><td>Enables or disables the instruction cache
Set CF = 1 to freeze the cache (cache is not updated), including LRU stack manipulation. If the cache is enabled (CE = 1), fetches from the cache are allowed, but modification of the cache contents is not allowed. Cache clearing (CC = 1) is allowed. At reset, this bit is cleared to 0, but it is set to 1 after reset.
When CF = 0, the cache is automatically updated by instruction fetches from external memory. Also, when CF = 0, cache clearing (CC = 1) is allowed.
The following table summarizes the CE and CF bits:
<table>
<tr><th>CE</th><th>CF</th><th>Effect</th></tr>
<tr><td>0</td><td>0</td><td>Cache not enabled</td></tr>
<tr><td>0</td><td>1</td><td>Cache not enabled</td></tr>
<tr><td>1</td><td>0</td><td>Cache enabled and not frozen</td></tr>
<tr><td>1</td><td>1</td><td>Cache enabled but frozen (cache read only)</td></tr>
</table></td></tr>
<tr><td>CC</td><td>0</td><td>Cache clear</td><td>CC = 1 invalidates all entries in the cache. This bit is always cleared after it is written to, and is always read as 0. At reset, 0 is written to this bit.</td></tr>
<tr><td>GIE</td><td>0</td><td>Global interrupt-enable</td><td>If GIE = 1, the CPU responds to an enabled interrupt.
If GIE = 0, the CPU does not respond to an enabled interrupt.</td></tr>
<tr><td>INT config</td><td>0</td><td>Interrupt configuration (‘C32 only)</td><td>Sets the external interrupt signals INT3 – INT0 for level- or edge-triggered interrupts.
<table>
<tr><th>INT Config</th><th>Effect</th></tr>
<tr><td>0</td><td>All the external interrupts (INT3 – INT0) are configured as <b>level-triggered interrupts.</b> Multiple interrupts may be triggered when the signal is active for a long period of time.</td></tr>
<tr><td>1</td><td>All the external interrupts (INT3 – INT0) are configured as <b>edge-triggered interrupts.</b> Edge and duration are required for all interrupts to be recognized.</td></tr>
</table></td></tr>
</table>

Note: If a load of the status register occurs simultaneously with a CPU interrupt pulse trying to reset GIE, GIE is reset.

Table 3–2. Status Register Bits (Continued)

Bit Name	Reset Value	Name	Description
PRGW	Dependent on PRGW pin level	Program width status (‘C32 only)	Indicates the status of the external input PRGW pin. When the signal of the PRGW pin is high, the PRGW status bit is set to 1, indicating a 16-bit memory width. The ‘C32 performs two fetches to retrieve a single 32-bit instruction word. The PRGW bit is a read-only bit, and can have the following values: **PRG** **Effect** 0 Instruction fetches use one 32-bit external program memory read. 1 Instruction fetches use two 16-bit external program memory reads.

Note: If a load of the status register occurs simultaneously with a CPU interrupt pulse trying to reset GIE, GIE is reset.

3.1.8 CPU/DMA Interrupt-Enable (IE) Register

The CPU/DMA interrupt-enable (IE) register of the ’C30, ’C31, and ’C32 are 32-bit registers (see Figure 3–5 and Figure 3–6). The CPU interrupt-enable bits are in locations 10–0 for ’C30 and ’C31 devices, and 11–0 for ’C32 devices. The direct memory access (DMA) interrupt-enable bits are in locations 26–16 for ‘C30 and ‘C31 devices, and 31–16 for ’C32 devices. A 1 in a CPU/DMA IE bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. At reset, 0 is written to this register.

Table 3–3 describes the interrupt-enable register bits, their names, and their functions.

Figure 3–5. CPU/DMA Interrupt-Enable (IE) Register (TMS320C30 and TMS320C31)

’C30 ’C31

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	EDINT (DMA)	ETINT1 (DMA)	ETINT0 (DMA)	ERINT1 (DMA)	EXINT1 (DMA)	ERINT0 (DMA)	EXINT0 (DMA)	EINT3 (DMA)	EINT2 (DMA)	EINT1 (DMA)	EINT0 (DMA)
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	xx	EDINT (CPU)	ETINT1 (CPU)	ETINT0 (CPU)	ERINT1 (CPU)	EXINT1 (CPU)	ERINT0 (CPU)	EXINT0 (CPU)	EINT3 (CPU)	EINT2 (CPU)	EINT1 (CPU)	EINT0 (CPU)
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) R = read, W = write

Figure 3–6. CPU/DMA Interrupt-Enable (IE) Register (TMS320C32)

’C32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
EINT3 (DMA1)	EINT2 (DMA1)	EINT1 (DMA1)	EINT0 (DMA1)	EDINT0 (DMA1)	EDINT1 (DMA0)	ETINT1 (DMA0)	ETINT0 (DMA0)	ETINT1 (DMA1)	ETINT0 (DMA1)	ERINT0 (DMA1)	EXINT0 (DMA0)	EINT3 (DMA0)	EINT2 (DMA0)	EINT1 (DMA0)	EINT0 (DMA0)
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	EDINT1 (CPU)	EDINT0 (CPU)	ETINT1 (CPU)	ETINT0 (CPU)	xx	xx	ERINT0 (CPU)	EXINT0 (CPU)	EINT3 (CPU)	EINT2 (CPU)	EINT1 (CPU)	EINT0 (CPU)
				R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) R = read, W = write

Table 3–3. IE Bits and Functions

Abbreviation	Reset Value	Description
EINT0 (CPU)	0	CPU external interrupt 0 enable
EINT1 (CPU)	0	CPU external interrupt 1 enable
EINT2 (CPU)	0	CPU external interrupt 2 enable
EINT3 (CPU)	0	CPU external interrupt 3 enable
EXINT0 (CPU)	0	CPU serial port 0 transmit interrupt enable
ERINT0 (CPU)	0	CPU serial port 0 receive interrupt enable
EXINT1 (CPU)	0	CPU serial port 1 transmit interrupt enable (’C30 only)
ERINT1 (CPU)	0	CPU serial port 1 receive interrupt enable (’C30 only)
ETINT0 (CPU)	0	CPU timer0 interrupt enable
ETINT1 (CPU)	0	CPU timer1 interrupt enable
EDINT (CPU)	0	CPU DMA controller interrupt enable (’C30 and ’C31 only)
EDINT0 (CPU)	0	CPU DMA0 controller interrupt enable (’C32 only)
EDINT1 (CPU)	0	CPU DMA1 controller interrupt enable (’C32 only)
EINT0 (DMA)	0	DMA external interrupt 0 enable (’C30 and ’C31 only)
EINT1 (DMA)	0	DMA external interrupt 1 enable (’C30 and ’C31 only)
EINT2 (DMA)	0	DMA external interrupt 2 enable (’C30 and ’C31 only)
EINT3 (DMA)	0	DMA external interrupt 3 enable (’C30 and ’C31 only)
EINT0 (DMA0)	0	DMA0 external interrupt 0 enable (’C32 only)
EINT1 (DMA0)	0	DMA0 external interrupt 1 enable (’C32 only)
EINT2 (DMA0)	0	DMA0 external interrupt 2 enable (’C32 only)
EINT3 (DMA0)	0	DMA0 external interrupt 3 enable (’C32 only)
EXINT0 (DMA)	0	DMA serial port 0 transmit interrupt enable (’C30 and ’C31 only)
ERINT0 (DMA)	0	DMA serial port 0 receive interrupt enable (’C30 and ’C31 only)
EXINT1 (DMA)	0	DMA serial port 1 transmit interrupt enable (’C30 only)
ERINT1 (DMA)	0	DMA serial port 1 receive interrupt enable (’C30 only)
EXINT0 (DMA0)	0	DMA0 serial port 1 transmit interrupt enable (’C32 only)
ERINT0 (DMA1)	0	DMA1 serial port 1 receive interrupt enable (’C32 only)

Table 3–3. IE Bits and Functions(Continued)

Abbreviation	Reset Value	Description
ETINT0 (DMA)	0	DMA timer0 interrupt enable (’C30 and ’C31)
ETINT1 (DMA)	0	DMA timer1 interrupt enable (’C30 and ’C31 only)
ETINT0 (DMA0)	0	DMA0 timer1 interrupt enable (’C32 only)
ETINT1 (DMA0)	0	DMA0 timer1 interrupt enable (’C32 only)
ETINT0 (DMA1)	0	DMA1 timer0 interrupt enable (’C32 only)
ETINT1 (DMA1)	0	DMA1 timer1 interrupt enable (’C32 only)
EDINT (DMA)	0	DMA controller interrupt enable (’C30 and ’C31 only)
EDINT1 (DMA0)	0	DMA0-DMA1 controller interrupt enable (’C32 only)
EDINT0 (DMA1)	0	DMA1-DMA0 controller interrupt enable (’C32 only)
EINT0 (DMA1)	0	DMA1 external interrupt 0 enable (’C32 only)
EINT1 (DMA1)	0	DMA1 external interrupt 1 enable (’C32 only)
EINT2 (DMA1)	0	DMA1 external interrupt 2 enable (’C32 only)
EINT3 (DMA1)	0	DMA1 external interrupt 2 enable (’C32 only)

3.1.9 CPU Interrupt Flag (IF) Register

Figure 3–7, Figure 3–8, and Figure 3–9 show the 32-bit CPU interrupt flag registers (IF) for the ‘C30, ‘C31, and ‘C32 devices, respectively. A 1 in a CPU IF register bit indicates that the corresponding interrupt is set. The IF bits are set to 1 when an interrupt occurs. They may also be set to 1 through software to cause an interrupt. A 0 indicates that the corresponding interrupt is not set. If a 0 is written to an IF register bit, the corresponding interrupt is cleared. At reset, 0 is written to this register. Table 3–4 describes the interrupt flag register bits, their names, and their functions.

’C30

Figure 3–7. TMS320C30 CPU Interrupt Flag (IF) Register

31–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
xx	yy	yy	DINT	TINT1	TINT0	RINT1	XINT1	RINT0	XINT0	INT3	INT2	INT1	INT0
		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) yy = reserved bit, set to 0 at reset; can store value

3) R = read, W = write

’C31

Figure 3–8. TMS320C31 CPU Interrupt Flag (IF) Register

31–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
xx	yy	yy	DINT	TINT1	TINT0	xx	xx	RINT0	XINT0	INT3	INT2	INT1	INT0
			R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) yy = reserved bit, set to 0 at reset

3) R = read, W = write

’C32

Figure 3–9. TMS320C32 CPU Interrupt Flag (IF) Register

31–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
ITTP	xx	DINT1	DINT0	TINT1	TINT0	xx	xx	RINT0	XINT0	INT3	INT2	INT1	INT0
		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

2) R = read, W = write

Table 3–4. IF Bits and Functions

Bit Name	Reset Value	Function
INT0	0	External interrupt 0 flag
INT1	0	External interrupt 1 flag
INT2	0	External interrupt 2 flag
INT3	0	External interrupt 3 flag
XINT0	0	Serial port 0 transmit flag
RINT0	0	Serial port 0 receive flag
XINT1	0	Serial port 1 transmit flag (‘C30 only)
RINT1	0	Serial port 1 receive interrupt flag (‘C30 only)
TINT0	0	Timer 0 interrupt flag
TINT1	0	Timer 1 interrupt flag
DINT	0	DMA channel interrupt flag (‘C30 and ‘C31 only)
DINT0	0	DMA0 channel interrupt flag (‘C32 only)
DINT1	0	DMA1 channel interrupt flag (‘C32 only)
ITTP	0	Interrupt-trap table pointer (see Section 3.1.9.1) Allows the relocation of interrupt and trap vector tables (‘C32 only)

Note: If a load of the interrupt flag (IF) register occurs simultaneously with a set of a flag by an interrupt pulse, the loading of the flag has higher priority and overwrites the value of the interrupt flag register.

3.1.9.1 Interrupt-Trap Table Pointer (ITTP)

’C32

Similar to the rest of the ‘C3x device family, the ’C32’s reset vector location remains at address 0. However, the interrupt and trap vectors are relocatable. This is achieved by the interrupt-trap table pointer (ITTP) bit field in the CPU interrupt flag register, shown in Figure 3–9. The ITTP bit field dictates the starting location (base) of the interrupt-trap vector table. This base address is formed by left shifting by eight bits the value of the ITTP bit field. This shifted value is called the effective base address and is referenced as EA[ITTP], as shown in Figure 3–10. Therefore, the location of an interrupt or trap vector is given by the addition of the effective base address formed by the ITTP bit field (EA[ITTP]) and the offset of the interrupt or trap vector in the interrupt-trap vector table, as shown in Figure 3–11. For example, if the ITTP contains the value 100h, the serial port transmit interrupt vector is located at 10005h. Note that the vectors stored in the interrupt-trap vector table are the addresses of the start of the respective interrupt and trap routines. Furthermore, the interrupt-trap vector table must lie on a 256-word boundary, since the eight LSBs of the effective base address of the interrupt-trap vector table are 0.

See Section 7.6, *Interrupts*, on page 7-26 for more information on interrupt vector tables.

Figure 3–10. Effective Base Address of the Interrupt-Trap Vector Table

	23 8	7 0
EA[ITTP] =	Bits 31 – 16 of the CPU interrupt flag register	00000000

Figure 3–11. Interrupt and Trap Vector Locations

’C32

EA (ITTP) + 00h	Reserved
EA (ITTP) + 01h	INT0
EA (ITTP) + 02h	INT1
EA (ITTP) + 03h	INT2
EA (ITTP) + 04h	INT3
EA (ITTP) + 05h	XINT0
EA (ITTP) + 06h	RINT0
EA (ITTP) + 07h	Reserved
EA (ITTP) + 08h	Reserved
EA (ITTP) + 09h	TINT0
EA (ITTP) + 0Ah	TINT1
EA (ITTP) + 0Bh	DINT0
EA (ITTP) + 0Ch	DINT1
EA (ITTP) + 0Dh	Reserved
EA (ITTP) + 1Fh	
EA (ITTP) + 20h	TRAP0
	⋮
EA (ITTP) + 3Bh	TRAP27
EA (ITTP) + 3Ch	TRAP28 (reserved)
EA (ITTP) + 3Dh	TRAP29 (reserved)
EA (ITTP) + 3Eh	TRAP30 (reserved)
EA (ITTP) + 3Fh	TRAP31 (reserved)

3.1.10 I/O Flag (IOF) Register

The I/O flag (IOF) register is shown in Figure 3–12 and controls the function of the dedicated external pins, XF0 and XF1. These pins can be configured for input or output. The pins can also be read from and written to. At reset, 0 is written to this register. Table 3–5 describes the I/O flags register bits, their names, and their functions.

Figure 3–12. I/O Flag (IOF) Register

31–16	15–12	11–8	7	6	5	4	3	2	1	0
xx	x	xx	INXF1	OUTXF1	$\overline{\text{I}}$/OXF1	xx	INXF0	OUTXF0	$\overline{\text{I}}$/OXF0	xx
			R	R/W	R/W		R	R/W	R/W	

Notes: 1) xx = reserved bit, read as 0

2) R = read, W = write

Table 3–5. IOF Bits and Functions

Bit Name	Reset Value	Function
$\overline{\text{I}}$/OXF0	0	If 0, XF0 is configured a general-purpose input pin. If 1, XF0 is configured a general-purpose output pin.
OUTXF0	0	Data output on XF0.
INXF0	0	Data input on XF0. A write has no effect.
$\overline{\text{I}}$/OXF1	0	If 0, XF1 is configured a general-purpose input pin. If 1, XF1 is configured a general-purpose output pin.
OUTXF1	0	Data output on XF1.
INXF1	0	Data input on XF1. A write has no effect.

3.1.11 Repeat-Counter (RC) and Block-Repeat (RS, RE) Registers

The repeat-counter (RC) register is a 32-bit register that specifies the number of times a block of code is to be repeated when a block repeat is performed. If RC contains the number *n*, the loop is executed *n* + 1 times.

The 32-bit repeat start-address (RS) register contains the starting address of the program-memory block to be repeated when the CPU is operating in the repeat mode.

The 32-bit repeat end-address (RE) register contains the ending address of the program-memory block to be repeated when the CPU is operating in the repeat mode.

Note: RE < RS

If RE< RS and the block mode is enabled, the code between RE and RS is bypassed when the program counter encounters the repeat end (RE) address.

3.2 Other Registers

3.2.1 Program-Counter (PC) Register

The program counter (PC) is a 32-bit register containing the address of the next instruction fetch. While the program-counter register is not part of the CPU register file, it can be modified by instructions that modify the program flow.

3.2.2 Instruction Register (IR)

The instruction register (IR) is a 32-bit register that holds the instruction opcode during the decode phase of the instruction. This register is used by the instruction decode control circuitry and is not accessible to the CPU.

3.3 Reserved Bits and Compatibility

To retain compatibility with future members of the ’C3x family of microprocessors, reserved bits that are read as 0 must be written as 0. You must not modify the current value of a reserved bit that has an undefined value. In other cases, you should maintain the reserved bits as specified.

Chapter 4

Memory and the Instruction Cache

The ’C3x provides a total memory space of 16M (million) 32-bit words that contain program, data, and I/O space. Two RAM blocks of 1K × 32 bits each (available on the ’C30 and ’C31) or two RAM blocks of 256 × 32 bits (available on the ’C32) and a ROM block of 4K × 32 bits (available only on the ’C30) or boot loader (available on the ’C31 and the ’C32) permit two CPU accesses in a single cycle.

A 64 × 32-bit instruction cache stores often-repeated sections of code, greatly reducing the number of off-chip accesses and allowing code to be stored off-chip in slower, lower-cost memories.

Topic **Page**

4.1 Memory

The ’C3x accesses a total memory space of 16M (million) 32-bit words of program, data, and I/O space and allows tables, coefficients, program code, or data to be stored in either RAM or ROM. In this way, you can maximize memory usage and allocate memory space as desired.

RAM blocks 0 and 1 are each 1K × 32 bits on the ’C30 and ’C31. The ROM block is 4K × 32 bits on the ’C30. The ’C31 and ’C32 have a boot ROM. By manipulating one external pin (MC/$\overline{\text{MP}}$ or MCBL/$\overline{\text{MP}}$), you can configure the first 1000h words of memory to address the on-chip ROM or external RAM. Each on-chip RAM and ROM block can support two CPU accesses in a single cycle. The separate program buses, data buses, and DMA buses allow for parallel program fetches, data reads/writes, and DMA operations, which are covered in Chapter 11, *Peripherals*.

4.1.1 Memory Maps

The following sections describe the memory maps for the ’C30, ’C31, and ’C32.

4.1.1.1 TMS320C30 Memory Map

’C30

The memory map depends on whether the processor is running in microprocessor mode (MC/$\overline{\text{MP}}$ = 0) or microcomputer mode (MC/$\overline{\text{MP}}$ = 1). The memory maps for these modes are similar (see Figure 4–1 on page 4-4). Locations 800000h–801FFFh are mapped to the expansion bus. When this region is accessed, $\overline{\text{MSTRB}}$ is active. Locations 802000h–803FFFh are reserved. Locations 804000h–805FFFh are mapped to the expansion bus. When this region is accessed, $\overline{\text{IOSTRB}}$ is active. Locations 806000h–807FFFh are reserved. All of the memory-mapped peripheral bus registers are in locations 808000h–8097FFh. In both modes, RAM block 0 is located at addresses 809800h–809BFFh, and RAM block 1 is located at addresses 809C00h–809FFFh. Locations 80A000h–0FFFFFFh are accessed over the external memory port ($\overline{\text{STRB}}$ active).

❑ **Microprocessor Mode**

In microprocessor mode, the 4K on-chip ROM is not mapped into the ’C3x memory map. Locations 0h–03Fh consist of interrupt vector, trap vector, and reserved locations, all of which are accessed over the external memory port ($\overline{\text{STRB}}$ active) (see Figure 4–1 on page 4-4). Locations 040h–7FFFFFh are also accessed over the external memory port.

- **Microcomputer Mode**

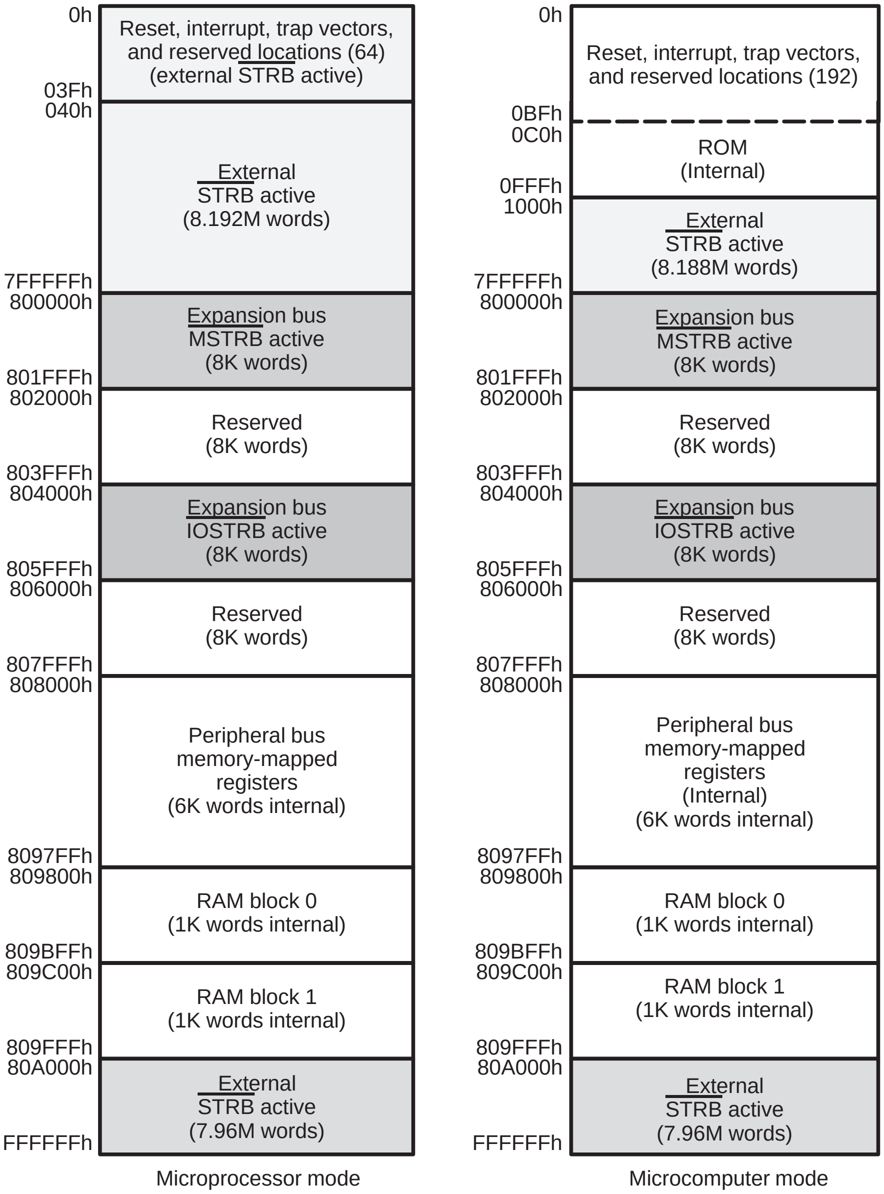
  In microcomputer mode, the 4K on-chip ROM is mapped into locations 0h–0FFFh. There are 192 locations (0h–0BFh) within this block for interrupt vectors, trap vectors, and a reserved space (’C30). Locations 1000h–7FFFFFh are accessed over the external memory port ($\overline{\text{STRB}}$ active).

Section 4.1.2, *Peripheral Bus Memory Map*, on page 4-9 describes the peripheral memory maps in greater detail and Section 4.2, *Reset/Interrupt/Trap Vector Map*, on page 4-14 provides the vector locations for reset, interrupts, and traps.

CAUTION

Be careful! Access to a reserved area produces unpredictable results.

Figure 4–1. TMS320C30 Memory Maps


0h
Reset, interrupt, trap vectors, and reserved locations (64) (external STRB active)
03Fh
040h
External STRB active (8.192M words)
7FFFFFh
800000h
Expansion bus MSTRB active (8K words)
801FFFh
802000h
Reserved (8K words)
803FFFh
804000h
Expansion bus IOSTRB active (8K words)
805FFFh
806000h
Reserved (8K words)
807FFFh
808000h
Peripheral bus memory-mapped registers (6K words internal)
8097FFh
809800h
RAM block 0 (1K words internal)
809BFFh
809C00h
RAM block 1 (1K words internal)
809FFFh
80A000h
External STRB active (7.96M words)
FFFFFFh
Microprocessor mode
0h
Reset, interrupt, trap vectors, and reserved locations (192)
0BFh
0C0h
ROM (Internal)
0FFFh
1000h
External STRB active (8.188M words)
7FFFFFh
800000h
Expansion bus MSTRB active (8K words)
801FFFh
802000h
Reserved (8K words)
803FFFh
804000h
Expansion bus IOSTRB active (8K words)
805FFFh
806000h
Reserved (8K words)
807FFFh
808000h
Peripheral bus memory-mapped registers (Internal) (6K words internal)
8097FFh
809800h
RAM block 0 (1K words internal)
809BFFh
809C00h
RAM block 1 (1K words internal)
809FFFh
80A000h
External STRB active (7.96M words)
FFFFFFh
Microcomputer mode

4.1.1.2 TMS320C31 Memory Map

’C31

The memory map depends on whether the processor is running in microprocessor mode (MCBL/$\overline{\text{MP}}$ = 0) or microcomputer mode (MCBL/$\overline{\text{MP}}$ = 1). The memory maps for these modes are similar (see Figure 4–2 on page 4-6). Locations 800000h–807FFFh are reserved. All of the memory-mapped peripheral bus registers are in locations 808000h–8097FFh. In both modes, RAM block 0 is located at addresses 809800h–809BFFh, and RAM block 1 is located at addresses 809C00h–809FFFh. Locations 80A000h–0FFFFFFh are accessed over the external memory port ($\overline{\text{STRB}}$ active).

❑ **Microprocessor Mode**

In microprocessor mode, the boot loader is not mapped into the ’C3x memory map. Locations 0h–03Fh consist of interrupt vector, trap vector, and reserved locations, all of which are accessed over the external memory port ($\overline{\text{STRB}}$ active) (see Figure 4–2 on page 4-6). Locations 040h–7FFFFFh are also accessed over the external memory port.

❑ **Microcomputer Mode**

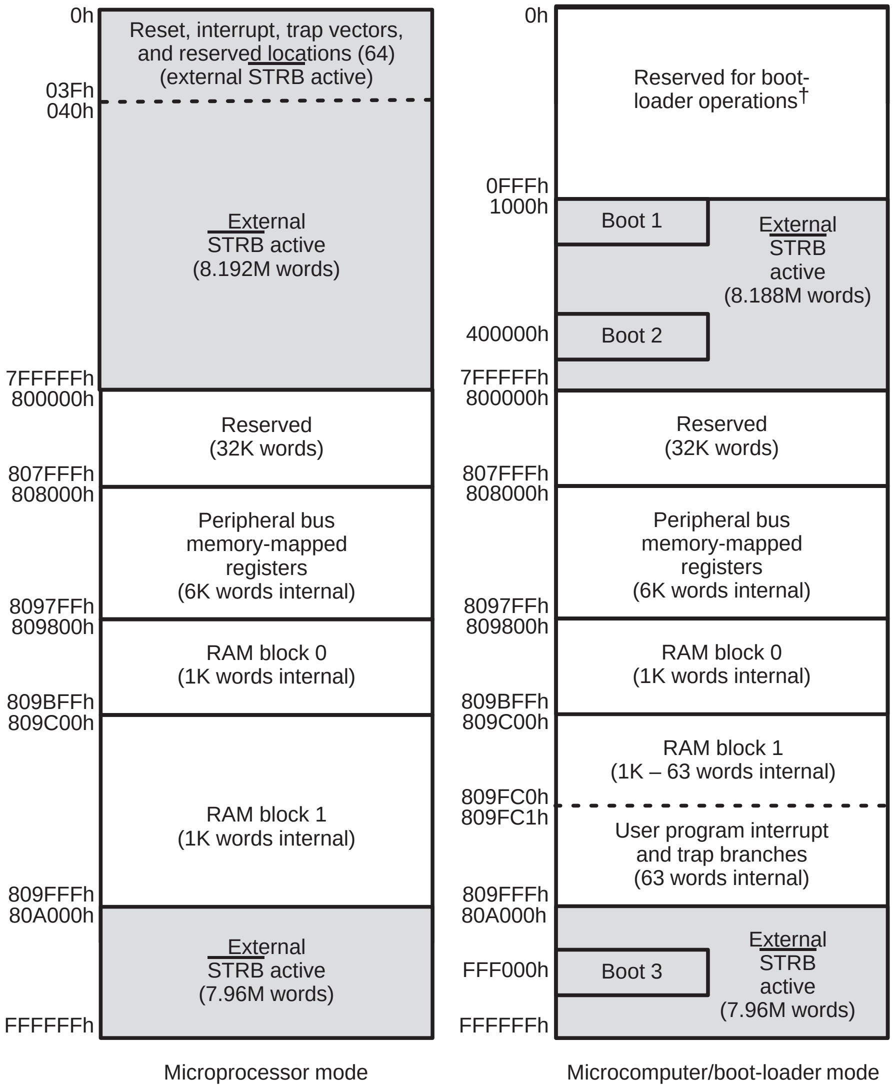
In microcomputer mode, the boot loader ROM is mapped into locations 0h–0FFFh. The last 63 words (809FC1h to 809FFFh) of internal RAM Block 1 are used for interrupt and trap *branches* (see Figure 4–2 on page 4-6). Locations 1000h–7FFFFFh are accessed over the external memory port ($\overline{\text{STRB}}$ active).

Section 4.1.2, *Peripheral Bus Memory Map*, on page 4-9 describes the peripheral memory maps in greater detail and Section 4.2, *Reset/Interrupt/Trap Vector Map*, on page 4-14 provides the vector locations for reset, interrupts, and traps.

Be careful! Access to a reserved area produces unpredictable results.

CAUTION

Figure 4–2. TMS320C31 Memory Maps


0h
Reset, interrupt, trap vectors, and reserved locations (64) (external STRB active)
03Fh
040h
External STRB active (8.192M words)
7FFFFFh
800000h
Reserved (32K words)
807FFFh
808000h
Peripheral bus memory-mapped registers (6K words internal)
8097FFh
809800h
RAM block 0 (1K words internal)
809BFFh
809C00h
RAM block 1 (1K words internal)
809FFFh
80A000h
External STRB active (7.96M words)
FFFFFFh
Microprocessor mode
0h
Reserved for boot-loader operations†
0FFFh
1000h
Boot 1
External STRB active (8.188M words)
400000h
Boot 2
7FFFFFh
800000h
Reserved (32K words)
807FFFh
808000h
Peripheral bus memory-mapped registers (6K words internal)
8097FFh
809800h
RAM block 0 (1K words internal)
809BFFh
809C00h
RAM block 1 (1K – 63 words internal)
809FC0h
809FC1h
User program interrupt and trap branches (63 words internal)
809FFFh
80A000h
External STRB active (7.96M words)
FFF000h
Boot 3
FFFFFFh
Microcomputer/boot-loader mode

† See Section 3.1.3, *Data-Page Pointer (DP)*, on page 3-4 for more information.

4.1.1.3 TMS320C32 Memory Map

’C32

The memory map depends on whether the processor is running in microprocessor mode (MCBL/$\overline{\text{MP}}$ = 0) or microcomputer mode (MCBL/$\overline{\text{MP}}$ = 1). The memory maps for these modes are similar (see Figure 4–3 on page 4-8). Locations 800000h–807FFFh, 809800h–80FFFh, and 830000H–87FDFFh are reserved. Locations 810000h–82FFFFh are mapped to the external bus with $\overline{\text{IOSTRB}}$ active. All of the memory-mapped peripheral bus registers are in locations 808000h–8097FFh. In both modes, RAM block 0 is located at addresses 87FE00h–87FEFFh, and RAM block 1 is located at addresses 87FF00h–87FFFFh. Locations 900000h–FFFFFFh are mapped to the external bus with $\overline{\text{STRB1}}$ active.

Unlike the fixed interrupt-trap vector table location of the ’C30 and ’C31 devices, the ’C32 has a user-relocatable interrupt-trap vector table. The interrupt-trap vector table must start on a 256-word boundary. The starting location is programmed through the interrupt-trap table pointer (ITTP) bit field in the CPU interrupt flag (IF) register. See Section 3.1.9.1, *Interrupt-Trap Table Pointer (ITTP)*, on page 3-14.

❑ **Microprocessor Mode**

  In microprocessor mode, the boot loader is not mapped into the ’C3x memory map. Locations 0h–7FFFFFFh are accessed over the external memory port ($\overline{\text{STRB0}}$ active) with location 0h containing the reset vector.

❑ **Microcomputer Mode**

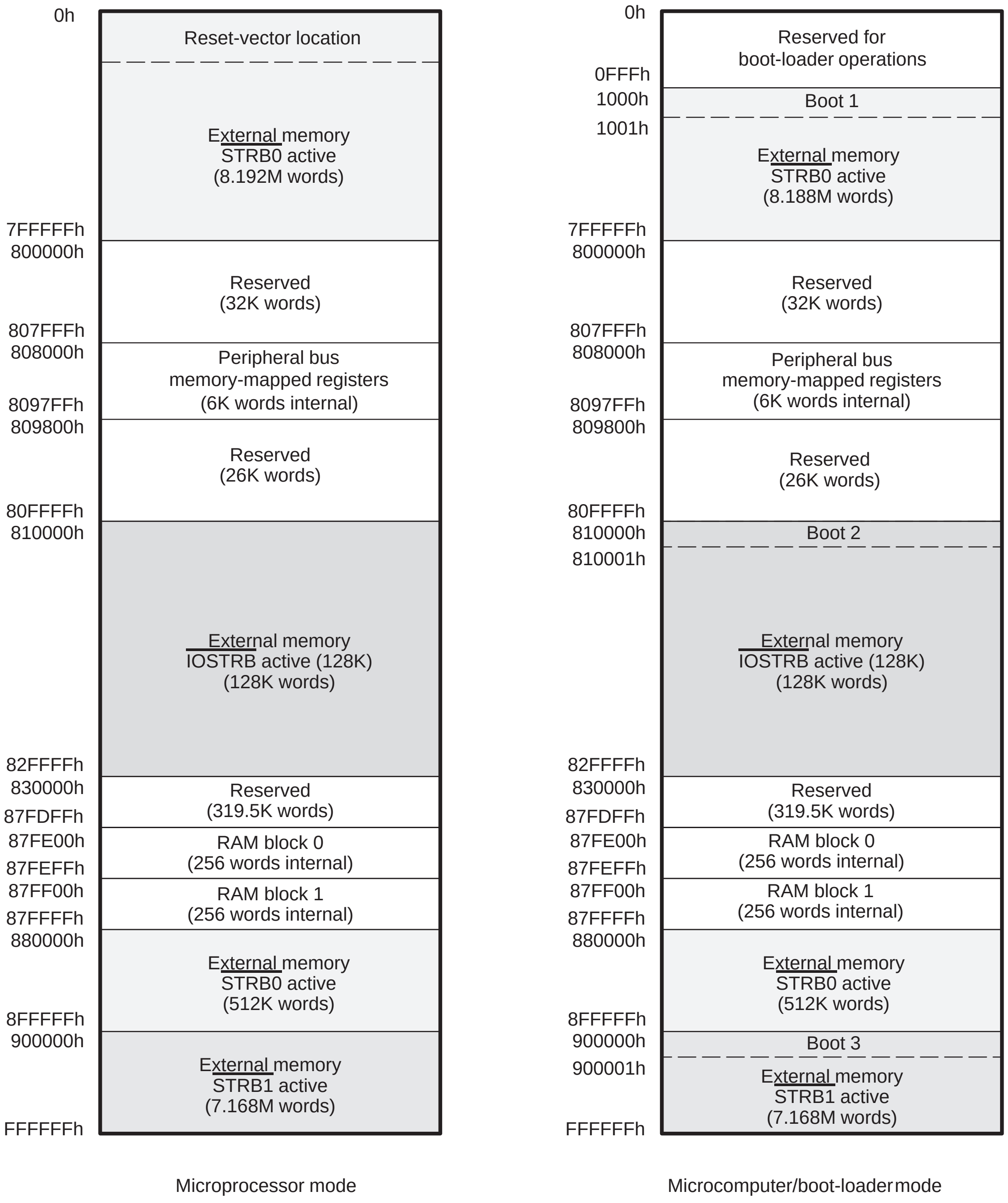
  In microcomputer mode, the on-chip boot loader ROM is mapped into locations 0h–0FFFh. Locations 1000h–7FFFFFh are accessed over the external memory port ($\overline{\text{STRB0}}$ active).

The ’C32 boot loader has additional modes over the ’C31 boot loader to handle the data types, sizes, and memory widths supported by the external memory interface. The memory boot load supports data transfer with and without handshaking. The handshake mode allows synchronous program transfer by using two pins as data-acknowledge and data-ready signals.

See Section 4.1.2, *Peripheral Bus Memory Map*, on page 4-9 and Section 4.2, *Reset/Interrupt/Trap Vector Map*, on page 4-14 for more information.

> CAUTION
>
> **Be careful! Access to a reserved area produces unpredictable results.**

Figure 4–3. TMS320C32 Memory Maps



Microprocessor mode

Microcomputer/boot-loadermode

4.1.2 Peripheral Bus Memory Map

The following sections describe the peripherial bus memory maps for the ’C30, ’C31, and ’C32.

4.1.2.1 TMS320C30 Peripheral Bus Memory Map

The ’C30 memory-mapped peripheral registers are located starting at address 808000h. Figure 4–4 on page 4-10 shows the peripheral bus memory map. The shaded blocks are reserved.

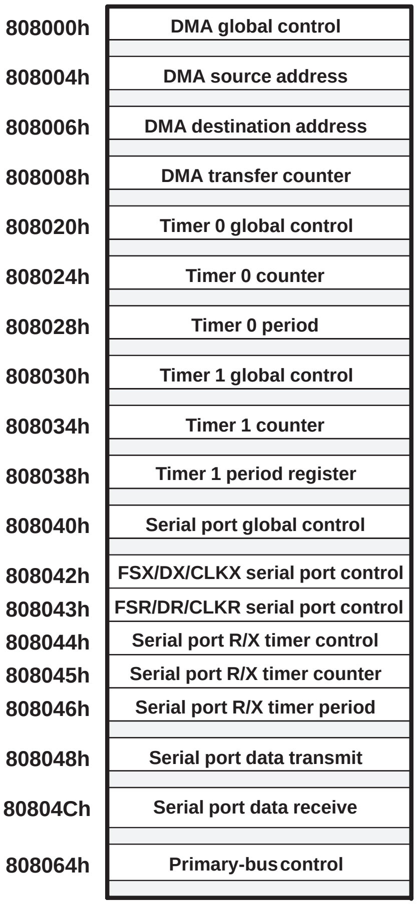
Figure 4–4. TMS320C30 Peripheral Bus Memory-Mapped Registers

808000h	DMA global control
808004h	DMA source address
808006h	DMA destination address
808008h	DMA transfer counter
808020h	Timer 0 global control
808024h	Timer 0 counter
808028h	Timer 0 period
808030h	Timer 1 global control
808034h	Timer 1 counter
808038h	Timer 1 period register
808040h	Serial port 0 global control
808042h	FSX/DX/CLKX serial port 0 control
808043h	FSR/DR/CLKR serial port 0 control
808044h	Serial port 0 R/X timer control
808045h	Serial port 0 R/X timer counter
808046h	Serial port 0 R/X timer period
808048h	Serial port 0 data transmit
80804Ch	Serial port 0 data receive
808050h	Serial port 1 global control
808052h	FSX/DX/CLKX serial port 1 control
808053h	FSR/DR/CLKR serial port 1 control
808054h	Serial port 1 R/X timer control
808055h	Serial port 1 R/X timer counter
808056h	Serial port 1 R/X timer period
808058h	Serial port 1 data transmit
80805Ch	Serial port 1 data receive
808060h	Expansion-bus control
808064h	Primary-bus control

4.1.2.2 TMS320C31 Peripheral Bus Memory Map

The ’C31 memory-mapped peripheral registers are located starting at address 808000h. Figure 4–5 shows the peripheral bus memory map. The shaded blocks are reserved.

Figure 4–5. TMS320C31 Peripheral Bus Memory-Mapped Registers



808000h	DMA global control
808004h	DMA source address
808006h	DMA destination address
808008h	DMA transfer counter
808020h	Timer 0 global control
808024h	Timer 0 counter
808028h	Timer 0 period
808030h	Timer 1 global control
808034h	Timer 1 counter
808038h	Timer 1 period register
808040h	Serial port global control
808042h	FSX/DX/CLKX serial port control
808043h	FSR/DR/CLKR serial port control
808044h	Serial port R/X timer control
808045h	Serial port R/X timer counter
808046h	Serial port R/X timer period
808048h	Serial port data transmit
80804Ch	Serial port data receive
808064h	Primary-bus control

4.1.2.3 TMS320C32 Peripheral Bus Memory Map

’C32

The ’C32’s memory-mapped peripheral and external-bus control registers are located starting at address 808000h, as shown in Figure 4–6 on page 4-13. The shaded blocks are reserved.

Figure 4–6. TMS320C32 Peripheral Bus Memory-Mapped Registers

808000h	DMA 0 global control
808004h	DMA 0 source address
808006h	DMA 0 destination address
808008h	DMA 0 transfer counter
808010h	DMA 1 global control
808014h	DMA 1 source address
808016h	DMA 1 destination address
808018h	DMA 1 transfer counter
808020h	Timer 0 global control
808024h	Timer 0 counter
808028h	Timer 0 period
808030h	Timer 1 global control
808034h	Timer 1 counter
808038h	Timer 1 period register
808040h	Serial port global control
808042h	FSX/DX/CLKX serial port control
808043h	FSR/DR/CLKR serial port control
808044h	Serial port R/X timer control
808045h	Serial port R/X timer counter
808046h	Serial port R/X timer period
808048h	Serial port data transmit
80804Ch	Serial port data receive
808060h	$\overline{\text{IOSTRB}}$ bus control
808064h	$\overline{\text{STRB0}}$ bus control
808068h	$\overline{\text{STRB1}}$ bus control
8097FFh	

4.2 Reset/Interrupt/Trap Vector Map

The addresses for the reset, interrupt, and trap vectors are 00h–3Fh, as shown in Figure 4–7 and Figure 4–8. The reset vector contains the address of the reset routine.

’C30 ’C31

- **’C30 and ’C31 Microprocessor and Microcomputer Modes**

  In the microprocessor mode of the ’C30 and ’C31 and the microcomputer mode of the ’C30, the reset interrupt and trap vectors stored in locations 0h–3Fh are the addresses of the starts of the respective reset, interrupt, and trap routines. For example, at reset, the content of memory location 00h (reset vector) is loaded into the PC, and execution begins from that address (see Figure 4–8 on page 4-16).

’C31

- **’C31 Microcomputer/Boot-Loader Mode**

  In the microcomputer/boot-loader mode of the ’C31, the interrupt and trap vectors stored in locations 809FC1h–809FFFh are *branch* instructions to the start of the respective interrupt and trap routines (see Figure 4–9 on page 4-17).

’C32

- **’C32 Microprocessor and Microcomputer/Boot-Loader Mode**

  The ’C32 has a user-relocatable interrupt-trap vector table. The interrupt-trap vector table must start on a 256-word boundary. The starting location is programmed through the interrupt-trap table pointer (ITTP) bit field in the CPU interrupt flag (IF) register. See Section 3.1.9.1, *Interrupt-Trap Table Pointer (ITTP)*, on page 3-14. The reset vector is stored at location 0h in microprocessor mode.

Figure 4–7. Reset, Interrupt, and Trap Vector Locations for the TMS320C30 Microprocessor Mode

00h	RESET
01h	$\overline{\text{INT0}}$
02h	$\overline{\text{INT1}}$
03h	$\overline{\text{INT2}}$
04h	$\overline{\text{INT3}}$
05h	XINT0
06h	RINT0
07h	XINT1
08h	RINT1
09h	TINT0
0Ah	TINT1
0Bh	DINT
0Ch 1Fh	Reserved
20h	TRAP 0
	• • •
3Bh	TRAP 27
3Ch	TRAP 28 (reserved)
3Dh	TRAP 29 (reserved)
3Eh	TRAP 30 (reserved)
3Fh	TRAP 31 (reserved)

Note: Traps 28–31

Traps 28–31 are reserved; **do not use them**.

’C31

Figure 4–8. Reset, Interrupt, and Trap Vector Locations for theTMS320C31 Microprocessor Mode

00h	RESET
01h	$\overline{\text{INT0}}$
02h	$\overline{\text{INT1}}$
03h	$\overline{\text{INT2}}$
04h	$\overline{\text{INT3}}$
05h	XINT0
06h	RINT0
07h	XINT1 (Reserved)
08h	RINT1 (Reserved)
09h	TINT0
0Ah	TINT1
0Bh	DINT
0Ch 1Fh	Reserved
20h	TRAP 0
	• • •
3Bh	TRAP 27
3Ch	TRAP 28 (reserved)
3Dh	TRAP 29 (reserved)
3Eh	TRAP 30 (reserved)
3Fh	TRAP 31 (reserved)

Note: Traps 28–31

Traps 28–31 are reserved; **do not use them**.

Figure 4–9. Interrupt and Trap Branch Instructions for the TMS320C31 Microcomputer Mode

’C31

809FC1h	$\overline{\text{INT0}}$
809FC2h	$\overline{\text{INT1}}$
809FC3h	$\overline{\text{INT2}}$
809FC4h	$\overline{\text{INT3}}$
809FC5h	XINT0
809FC6h	RINT0
809FC7h	XINT1 (reserved)
809FC8h	RINT1 (reserved)
809FC9h	TINT0
809FCAh	TINT1
809FCBh	DINT
809FCCh 809FDFh	Reserved
809FE0h	TRAP 0
809FE1h	TRAP 1
	• • •
809FFBh	TRAP 27
809FFCh	TRAP 28 (reserved)
809FFDh	TRAP 29 (reserved)
809FFEh	TRAP 30 (reserved)
809FFFh	TRAP 31 (reserved)

Note: Traps 28–31

Traps 28–31 are reserved; **do not use them**.

Unlike the ’C31’s microprocessor mode, the ’C31 microcomputer/boot loader mode uses a dual-vectoring scheme to service interrupts and trap requests. In this dual vectoring scheme, a **branch instruction** rather than a vector address is used.

Figure 4–10. Interrupt and Trap Vector Locations for TMS320C32

’C32

EA (ITTP) + 00h	Reserved
EA (ITTP) + 01h	$\overline{\text{INT0}}$
EA (ITTP) + 02h	$\overline{\text{INT1}}$
EA (ITTP) + 03h	$\overline{\text{INT2}}$
EA (ITTP) + 04h	$\overline{\text{INT3}}$
EA (ITTP) + 05h	XINT0
EA (ITTP) + 06h	RINT0
EA (ITTP) + 07h	Reserved
EA (ITTP) + 08h	Reserved
EA (ITTP) + 09h	TINT0
EA (ITTP) + 0Ah	TINT1
EA (ITTP) + 0Bh	DINT0
EA (ITTP) + 0Ch	DINT1
EA (ITTP) + 0Dh EA (ITTP) + 1Fh	Reserved
EA (ITTP) + 20h	TRAP0
	⋮
EA (ITTP) + 3Bh	TRAP27
EA (ITTP) + 3Ch	TRAP28 (reserved)
EA (ITTP) + 3Dh	TRAP29 (reserved)
EA (ITTP) + 3Eh	TRAP30 (reserved)
EA (ITTP) + 3Fh	TRAP31 (reserved)

Note: Traps 28–31

Traps 28–31 are reserved; **do not use them**.

4.3 Instruction Cache

A 64 × 32-bit instruction cache speeds instruction fetches and lowers system cost by caching program fetches from external memory. The instruction cache allows the use of slow, external memories while still achieving single-cycle access performances. This reduces the number of off-chip accesses necessary and allows code to be stored off-chip in slower, lower-cost memories. The cache also frees external buses from program fetches so that they can be used by the DMA or other system elements.

The cache can operate automatically, with no user intervention. Subsection 4.3.2 describes a form of the least recently used (LRU) cache update algorithm.

4.3.1 Instruction-Cache Architecture

The instruction cache (see Figure 4–12) contains 64 32-bit words of RAM; it is divided into two 32-word segments. A 19-bit segment start address (SSA) register is associated with each segment. For each word in the cache, there is a corresponding single bit-present (P) flag.

When the CPU requests an instruction word from external memory, the cache algorithm checks to determine if the word is already contained in the instruction cache. Figure 4–11 shows how the cache-control algorithm partitions an instruction address. The algorithm uses the19 most significant bits (MSBs) of the instruction address to select the segment; the five least significant bits (LSBs) define the address of the instruction word within the pertinent segment. The algorithm compares the 19 MSBs of the instruction address with the two SSA registers. If there is a match, the algorithm checks the relevant P flag. The P flag indicates if a word within a particular segment is already present in cache memory:

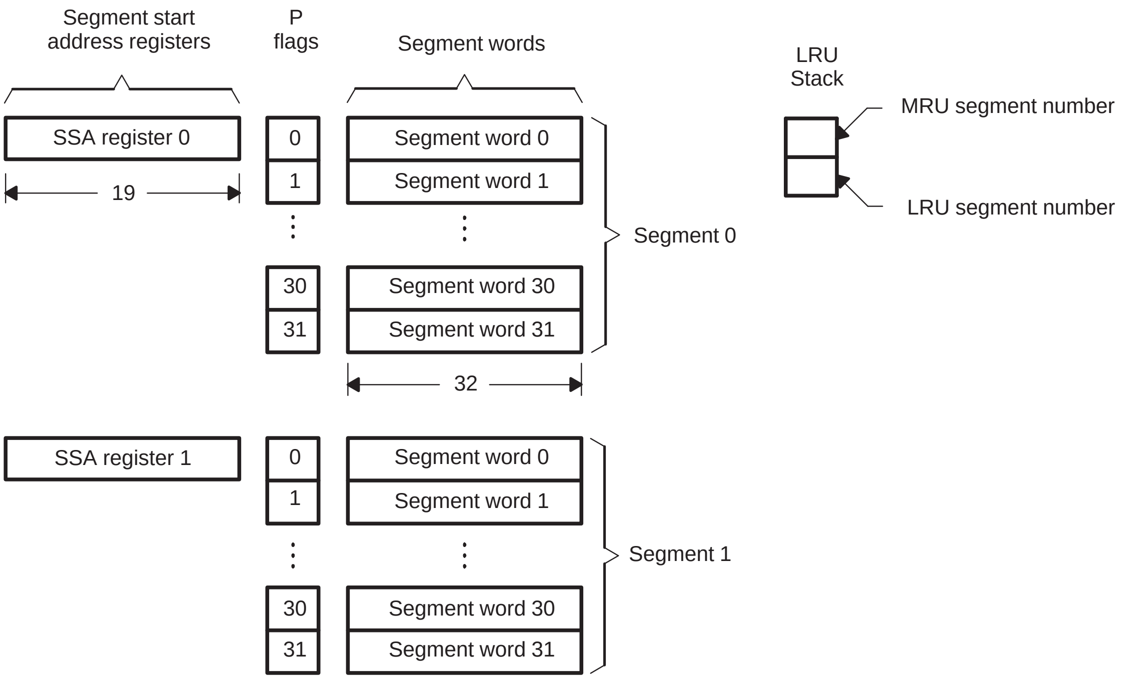
- ❑ P = 1: the word is already present in cache memory
- ❑ P = 0: the location cache is invalid

Figure 4–11. Address Partitioning for Cache Control Algorithm

23 — 5	4 — 0
Segment start address (SSA)	Instruction word address within segment

If there is no match, one of the segments must be replaced by the new data. The segment replaced in this circumstance is determined by the LRU algorithm. The LRU stack (see Figure 4–12) is maintained for this purpose.

Figure 4–12. Instruction-Cache Architecture



The LRU stack determines which of the two segments qualifies as the least recently used after each access to the cache. Each time a segment is accessed, its segment number is removed from the LRU stack and pushed onto the top of the LRU stack. Therefore, the number at the top of the stack is the most recently used (MRU) segment number, and the number at the bottom of the stack is the least recently used segment number.

At reset, the LRU stack is initialized with 0 at the top and 1 at the bottom. All P flags in the instruction cache are cleared.

When a replacement is necessary, the LRU segment is selected for replacement. Also, the 32 P flags for the segment to be replaced are set to 0, and the segment’s SSA register is replaced with the 19 MSBs of the instruction address.

4.3.2 Instruction-Cache Algorithm

When the ’C3x requests an instruction word from external memory, one of two possible actions occurs: a *cache hit* or a *cache miss*.

- **Cache Hit.** The cache contains the requested instruction, and the following actions occur:
  - The instruction word is read from the cache.
  - The number of the segment containing the word is removed from the LRU stack and pushed to the top of the LRU stack (if it is not already at the top), thus moving the other segment number to the bottom of the stack.
- **Cache Miss.** The cache does not contain the instruction. There are two types of cache misses:
  - **Subsegment miss**. The segment address register matches the instruction address, but the relevant P flag is not set. The following actions occur in parallel:
    - The instruction word is read from memory and copied into the cache.
    - The number of the segment containing the word is removed from the LRU stack and pushed to the top of the LRU stack (if it is not already at the top), thus moving the other segment number to the bottom of the stack.
    - The relevant P flag is set.
  - **Segment miss.** Neither of the segment addresses matches the instruction address. The following actions occur in parallel:
    - The LRU segment is selected for replacement. The P flags for all 32 words are cleared.
    - The SSA register for the selected segment is loaded with the 19 MSBs of the address of the requested instruction word.
    - The instruction word is fetched and copied into the cache. It goes into the appropriate word of the LRU segment. The P flag for that word is set to 1.
    - The number of the segment containing the instruction word is removed from the LRU stack and pushed to the top of the LRU stack, thus moving the other segment number to the bottom of the stack.

Only instructions may be fetched from the program cache. All reads and writes of data in memory bypass the cache. Program fetches from internal memory do not modify the cache and do not generate cache hits or misses. The program cache is a single-access memory block. Dummy program fetches (for example, those following a branch) are treated by the cache as valid program fetches and can generate cache misses and cache updates.

Notes: Using Self-Modifying Code

Be careful when using self-modifying code. If an instruction resides in the cache and the corresponding location in primary memory is modified, the copy of the instruction in the cache is not modified.

You can use the cache more efficiently by aligning program code on 32-word address boundaries. Do this with the .*align* directive when coding assembly language.

4.3.3 Cache Control Bits

Three cache control bits are located in the CPU status register:

❑ **Cache Clear Bit (CC)**. Set CC = 1 to invalidate all entries in the cache. This bit is always cleared after it is written to; it is always read as a 0. At reset, the cache is cleared, and 0 is written to this bit.

❑ **Cache Enable Bit (CE)**. Set CE = 1 to enable the cache, allowing the cache to be used according to the LRU cache algorithm. Set CE = 0 to disable the cache; this prevents cache update modifications (thus, no cache fetches can be made). At reset, 0 is written to this bit. Cache clearing (CC = 1) is allowed when CE = 0.

❑ **Cache Freeze Bit (CF)**. Set CF = 1 to freeze both the cache and LRU stack manipulation. If the cache is enabled (CE = 1) and the cache is frozen (CF = 1), fetches from the cache are allowed, but modification of cache contents is not allowed. Cache clearing (CC = 1) is allowed when CF = 1 or CF = 0. At reset, this CF bit is cleared to 0.

Table 4–1 shows the combined effect of the CE and CF.

Table 4–1. Combined Effect of the CE and CF Bits

CE	CF	Effect
0	0	Cache not enabled
0	1	Cache not enabled
1	0	Cache enabled and not frozen
1	1	Cache enabled and frozen

When the CE or CF bits of the CPU status register are modified, the following four instructions may or may not be fetched from the cache or external memory (see Example 4–1).

When the CC bit of the CPU status register is modified, the following five instructions may or may not be fetched from the cache before the cache is cleared (see Example 4–1).

Example 4–1. Pipeline Effects of Modifying the Cache Control Bits

Pipeline Operation

Cycle	**Fetch**	**Decode**	**Read**	**Execute**
n	`LDI 1000h, ST`			
n+1	`LDI 1h, R1`	`LDI 1000h, ST`		
n+2	`LDI 2h, R2`	`LDI 1h, R1`	`LDI 1000h, ST`	
n+3	`LDI 3h, R3`	`LDI 2h, R2`	`LDI 1h, R1`	`LDI 1000h, ST`
n+4	`LDI 4h, R4`	`LDI 3h, R3`	`LDI 2h, R2`	`LDI 1h, R1`
n+5	`LDI 5h, R5`	`LDI 4h, R4`	`LDI 3h, R3`	`LDI 2h, R2`
n+6		`LDI 5h, R5`	`LDI 4h, R4`	`LDI 3h, R3`
n+7			`LDI 5h, R5`	`LDI 4h, R4`
n+8				`LDI 5h, R5`

Instructions may be fetched before cache is enabled or frozen.

Cache cleared

Instructions may be fetched before cache cleared.

Chapter 5

Data Formats and Floating-Point Operation

In the ’C3x architecture, data is organized into three fundamental types: integer, unsigned integer, and floating-point. The terms integer and signed integer are equivalent. The ’C3x supports short and single-precision formats for signed and unsigned integers. It also supports short, single-precision, and extended-precision formats for floating-point data.

Floating-point operations make fast, trouble-free, accurate, and precise computations. Specifically, the ’C3x implementation of floating-point arithmetic facilitates floating-point operations at integer speeds, while preventing problems with overflow, operand alignment, and other burdensome tasks that are common in integer operations.

This chapter discusses data formats and floating-point operations supported in the ’C3x.

Topic **Page**

5.1 Integer Formats

The ’C3x supports two integer formats: a 16-bit short-integer format and a 32-bit single-precision integer format.

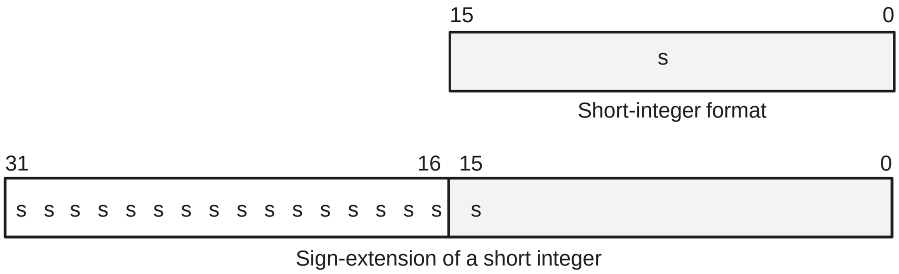
Note:

When extended-precision registers are used as integer operands, only bits 31–0 are used; bits 39–32 remain unchanged.

5.1.1 Short-Integer Format

The short-integer format is a 16-bit 2s-complement integer format for immediate-integer operands. For those instructions that assume integer operands, this format is sign-extended to 32 bits (see Figure 5–1). The range of an integer *si,* represented in the short-integer format, is $-2^{15} \le si \le 2^{15} - 1$. In Figure 5–1, s = signed bit.

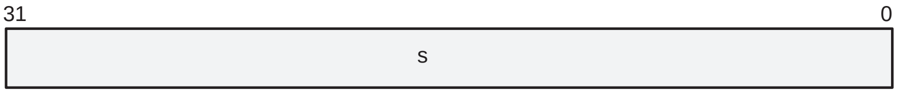
Figure 5–1. Short-Integer Format and Sign-Extension of Short Integers


15
0
s
Short-integer format
31
16
15
0
s s s s s s s s s s s s s s s s
s
Sign-extension of a short integer

5.1.2 Single-Precision Integer Format

In the single-precision integer format, the integer is represented in 2s-complement notation. The range of an integer *sp*, represented in the single-precision integer format, is $-2^{31} \le sp \le 2^{31} - 1$. Figure 5–2 shows the single-precision integer format.

Figure 5–2. Single-Precision Integer Format



5.2 Unsigned-Integer Formats

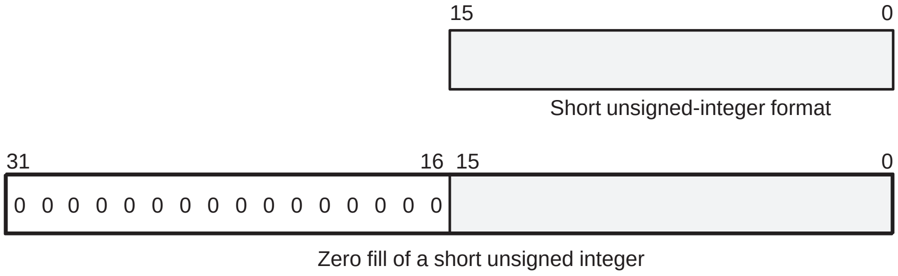
The ’C3x supports two unsigned-integer formats: a 16-bit short format and a 32-bit single-precision format.

> **Note:**
>
> In extended-precision registers, the unsigned-integer operands use only bits 31–0; bits 39–32 remain unchanged.

5.2.1 Short Unsigned-Integer Format

Figure 5–3 shows the16-bit, short, unsigned-integer format for immediate unsigned-integer operands. For those instructions which assume unsigned-integer operands, this format is zero filled to 32 bits. The range of a short unsigned integer is 0 ≤ *si* ≤ 2^{16}.

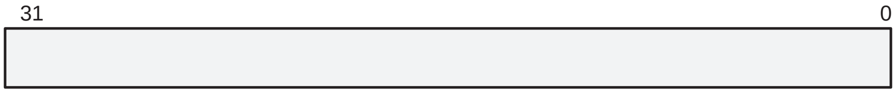
Figure 5–3. Short Unsigned-Integer Format and Zero Fill


15
0
Short unsigned-integer format
31
16
15
0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Zero fill of a short unsigned integer

5.2.2 Single-Precision Unsigned-Integer Format

In the single-precision unsigned-integer format, the number is represented as a 32-bit value, as shown in Figure 5–4. The range of a single-precision unsigned integer is 0 ≤ *sp* ≤ 2^{32}.

Figure 5–4. Single-Precision Unsigned-Integer Format



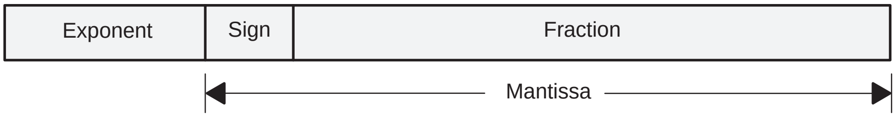
5.3 Floating-Point Formats

The ’C3x supports four floating-point formats:

- A short floating-point format for immediate floating-point operands, consisting of a 4-bit exponent, a sign bit, and an 11-bit fraction
- (’C32 only) A short floating-point format for use with 16-bit floating-point data types, consisting of a 2s-complement, 8-bit exponent field, a sign bit, and a 7-bit fraction
- A single-precision floating-point format by an 8-bit exponent field, a sign bit, and a 23-bit fraction
- An extended-precision floating-point format consisting of an 8-bit exponent field, a sign bit, and a 31-bit fraction.

All ’C3x floating-point formats consist of three fields: an *exponent* field *(e),* a *single-bit sign* field *(s), and a fraction* field *(f)*. The sign field and fraction field may be considered as one unit and referred to as the *mantissa* field (*man*).

Figure 5–5. General Floating-Point Format



The general equation for calculating the value in a floating-point number is:

$$x = s\bar{s}.f_2 \times 2^e$$

In the equation, *s* is the value of the sign bit, $\bar{s}$ is the inverse of the value of the sign bit, *f* is the binary value of the fraction field, and *e* is the decimal equivalent of the exponent field.

The mantissa represents a normalized 2s-complement number. In a normalized representation, a most significant nonsign bit is implied, thus providing an additional bit of precision. The implied sign bit is used as follows:

- If *s* = 0, then the leading two bits of the mantissa are 01.
- If *s* = 1, then the leading two bits of the mantissa are 10.

If the sign bit, *s*, is equal to 0, the mantissa becomes $01.f_2$, where *f* is the binary representation of the fraction field. If *s* is 1, the mantissa becomes $10.f_2$, where *f* is the binary representation of the fraction field.

For example, if f = 00000000001_2 and *s* = 0, the value of the mantissa (man) is 01.00000000001_2. If *s* = 1 for the same value of *f*, the value of *man* is 10.00000000001_2.

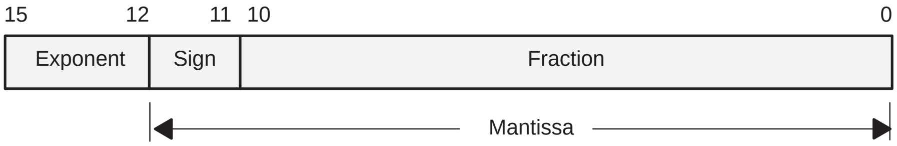
The exponent field is a 2s-complement number that determines the factor of 2 by which the number is multiplied. Essentially, the exponent field shifts the binary point in the mantissa. If the exponent is positive, then the binary point is shifted to the right. If the exponent is negative, then the binary point is shifted to the left.

For example, if *man* = 01.00000000001_2 and the *e* = 11_{10}, then the binary point is shifted 11 places to the right, producing the number: 0100000000001_2, which is equal to 2049 decimal.

5.3.1 Short Floating-Point Format

In the short floating-point format, floating-point numbers are represented by a 2s-complement, 4-bit exponent field (e) and a 2s-complement, 12-bit mantissa field (*man*) with an implied most significant nonsign bit (see Figure 5–6).

Figure 5–6. Short Floating-Point Format



Operations are performed with an implied binary point between bits 11 and 10. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point 2s-complement number *x* in the short floating-point format is given by the following:

$x = 01.\text{f} \times 2^{\text{e}}$ if $s = 0$
$x = 10.\text{f} \times 2^{\text{e}}$ if $s = 1$
$x = 0$ if $e = -8$

You must use the following reserved values to represent 0 in the short floating-point format:

$e = -8$
$s = 0$
$f = 0$

The following examples illustrate the range and precision of the short floating-point format:

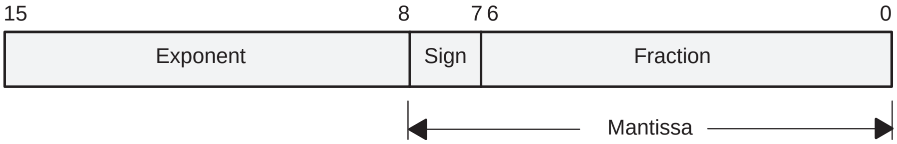
Most positive:	$x = (2 - 2^{-11}) \times 2^{7} = 2.5594 \times 10^{2}$
Least positive:	$x = 1 \times 2^{-7} = 7.8125 \times 10^{-3}$
Least negative:	$x = (-1 - 2^{-11}) \times 2^{-7} = -7.8163 \times 10^{-3}$
Most negative:	$x = -2 \times 2^{7} = -2.5600 \times 10^{2}$

5.3.2 TMS320C32 Short Floating-Point Format for External 16-Bit Data

To facilitate the handling of 16-bit floating-point data types, the ‘C32 uses a new short floating-point format for external 16-bit data types. Note that the following short floating-point format is used only in external 16-bit floating-point data access. This format is different than the 16-bit immediate short floating-point data format used in the ‘C32’s instruction set.

In the short floating-point format for external 16-bit data-type size, floating-point numbers are represented by a 2s-complement, 8-bit exponent field (*e*), a sign bit (*s*), and an 8-bit mantissa field (*man*) with an implied most significant nonsign bit.

Figure 5–7. TMS320C32 Short Floating-Point Format for External 16-Bit Data



Operations are performed with an implied binary point between bits 7 and 6. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point 2s-complement number *x* in the short floating-point format is given by:

$x = 01.\mathrm{f} \times 2^{e}$ if $s = 0$
$x = 10.\mathrm{f} \times 2^{e}$ if $s = 1$
$x = 0$ if $e = -128$

You must use the following reserved values to represent 0 in the ‘C32 short floating-point format for external 16-bit data:

$e = -128$
$s = 0$
$f = 0$

The following examples illustrate the range and precision of the ‘C32 short floating-point format for external 16-bit data:

Most positive:	$x = (2–2^{–8}) \times 2^{127} = 3.3961775 \times 10^{38}$
Least positive	$x = 1 \times 2^{–127} = 5.8774717541 \times 10^{–39}$
Least negative:	$x = (–1–2^{–8}) \times 2^{–127} = –5.9004306 \times 10^{–39}$
Most negative:	$x = (–2 \times 2^{127}) = –3.4028236 \times 10^{38}$

Note that the floating-point instructions (such as LDF, MPYF, ADDF) and the integer instructions (such as LDI, MPYI, ADDI) produce different results when accessing the same memory location. The *integer* load instructions store the value in the LSBs of the ‘C32’s registers. A bit field in the strobe control register controls sign extension or zero fill of the MSBs of the integer value. On the other hand, the *floating-point* load instructions store the value in the MSBs of the ‘C32’s registers. For example:

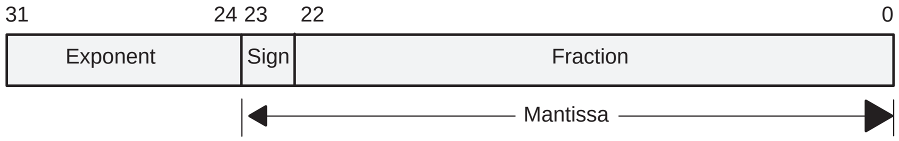
If AR1 = 4000h, R1 = 00 0000 0000h, the value stored at memory location 4000h is 0180h, and $\overline{\text{STRB0}}$ is configured for a physical memory size and data type size of 16 bits.

The result of:	ADDI *AR1,R1	is R1 =	00 0000 0180h, while
The result of:	ADDF *AR1,R1	is R1 =	01 C000 0000h (= – 3.0), since – 4.0 + 1.0 = – 3.0

5.3.3 Single-Precision Floating-Point Format

In the single-precision format, the floating-point number is represented by an 8-bit exponent field (*e*) and a 2s-complement 24-bit mantissa field (*man*) with an implied most significant nonsign bit (see Figure 5–8).

Figure 5–8. Single-Precision Floating-Point Format



Operations are performed with an implied binary point between bits 23 and 22. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point number *x* is given by the following:

$$x = 01.f \times 2^{e} \quad \text{if } s = 0$$
$$x = 10.f \times 2^{e} \quad \text{if } s = 1$$
$$x = 0 \quad \text{if } e = –128$$

You must use the following reserved values to represent 0 in the single-precision floating-point format:

e = – 128
s = 0
f = 0

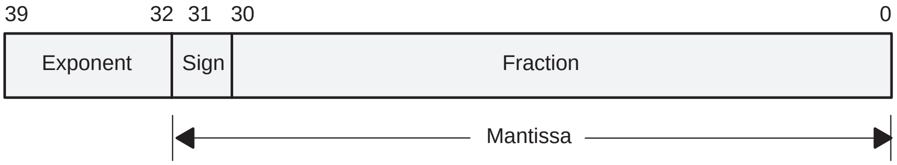
The following examples illustrate the range and precision of the single-precision floating-point format:

Most positive:	$x = (2 - 2^{-23}) \times 2^{127} = 3.4028234 \times 10^{38}$
Least positive:	$x = 1 \times 2^{-127} = 5.8774717 \times 10^{-39}$
Least negative:	$x = (-1-2^{-23}) \times 2^{-127} = -5.8774724 \times 10^{-39}$
Most negative:	$x = -2 \times 2^{127} = -3.4028236 \times 10^{38}$

5.3.4 Extended-Precision Floating-Point Format

In the extended-precision format, the floating-point number is represented by an 8-bit exponent field (*e*) and a 32-bit mantissa field (*man*) with an implied most significant nonsign bit (see Figure 5–9).

Figure 5–9. Extended-Precision Floating-Point Format



Operations are performed with an implied binary point between bits 31 and 30. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point number *x* is given by the following:

$x = 01.f \times 2^{e}$	if $s = 0$
$x = 10.f \times 2^{e}$	if $s = 1$
$x = 0$	if $e = -128$

You must use the following reserved values to represent 0 in the extended-precision floating-point format:

e = –128
s = 0
f = 0

The following examples illustrate the range and precision of the extended-precision floating-point format:

Most positive:	$x = (2 - 2^{-23}) \times 2^{127} = 3.4028234 \times 10^{38}$
Least positive:	$x = 1 \times 2^{-127} = 5.8774717541 \times 10^{38}$
Least negative:	$x = (-1-2^{-31}) \times 2^{-127} = -5.8774717569 \times 10^{-39}$
Most negative:	$x = -2 \times 2^{127} = -3.4028236691 \times 10^{38}$

5.3.5 Determining the Decimal Equivalent of a TMS320C3x Floating-Point Format

To convert a ‘C3x floating-point number to its decimal equivalent, follow these steps:

Step 1: Convert the exponent field to its decimal representation.

The exponent field is a 2s-complement number. To convert a 2s-complement number, look at the MSB. If it is 0, then convert the binary number to a decimal number. If the MSB is 1, then complement the binary number, add 1 to the result, and then convert this binary number to a decimal number.

Step 2: Convert the mantissa field to its decimal representation.

The mantissa field is represented as a sign-mantissa number with an implied 1 and an implied binary point between the sign bit and the fraction field. If the sign bit is cleared ($s = 0$), form the mantissa by writing 01, and appending the bits in the fraction field after the binary point. For example, if $f = 10100000000_2$, then $man = 01.10100000000_2$:

s	Fraction										
0	1	0	1	0	0	0	0	0	0	0	0

Rewrite the mantissa as:

Mantissa													
0	1	.	1	0	1	0	0	0	0	0	0	0	0

If the sign bit is set ($s = 1$), form the mantissa by writing 10 and appending the bits in the fraction field after the binary point. For example, if $f = 10100000000_2$, then $man = 10.10100000000_2$.

s	Fraction										
1	1	0	1	0	0	0	0	0	0	0	0

Rewrite the mantissa as:

Mantissa

1	0	.	1	0	1	0	0	0	0	0	0	0	0

Step 3: Shift the decimal point of the mantissa according to the value of the exponent.

If the exponent is positive, shift the binary point to the right by the value of the exponent. If the exponent is negative, shift the binary point to the left.

For example, if $e = 2_{10}$ and the $man = 01.11000000000_2$, then the shifted mantissa becomes 0111.000000000_2, which is equivalent to 7 in decimal.

If, on the other hand, $e = –2_{10}$ and $man = 01.10000000000_2$, then the shifted mantissa becomes 0.0110000000000_2, which is equivalent to 3/8 in decimal.

The following examples illustrate how you can obtain the equivalent floating-point value of a number in ‘C3x floating-point format. Each of the examples uses the single-precision floating-point format.

Example 5–1. Positive Number

```
 0    2    4    0    0    0    0    0       Hex value
0000 0010 0100 0000 0000 0000 0000 0000     Binary value
```

Exponent = $0000\ 0010_2 = 2$

Sign = 0

Fraction = $.10000_2$

Value = $01.1_2 \times 2^2 = 0110_2. = 6$

Fraction

Implied

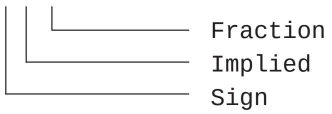
Sign

Example 5–2. Negative Number

```
 0    1    C    0    0    0    0    0      Hex value
0000 0001 1100 0000 0000 0000 0000 0000    Binary value
```

Exponent = $0000\ 0001_2 = 1$
Sign = 1
Fraction = $.10000_2$

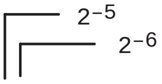
Value = $10.1_2 \times 2^1 = 101_2. = -3$



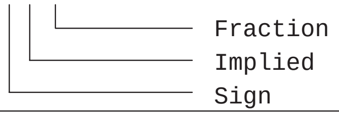
Example 5–3. Fractional Number

```
 F    B    4    0    0    0    0    0      Hex value
1111 1011 0100 0000 0000 0000 0000 0000    Binary value
```

Exponent = $1111\ 1011_2 = -5$
Sign = 0
Fraction = $.10000_2$



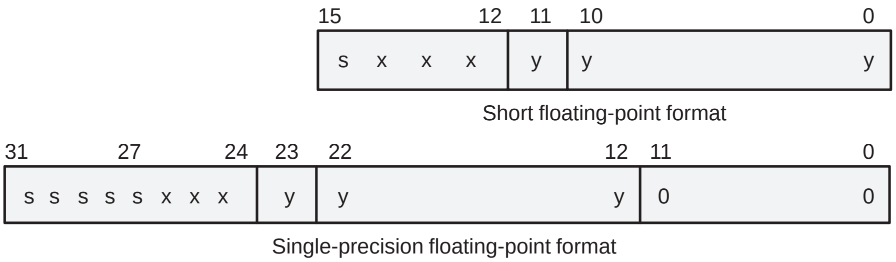
Value = $01.1_2 \times 2^{-5} = .000011_2 = 3/64$



5.3.6 Conversion Between Floating-Point Formats

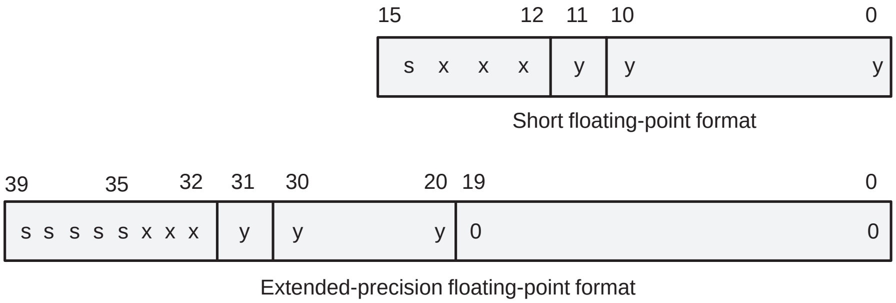
Floating-point operations assume several different formats for inputs and outputs. These formats often require conversion from one floating-point format to another (for example, short floating-point format to extended-precision floating-point format). Format conversions occur automatically in hardware, with no overhead, as a part of the floating-point operations. Examples of the four conversions are shown in Figure 5–10 through Figure 5–13. When a floating-point format 0 is converted to a greater-precision format, it is always converted to a valid representation of 0 in that format. In Figure 5–10 through Figure 5–13, *s* = sign bit of the exponent, *y* = short mantissa, and *x* = short exponent.

Figure 5–10. Converting from Short Floating-Point Format to Single-Precision Floating-Point Format


15
12
11
10
0
s x x x
y
y
y
Short floating-point format
31
27
24
23
22
12
11
0
s s s s s x x x
y
y
y
0
0
Single-precision floating-point format

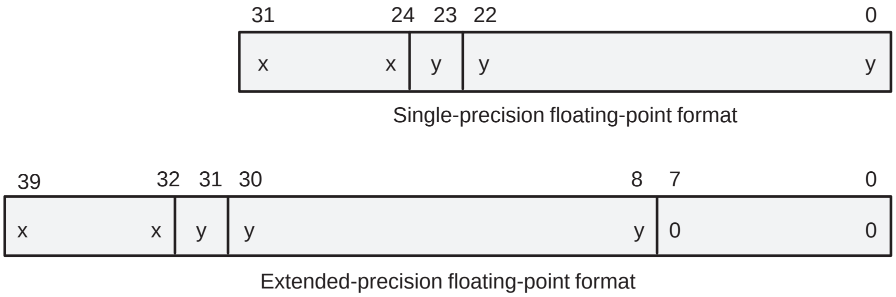
In this format, the exponent field is sign extended, and the 12 LSBs of the mantissa field are filled with 0s.

Figure 5–11. Converting from Short Floating-Point Format to Extended-Precision Floating-Point Format


15
12
11
10
0
s x x x
y
y
y
Short floating-point format
39
35
32
31
30
20
19
0
s s s s s x x x
y
y
y
0
0
Extended-precision floating-point format

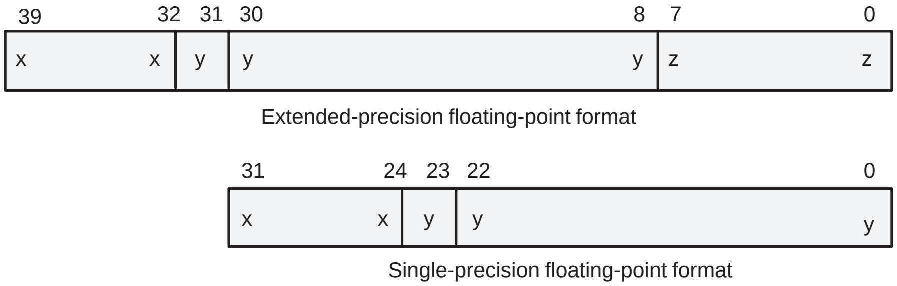
The exponent field in this format is sign extended, and the 20 LSBs of the mantissa field are filled with 0s.

Figure 5–12. Converting from Single-Precision Floating-Point Format to Extended-Precision Floating-Point Format


31
24
23
22
0
x
x
y
y
y
Single-precision floating-point format
39
32
31
30
8
7
0
x
x
y
y
y
0
0
Extended-precision floating-point format

The 8 LSBs of the mantissa field are filled with 0s.

Figure 5–13. Converting from Extended-Precision Floating-Point Format to Single-Precision Floating-Point Format


39
32
31
30
8
7
0
x
x
y
y
y
z
z
Extended-precision floating-point format
31
24
23
22
0
x
x
y
y
y
Single-precision floating-point format

The 8 LSBs of the mantissa field are truncated.

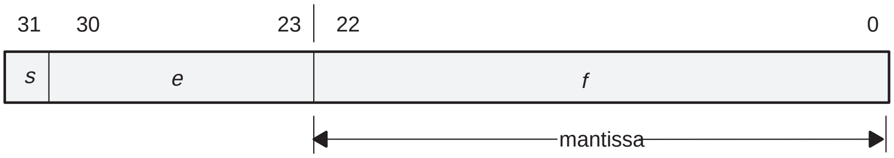
5.4 Floating-Point Conversion (IEEE Std. 754)

The ‘C3x floating-point format is not compatible with the IEEE standard 754 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 next 23 bits represent the absolute value of the mantissa with the most significant 1 implied. The binary point follows this most significant 1. In other words, the mantissa actually has 24 bits (see Figure 5–14). There are several special cases, summarized below.

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

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

Figure 5–14. IEEE Single-Precision Std. 754 Floating-Point Format



The following five cases define the value *v* of a number expressed in the IEEE format:

1) If $e = 255$ and $f \neq 0$, then $v = \text{NaN}$

2) If $e = 255$ and $f = 0$, then $v = (-1)^{s}$ infinite

3) If $0 < e < 255$, then $v = (-1)^{s} \times 2^{e-127}(1.f)$

4) If $e = 0$ and $f \neq 0$, then $v = (-1)^{s} \times 2^{-126}(0.f)$

5) If $e = 0$ and $f = 0$, then $v = (-1)^{s} \times 0$

where:

s = sign bit

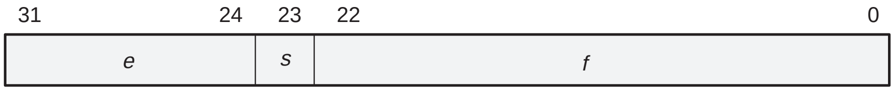
e = the exponent field

f = the fraction field

NaN = not a number

For the above five representations, *e* is treated as an unsigned integer. Case 1 generates NaN (not an number) and is primarily used for software signaling. Case 4 represents a denormalized number. Case 5 represents positive and negative 0.

Figure 5–15. TMS320C3x Single-Precision 2s-Complement Floating-Point Format



Note: Same format as for the ’C4x

In comparison, Figure 5–15 shows the the ‘C3x 2s-complement floating-point format. In this format, two cases can be used to define value *v* of a number:

1) If $e = –128$ then $v = 0$

2) If $e \neq –128$ then $v = s\overline{s}.f_2 \times 2^e$

where:

s = sign bit

e = the exponent field

f = the fraction field

For this representation, *e* is treated as a 2s-complement integer. The fraction and sign bit form a normalized 2s-complement mantissa.

Note: Differentiating Symbols for IEEE and TMS320C3x Formats

To differentiate between the symbols that define these two formats, all IEEE fields are subscripted with an IEEE (for example, e_{IEEE}, s_{IEEE}, and so forth). Similarly, all 2s-complement fields are subscripted with 2 (that is, e_2, s_2, f_2).

5.4.1 Converting IEEE Format to 2s-Complement TMS320C3x Floating-Point Format

The most common conversion is the IEEE-to-2s-complement format. This conversion is done according to rules in Table 5–1.

Table 5–1. Converting IEEE Format to 2s-Complement Floating-Point Format

		If these values are present			Then these values equal		
Description	**Case**	e_{IEEE}	s_{IEEE}	f_{IEEE}	e_2	s_2	f_2
max neg ∞	1	255	1	any	7Fh	1	00 0000h
max pos ∞	2	255	0	any	7Fh	0	7F FFFFh
	3	$0 < e_{IEEE} < 255$	0	f_{IEEE}	e_{IEEE} – 7Fh	0	f_{IEEE}
	4	$0 < e_{IEEE} < 255$	1	≠0	e_{IEEE} – 7Fh	1	$\overline{f}_{IEEE} + 1$†
	5	$0 < e_{IEEE} < 255$	1	0	e_{IEEE} – 80h	1	0
zero	6	0	any	any	80h	0	00 0000h

† $\overline{f}_{IEEE}$ = 1s complement of f_{IEEE}

Case 1 maps the IEEE positive NaNs and positive infinity to the single-precision 2s-complement most positive number. Overflow is also signaled to allow you to check for these special cases.

Case 2 maps the IEEE negative NaNs and negative infinity to the single-precision 2s-complement most negative number. Overflow is also signaled to allow you to check for these special cases.

Case 3 maps the IEEE positive normalized numbers to the identical value in the 2s-complement positive number.

Case 4 maps the IEEE negative normalized numbers with a nonzero fraction to the identical value in the 2s-complement negative number.

Case 5 maps the IEEE negative normalized numbers with a 0 fraction to the identical value in the 2s-complement negative number.

Case 6 maps the IEEE positive and negative denormalized numbers and positive and negative 0s to a 2s-complement 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 it 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.

5.4.1.1 IEEE-to-TMS320C3x Floating-Point Format Conversion

Example 5–4 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, will give erroneous results.

The fast version of the IEEE-to ’C3x conversion routine was originally developed by Keith Henry of Apollo Computer, Inc. The other routines were based on this initial input.

Example 5–4. 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
*
```

Example 5–4.IEEE-to-TMS320C3x Conversion (Fast Version) (Continued)

```
*  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
*
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 5–5 shows the complete conversion between 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 5–5. 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 5–5.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
```

5.4.2 Converting 2s-Complement TMS320C3x Floating-Point Format to IEEE Format

This conversion is performed according to the following table:

Table 5–2. Converting 2s-Complement Floating-Point Format to IEEE Format

	If these values are present			Then these values equal		
Case	$\mathbf{e_2}$	$\mathbf{s_2}$	$\mathbf{f_2}$	$\mathbf{e_{IEEE}}$	$\mathbf{s_{IEEE}}$	$\mathbf{f_{IEEE}}$
1	–128			00h	0	00 0000h
2	–127			00h	0	00 0000h
3	$-126 \le e_2 \le 127$	0		e_2+7Fh	0	f_2
4	$-126 \le e_2 \le 127$	1	$\neq 0$	e_2+7Fh	0	$\bar{f}_2+1$†
5	$-126 \le e_2 \le 127$	1	0	e_2+80h	1	00 0000h
6	127	1	0	FFh	1	00 0000h

† $\bar{f}_2$ = 2s-complement of f_2.

Case 1 maps a 2s-complement 0 to a positive IEEE 0.

Case 2 maps the 2s-complement numbers that are too small to be represented as normalized IEEE numbers to a positive IEEE 0.

Case 3 maps the positive 2s-complement numbers that are not covered by case 2 into the identically valued IEEE number.

Case 4 maps the negative 2s-complement numbers with a nonzero fraction that are not covered in case 2 into the identically valued IEEE number.

Case 5 maps all the negative 2s-complement numbers with a 0 fraction, except for the most negative 2s-complement number and those that are not covered in case 2, into the identically valued IEEE number.

Case 6 maps the most negative 2s-complement number to the IEEE negative infinity.

5.4.2.1 TMS320C3x-to-IEEE Floating-Point Format Conversion

The vast 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, while it is treated properly in the complete version. Example 5–6 shows the fast version, and Example 5–7 shows the complete version of the ’C3x-to-IEEE conversion.

Example 5–6. 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 5–6.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 5–7. 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 5–7.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
```

5.5 Floating-Point Multiplication

A floating-point number α can be written in floating-point format as in the following formula, where α(*man*) is the mantissa and α(*exp*) is the exponent:

$$\alpha = \alpha(man) \times 2^{\alpha(exp)}$$

The product of α and *b* is *c*, defined as:

$$c = \alpha \times b = \alpha(man) \times b(man) \times 2^{(\alpha(exp) + b(exp))}$$

thus:

$$c(man) = \alpha(man) \times b(man)$$
$$c(exp) = \alpha(exp) + b(exp)$$

During floating-point multiplication, source operands are in the single-precision floating-point format. If the source operands are in short floating-point format, they are converted to single-precision floating-point format. If the source operands are in extended-precision floating-point format, they are truncated to single-precision format. These conversions occur automatically in hardware with no overhead. All results of floating-point multiplications are in the extended-precision format. These multiplications occur in a single cycle.

Figure 5–16 is a flowchart that shows the steps involved in floating-point multiplication. Each step is labelled with a number in parenthesis.

❑ In step 1, the 24-bit source operand mantissas are multiplied, producing a 50-bit result *c*(*man*). (Input and output data are always represented as normalized numbers.)

❑ In step 2, the exponents, α(*exp*) and *b*(*exp*), are added, yielding *c*(*exp*).

❑ Step 3 checks for whether *c*(*man*) in extended-precision format is equal to 0. If *c*(*man*) is 0, step 7 sets *c*(*exp*) to –128, thus yielding the representation for 0.

❑ Steps 4 and 5 normalize the result.

❑ If a right shift of 1 is necessary, then in step 8, *c*(*man*) is right-shifted one bit, thus adding 1 to *c*(*exp*).

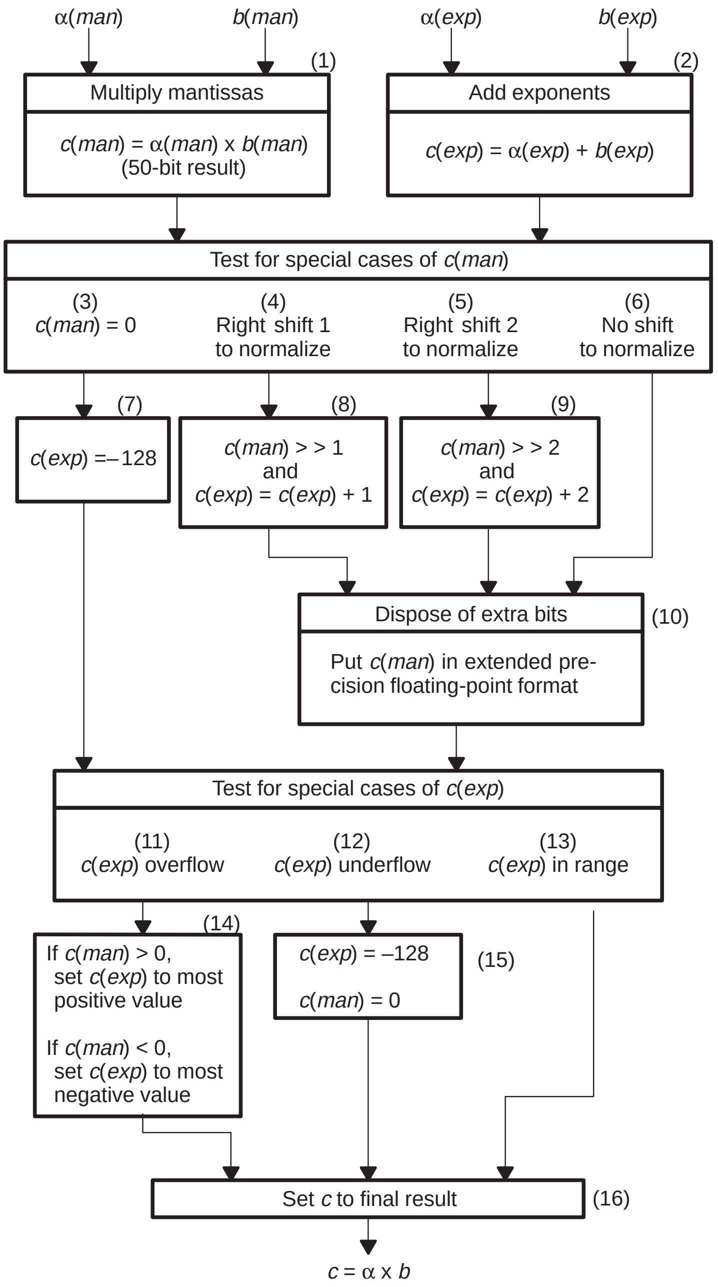
❑ If a right shift of 2 is necessary, then in step 9, *c*(*man*) is right-shifted two bits, thus adding 2 to *c*(*exp*). Step 6 occurs when the result is normalized.

❑ In step 10, *c*(*man*) is set in the extended-precision floating-point format.

❑ Steps 11 through 16 check for special cases of *c*(*exp*).

- ❑ If $c(exp)$ has overflowed (step 11) in the positive direction, then step 14 sets $c(exp)$ to the most positive extended-precision format value. If $c(exp)$ has overflowed in the negative direction, then step 14 sets $c(exp)$ to the most negative extended-precision format value.
- ❑ If $c(exp)$ has underflowed (step 12), then step 15 sets c to 0; that is, $c(man) = 0$ and $c(exp) = –128$.

Figure 5–16. Flowchart for Floating-Point Multiplication



Example 5–8 through Example 5–12 illustrate how floating-point multiplication is performed on the ’C3x. For these examples, the implied most significant nonsign bit is made explicit.

Example 5–8. Floating-Point Multiply (Both Mantissas = –2.0)

Let:

$$\begin{aligned} \alpha &= -2.0 \times 2^{\alpha(\mathit{exp})} = 10.00000000000000000000000 \times 2^{\alpha(\mathit{exp})} \\ b &= -2.0 \times 2^{b(\mathit{exp})} = 10.00000000000000000000000 \times 2^{b(\mathit{exp})} \end{aligned}$$

Where:

α and b are both represented in binary form according to the normalized single-precision floating-point format.

Then:

$$\begin{array}{l} \ 10.00000000000000000000000 \times 2^{\alpha(\mathit{exp})} \\ \times\ 10.00000000000000000000000 \times 2^{b(\mathit{exp})} \\ \hline 0100.000000000000000000000000000000000000000000 \times 2^{(\alpha(\mathit{exp}) + b(\mathit{exp}))} \end{array}$$

To place this number in the proper normalized format, it is necessary to shift the mantissa two places to the right and add 2 to the exponent. This yields:

$$\begin{array}{l} \ 10.00000000000000000000000 \times 2^{\alpha(\mathit{exp})} \\ \times\ 10.00000000000000000000000 \times 2^{b(\mathit{exp})} \\ \hline 0100.0000000000000000000000000000000000000000000 \times 2^{(\alpha(\mathit{exp}) + b(\mathit{exp}))} \end{array}$$

In floating-point multiplication, the exponent of the result may overflow. This can occur when the exponents are initially added or when the exponent is modified during normalization.

Example 5–9. Floating-Point Multiply (Both Mantissas = 1.5)

Let:

$$\begin{array}{l} \alpha = 1.5 \times 2^{\alpha(exp)} = 01.0000000000000000000000 \times 2^{\alpha(exp)} \\ b = 1.5 \times 2^{b(exp)} = 01.0000000000000000000000 \times 2^{b(exp)} \end{array}$$

Where:

a and *b* are both represented in binary form according to the single-precision floating-point format.

Then:

$$\begin{array}{rl} & 10.00000000000000000000000 \times 2^{\alpha(exp)} \\ \text{x} & 10.00000000000000000000000 \times 2^{b(exp)} \\ \hline & 01.00000000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp) + 2)} \end{array}$$

To place this number in the proper normalized format, it is necessary to shift the mantissa one place to the right and add 1 to the exponent. This yields:

$$\begin{array}{rl} & 01.0000000000000000000000 \times 2^{\alpha(exp)} \\ \times & 01.0000000000000000000000 \times 2^{b(exp)} \\ \hline & 01.\ 00100000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp) + 1)} \end{array}$$

Example 5–10. Floating-Point Multiply (Both Mantissas = 1.0)

Let:

$$\begin{array}{l} \alpha = 1.0 \times 2^{\alpha(exp)} = 01.00000000000000000000000 \times 2^{\alpha(exp)} \\ b = 1.0 \times 2^{b(exp)} = 01.00000000000000000000000 \times 2^{b(exp)} \end{array}$$

Where:

a and *b* are both represented in binary form according to the single-precision floating-point format.

Then:

$$\begin{array}{rl} & 01.00000000000000000000000 \times 2^{\alpha(exp)} \\ \times & 01.00000000000000000000000 \times 2^{b(exp)} \\ \hline & 0001.00000000000000000000000000000000000000000000 \ \text{y} \ 2^{(a(exp) + b(exp))} \end{array}$$

This number is in the proper normalized format. Therefore, no shift of the mantissa or modification of the exponent is necessary.

The previous examples show cases where the product of two normalized numbers can be normalized with a shift of 0, 1, or 2. The floating-point format of the ‘C3x makes this possible.

Example 5–11. Floating-Point Multiply Between Positive and Negative Numbers

Let:

$$\alpha = 1.0 \text{ x } 2^{\alpha(exp)} = 01.00000000000000000000000 \text{ x } 2^{\alpha(exp)}$$
$$b = -2.0 \text{ x } 2^{b(exp)} = 10.00000000000000000000000 \text{ x } 2^{b(exp)}$$

Then:

$$\begin{array}{rl} & 01.00000000000000000000000 \times 2^{\alpha(exp)} \\ \text{x} & 10.00000000000000000000000 \times 2^{b(exp)} \\ \hline \end{array}$$
$$1110.0000000000000000000000000000000000000000000000 \times 2^{(\alpha(exp)\,+\,b(exp))}$$

The result is: $c = -2.0 \text{ x } 2^{(\alpha(exp)\,+\,b(exp))}$

Example 5–12. Floating-Point Multiply by 0

All multiplications by a floating-point 0 yield a result of 0 (*f* = 0, *s* = 0, and *exp* = –128).

5.6 Floating-Point Addition and Subtraction

In floating-point addition and subtraction, two floating-point numbers α and *b* can be defined as:

$$\begin{aligned} \alpha &= \alpha(\mathit{man}) \times 2^{\alpha(\mathit{exp})} \\ b &= b(\mathit{man}) \times 2^{b(\mathit{exp})} \end{aligned}$$

The sum (or difference) of α and *b* can be defined as:

$$\begin{aligned} c &= \alpha \pm b \\ &= (\alpha(\mathit{man}) \pm (b(\mathit{man}) \times 2^{-(\alpha(\mathit{exp}) - b(\mathit{exp}))})) \times 2^{\alpha(\mathit{exp})}, \text{ if } \alpha(\mathit{exp}) \ge b(\mathit{exp}) \\ &= (\alpha(\mathit{man}) \times 2^{-(b(\mathit{exp}) - \alpha(\mathit{exp}))}) \pm b(\mathit{man})) \times 2^{b(\mathit{exp})}, \text{ if } \alpha(\mathit{exp}) < b(\mathit{exp}) \end{aligned}$$

Figure 5–17 shows the flowchart for floating-point addition. Because this flowchart assumes signed data, it is also appropriate for floating-point subtraction. In this figure, it is assumed that α(*exp*) ≤ *b*(*exp*).

- In step 1, the source exponents, α(*exp*) and *b*(*exp*), are compared, and *c*(*exp*) is set equal to the largest of the two source exponents.

- In step 2, *d* is set to the difference of the two exponents.

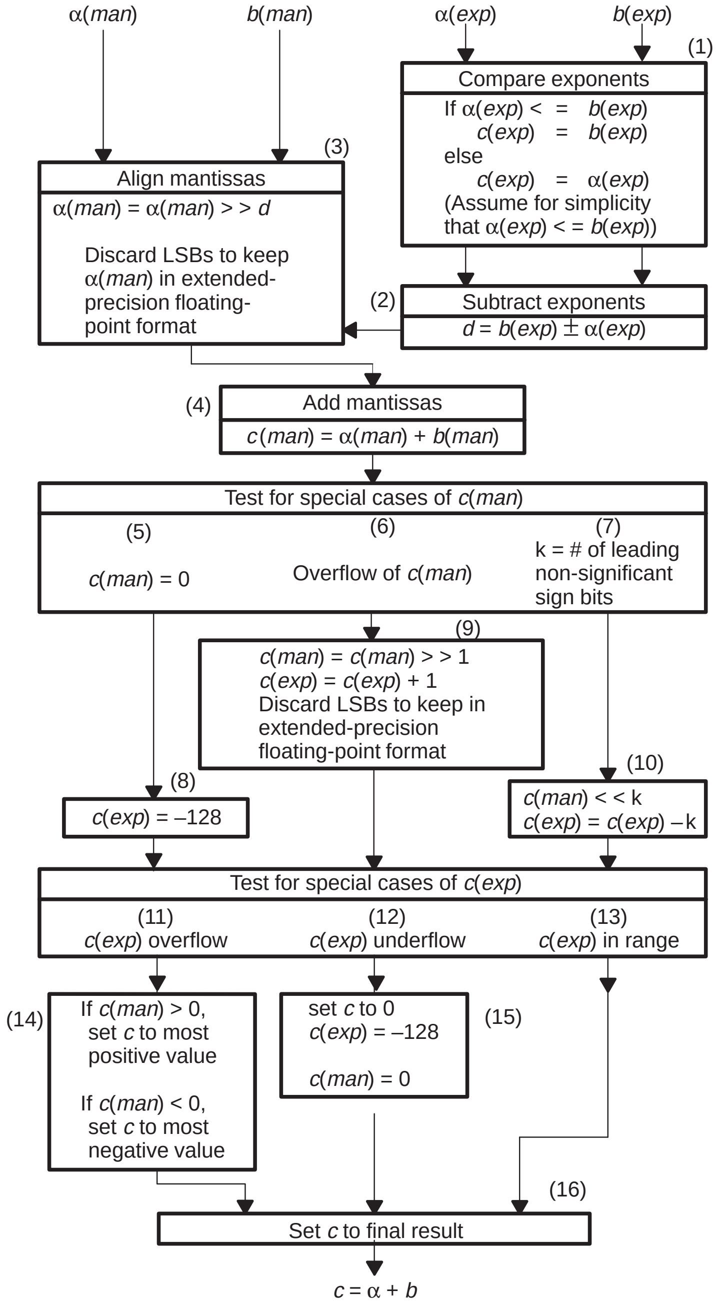
- In step 3, the mantissa with the smallest exponent, in this case α(*man*), is right-shifted *d* bits to align the mantissas.

- In step 4, after the mantissas have been aligned, they are added.

- In steps 5 through 7, a check for a special case of *c*(*man*). If *c*(*man*) is 0 (step 5), then *c*(*exp*) is set to its most negative value (step 8) to yield the correct representation of 0. If *c*(*man*) has overflowed *c* (step 6), then in step 9 *c*(*man*) is right-shifted one bit and 1 is added to *c*(*exp*). In step 10, the result is normalized.

- Steps 11 through 13 check for special cases of *c*(*exp*). If *c*(*exp*) has overflowed (step 11) in the positive direction, then step 14 sets *c*(*exp*) to the most positive extended-precision format value. If *c*(*exp*) has overflowed (step 11) in the negative direction, then step 14 sets *c*(*exp*) to the most negative extended-precision format value. If *c*(*exp*) has underflowed (step 12), then step 15 sets *c* to 0; that is, *c*(*man*) = 0 and *c*(*exp*) = –128. If no overflow or underflow occurred, then *c* is not modified.

Figure 5–17. Flowchart for Floating-Point Addition



The following examples describe the floating-point addition and subtraction operations. It is assumed that the data is in the extended-precision floating-point format.

Example 5–13. Floating-Point Addition

In the case of two normalized numbers to be summed, let

$$\alpha = 1.5 = 01.100000000000000000000000000000 \times 2^{0}$$
$$b = 0.5 = 01.000000000000000000000000000000 \times 2^{-1}$$

It is necessary to shift b to the right by 1 so that α and *b* have the same exponent. This yields:

$$b = 0.5 = 00.100000000000000000000000000000 \times 2^{0}$$

Then:

$$\begin{array}{rl} 01.1000000000000000000000000000000 & \times\, 2 \\ +00.1000000000000000000000000000000 & \times\, 2^{0} \\ \hline 010.0000000000000000000000000000000 & \times\, 2^{0} \end{array}$$

As in the case of multiplication, it is necessary to shift the binary point one place to the left and add 1 to the exponent. This yields:

$$\begin{array}{rl} 01.100000000000000000000000000000 & \times\, 2^{0} \\ \pm\, 00.100000000000000000000000000000 & \times\, 2^{0} \\ \hline 01.000000000000000000000000000000 & \times\, 2^{1} \end{array}$$

Example 5–14. Floating-Point Subtraction

A subtraction is performed in this example. Let:

$$\alpha = 01.0000000000000000000000000000001 \times 2^{0}$$
$$b = 01.0000000000000000000000000000000 \times 2^{0}$$

The operation performed is $\alpha - b$. The mantissas are already aligned because the two numbers have the same exponent. The result is a large cancellation of the upper bits, as shown below.

$$\begin{array}{rl} 01.0000000000000000000000000000001 & \times\ 2^{0} \\ -01.0000000000000000000000000000000 & \times\ 2^{0} \\ \hline 00.0000000000000000000000000000001 & \times\ 2^{0} \end{array}$$

The result must be normalized. In this case, a left shift of 31 is required. The exponent of the result is modified accordingly. The result is:

$$\begin{array}{rl} 01.0000000000000000000000000000001 & \times\ 2^{0} \\ -\ 01.0000000000000000000000000000000 & \times\ 2^{0} \\ \hline 01.0000000000000000000000000000000 & \times\ 2^{-31} \end{array}$$

Example 5–15. Floating-Point Addition With a 32-Bit Shift

This example illustrates a situation where a full 32-bit shift is necessary to normalize the result. Let:

$$\alpha = 01.1111111111111111111111111111111 \times 2^{127}$$
$$b = 10.0000000000000000000000000000000 \times 2^{127}$$

The operation to be performed is $\alpha + b$.

$$\begin{array}{rl} 01.1111111111111111111111111111111 & \times\ 2^{127} \\ +10.0000000000000000000000000000000 & \times\ 2^{127} \\ \hline 11.1111111111111111111111111111111 & \times\ 2^{127} \end{array}$$

Normalizing the result requires a left shift of 32 and a subtraction of 32 from the exponent. The result is:

$$\begin{array}{rl} 01.1111111111111111111111111111111 & \times\ 2^{127} \\ +10.0000000000000000000000000000000 & \times\ 2^{127} \\ \hline 11.1111111111111111111111111111111 & \times\ 2^{127} \end{array}$$

Example 5–16. Floating-Point Addition/Subtraction With Floating-Point 0

When floating-point addition and subtraction are performed with a floating-point 0, the following identities are satisfied:

$\alpha \pm 0 = \alpha \ (\alpha \neq 0)$
$0 \pm 0 = 0$
$0 - \alpha = -\alpha \ (\alpha \neq 0)$

5.7 Normalization Using the NORM Instruction

The NORM instruction normalizes an extended-precision floating-point number that is assumed to be unnormalized (see Example 5–17). Since the number is assumed to be unnormalized, no implied most significant nonsign bit is assumed. The NORM instruction:

1) Locates the most significant nonsign bit of the floating-point number
2) Left shifts to normalize the number
3) Adjusts the exponent

Example 5–17. NORM Instruction

Assume that an extended-precision register contains the value:

man = 00000000000000000001000000000001, *exp* = 0

When the normalization is performed on a number assumed to be unnormalized, the binary point is assumed to be:

man = 0.00000000000000000001000000000001, *exp* = 0

This number is then sign-extended one bit so that the mantissa contains 33 bits:

man = 00.00000000000000000001000000000001, *exp* = 0

The intermediate result after the most significant nonsign bit is located and the shift performed is:

man = 01.00000000000100000000000000000000, *exp* = –19

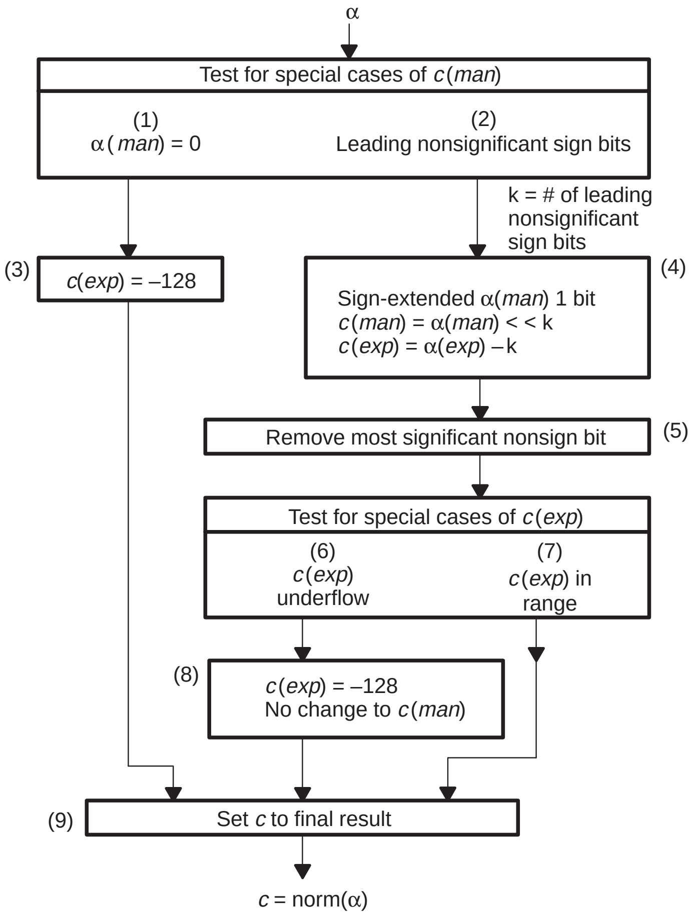
The final 32-bit value output after removing the redundant bit is:

man = 00000000000010000000000000000000, *exp* = –19

The NORM instruction is useful for counting the number of leading 0s or leading 1s in a 32-bit field. If the exponent is initially 0, the absolute value of the final value of the exponent is the number of leading 1s or 0s. This instruction is also useful for manipulating unnormalized floating-point numbers.

Given the extended-precision floating-point value *a* to be normalized, the normalization, norm ( ), is performed as shown in Figure 5–18.

Figure 5–18. Flowchart for NORM Instruction Operation


α
Test for special cases of c(man)
(1)
α(man) = 0
(2)
Leading nonsignificant sign bits
k = # of leading nonsignificant sign bits
(3)
c(exp) = –128
(4)
Sign-extended α(man) 1 bit
c(man) = α(man) < < k
c(exp) = α(exp) – k
Remove most significant nonsign bit
(5)
Test for special cases of c(exp)
(6)
c(exp) underflow
(7)
c(exp) in range
(8)
c(exp) = –128
No change to c(man)
(9)
Set c to final result
c = norm(α)

5.8 Rounding (RND Instruction)

The RND instruction rounds a number from the extended-precision floating-point format to the single-precision floating-point format. Rounding is similar to floating-point addition. Given the number *a* to be rounded, the following operation is performed first.

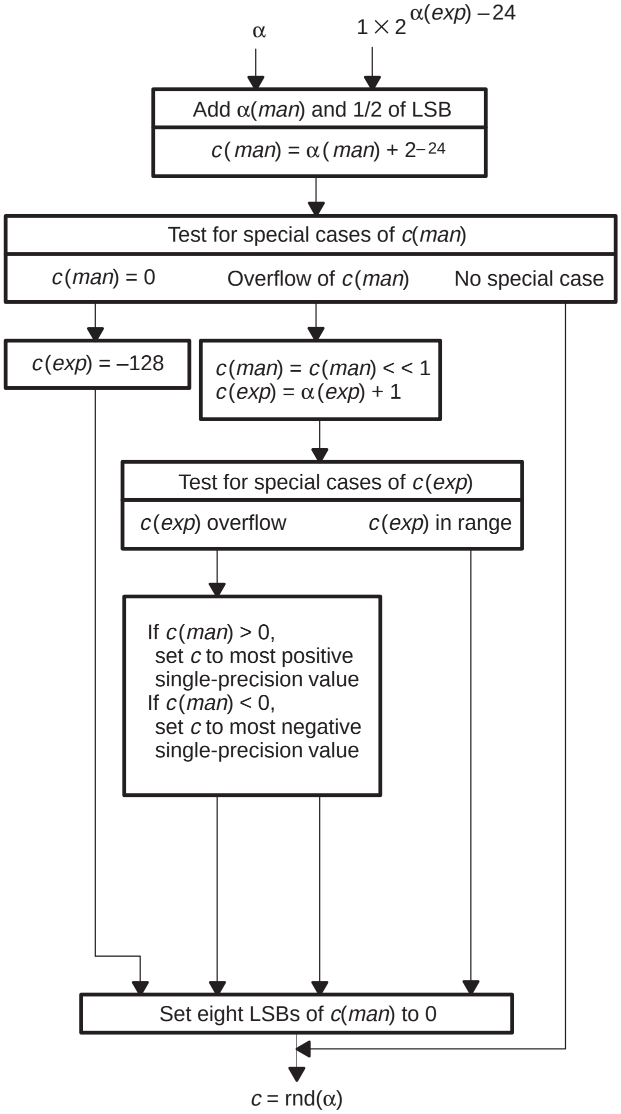
$c = \alpha(man) \times 2^{\alpha(exp)} + (1 \times 2^{\alpha(exp)-24})$

Next, a conversion from extended-precision floating-point to single-precision floating-point format is performed. Given the extended-precision floating-point value, the rounding, rnd( ), is performed as shown in Figure 5–19.

Note:

RND, *src*, *dst*—where (*src*) = 0—does not set the 0 conditions flag (bit 2 in the status register). Instead, it sets the underflow condition flag (bit 4 in the status register). When required, check for the underflow condition instead of the 0 condition.

Figure 5–19. Flowchart for Floating-Point Rounding by the RND Instruction


α
1 × 2 α(exp) – 24
Add α(man) and 1/2 of LSB
c(man) = α (man) + 2– 24
Test for special cases of c(man)
c(man) = 0
Overflow of c(man)
No special case
c(exp) = –128
c(man) = c(man) < < 1
c(exp) = α (exp) + 1
Test for special cases of c(exp)
c(exp) overflow
c(exp) in range
If c(man) > 0, set c to most positive single-precision value
If c(man) < 0, set c to most negative single-precision value
Set eight LSBs of c(man) to 0
c = rnd(α)

5.9 Floating-Point to Integer Conversion (FIX Instruction)

Using the FIX instruction, you can convert an extended-precision floating-point number to a single-precision integer in a single cycle. The floating-point to integer conversion of the value *x* is referred to here as fix(*x*). The conversion does not overflow if *a*, the number to be converted, is in the range:

$-2^{31} \leq \alpha \leq 2^{31} - 1$

First, you must be certain that

$\alpha(exp) \leq 30$

If these bounds are not met, an overflow occurs. If an overflow occurs in the positive direction, the output is the most positive integer. If an overflow occurs in the negative direction, the output is the most negative integer. If $\alpha(exp)$ is within the valid range, then $\alpha(man)$, with implied bit included, is sign-extended and right-shifted (rs) by the amount

$rs = 31 - \alpha(exp)$

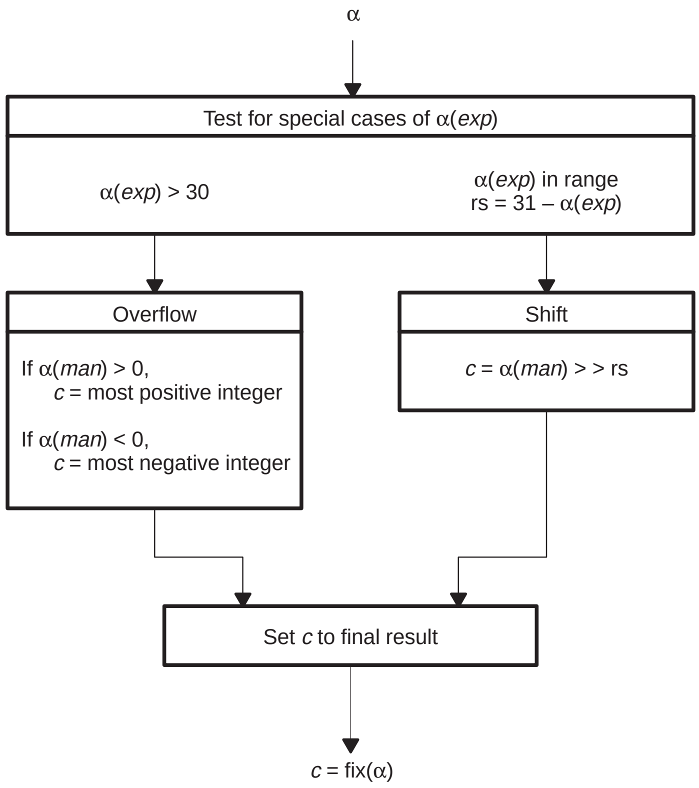
This right shift (rs) shifts out those bits corresponding to the fractional part of the mantissa. For example:

If $0 \leq \times < 1$, then fix(*x*) = 0

If $-1 \leq \times < 0$, then fix(*x*) = –1

Figure 5–20 shows the flowchart for the floating-point-to-integer conversion.

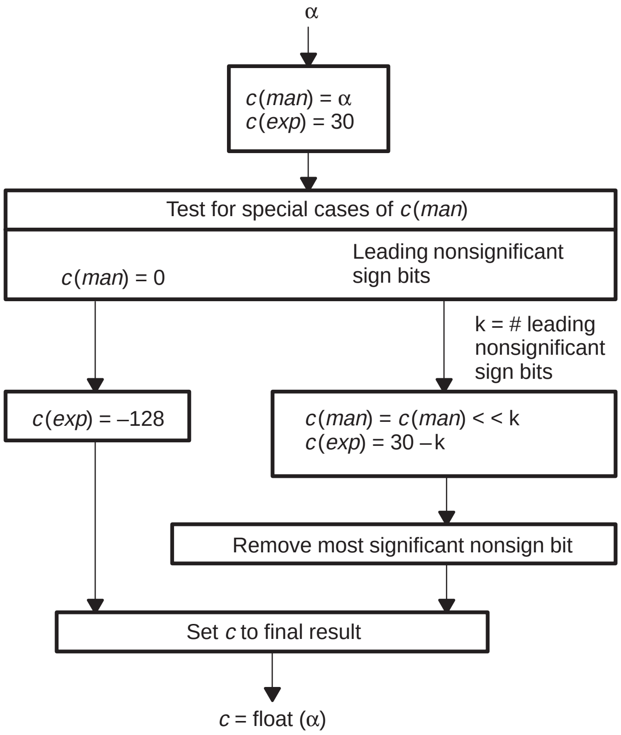
Figure 5–20. Flowchart for Floating-Point to Integer Conversion by FIX Instruction



5.10 Integer to Floating-Point Conversion (FLOAT Instruction)

Integer to floating-point conversion, using the FLOAT instruction, allows single-precision integers to be converted to extended-precision floating-point numbers. The flowchart for this conversion is shown in Figure 5–21.

Figure 5–21. Flowchart for Integer to Floating-Point Conversion by FLOAT Instruction


α
c(man) = α
c(exp) = 30
Test for special cases of c(man)
c(man) = 0
Leading nonsignificant sign bits
k = # leading nonsignificant sign bits
c(exp) = –128
c(man) = c(man) < < k
c(exp) = 30 – k
Remove most significant nonsign bit
Set c to final result
c = float (α)

5.11 Fast Logarithms on a Floating-Point Device

The following TMS320C30/C40 function calculates the log base two of a number in about half the time of conventional algorithms. Furthermore, the method can easily be scaled for faster execution if less accuracy is desired. The method is efficient because the algorithm uses the floating-point multipliers' exponent/normalization hardware in a unique way. The following is a proof of the algorithm. The value of a floating point number X is given by:

$X = 2^{\wedge}EXP\_old * mant\_old$

Since the bit fields used to store the exponent and mantissa are actually integer, the exponent is already in log2 (log base 2) form. In fact, the exponent is nothing more than a normalizing shift value. By converting both sides of the first equation to a logarithm, the logarithm of the value becomes the sum of the exponent and mantissa in log form:

$log2(X) = EXP\_old + log2(mant\_old)$ *(Log base two)*

Since EXP is in the exponent register, no calculation is needed and the value can be used directly as an integer. To extract the value of the exponent, PUSH, POP, and masking operations are used. The remaining mantissa conversion is done by first forcing the exponent bits to zero using an LDE 1.0 instruction. This causes the exponent term 2^EXP to equal 1.0, leaving 1.0 <= Value < 2.0. Then, by using the following identity, the logarithm of the mantissa can be extracted from the final results exponent. If the value (mant_old) is repeatedly squared, the sequence becomes:

$X\_new = mant\_old^{\wedge}N$

where:

$1.0 \quad X\_new < 2^{\wedge}N$

$N = 1,2,4,8,16...$

Since the hardware multiplier restructures the new value (X_new) during each squaring operation, X_new is represented by a new exponent (EXP_new) and mantissa (mant_new):

$X\_new = 2^{\wedge}EXP\_New * mant\_new$

By then applying familiar logarithm rules, we find that EXP_new holds the logarithm of Old_mant. This is best shown by setting the previous two equations equal to each other and taking the logarithm of both sides:

$mant\_old^{\wedge}N = 2^{\wedge}EXP\_new * mant\_new$

$N=1,2,4,8,16...$

$$N * \log2(mant\_old) = EXP\_new + \log2(mant\_new)$$

$$\log2(mant\_old) = EXP\_new / N + \log2(mant\_new) / N$$

This last equation shows that the logarithm of mant_old is indeed related to EXP_new. And as shown earlier, EXP_new can be separated from the new mantissa and used as the logarithm of the original mantissa.

We also need to consider the divisor N, which is defined to be the series 1, 2, 4, 8, 16... , and EXP_new is an integer. The division by N becomes a shift for each squaring operation. What remains is to concatenate the bits of EXP_new to EXP_old and then repeat the process until the desired accuracy is achieved.

5.11.1 Example of Fast Logarithm on a Floating-Point Device

Consider a mantissa value of 1.5 and an exponent value of 0 (giving an exponent multiplier 2^0, or 1.0). The TMS320C30/C40 extended register bit pattern for the algorithm sequence is shown below.

Table 5–3. Squaring Operation of F0 = 1.5

Squaring Operation of F0 = 1.5

Exp	S	Mantissa			
00000000	0	1000000000000000000000000000000	X	=1.5	Exp=0
00000001	0	0010000000000000000000000000000	X^2	=2.25	Exp=1
00000010	0	0100010000000000000000000000000	X^4	=5.0625	Exp=2
00000100	0	1001101000010000000000000000000	X^8	=25.628906	Exp=4
00100100	0	0100100001101011101000001000000	X^16	=656.84083	Exp=9
00010010	0	1010010101010011111101110011111	X^32	=431.43988–E3	Exp=18
00100101	0	0101101010110110101000010101001	X^64	=186.14037–E9	Exp=37
01001010	0	1101010110010010001010101100011	X^128	=34.648238–E21	Exp=74
XXXXXXXX	S	MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM			
← Exp →	S	← Mantissa →			

Hand-calculated value of log2(1.5)

$\log2(1.5) = 0.58496250 = 1001010\ 111000000$

xxxxxxx– first 7 bits (exponent)

mmm– quick 3 bits (mantissa)

If you compare the hand-calculated value and the binary representation of log2(1.5), you find that the sequence of bits in the exponent (seven bits worth) are equivalent to the seven MSBs of the logarithm. If the exponent could hold all the bits needed for full accuracy, then it would be possible to continue the operation for all 24 bits of the mantissa. Since there are only eight bits in the exponent and the MSBs are used for negative values, only seven iterations are possible before the exponent must be off-loaded and reinitialized to zero.

By concatenating EXP_new to the previous exponent, longer strings of bits can be built for greater accuracy. The process is then repeated until the desired accuracy is achieved. Also remember that the original numbers exponent, which represents the whole number part of the result, becomes the eight MSBs of the final result.

Another technique is to look at the three MSBs of the mantissa and apply a roundup from the fourth bit. Those same MSBs can be used as a quick extension of the exponent (logarithm). To visualize this, consider the following tabulated values and graph.

Mant	**log^2(Mant)**
1.000	0.000
1.250	0.322
1.500	0.585
1.7500	0.807
1.999	0.999

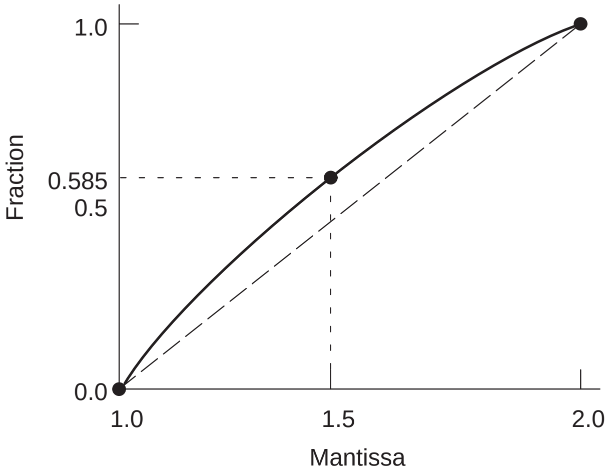


Figure 5–22. Tabulated Values for Mantissa

Note:

The fractional part is the same at the endpoints.

In the middle, only a slight bowing exists which can either be ignored or optionally rounded for better accuracy. The maximum actually occurs at a mantissa value of

$$\frac{1}{\mathit{ln}(2.0)}$$

or 1.442695. The value of log2(mant) at that point is 0.52876637, giving a maximum error of 0.086071.

When finished, the bits representing the finished logarithm are in a fixed-point notation and need to be scaled. This is done by using the FLOAT instruction followed by a multiplication by a constant scaling factor. If the final result needs to be in any other base, the scaling factor is simply adjusted for that base.

5.11.2 Points to Consider

The round-off accuracy of the first three squaring operations affect the final result if >21 mantissa bits are desired. A RND instruction placed after the first three MPYF R0,R0 instructions remedy this, but adds to the cycle count.

When the input value approaches 1.0, the result is driven close to zero and accuracy suffers. In this case, an input range comparison and a branch to a McLauren series expansion is used as a solution with minimal degradation in speed. This is because the power series converges quickly for input values close to 1.0.

If you only need to calculate a visual quality logarithm, such as in spectrum analysis, the logarithm can often be calculated in one cycle. In this case the mantissa is substituted directly into the fractional bits of the logarithm giving a maximum error of 0.086 (about 3.5 bits). The one cycle arises from the need to remove the 2's compliment sign bit in the 'C3x's mantissa.

Figure 5–23. Fast Logarithm for FFT Displays

```
******************************************************************* *
*                  FAST Logarithm for FFT displays                  *
*     >>>> NEED ONLY ADD ONE INSTRUCTION IN MANY CASES  <<<<        *
*********************************************************************

            ||      ||                ;
          MPYF    REAL,REAL,R0        ; calculate the magnitude
          MYPF    IMAG,IMAG,R1        ; Note: sign bit is zero
          ADDF    R1,R0               ;
          ASH     –1,R0               ; <–One instruction logarithm!
          STR     R0,OUT              ;   scaled externally in DAC
            ||      ||                ;
*********************************************************************
* _log_E.asm                          DEVICE: TMS320C30             *
*********************************************************************
          .global_log_E
_log_E:POP       AR1                  ; return address –> AR1
          POPF   R0                   ; X –> R0
          LDF    R0,R1                ; use R1 to accumulate answer
          LDI    2,RC                 ; repeat 3x
          RPTB   loop                 ;
          ASH    7,R1                 ;
          LDE    1.0,R0               ; EXP = 0
          MPYF   R0,R0                ; mant^2
          MPYF   R0,R0                ; mant^4
          MPYF   R0,R0                ; mant^8
          MPYF   R0,R0                ; mant^16
          MPYF   R0,R0                ; mant^32
          MPYF   R0,R0                ; mant^64
          MPYF   R0,R0                ; mant^128
          PUSHF  R1                   ; PUSH EXP and Mantissa (sign is now data!)
          POP    R0                   ; POP as ianteger (EXP+FRACTION)
          BD     AR1                  ;
          FLOAT  R0                   ; convert EXP+FRACTION to float
          MPYF   @CONST,R0            ; scale the result by 2^–24 and change base
          ADDI   1,SP                 ; restore stack pointer
          .data
CONST_ADR:       .word CONST
CONST            .long    0e7317219h               ;;Base e hand calc w/1 lsb round
                 .end
```

Chapter 6

Addressing Modes

The 'C3x supports five groups of powerful addressing modes. Six types of addressing that allow data access from memory, registers, and the instruction word can be used within the groups. This chapter describes the operation, encoding, and implementation of the addressing modes. It also discusses the management of system stacks, queues, and dequeues in memory.

Topic **Page**

6.1 Addressing Types

You can access data from memory, registers, and the instruction word by using five types of addressing:

- ❑ **Register addressing**. A CPU register contains the operand.
- ❑ **Direct addressing**. The data address is formed by concatenating the eight least significant bits (LSBs) of the data-page (DP) register and the 16 LSBs of the instruction.
- ❑ **Indirect addressing**. An auxiliary register contains the address of the operand.
- ❑ **Immediate addressing**. The operand is a 16-bit or 24-bit immediate value.
- ❑ **PC-relative addressing**. A 16-bit or 24-bit displacement to the program counter (PC).

Two specialized modes are available for use in filters, FFTs, and DSP algorithms:

- ❑ **Circular addressing**. An auxiliary register is incremented/decremented with regards to a circular buffer boundary.
- ❑ **Bit-reverse addressing**. An auxiliary register is transferred to its bit-reversed representation that contains the address of the operand.

6.2 Register Addressing

In register addressing, a CPU register contains the operand, as shown in this example:

```
ABSF    R1          ; R1 = |R1|
```

The syntax for the CPU registers, the assembler syntax, and the assigned function for those registers are listed in Table 6–1.

Table 6–1. CPU Register Address/Assembler Syntax and Function

Register Name	Machine Address	Assigned Function
R0	00h	Extended-precision register 0
R1	01h	Extended-precision register 1
R2	02h	Extended-precision register 2
R3	03h	Extended-precision register 3
R4	04h	Extended-precision register 4
R5	05h	Extended-precision register 5
R6	06h	Extended-precision register 6
R7	07h	Extended-precision register 7
AR0	08h	Auxiliary register 0
AR1	09h	Auxiliary register 1
AR2	0Ah	Auxiliary register 2
AR3	0Bh	Auxiliary register 3
AR4	0Ch	Auxiliary register 4
AR5	0Dh	Auxiliary register 5
AR6	0Eh	Auxiliary register 6
AR7	0FH	Auxiliary register 7
DP	10h	Data-page pointer
IR0	11h	Index register 0
IR1	12h	Index register 1
BK	13h	Block-size register
SP	14h	Active stack pointer
ST	15h	Status register
IE	16h	CPU/DMA interrupt-enable
IF	17h	CPU interrupt flags
IOF	18h	I/O flags
RS	19h	Repeat start address
RE	1Ah	Repeat end address
RC	1Bh	Repeat counter

6.3 Direct Addressing

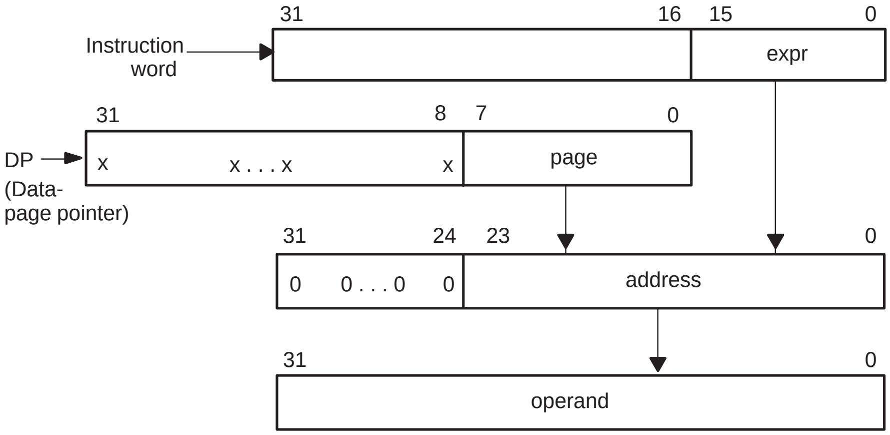
In direct addressing, the data address is formed by the concatenation of the eight LSBs of the data-page pointer (DP) with the 16 LSBs of the instruction word (expr). This results in 256 pages (64K words per page), allowing you to access a large address space without requiring a change of the page pointer. The syntax and operation for direct addressing are:

Syntax: @expr

Operation: address = DP concatenated with expr

Figure 6–1 shows the formation of the data address. Example 6–1 is an instruction example with data before and after instruction execution.

Figure 6–1. Direct Addressing



Example 6–1. Direct Addressing

```
ADDI   @0BCDEh,R7
                      Before Instruction              After Instruction
                 R7   00 0000 0000              R7    00 1234 5678
                 DP             8A              DP              8A
Data memory
           8ABCDEh       1234 5678         8ABCDEh       1234 5678
```

6.4 Indirect Addressing

Indirect addressing specifies the address of an operand in memory through the contents of an auxiliary register, optional displacements, and index registers as shown in Example 6–2. Only the 24 LSBs of the auxiliary registers and index registers are used in indirect addressing. The auxiliary register arithmetic units (ARAUs) perform the unsigned arithmetic on these lower 24 bits. The upper eight bits are unmodified.

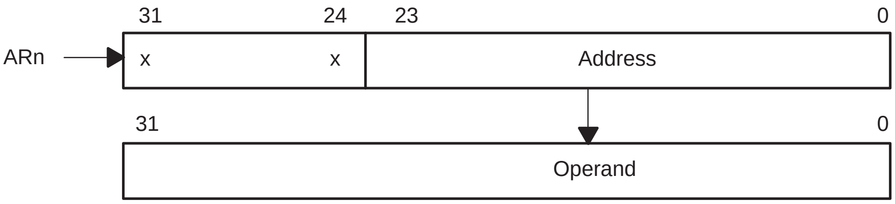
Example 6–2. Auxiliary Register Indirect

An auxiliary register (AR*n*) contains the address of the operand to be fetched.

Operation: operand address = AR*n*

Assembler Syntax: *AR*n*

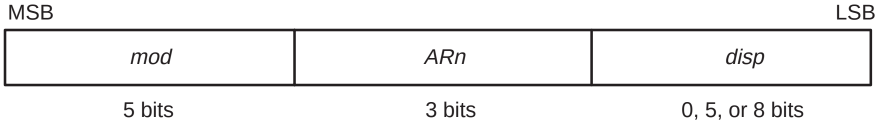
Modification Field: 11000



The flexibility of indirect addressing is possible because the ARAUs on the ’C3x modify auxiliary registers in parallel with operations within the main CPU. Indirect addressing is specified by a 5-bit field in the instruction word, referred to as the mod field (shown in Table 6–2). A displacement is either an explicit unsigned 8-bit integer contained in the instruction word or an implicit displacement of 1. Two index registers, IR0 and IR1, can also be used in indirect addressing, enabling the use of 24-bit indirect displacement. In some cases, an addressing scheme using circular or bit-reversed addressing is optional. Generating addresses in circular addressing is discussed in Section 6.7 on page 6-21; bit-reversed addressing is discussed in Section 6.8 on page 6-26.

Table 6–2 lists the various kinds of indirect addressing, along with the value of the modification (mod) field, assembler syntax, operation, and function for each. The succeeding 17 examples show the operation for each kind of indirect addressing. Figure 6–3 on page 6-20 shows the format in the instruction encoding.

Figure 6–2. Indirect Addressing Operand Encoding



Note: Auxiliary Register

The auxiliary register (AR*n*) is encoded in the instruction word according to its binary representation *n* (for example, AR3 is encoded as 11_2), not its register machine address (shown in Table 6–1).

Table 6–2. Indirect Addressing

(a) Indirect addressing with displacement

Mod Field	Syntax	Operation	Description
00000	*+AR_n_(_disp_)	addr = AR_n_ + _disp_	With predisplacement add
00001	*–AR_n_(_disp_)	addr = AR_n_ – _disp_	With predisplacement subtract

00010	*++AR_n_(_disp_)	addr = AR_n_ + _disp_
		AR_n_ = AR_n_ + _disp_
			With predisplacement add and modify

00011	*– –AR_n_(_disp_)	addr = AR_n_ – _disp_
		AR_n_ = AR_n_ – _disp_
			With predisplacement subtract and modify

00100	*AR_n_++(_disp_)	addr = AR_n_
		AR_n_ = AR_n_ + _disp_
			With postdisplacement add and modify

00101	*AR_n_– –(_disp_)	addr = AR_n_
		AR_n_ = AR_n_ – _disp_
			With postdisplacement subtract and modify

00110	*AR_n_++(_disp_)%	addr = AR_n_
		AR_n_ = circ(AR_n_ + _disp_)
			With postdisplacement add and circular modify

00111	*AR_n_– –(_disp_)%	addr = AR_n_
		AR_n_ = circ(AR_n_ – _disp_)
			With postdisplacement subtract and circular modify

(b) Indirect addressing with index register IR0

Mod Field	Syntax	Operation	Description
01000	*+AR_n_(IR0)	addr = AR_n_ + IR0	With preindex (IR0) add
01001	*–AR_n_(IR0)	addr = AR_n_ – IR0	With preindex (IR0) subtract

01010	*++AR_n_(IR0)	addr = AR_n_ + IR0
		AR_n_ = AR_n_ + IR0
			With preindex (IR0) add and modify

01011	*– –AR_n_(IR0)	addr = AR_n_ – IR0
		AR_n_ = AR_n_ – IR0
			With preindex (IR0) subtract and modify

01100	*AR_n_++(IR0)	addr = AR_n_
		AR_n_ = AR_n_ + IR0
			With postindex (IR0) add and modify

01101	*AR_n_– –(IR0)	addr= AR_n_
		AR_n_ = AR_n_ – IR0
			With postindex (IR0) subtract and modify

01110	*AR_n_++(IR0)%	addr = AR_n_
		AR_n_ = circ(AR_n_ + IR0)
			With postindex (IR0) add and circular modify

01111	*AR_n_– –(IR0)%	addr = AR_n_
		AR_n_ = circ(AR_n_– IR0)
			With postindex (IR0) subtract and circular modify

Legend:	addr	memory address	++	add and modify
	AR_n_	auxiliary registers AR0–AR7	– –	subtract and modify
	circ( )	address in circular addressing	%	where circular addressing is performed
	disp	displacement	IR_n_	index register IR0 or IR1

Table 6–2. Indirect Addressing (Continued)

(c) Indirect addressing with index register IR1

Mod Field	**Syntax**	**Operation**	**Description**
10000	*+AR*n*(IR1)	addr = AR*n* + IR1	With preindex (IR1) add
10001	*–AR*n*(IR1)	addr = AR*n* – IR1	With preindex (IR1) subtract
10010	*++AR*n*(IR1)	addr = AR*n* + IR1 AR*n* = AR*n* + IR1	With preindex (IR1) add and modify
10011	*––AR*n*(IR1)	addr = AR*n* – IR1 AR*n* = AR*n* – IR1	With preindex (IR1) subtract and modify
10100	*AR*n*++(IR1)	addr = AR*n* AR*n* = AR*n* + IR1	With postindex (IR1) add and modify
10101	*AR*n*––(IR1)	addr = AR*n* AR*n* = AR*n* – IR1	With postindex (IR1) subtract and modify
10110	*AR*n*++(IR1)%	addr = AR*n* AR*n* = circ(AR*n* + IR1)	With postindex (IR1) add and circular modify
10111	*AR*n*––(IR1)%	addr = AR*n* AR*n* = circ(AR*n* – IR1)	With postindex (IR1) subtract and circular modify

(d) Indirect addressing (special cases)

Mod Field	**Syntax**	**Operation**	**Description**
11000	*AR*n*	addr = AR*n*	Indirect
11001	*AR*n*++(IR0)B	addr = AR*n* AR*n* = B(AR*n* + IR0)	With postindex (IR0) add and bit-reversed modify

Legend:	addr	memory address	circ( )	address in circular addressing
	AR*n*	auxiliary registers AR0–AR7	++	add and modify by 1
	B	where bit-reversed addressing is performed	– –	subtract and modify by 1
	B()	bit-reversed address	%	where circular addressing is performed
			IR*n*	index register IR0 or IR1

Example 6–3 through Example 6–19 show the operation for each type of indirect addressing.

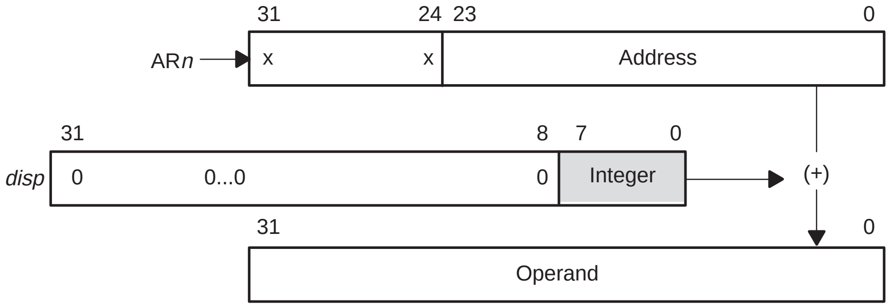
Example 6–3. Indirect Addressing With Predisplacement Add

The address of the operand to fetch is the sum of an auxiliary register (AR*n*) and the displacement (*disp*). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n* + *disp*

Assembler Syntax: *+AR*n*(*disp*)

Modification Field: 00000



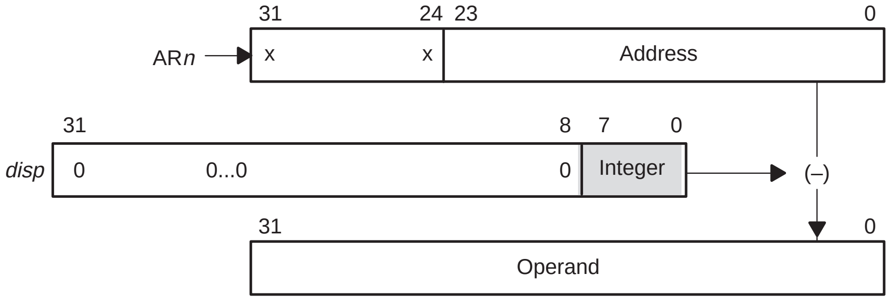
Example 6–4. Indirect Addressing With Predisplacement Subtract

The address of the operand to fetch is the contents of an auxiliary register (AR*n*) minus the displacement (*disp*). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n* – *disp*

Assembler Syntax: *–AR*n*(*disp*)

Modification Field: 00001



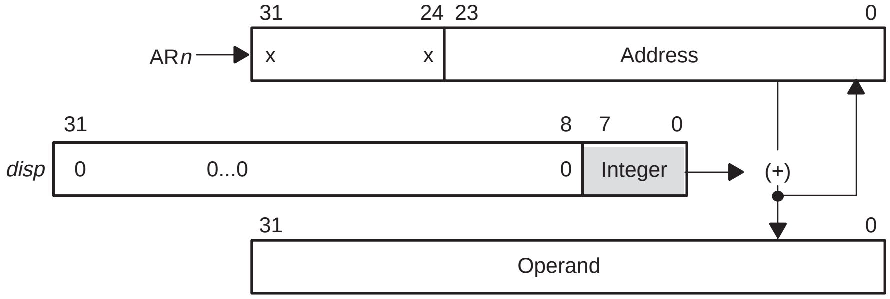
Example 6–5. Indirect Addressing With Predisplacement Add and Modify

The address of the operand to fetch is the sum of an auxiliary register (AR*n*) and the displacement (*disp*). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1. After the data is fetched, the auxiliary register is updated with the address generated.

Operation: operand address = AR*n* + *disp*
AR*n* = AR*n* + *disp*

Assembler Syntax: *++AR*n* (*disp*)

Modification Field: 00010



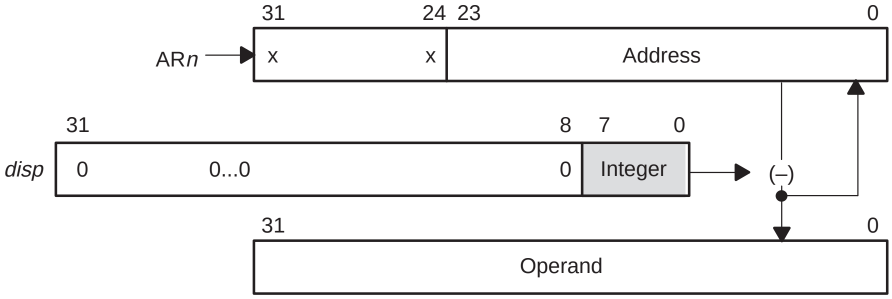
Example 6–6. Indirect Addressing With Predisplacement Subtract and Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*) minus the displacement (*disp*). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1. After the data is fetched, the auxiliary register is updated with the address generated.

Operation: operand address = AR*n* – *disp*
AR*n* = AR*n* – *disp*

Assembler Syntax: *––AR*n*(*disp*)

Modification Field: 00011



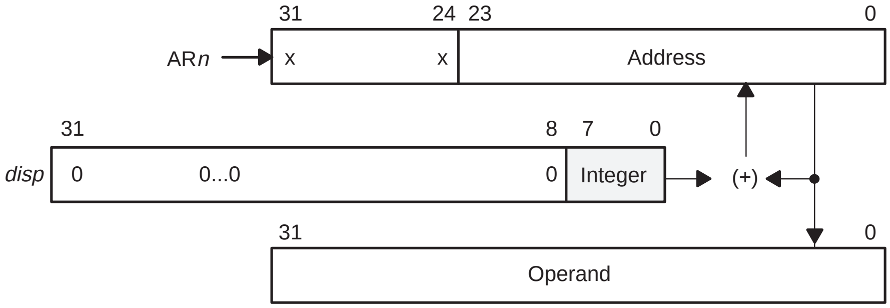
Example 6–7. Indirect Addressing With Postdisplacement Add and Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the displacement (*disp*) is added to the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n* = AR*n* + *disp*

Assembler Syntax: *AR*n*++(*disp*)

Modification Field: 00100



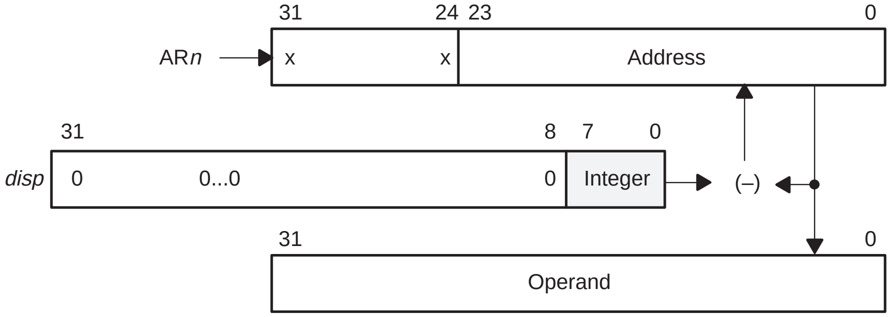
Example 6–8. Indirect Addressing With Postdisplacement Subtract and Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the displacement (*disp*) is subtracted from the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n* = AR*n* – *disp*

Assembler Syntax: *AR*n*––(*disp*)

Modification Field: 00101



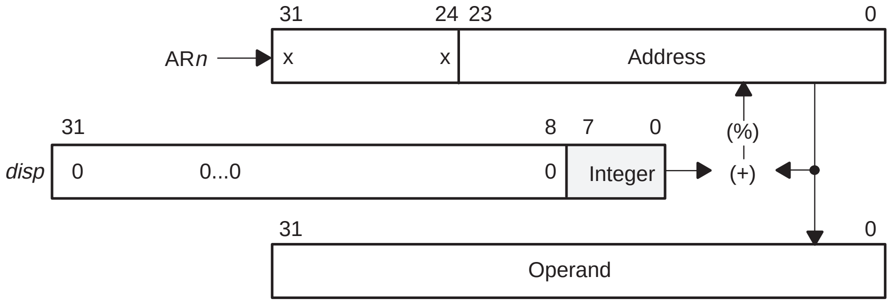
Example 6–9. Indirect Addressing With Postdisplacement Add and Circular Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the displacement (*disp*) is added to the contents of the auxiliary register using circular addressing. This result is used to update the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n* = circ(AR*n*+ *disp*)

Assembler Syntax: *AR*n* ++(*disp*)%

Modification Field: 00110



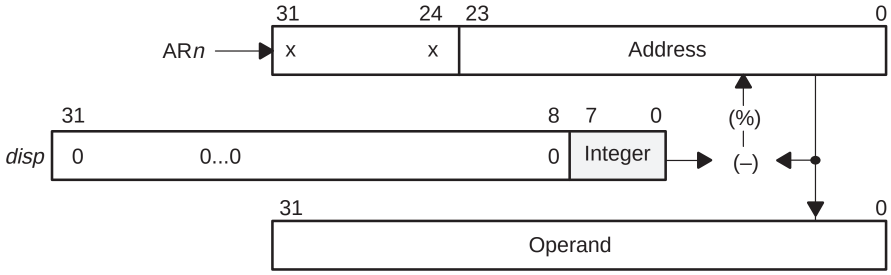
Example 6–10. Indirect Addressing With Postdisplacement Subtract and Circular Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the displacement (*disp*) is subtracted from the contents of the auxiliary register using circular addressing. This result is used to update the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n* = circ(AR *n* – *disp*)

Assembler Syntax: *AR*n*––(*disp*)%

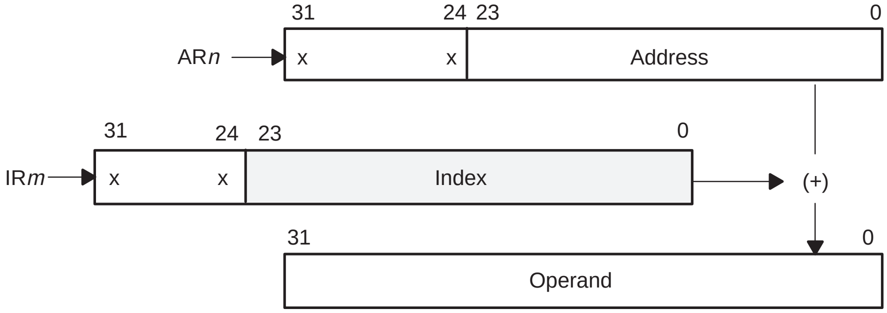
Modification Field: 00111



Example 6–11. Indirect Addressing With Preindex Add

The address of the operand to fetch is the sum of an auxiliary register (AR*n*) and an index register (IR0 or IR1).

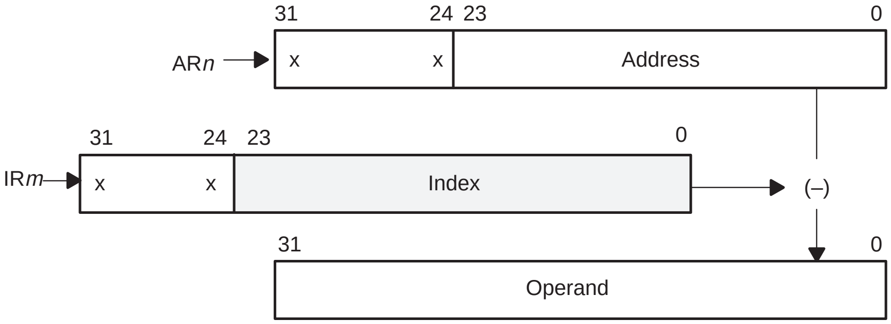
Operation:	operand address = AR*n* + IR*m*	
Assembler Syntax:	*+AR*n*(IR*m*)	
Modification Field:	01000	if *m* = 0
	10000	if *m* = 1



Example 6–12. Indirect Addressing With Preindex Subtract

The address of the operand to fetch is the difference of an auxiliary register (AR*n*) and an index register (IR0 or IR1).

Operation:	operand address = AR*n* – IR*m*	
Assembler Syntax:	*–AR*n*(IR*m*)	
Modification Field:	01001	if *m* = 0
	10001	if *m* = 1



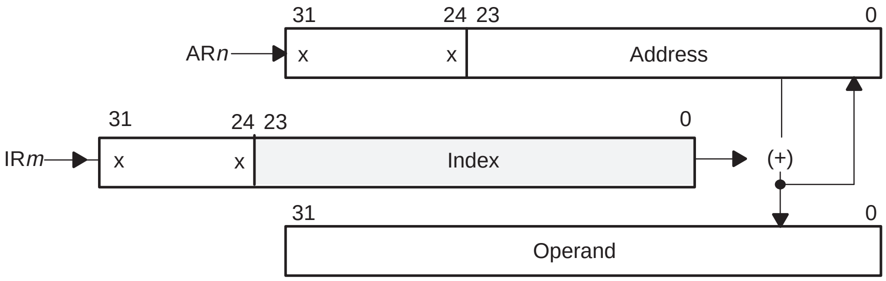
Example 6–13. Indirect Addressing With Preindex Add and Modify

The address of the operand to fetch is the sum of an auxiliary register (AR*n*) and an index register (IR0 or IR1). After the data is fetched, the auxiliary register is updated with the generated address.

Operation: operand address = AR*n* + IR*m*
AR*n* = AR*n* + IR*m*

Assembler Syntax: *++AR*n*(IR*m*)

Modification Field: 01010 if *m* = 0
10010 if *m* = 1



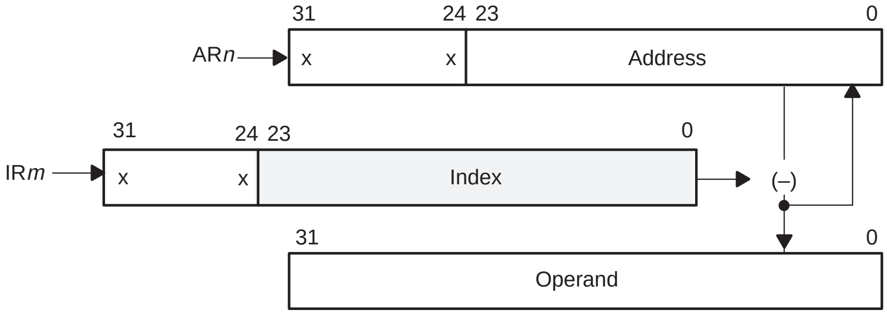
Example 6–14. Indirect Addressing With Preindex Subtract and Modify

The address of the operand to fetch is the difference between an auxiliary register (AR*n*) and an index register (IR0 or IR1). The resulting address becomes the new contents of the auxiliary register.

Operation: operand address = AR*n* – IR*m*
AR*n* = AR*n* – IR*m*

Assembler Syntax: *– –AR*n*(IR*m*)

Modification Field: 01011 if *m* = 0
10011 if *m* = 1



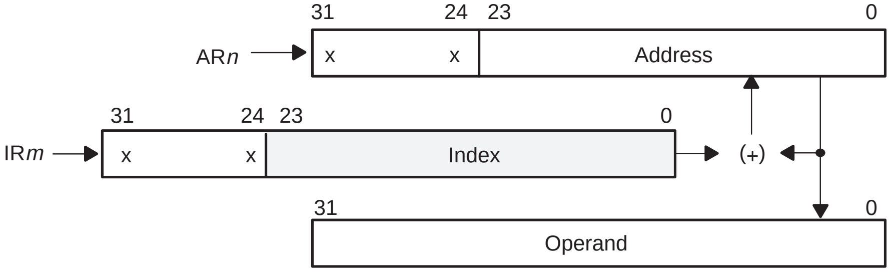
Example 6–15. Indirect Addressing With Postindex Add and Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register.

Operation: operand address = AR*n*
AR*n* = AR*n* + IR*m*

Assembler Syntax: *AR*n*++(IR*m*)

Modification Field: 01100 if *m* = 0
10100 if *m* = 1



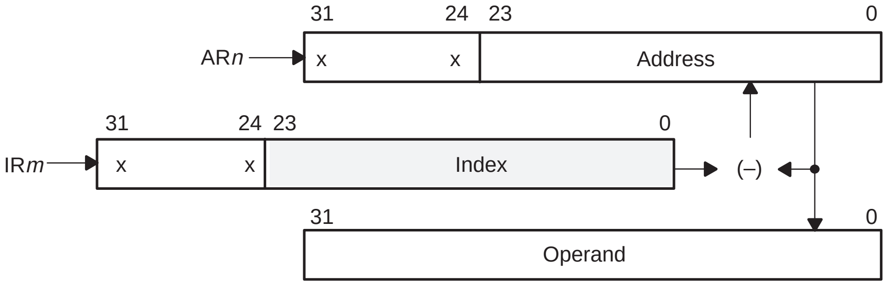
Example 6–16. Indirect Addressing With Postindex Subtract and Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the index register (IR0 or IR1) is subtracted from the auxiliary register.

Operation: operand address = AR*n*
AR*n* = AR*n* – IR*m*

Assembler Syntax: *AR*n*––(IR*m*)

Modification Field: 01101 if *m* = 0
10101 if *m* = 1



Example 6–17. Indirect Addressing With Postindex Add and Circular Modify

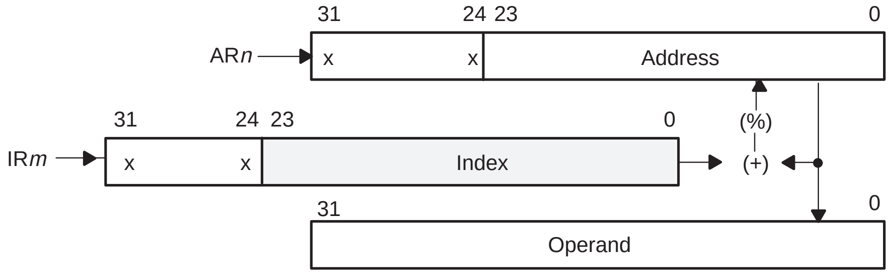
The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register. This value is evaluated using circular addressing and replaces the contents of the auxiliary register.

Operation: operand address = AR*n*
AR*n* = circ(AR*n* + IR*m*)

Assembler Syntax: *AR*n*++(IR*m*)%

Modification Field:

	01110	if *m* = 0
	10110	if *m* = 1



Example 6–18. Indirect Addressing With Postindex Subtract and Circular Modify

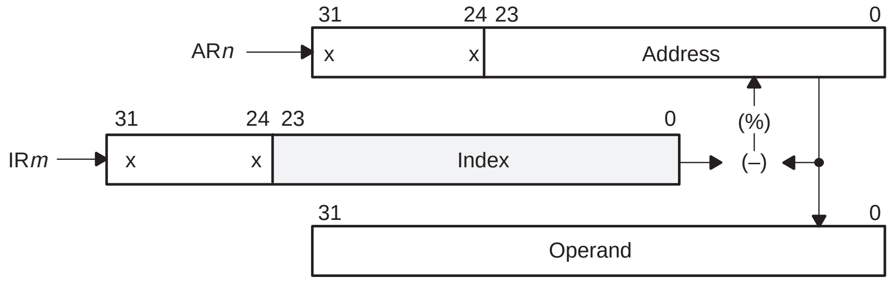
The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the index register (IR0 or IR1) is subtracted from the auxiliary register. This result is evaluated using circular addressing and replaces the contents of the auxiliary register.

Operation: operand address = AR*n*
AR*n* = circ(AR*n* – IR*m*)

Assembler Syntax: *AR*n*––(IR*m*)%

Modification Field:

	01111	if *m* = 0
	10111	if *m* = 1



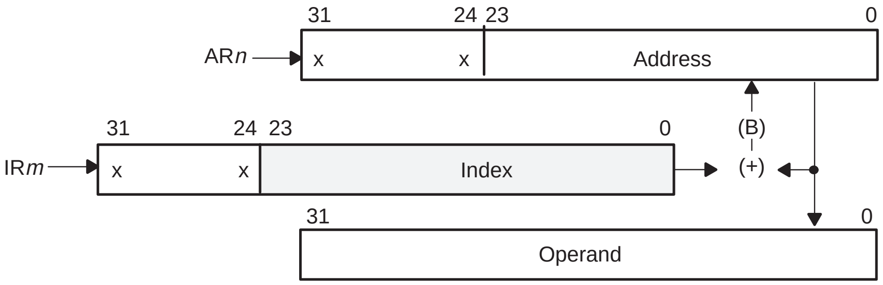
Example 6–19. Indirect Addressing With Postindex Add and Bit-Reversed Modify

The address of the operand to fetch is the contents of an auxiliary register (AR*n*). After the operand is fetched, the index register (IR0) is added to the auxiliary register. This addition is performed with a reverse-carry propagation and can be used to yield a bit-reversed (B) address. This value replaces the contents of the auxiliary register.

Operation: operand address = AR*n*
AR*n* = B(AR*n* + IR0)

Assembler Syntax: *AR*n*++(IR0)B

Modification Field: 11001



6.5 Immediate Addressing

In immediate addressing, the operand is a 16-bit (short) or 24-bit (long) immediate value contained in the 16 or 24 LSBs of the instruction word (expr). Depending on the data types assumed for the instruction, the short-immediate operand can be a 2s-complement integer, an unsigned integer, or a floating-point number. This is the syntax for this mode:

Syntax: **expr**

Example 6–20 illustrates an instruction example with data before and after the instruction is executed.

Example 6–20. Short-Immediate Addressing

```
SUBI 1,R0
```

	Before Instruction		After Instruction
R0	00 0000 0000	R0	00 FFFF FFFF

In long-immediate addressing, the operand is a 24-bit unsigned immediate value contained in the 24 LSBs of the instruction word (expr). This is the syntax for this mode:

Syntax: **expr**

Example 6–21 illustrates an instruction example with data from before and after the instruction is executed.

Example 6–21. Long-Immediate Addressing

```
BR 8000h
```

	Before Instruction		After Instruction
PC	0000	PC	8000

6.6 PC-Relative Addressing

Program counter (PC)-relative addressing is used for branching. It adds the contents of the 16 or 24 LSBs of the instruction word to the PC register. The assembler takes the *src* (a label or address) specified by the user and generates a displacement. If the branch is a standard branch, this displacement is equal to [label – (instruction address +1)]. If the branch is a delayed branch, this displacement is equal to [label – (instruction address+3)].

The displacement is stored as a 16-bit or 24-bit signed integer in the LSBs of the instruction word. The displacement is added to the PC during the pipeline decode phase. Notice that because the PC is incremented by 1 in the fetch phase, the displacement is added to this incremented PC value.

Syntax: **expr** (*src*)

Example 6–22 illustrates an example with data from before and after the instruction is executed.

Example 6–22. PC-Relative Addressing

```
BU     NEWPC  ; pc=1001h, NEWPC label = 1005h, displacement = 3
```

Before Instruction decode phase:

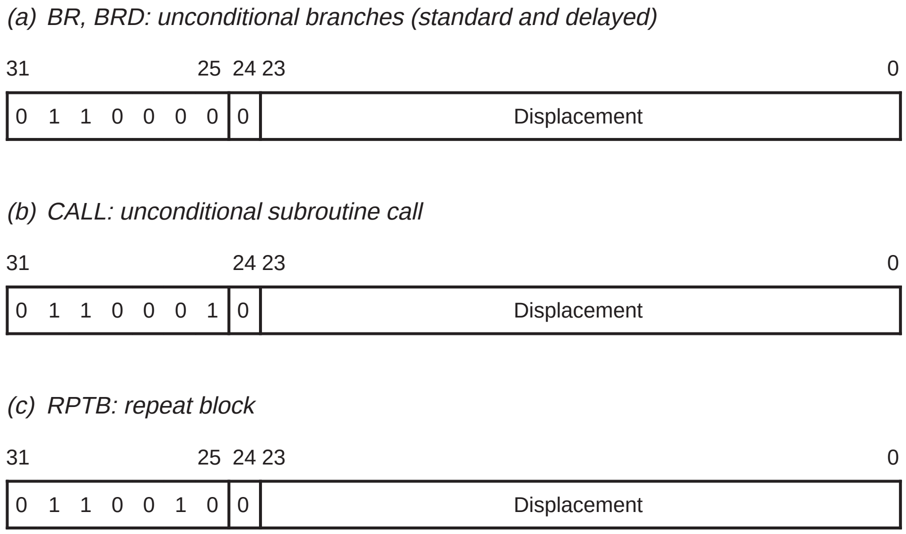
PC 1002

After Instruction execution phase:

PC 1005

The 24-bit addressing mode encodes the program-control instructions (for example, BR, BRD, CALL, RPTB, and RPTBD). Depending on the instruction, the new PC value is derived by adding a 24-bit signed value in the instruction word with the present PC value. Bit 24 determines the type of branch (D = 0 for a standard branch or D = 1 for a delayed branch). Some of the instructions are encoded in Figure 6–3.

Figure 6–3. Encoding for 24-Bit PC-Relative Addressing Mode



(a) BR, BRD: unconditional branches (standard and delayed)

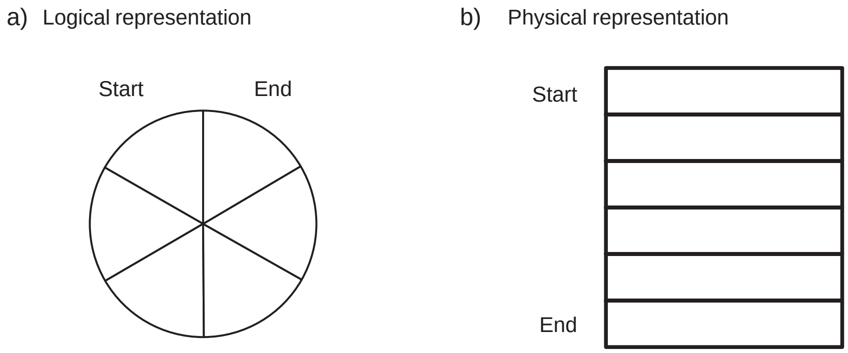
(b) CALL: unconditional subroutine call

(c) RPTB: repeat block

6.7 Circular Addressing

Many DSP algorithms, such as convolution and correlation, require a circular buffer in memory. In convolution and correlation, the circular buffer acts as a sliding window that contains the most recent data to process. As new data is brought in, the new data overwrites the oldest data by increasing the pointer to the data through the buffer in counter-clockwise fashion. When the pointer accesses the end of the buffer, the device sets the pointer to the beginning of the buffer. For example, Figure 6–4a shows a circular buffer that holds six values. Figure 6–4b shows how this buffer is implemented in the ’C3x memory space. Figure 6–5 shows this buffer after writing three values. Figure 6–6 shows this buffer after writing eight values.

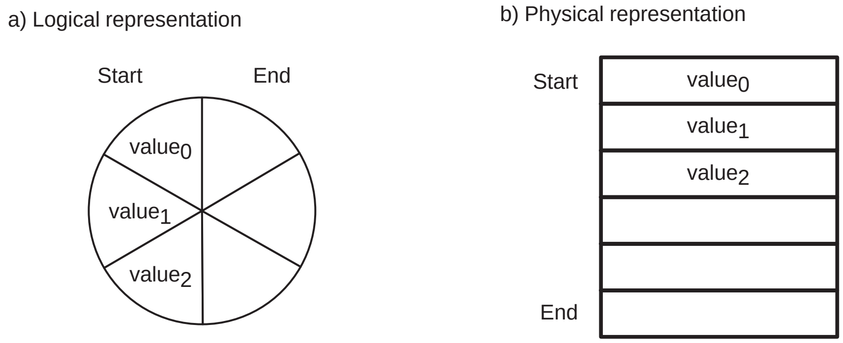
Figure 6–4. Logical and Physical Representation of Circular Buffer



a) Logical representation

b) Physical representation

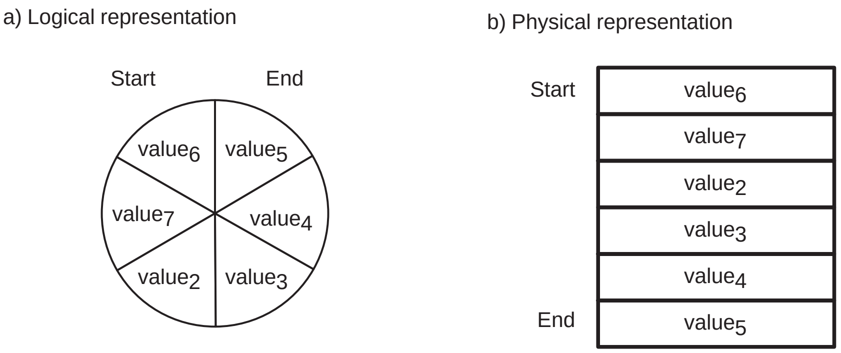
Figure 6–5. Logical and Physical Representation of Circular Buffer after Writing Three Values



a) Logical representation

b) Physical representation

Figure 6–6. Logical and Physical Representation of Circular Buffer after Writing Eight Values


a) Logical representation

b) Physical representation

To implement a circular buffer in the ’C3x, the following criteria must be satisfied (more than one circular buffer can be implemented on the ’C3x as long as the size of the buffers are identical):

- ❏ Specify the size of the circular buffer (R) by storing the length of the buffer in the block-size register (BK). The size of the buffer must be less than or equal to 64K (16 bits).
- ❏ Align the start of the buffer to a K-bit boundary (that is, the K LSBs of the starting address of the circular buffer must be 0) by satisfying the following formula:

  $2^K > R$

  where:

  R = length of circular buffer
  K = number of 0s in the LSBs of the circular buffer starting address

Example 6–23. Examples of Formula $2^K > R$

Length of Buffer	BK Register Value	Starting Address of Buffer
31	31	$\text{XXXXXXXXXXXXXXXXXXX00000}_2$
32	32	$\text{XXXXXXXXXXXXXXXXXX000000}_2$
1024	1024	$\text{XXXXXXXXXXXXX00000000000}_2$

In circular addressing, *index* refers to the K LSBs (from the K-bit boundary criteria) of the auxiliary register selected, and *step* is the quantity being added to or subtracted from the auxiliary register. Follow these two rules when you use circular addressing:

- ❑ The step used must be less than or equal to the block size. The step size is treated as an unsigned integer. If an index register (IR) is used as a step increment or decrement, it is also treated as an unsigned integer.
- ❑ The first time the circular queue is addressed, the auxiliary register must be pointing to an element in the circular queue.

The algorithm for circular addressing is as follows:

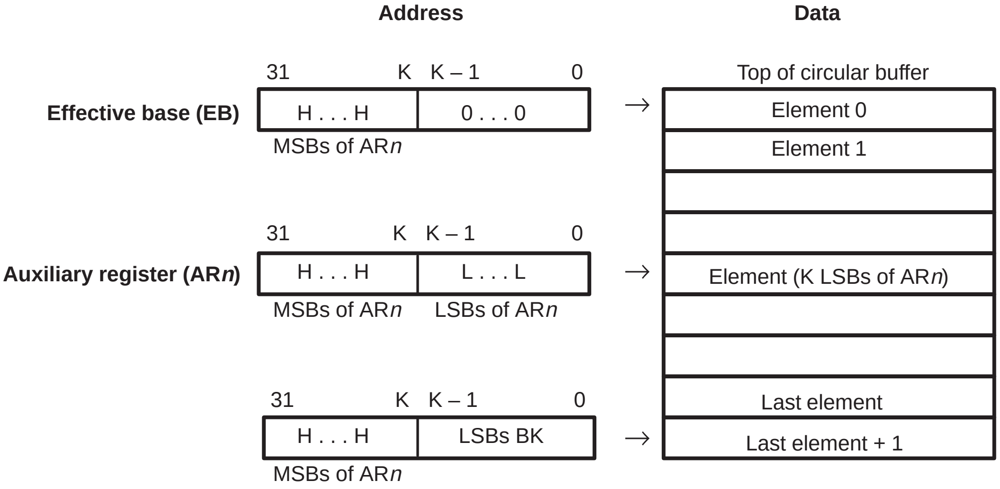
If 0 ≤ index + step < BK: index = index + step.

Else if index + step ≥ BK: index = index + step – BK.

Else if index + step < 0: index = index + step + BK.

Figure 6–7 shows how the circular buffer is implemented and illustrates the relationship of the quantities generated and the elements in the circular buffer.

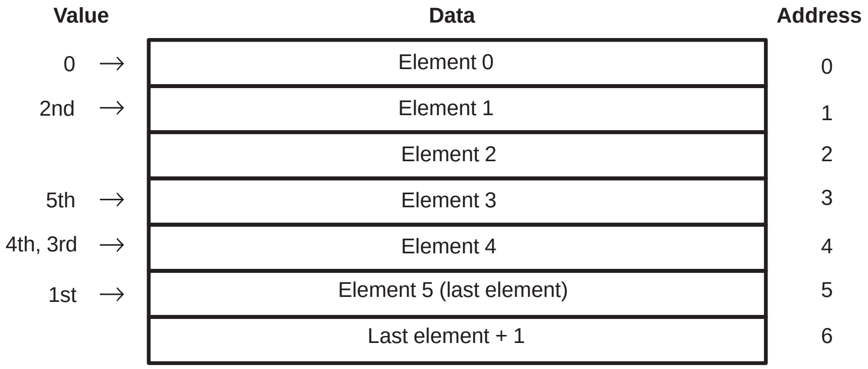
Figure 6–7. Circular Buffer Implementation



Example 6–24 shows circular addressing operation. Assuming that all ARs are four bits, let AR0 = 0000 and BK = 0110 (block size of 6). Example 6–24 shows a sequence of modifications and the resulting value of AR0. Example 6–24 also shows how the pointer steps through the circular queue with a variety of step sizes (both incrementing and decrementing).

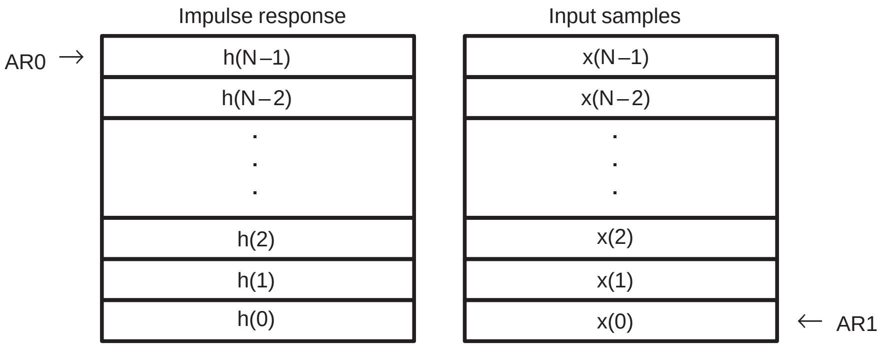
Example 6–24. Circular Addressing

```
*AR0++(5)%      ;  AR0  =  0   (0 value)
*AR0++(2)%      ;  AR0  =  5   (1st value)
*AR0– –(3)%     ;  AR0  =  1   (2nd value)
*AR0++(6)%      ;  AR0  =  4   (3rd value)
*AR0– –%        ;  AR0  =  4   (4th value)
*AR0            ;  AR0  =  3   (5th value)
```



Circular addressing is especially useful for the implementation of FIR filters. Figure 6–8 shows one possible data structure for FIR filters. Note that the initial value of AR0 points to h(N–1), and the initial value of AR1 points to x(0). Circular addressing is used in the ’C3x code for the FIR filter shown in Example 6–25.

Figure 6–8. Data Structure for FIR Filters



Example 6–25. FIR Filter Code Using Circular Addressing

```
*  Impulse Response
       .sect  ”Impulse_Resp”
H      .float 1.0
       .float 0.99
       .float 0.95
       .
       .
       .
       .float 0.1

*  Input Buffer
X      .usect ”Input_Buf”,128

       .data
HADDR  .word  H
XADDR  .word  X
N      .word  128

*  Initialization
*
       LDP    HADDR
       LDI    @N,BK            ; Load block size.
       LDI    @HADDR,AR0       ; Load pointer to impulse re–
                               ; sponse.
       LDI    @XADDR,AR1       ; Load pointer to bottom of
                               ; input sample buffer.
*
TOP    LDF    IN,R3            ;Read input sample.
       STF    R3,*AR1++%       ;Store with other samples,
                               ;and point to top of buffer.
       LDF    0,R0             ;Initialize R0.
       LDF    0,R2             ;Initialize R2.
*
*      Filter
*
       RPTS   N – 1            ;Repeat next instruction.
       MPYF3  *AR0++%,*AR1++%,R0
||     ADDF3  R0,R2,R2         ;Multiply and accumulate.
       ADDF   R0,R2            ;Last product accumulated.
*
       STF    R2,Y             ;Save result.
       B      TOP              ;Repeat.
```

6.8 Bit-Reversed Addressing

The ’C3x can implement fast Fourier transforms (FFT) with bit-reversed addressing. Whenever data in increasing sequence order is transformed by an FFT, the resulting data is presented in bit-reversed order. To recover this data in the correct order, certain memory locations must be swapped. By using the bit-reversed addressing mode, swapping data is unnecessary. The data is accessed by the CPU in bit-reversed order rather than sequentially. For correct bit-reversed access, the base address of bit-reversed access, the base address must be located on a boundary given by the size of the FFT table. Similar to circular addressing, the base address of bit-reversed addressing must follow this criteria.

❑ Base address must be aligned to a K-bit boundary (that is, the K LSBs of the starting address of the buffer/table must be 0) as follows:

  $2^K > R$

  where:

  $R =$ length of table/buffer
  $K =$ number of 0s in the LSBs of the buffer/table starting address

❑ Size of the buffer/table must be less than or equal to 64K (16 bits)

The CPU bit-reversed operation can be illustrated by assuming an FFT table of size 2^n. When real and imaginary data are stored in separate arrays, the n LSBs of the base address must be 0, and IR0 must be equal to $2^{n–1}$ (half of the FFT size). When real and imaginary data are stored in consecutive memory locations (Real_0, Imaginary_0, Real_1, Imaginary_1, Real_2, Imaginary_2, etc.), the n+1 LSBs of the base address must be 0, and IR0 must be equal to 2^n (FFT size).

For CPU bit-reversed addressing, one auxiliary register points to the physical location of data. Adding IR0 (in bit-reversed addressing) to this auxiliary register performs a reverse-carry propagation. IR0 is treated as an unsigned integer.

To illustrate bit-reversed addressing, assume 8-bit auxiliary registers. Let AR2 contain the value 0110 0000 (96). This is the base address of the data in memory assuming a 16-entry table. Let IR0 contain the value 0000 1000 (8). Example 6–26 shows a sequence of modifications of AR2 and the resulting values of AR2.

Example 6–26. Bit-Reversed Addressing

```
*AR2++(IR0)B ;  AR2=  0110 0000 (0th value)
*AR2++(IR0)B ;  AR2=  0110 1000 (1st value)
*AR2++(IR0)B ;  AR2=  0110 0100 (2nd value)
*AR2++(IR0)B ;  AR2=  0110 1100 (3rd value)
*AR2++(IR0)B ;  AR2=  0110 0010 (4th value)
*AR2++(IR0)B ;  AR2=  0110 1010 (5th value)
*AR2++(IR0)B ;  AR2=  0110 0110 (6th value)
*AR2         ;  AR2=  0110 1110 (7th value)
```

Table 6–3 shows the relationship of the index steps and the four LSBs of AR2. You can find the four LSBs by reversing the bit pattern of the steps.

Table 6–3. Index Steps and Bit-Reversed Addressing

Step	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Step
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

6.9 Aligning Buffers With the TMS320 Floating-Point DSP Assembly Language Tools

To align buffers to a K-bit boundary, you can use the .sect or .usect assembly directives to define a section in conjunction with the align memory allocation parameter of the sections directive of the linker command file. For the FIR filter of Example 6–25 with a length of 32, the linker command file is:

```
MEMORY
{
   RAM origin = 0h, length = 1000h
}
SECTIONS
{
      .text: > RAM
      Impulse_Resp ALIGN(64): > RAM
      Input_Buf    ALIGN(64): > RAM
}
```

6.10 System and User Stack Management

The ’C3x provides a dedicated system-stack pointer (SP) for building stacks in memory. The auxiliary registers can also be used to build a variety of more general linear lists. This section discusses the implementation of the following types of linear lists:

❑ **Stack**

The stack is a linear list for which all insertions and deletions are made at one end of the list.

❑ **Queue**

The queue is 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**

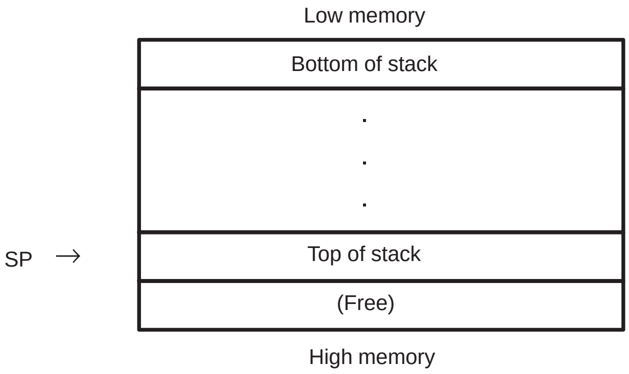
The dequeue is a double-ended queue linear list for which insertions and deletions are made at either end of the list.

6.10.1 System-Stack Pointer

The system-stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The system stack fills from low-memory address to high-memory address (see Figure 6–9). 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 system-stack pointer.

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 system stack can be pushed and popped using the PUSH, POP, PUSHF, and POPF instructions.

Figure 6–9. System Stack Configuration



6.10.2 Stacks

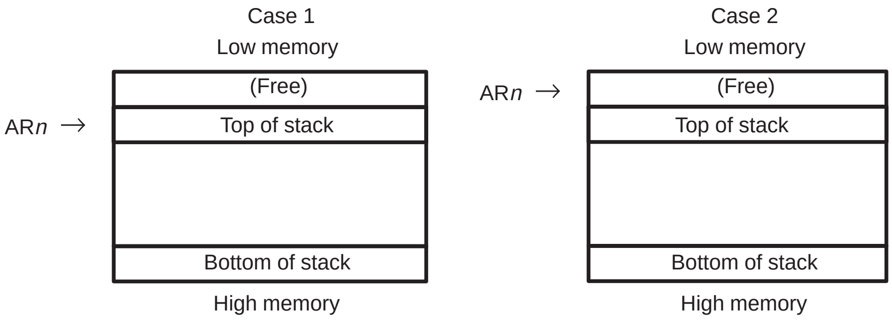
Stacks can be built from low to high memory or high to low memory. Two cases for each type of stack are shown. Stacks can be built using the preincrement/ decrement and postincrement/decrement modes of modifying the auxiliary registers (AR). Stack growth from high-to-low memory can be implemented in two ways:

CASE 1: Stores to memory using *– –AR*n* to push data onto the stack and reads from memory using *AR*n*++ to pop data off the stack.

CASE 2: Stores to memory using *AR*n*– –to push data onto the stack and reads from memory using * ++AR*n* to pop data off the stack.

Figure 6–10 illustrates these two cases. The only difference is that in case 1, the AR always points to the top of the stack, and in case 2, the AR always points to the next free location on the stack.

Figure 6–10. Implementations of High-to-Low Memory Stacks



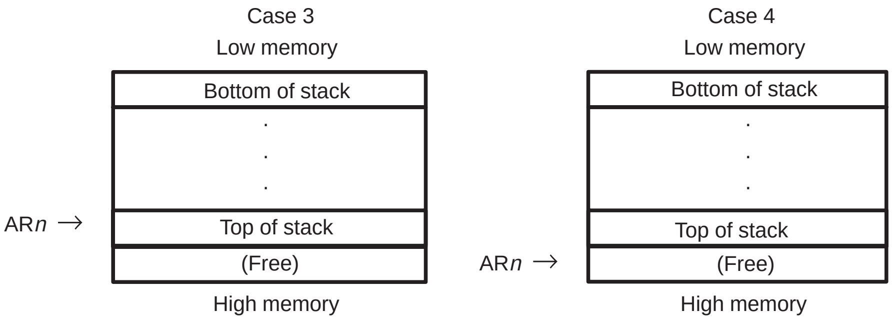
Stack growth from low-to-high memory can be implemented in two ways:

CASE 3: Stores to memory using *++AR*n* to push data onto the stack and reads from memory using *AR*n*– –to pop data off the stack.

CASE 4: Stores to memory using *AR*n*++ to push data onto the stack and reads from memory using *– –AR*n* to pop data off the stack.

Figure 6–11 shows these two cases. In case 3, the AR always points to the top of the stack. In case 4, the AR always points to the next free location on the stack.

Figure 6–11. Implementations of Low-to-High Memory Stacks



6.10.3 Queues

A queue is like a FIFO. The implementation of queues is based on the manipulation of auxiliary registers. Two auxiliary registers are used: one to mark the front of the queue from which data is popped (or dequeued) and the other to mark the rear of the queue where data is pushed. With proper management of the auxiliary registers, the queue can also be circular. (A queue is circular when the rear pointer is allowed to point to the beginning of the queue memory after it has pointed to the end of the queue memory.)

Chapter 7

Program Flow Control

The TMS320C3x provides a complete set of constructs that facilitate software and hardware control of the program flow. Software control includes repeats, branches, calls, traps, and returns. Hardware control includes reset operation, interrupts, and power management. You can select the constructs best suited for your particular application.

Topic **Page**

7.1 Repeat Modes

The repeat modes of the ’C3x can implement zero-overhead looping. For many algorithms, most execution time is spent in an inner kernel of code. Using the repeat modes allows these time-critical sections of code to be executed in the shortest possible time.

The ’C3x provides two instructions to support zero-overhead looping:

❑ **RPTB** (repeat a block of code). RPTB repeats execution of a block of code a specified number of times.

❑ **RPTS** (repeat a single instruction). RPTS fetches a single instruction once and then repeats its execution a number of times. Since the instruction is fetched only once, bus traffic is minimized.

RPTB and RPTS are 4-cycle instructions. These four cycles of overhead occur during the initial execution of the loop. All subsequent executions of the loop have no overhead (0 cycle).

Three registers (RS, RE, and RC) control the updating of the program-counter (PC) when it is modified in a repeat mode. Table 7–1 describes these registers.

Table 7–1. Repeat-Mode Registers

Register	Function
RS	Repeat start-address register. Holds the address of the first instruction of the block of code to be repeated.
RE	Repeat end-address register. Holds the address of the last instruction of the block of code to be repeated. RE ≥ RS (see subsection 7.1.2).
RC	Repeat-counter register. Contains 1 less than the number of times the block remains to be repeated. For example, to execute a block *n* times, load *n* – 1 into RC.

Correct operation of the repeat modes requires that all of the above registers must be initialized correctly. RPTB and RPTS perform this initialization in slightly different ways.

7.1.1 Repeat-Mode Control Bits

Two bits are important to the operation of RPTB and RPTS:

- ❑ **RM bit.** The repeat-mode (RM) flag bit in the status register specifies whether the processor is running in the repeat mode.
  - ■ RM = 0: Fetches are not made in repeat mode.
  - ■ RM = 1: Fetches are made in repeat mode.
- ❑ **S bit**. The S bit is internal to the processor and cannot be programmed, but this bit is necessary to fully describe the operation of RPTB and RPTS.
  - ■ If RM = 1 and S = 0, RPTB is executing. Program fetches occur from memory.
  - ■ If RM = 1 and S = 1, RPTS is executing. After the first fetch from memory, program fetches occur from the instruction register.

7.1.2 Repeat-Mode Operation

Information in the repeat-mode registers and associated control bits controls the modification of the PC during repeat-mode fetches. The repeat modes compare the contents of the RE register (repeat-end-address register) with the PC after the execution of each instruction. If they match and the repeat counter (RC) is nonnegative, the RC is decremented; the PC is loaded with the repeat-start-address, and the processing continues. The fetches and appropriate status bits are modified as necessary. Note that the RC is never modified when the RM flag is 0.

The repeat counter should be loaded with a value 1 less than the number of times to execute the block; for example, an RC value of 4 executes the block five times. The detailed algorithm for the update of the PC is shown in Example 7–1.

Note: Maximum Number of Repeats

1) The maximum number of repeats occurs when RC = 8000 0000h. This results in 8000 0001h repetitions. The minimum number of repeats occurs when RC = 0. This results in one repetition.

2) RE must be greater than or equal to RS (RE ≥ RS). Otherwise, the code does not repeat even though the RM bit remains set to 1.

3) By writing a 0 into the repeat counter or writing 0 into the RM bit of the status register, you can stop the repeating of the loop before completion.

Example 7–1. Repeat-Mode Control Algorithm

```
   if RM == 1                               ; If in repeat mode (RPTB or RPTS)
   if S == 1                                ; If RPTS
   if first time through                    ; If this is the first fetch
          fetch instruction from memory     ; Fetch instruction from memory
       else                                 ; If not the first fetch
          fetch instruction from IR         ; Fetch instruction from IR
RC – 1 → RC                                 ; Decrement RC
       if RC < 0                            ; If RC is negative
                                            ; Repeat single mode completed
          0 → ST(RM)                        ; Turn off repeat-mode bit
          0 → S                             ; Clear S
          PC + 1 → PC                       ; Increment PC
       else if S == 0                       ; If RPTB
          fetch instruction from memory     ; Fetch instruction from memory
       if PC == RE                          ; If this is the end of the block
          RC – 1 → RC                       ; Decrement RC
       if RC ≥ 0                            ; If RC is not negative
       RS → PC                              ; Set PC to start of block
   else if RC < 0                           ; If RC is negative
       0 → ST(RM)                           ; Turn off repeat mode bits
       0 → S                                ; Clear S
       PC + 1 → PC                          ; Increment PC
```

7.1.3 RPTB Instruction

The RPTB instruction repeats a block of code a specified number of times.

The number of times to repeat the block is the RC (repeat *count*) register value plus 1. Because the execution of RPTB does not load the RC, you must load this register yourself. The RC register must be loaded before the RPTB instruction is executed. A typical setup of the block repeat operation is shown in Example 7–2.

Example 7–2. RPTB Operation

```
      LDI 15,RC       ; Load repeat counter with 15
      RPTB   ENDLOOP  ; Execute the block of code
STLOOP                ; from STLOOP to ENDLOOP 16
                      ; times
   .
   .
   .
ENDLOOP
```

All block repeats initiated by RPTB can be interrupted. When RPTB *src* (source) instruction executes, it performs the following sequence:

1) Load the start address of the block into repeat-start-address register (RS). This is the next address following the instruction:

   RS ← PC (program-counter) of RPTB + 1

2) Load the end address of the block into repeat-end-address register (RE).

   ❑ In PC-relative mode, the end address is the 24-bit *src* operand plus RS:

   RE ← *src* + PC of RPTB + 1

   ❑ In register mode, the end address is the contents of the *src* register:

   RE ← *src* register

3) Set the status register to indicate the repeat-mode operation.

   RM ← 1

4) Indicate repeat-mode operation by clearing the S bit.

   S ← 0

Note:

You can stop the loop from repeating before its completion by writing a 0 to the repeat counter or writing a 0 to the RM bit of the status register.

7.1.4 RPTS Instruction

An RPTS *src* instruction repeats the instruction following the RPTS *(src* + 1) times. Repeats of a single instruction initiated by RPTS are not interruptible, because the RPTS fetches the instruction word only once and then keeps it in the instruction register for reuse. An interrupt in this situation would cause the instruction word to be lost. Refetching the instruction word from the instruction register reduces memory accesses and, in effect, acts as a one-word program cache. If you need a single instruction that is repeatable and interruptible, you can use the RPTB instruction.

When RPTS *src* is executed, the following sequence of operations occurs:

1) PC + 1 → RS
2) PC + 1 → RE
3) 1 → RM status register bit
4) 1 → S bit
5) *src* → RC (repeat *count* register)

The RPTS instruction loads all registers and mode bits necessary for the operation of the single-instruction repeat mode. Step 1 loads the start address of the block into RS. Step 2 loads the end address into the RE (end address of the block). Since this is a repeat of a single instruction, the start address and the end address are the same. Step 3 sets the status register to indicate the repeat mode of operation. Step 4 indicates that this is the repeat single-instruction mode of operation. Step 5 loads *src* into RC.

7.1.5 Repeat-Mode Restrictions

Because the block-repeat modes modify the program counter, no other instruction can modify the program counter at the same time. Two rules apply:

Rule 1: The last instruction in the block (or the only instruction in a block of size 1) cannot be a B*cond*, BR, DB*cond*, CALL, CALL*cond*, TRAP*cond*, RETI*cond*, RETS*cond*, IDLE, RPTB, or RPTS. Example 7–3 shows an incorrectly placed standard branch.

Rule 2: None of the last four instructions from the bottom of the block (or the only instruction in a block of size 1) can be a B*cond*D, BRD, or DB*cond*D. Example 7–4 shows an incorrectly placed delayed branch.

If either of these rules is violated, the PC is undefined.

Example 7–3. Incorrectly Placed Standard Branch

```
         LDI    15,RC     ; Load repeat counter with 15
         RPTB   ENDLOOP   ; Execute the block of code
STLOOP                    ; from STLOOP to ENDLOOP 16
                          ; times
      .
      .
      .
ENDLOOP  BR     OOPS      ; This branch violates rule 1
```

Example 7–4. Incorrectly Placed Delayed Branch

```
         LDI    15,RC     ; Load repeat counter with 15
         RPTB   ENDLOOP   ; Execute block of code
STLOOP                    ; from STLOOP to ENDLOOP 16
                          ; times
         .
         .
         .
         BRD    OOPS      ; This branch violates rule 2
         ADDF
         MPYF
ENDLOOP  SUBF
```

7.1.6 RC Register Value After Repeat Mode Completes

For the RPTB instruction, the RC register normally decrements to 0000 0000h unless the block size is 1; in which case, it decrements to FFFF FFFFh. However, if the RPTB instruction using a block size of 1 has a pipeline conflict in the instruction being executed, the RC register decrements to 0000 0000h. Example 7–5 illustrates a pipeline conflict. See Chapter 8 for pipeline information.

RPTS normally decrements the RC register to FFFF FFFFh. However, if the RPTS has a pipeline conflict on the last cycle, the RC register decrements to 0000 0000h.

Note: Number of Repetitions

In any case, the number of repetitions is always RC + 1.

Example 7–5. Pipeline Conflict in an RPTB Instruction

```
EDC      .word 40000000h    ; The program is located in 4000000Fh
         LDP      EDC
         LDI      @EDC,AR0
         LDI      15,RC     ; Load repeat counter with 15
         RPTB     ENDLOOP   ; Execute block of code
ENDLOOP  LDI      *AR0,R0   ; The *AR0 read conflicts with
                            ; the instruction fetching
                            ; Then RC decrements to 0
                            ; If cache is enabled, RC decrements
                            ; to FFFF FFFFh
```

7.1.7 Nested Block Repeats

Block repeats (RPTB) can be nested. Since the registers RS, RE, RC, and ST control the repeat-mode status, these registers must be saved and restored in order to nest block repeats. For example, if you write an interrupt service routine that requires the use of RPTB, it is possible that the interrupt associated with the routine may occur during another block repeat. The interrupt service routine can check the RM bit to determine whether the block repeat mode is active. If this RM is set, the interrupt routine must save ST, RS, RE, and RC, in that order. The interrupt routine can then perform a block repeat. Before returning to the interrupted routine, the interrupt routine must restore RC, RE, RS, and ST, in that order. If the RM bit is not set, you do not need to save and restore these registers.

Note: Saving/Restoring Registers in Correct Order

The order in which the registers are saved/restored is important to guarantee correct operation. The ST register must be restored last, after the RC, RE, and RS registers. ST must be restored after restoring RC, because the RM bit cannot be set to 1 if the RC register is 0 or –1. For this reason, if you execute a POP ST instruction (with ST (RM bit) = 1) while RC = 0, the POP instruction recovers all the ST register bits but not the RM bit that stays at 0 (repeat mode disabled). Also, RS and RE must be correctly set before you activate the repeat mode.

The RPTS instruction can be used in a block repeat loop if the proper registers are saved.

Because the program counter is modified at the end of the loop according to the contents of registers RS, RE, and RC, no operation should attempt to modify the repeat counter or the program-counter to a different value at the end of the loop. It takes four cycles in the pipeline to save and restore these registers. Hence, sometimes, it may be more economical to implement a nested loop by the more traditional method of using a register as a counter and then using a delayed branch or a decrement and branch-delayed instructions, rather than using nested repeat blocks. Often implementing the outer loop as a counter and the inner loop as RPTB instruction produces the fastest execution.

7.2 Delayed Branches

The ’C3x offers three main types of branching: standard, delayed, and conditional delayed.

Standard branches empty the pipeline before performing the branch, ensuring correct management of the program counter and resulting in a ’C3x branch taking four cycles. Included in this class are repeats, calls, returns, and traps.

Delayed branches on the ’C3x do not empty the pipeline, but rather execute the next three instructions before the program counter is modified by the branch. This results in a branch that requires only a single cycle, making the speed of the delayed branch very close to that of the optimal block repeat modes of the ’C3x. However, unlike block-repeat modes, delayed branches may be used in situations other than looping. Every delayed branch has a standard branch counterpart that is used when a delayed branch cannot be used. The delayed branches of the ’C3x are B*cond* D, BRD, and DB*cond* D.

Conditional delayed branches use the conditions that exist at the end of the instruction immediately preceding the delayed branch. They do not depend on the instructions following the delayed branch. The condition flags are set by a previous instruction only when the destination register is one of the extended-precision registers (R0–R7) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed. Delayed branches guarantee that the next three instructions will execute, regardless of other pipeline conflicts.

When a delayed branch is fetched, it remains pending until the three subsequent instructions are executed. The following instructions **cannot** be used in the three instructions after a delayed branch (see Example 7–6):

B*cond*
B*cond*D
BR
BRD
CALL
CALL*cond*
DB*cond*
DB*cond*D
IDLE
IDLE2
RETI*cond*
RETS*cond*
RPTB
RPTS
TRAP*cond*

Delayed branches disable interrupts until the completion of the three instructions that follow the delayed branch regardless of whether the branch is or is not performed.

Note: Incorrect Use of Delayed Branches

If delayed branches are used incorrectly, the PC is undefined.

Example 7–6. Incorrectly Placed Delayed Branches

```
   B1:    BD     L1
          NOP
          NOP
   B2:    B      L2     ; This branch is incorrectly placed.
          NOP
          NOP
          NOP
          .
          .
          .
```

For faster execution, it might still be advantageous to use a delayed branch followed by NOP instructions by trading increased program size for faster speed. This is shown in Example 7–7 where a NOP takes the place of the third unused instruction after the delayed branch.

Example 7–7. 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
```

7.3 Calls, Traps, and Returns

Calls and traps provide a means of executing a subroutine or function while providing a return to the calling routine.

The CALL, CALL*cond,* and TRAP*cond* instructions store the value of the PC on the stack before changing the PC's contents. The RETS*cond* or RETI*cond* instructions use the value on the stack to return execution from traps and calls. CALL is a 4-cycle instruction, while CALL*cond* and TRAP*cond* are 5-cycle instruction.

❑ The **CALL** instruction places the next PC value on the stack and places the *src* (source) operand into the PC. The *src* is a 24-bit immediate value. Figure 7–1 shows CALL response timing.

❑ The **CALL*cond*** instruction is similar to the CALL instruction except for two differences:

  - It executes only if a specific condition is true (the 20 conditions—including unconditional—are listed in Table 13–12 on page 13-30).
  - The *src* is either a PC-relative displacement or is in register-addressing mode.

  The condition flags are set by a previous instruction only when the destination register is one of the extended-precision registers (R0–R7) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

❑ The **TRAP*cond*** instruction also executes only if a specific condition is true (same conditions as for the CALL*cond* instruction). When executing, the following actions occur:

  1) Interrupts are disabled with 0 written to bit GIE of the ST.
  2) The next PC value is stored on the stack.
  3) The specified vector is retrieved from the trap-vector table and is loaded into the PC. The vector address corresponds to the trap number in the instruction.

  Using the RETI*cond* to return reenables interrupts by setting the bit field of the status register.

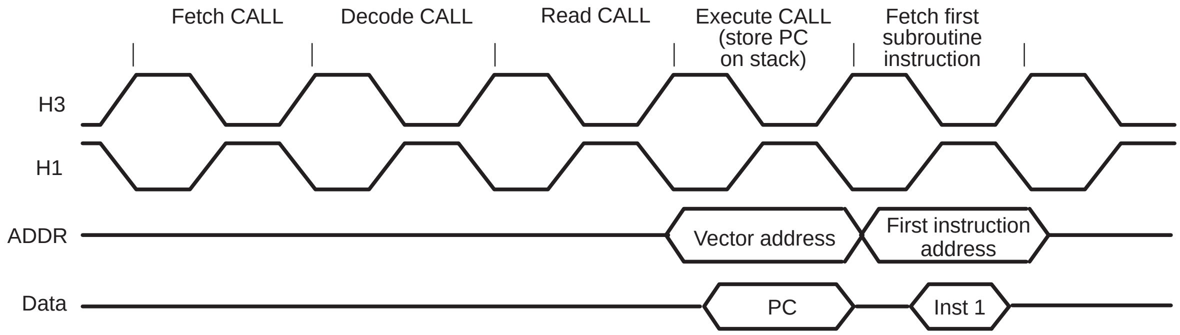
❑ **RETS*cond*** returns execution from any of the above three instructions by popping the top of the stack to the PC. For RETS*cond* to execute, the specified condition must be true. The conditions are the same as for the CALL*cond* instruction.

- **RETI*cond*** returns from traps or calls like the RETS*cond*, with the addition that RETI*cond* also sets the GIE bit of the status register, which enables all interrupts whose enabling bit is set to 1. The conditions for RETI*cond* are the same as for the CALL*cond* instruction.

Functionally, calls and traps accomplish the same task — a subfunction is called and executed, and control is then returned to the calling function. Traps offer two advantages over calls:

- Interrupts are automatically disabled when a trap is executed. This allows critical code to execute without risk of being interrupted. Traps are generally terminated with a RETI*cond* instruction to reenable interrupts.
- You can use traps to indirectly call functions. This is particularly beneficial when a kernel of code contains the basic subfunctions to be used by applications. In this case, you can modify the functions in the kernel and relocate them without recompiling each application.

Figure 7–1. CALL Response Timing


Fetch CALL
Decode CALL
Read CALL
Execute CALL (store PC on stack)
Fetch first subroutine instruction
H3
H1
ADDR
Data
Vector address
First instruction address
PC
Inst 1

7.4 Interlocked Operations

One of the most common parallel processing configurations is the sharing of global memory by multiple processors. For multiple processors to access this global memory and share data in a coherent manner, some sort of arbitration or handshaking is necessary. This requirement for arbitration is the purpose of the 'C3x interlocked operations.

The 'C3x provides a flexible means of multiprocessor support with five instructions, referred to as interlocked operations. Through the use of external signals, these instructions provide powerful synchronization mechanisms. They also guarantee the integrity of the communication and result in a high-speed operation. The interlocked-operation instruction group is listed in Table 7–2.

Table 7–2. Interlocked Operations

Mnemonic	Description	Operation
LDFI	Load floating-point value into a register, interlocked	Signal interlocked *src* → *dst*
LDII	Load integer into a register, interlocked	Signal interlocked *src* → *dst*
SIGI	Signal, interlocked	Signal interlocked Clear interlock
STFI	Store floating-point value to memory, interlocked	Clear interlock *src* → *dst*
STII	Store integer to memory, interlocked	Clear interlock *src* → *dst*

The interlocked operations use the two external flag pins, XF0 and XF1. XF0 must be configured as an output pin; XF1 is an input pin. When configured in this manner:

- XF0 signals an interlock operation request
- XF1 acts as an acknowledge signal for the requested interlocked operation. In this mode, XF0 and XF1 are treated as active-low signals.

The external timing for the interlocked loads and stores is the same as for standard loads and stores. The interlocked loads and stores may be extended like standard accesses by using the appropriate ready signal ($\overline{\text{RDY}}_{\text{int}}$ or $\overline{\text{XRDY}}_{\text{int}}$). ($\overline{\text{RDY}}_{\text{int}}$ and $\overline{\text{XRDY}}_{\text{int}}$ are a combination of external ready input and software wait states. see Chapter 9, *External Memory Interface,* for more information on ready generation.)

The LDFI and LDII instructions perform the following actions:

1) Simultaneously set XF0 to 0 and begin a read cycle. The timing of XF0 is similar to that of the address bus during a read cycle.

2) Execute an LDF or LDI instruction and extend the read cycle until XF1 is set to 0 and a ready ($\overline{\text{RDY}}_{\text{int}}$ or $\overline{\text{XRDY}}_{\text{int}}$) is signaled. The read cycle completes one H1/H3 cycle after the XF1 signal is detected.

3) Leave XF0 set to 0 and end the read cycle.

Note: Timing Diagrams for LDFI and LDII

The timing diagrams for LDFI and LDII shown on the data sheets depict a 0 wait state read cycle. Since the read cycle is extended for one H1/H3 cycle after XF1 signal is detected, the data sheets show the XF1 signal sampled one H1/H3 cycle before setting the XF0 signal low.

For the sequence of steps described here, the read cycle finishes one H1/H3 cycle after the XF1 signal is detected.

The read/write operation is identical to any other read/write cycle except for the special use of XF0 and XF1. The *src* operand for LDFI and LDII is always a direct or indirect memory address. XF0 is set to 0 only if the *src* is located off chip; that is, $\overline{\text{STRB}}$, $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, $\overline{\text{MSTRB}}$, or $\overline{\text{IOSTRB}}$ is active, or the *src* is one of the on-chip peripherals. If on-chip memory is accessed, then XF0 is not asserted, and the operation executes as an LDF or LDI from internal memory.

The STFI and STII instructions perform the following operations:

1) Simultaneously set XF0 to 1 and begin a write cycle. The timing of XF0 is similar to that of the address bus during a write cycle.

2) Execute an STF or STI instruction and extend the write cycle until a ready ($\overline{\text{RDY}}_{\text{int}}$ or $\overline{\text{XRDY}}_{\text{int}}$) is signaled.

As in the case for LDFI and LDII, the *dst* of STFI and STII affects XF0. If *dst* is located off chip ($\overline{\text{STRB}}$, $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, $\overline{\text{MSTRB}}$, or $\overline{\text{IOSTRB}}$ is active) or the *dst* is one of the on-chip peripherals, XF0 is set to 1. If on-chip memory is accessed, then XF0 is not asserted and the operation executes as an STF or STI to internal memory.

The SIGI instruction functions as follows:

1) Sets XF0 to 0
2) Idles until one H1/H3 cycle after XF1 is set to 0
3) Sets XF0 to 1 and ends the operation

Note: Timing Diagrams for SIGI

The timing diagrams for SIGI shown in the data sheets depict a zero wait state condition. Since the device idles until one cycle after XF1 is signaled, the data sheets show the XF1 signal sampled one H1/H3 cycle before setting the XF0 signal low.

For the sequence of steps described here, the device idles past one H1/H3 cycle after the $\overline{\text{XF1}}$ signal is detected.

7.4.1 Interrupting Interlocked Operations

While the LDFI, LDII, and SIGI instructions are waiting for XF1 to be set to 0, you can interrupt them. LDFI and LDII require a ready signal ($\overline{\text{RDY}}_{\text{int}}$ or $\overline{\text{XRDY}}_{\text{int}}$) in order to be interrupted. Because interrupts are taken on bus-cycle boundaries (see Section 7.6 on page 7-26), an interrupt may be taken any time after a valid ready. If the interrupted LDFI or LDII consists of a multicycle load, the load stops and an unknown value might be loaded into the register.

Interrupting an LDFI, LDII, or SIGI instruction allows you to implement protection mechanisms against deadlock conditions by interrupting an interlocked load that has taken too long. Upon return from the interrupt, the next instruction is executed. The STFI and STII instructions are not interruptible. Since the STFI and STII instructions complete when ready is signaled, the delay until an interrupt can occur is the same as for any other instruction.

7.4.2 Using Interlocked Operations

Note: Incorrect Use of Interlock Instructions

Do not place an STFI or STII back-to-back with an LDFI, LDII, or SIGI instruction as follows:

```
    STFI   R1,*AR1  ;
    LDFI   *AR1, R2 ; Incorrect use of interlock instructions
```

See Section 7.4.3, *Pipeline Effects of Interlocked Instructions*, on page 7-19.

Interlocked operations can be used to implement a busy-waiting loop, to manipulate a multiprocessor counter, to implement a simple semaphore mechanism, or to perform synchronization between two ’C3x devices. The following examples illustrate the usefulness of the interlocked operations instructions.

Example 7–8 shows the implementation of a busy-waiting loop. If location LOCK is the interlock for a critical section of code, and a nonzero means the lock is busy, the algorithm for a busy-waiting loop can be used as shown.

Example 7–8. Busy-Waiting Loop

```
       LDI    1,R0       ; Put 1 into R0
L1:    LDII   @LOCK,R1   ; Interlocked operation begun
                         ; Contents of LOCK → R1
       STII   R0,@LOCK   ; Put R0 (= 1) into LOCK, XF0 = 1
                         ; Interlocked operation ended
       BNZ    L1         ; Keep trying until LOCK = 0
```

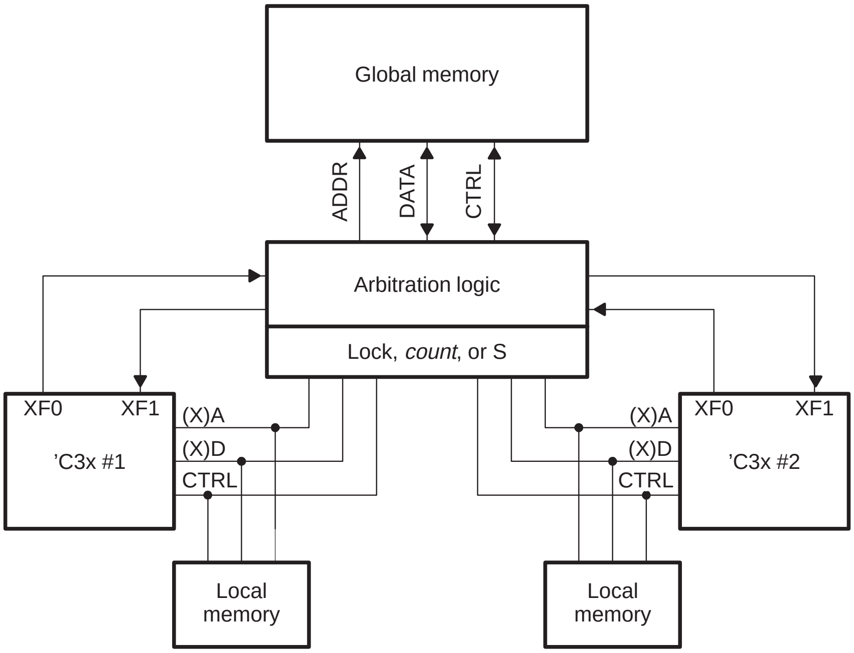
Example 7–9 shows how a location COUNT may contain a *count* of the number of times a particular operation must be performed. This operation may be performed by any processor in the system. If the *count* is 0, the processor waits until it is nonzero before beginning processing. The example also shows the algorithm for modifying COUNT correctly.

Example 7–9. Multiprocessor Counter Manipulation

```
CT:    OR        4,IOF         ; XF0 = 1
                               ; Interlocked operation ended
       LDII      @COUNT,R1     ; Interlocked operation begun
                               ; Contents of COUNT → R1
       BZ  CT                  ; If COUNT = 0, keep trying
       SUBI      1,R1          ; Decrement R1 (= COUNT)
       STII      R1,@COUNT     ; Update COUNT, XF0 = 1
                               ; Interlocked operation ended
```

Figure 7–2 illustrates multiple ’C3x devices sharing global memory and using the interlocked instructions as in Example 7–10, Example 7–11, and Example 7–12.

Figure 7–2. Multiple TMS320C3xs Sharing Global Memory



Sometimes it may be necessary for several processors to access some shared data or other common resources. The portion of code that must access the shared data is called a critical section.

To ease the programming of critical sections, semaphores may be used. Semaphores are variables that can take only nonnegative integer values. Two primitive, indivisible operations are defined on semaphores (with S being a semaphore):

```
V(S):      S + 1 → S
P(S):   P: if (S == 0), go to P
           else S – 1 → S
```

Indivisibility of V(S) and P(S) means that when these processes access and modify the semaphore S; they are the only processes doing so.

To enter a critical section, a P operation is performed on a common semaphore, say S (S is initialized to 1). The first processor performing P(S) will be able to enter its critical section. All other processors are blocked because S has become 0. After leaving its critical section, the processor performs a V(S), thus allowing another processor to execute P(S) successfully.

The ’C3x code for V(S) is shown in Example 7–10; code for P(S) is shown in Example 7–11. Compare the code in Example 7–11 to the code in Example 7–9, which does not use semaphores.

Example 7–10. Implementation of V(S)

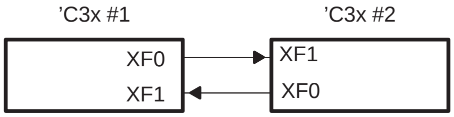
```
V:  LDII   @S,R0     ; Interlocked read of S begins (XFO = 0)
                     ; Contents of S → R0
    ADDI   1,R0      ; Increment R0 (= S)
    STII   R0,@S     ; Update S, end interlock (XF0 = 0)
```

Example 7–11. Implementation of P(S)

```
P:  OR     4,IOF  ; End interlock (XF0 = 1)
    NOP           ; Avoid potential pipeline conflicts when
                  ; executing out of cache, on-chip memory
                  ; or zero wait-state memory
    LDII   @S,R0  ; Interlocked read of S begins
                  ; Contents of S → R0
    BZ     P      ; If S = 0, go to P and try again
    SUBI   1,R0   ; Decrement R0 (= S)
    STII   R0,@S  ; Update S, end interlock (XF0 = 1)
```

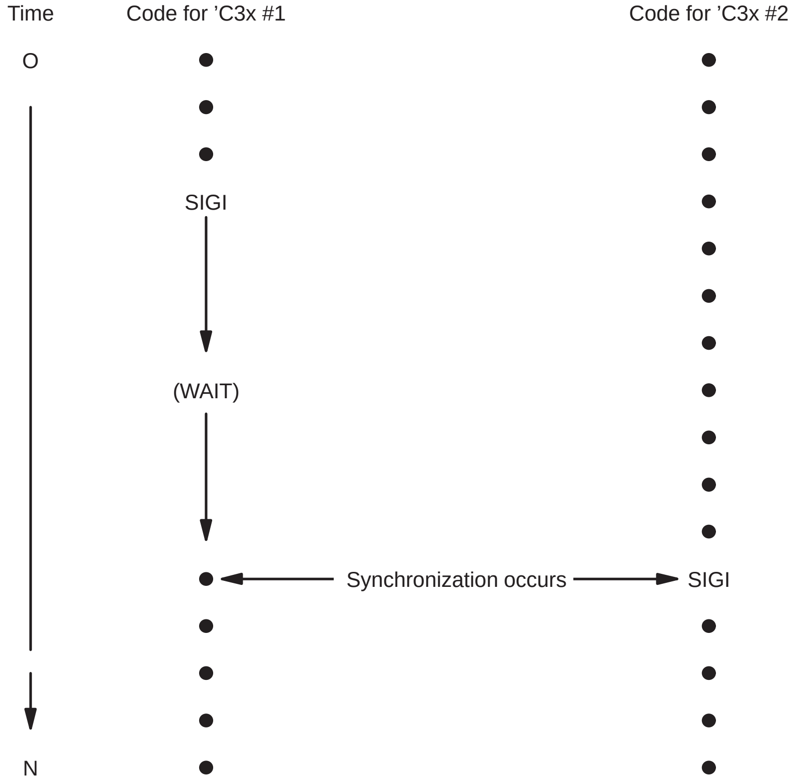
The SIGI operation can synchronize, at an instruction level, multiple ’C3xs. Consider two processors connected as shown in Figure 7–3. The code for the two processors is shown in Example 7–12.

Figure 7–3. Zero-Logic Interconnect of TMS320C3x Devices



Processor #1 runs until it executes the SIGI. It then waits until processor #2 executes a SIGI. At this point, the two processors are synchronized and continue execution.

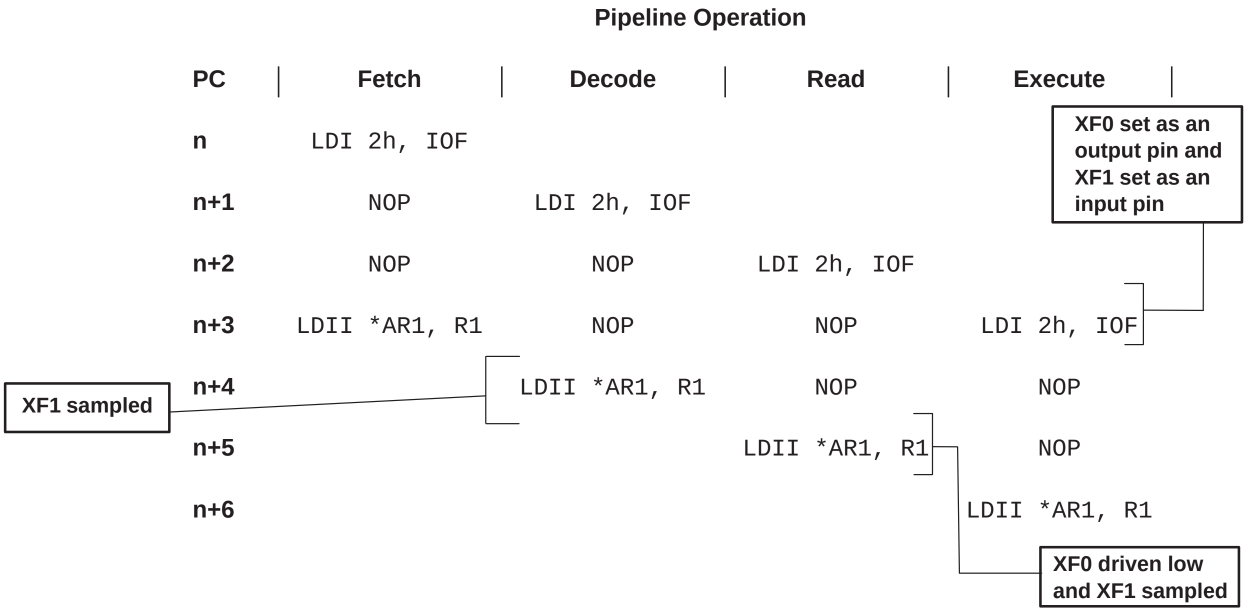
Example 7–12. Code to Synchronize Two TMS320C3x Devices at the Software Level



7.4.3 Pipeline Effects of Interlocked Instructions

Before performing an interlocked instruction, the XF0 pin must be configured as an output pin and the XF1 pin must be configured as an input pin through the IOF register (see subsection 3.1.10, *I/O Flag Register (IOF)*, on page 3-16). After the XF0 and XF1 pins are configured, no interlocked instruction can occur in the following two instructions.

Example 7–13. Pipeline Delay of XF Pin Configuration



Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	LDI 2h, IOF			
n+1	NOP	LDI 2h, IOF		
n+2	NOP	NOP	LDI 2h, IOF	
n+3	LDII *AR1, R1	NOP	NOP	LDI 2h, IOF
n+4		LDII *AR1, R1	NOP	NOP
n+5			LDII *AR1, R1	NOP
n+6				LDII *AR1, R1

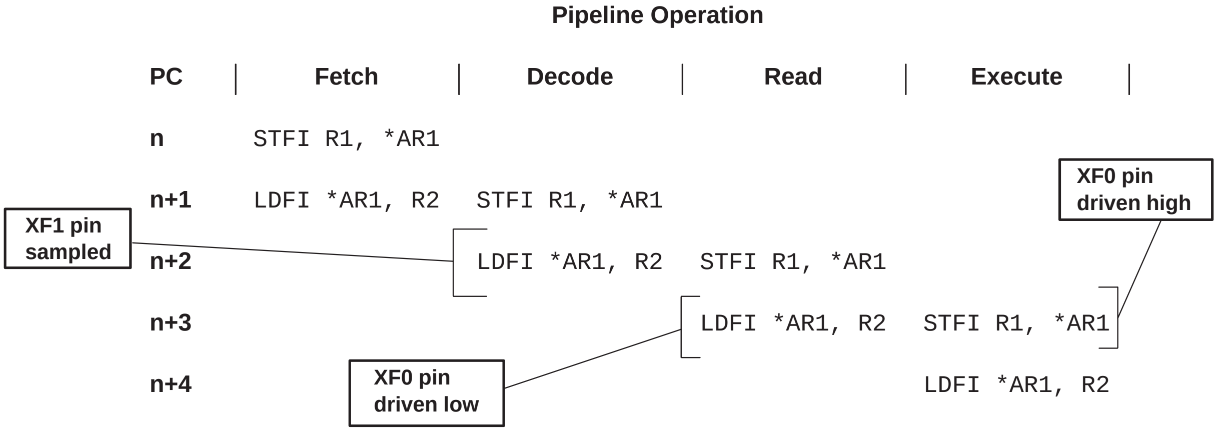
XF0 set as an output pin and XF1 set as an input pin

XF1 sampled

XF0 driven low and XF1 sampled

STFI and STII instructions drive the XF0 pin high during its execution phase. LDFI, LDII, and SIGI instructions sample the XF1 pin during its decode phase while driving the XF0 pin low during its read phase. Therefore, do not use an LDFI, LDII, or SIGI instruction immediately after an STFI or STII instruction (see Example 7–14).

Example 7–14. Incorrect Use of Interlocked Instructions



Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	STFI R1, *AR1			
n+1	LDFI *AR1, R2	STFI R1, *AR1		
n+2		LDFI *AR1, R2	STFI R1, *AR1	
n+3			LDFI *AR1, R2	STFI R1, *AR1
n+4				LDFI *AR1, R2

XF0 pin driven high

XF1 pin sampled

XF0 pin driven low

7.5 Reset Operation

The ’C3x supports a nonmaskable external reset signal ($\overline{\text{RESET}}$), which is used to perform system reset. This section discusses the reset operation.

At start-up, the state of the ’C3x processor is undefined. You can use the $\overline{\text{RESET}}$ signal to place the processor in a known state. This signal must be asserted low for ten or more H1 clock cycles to guarantee a system reset. H1 is an output clock signal generated by the ’C3x. (Check the datasheet for your device for the specific signal descriptions and electrical characteristics.)

Reset affects the other pins on the device in either a synchronous or asynchronous manner. The synchronous reset is gated by the ’C3x’s internal clocks. The asynchronous reset directly affects the pins and is faster than the synchronous reset. Table 7–3 shows the state of the ’C3x’s pins after $\overline{\text{RESET}}$ = 0. Each pin is described according to whether the pin is reset synchronously or asynchronously.

Table 7–3. TMS320C3x Pin Operation at Reset

		Device		
Signal	**Operation at Reset**	**‘C30**	**‘C31**	**‘C32**
	Primary Bus Interface Signals			
D31 – D0	Synchronous reset; placed in high-impedance state	✓	✓	✓
A23 – A0	Synchronous reset; placed in high-impedance state	✓	✓	✓
R/$\overline{\text{W}}$	Synchronous reset; deasserted by going to a high level	✓	✓	✓
$\overline{\text{IOSTRB}}$	Synchronous reset; deasserted by going to a high level	✓		✓
$\overline{\text{STRB0\_B3}}$/$A_{-1}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB0\_B2}}$/$A_{-2}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB0\_B1}}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB0\_B0}}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB1\_B3}}$/$A_{-1}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB1\_B2}}$/$A_{-2}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB1\_B1}}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB1\_B0}}$	Synchronous reset; deasserted by going to a high level			✓
$\overline{\text{STRB}}$	Synchronous reset; deasserted by going to a high level	✓	✓	
$\overline{\text{RDY}}$	Reset has no effect	✓	✓	✓
$\overline{\text{HOLD}}$	Reset has no effect	✓	✓	✓

Table 7–3. TMS320C3x Pin Operation at Reset (Continued)

		Device		
Signal	**Operation at Reset**	**‘C30**	**‘C31**	**‘C32**
$\overline{\text{HOLDA}}$	Reset has no effect	✓	✓	✓
PRGW	Reset has no effect			✓
	Expansion Bus Interface			
XD31 – XD0	Synchronous reset; placed in high-impedance state	✓		
XA12 – XA0	Synchronous reset; placed in high-impedance state	✓		
XR/$\overline{\text{W}}$	Synchronous reset; placed in high-impedance state	✓		
$\overline{\text{MSTRB}}$	Synchronous reset; deasserted by going to a high level	✓		
$\overline{\text{XRDY}}$	Reset has no effect	✓		
	Control Signals			
$\overline{\text{RESET}}$	Reset input pin	✓	✓	✓
$\overline{\text{INT3}}$ – $\overline{\text{INT0}}$	Reset has no effect	✓	✓	✓
$\overline{\text{IACK}}$	Synchronous reset; deasserted by going to a high level	✓	✓	✓
MC/$\overline{\text{MP}}$ or MCBL/$\overline{\text{MP}}$	Reset has no effect	✓	✓	✓
$\overline{\text{SHZ}}$	Reset has no effect	✓	✓	✓
XF1–XF0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
	Serial Port 0 Signals			
CLKX0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
DX0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
FSX0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
CLKR0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
DR0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
FSR0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
	Serial Port 1 Signals			
CLKX1	Asynchronous reset; placed in high-impedance state	✓		
DX1	Asynchronous reset; placed in high-impedance state	✓		
FSX1	Asynchronous reset; placed in high-impedance state	✓		
CLKR1	Asynchronous reset; placed in high-impedance state	✓		

Table 7–3. TMS320C3x Pin Operation at Reset (Continued)

		Device		
Signal	**Operation at Reset**	**‘C30**	**‘C31**	**‘C32**
DR1	Asynchronous reset; placed in high-impedance state	✓		
FSR1	Asynchronous reset; placed in high-impedance state	✓		
	Timer0 Signal			
TCLK0	Asynchronous reset; placed in high-impedance state	✓	✓	✓
	Timer1 Signal			
TCLK1	Asynchronous reset; placed in high-impedance state	✓	✓	✓
	Supply and Oscillator Signals			
V_{DD}	Reset has no effect	✓	✓	✓
IODV_{DD}	Reset has no effect	✓		
ADV_{DD}	Reset has no effect	✓		
PDV_{DD}	Reset has no effect	✓		
DDV_{DD}	Reset has no effect	✓		
MDV_{DD}	Reset has no effect	✓		
V_{SS}	Reset has no effect	✓	✓	✓
DV_{SS}	Reset has no effect	✓		✓
CV_{SS}	Reset has no effect	✓		✓
IV_{SS}	Reset has no effect	✓		✓
V_{BBP}	Reset has no effect	✓		✓
V_{SUBS}	Reset has no effect	✓	✓	✓
X1	Reset has no effect	✓	✓	
X2/CLKIN	Reset has no effect	✓	✓	✓
H1	Synchronous reset; will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition	✓	✓	✓
H3	Synchronous reset; will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition	✓	✓	✓

Table 7–3. TMS320C3x Pin Operation at Reset (Continued)

		Device		
Signal	**Operation at Reset**	**‘C30**	**‘C31**	**‘C32**
	Emulation, Test, and Reserved			
EMU0	Undefined	✓	✓	✓
EMU1	Undefined	✓	✓	✓
EMU2	Undefined	✓	✓	✓
EMU3	Undefined	✓	✓	✓
EMU4	Undefined	✓		
EMU5	Undefined	✓		
EMU6	Undefined	✓		
RSV0	Undefined	✓		
RSV1	Undefined	✓		
RSV2	Undefined	✓		
RSV3	Undefined	✓		
RSV4	Undefined	✓		
RSV5	Undefined	✓		
RSV6	Undefined	✓		
RSV7	Undefined	✓		
RSV8	Undefined	✓		
RSV9	Undefined	✓		
RSV10	Undefined	✓		

At system reset, the following additional operations are performed:

- ❑ The peripherals are reset. This is a synchronous operation. Peripheral reset is described in Chapter 12, *Peripherals*.
- ❑ The external bus control registers are reset. The reset values of the control registers are described in Chapter 9, *’C30 and ’C31 External-Memory Interface*.
- ❑ The following CPU registers are loaded with 0:
  - ■ ST (CPU status register), except in the ’C32, the PRGW status bit field is loaded with the status of the PRGW pin
  - ■ IE (CPU/DMA interrupt-enable flags)
  - ■ IF (CPU interrupt flags)
  - ■ IOF (I/O flags)
- ❑ The reset vector is read from memory location 0h. On the ’C32, this is a 32-bit data read. Once read, this value is loaded into the PC. This vector contains the start address of the system reset routine.
- ❑ At this point, code location is dictated by the PC.

Multiple ’C3x devices, driven by the same system clock, may be reset and synchronized. When the 1 to 0 transition of $\overline{\text{RESET}}$ occurs, the processor is placed on a well-defined internal phase, and all of the ’C3x devices come up on the same internal phase and all internal memory locations.

Unless otherwise specified, all registers are undefined after reset.

7.6 Interrupts

The ’C3x supports multiple internal and external interrupts, which can be used for a variety of applications. Internal interrupts are generated by the DMA controller, timers, and serial ports. Four external maskable interrupt pins include INT0 – INT3. Interrupts are automatically prioritized allowing interrupts to occur simultaneously and serviced in a predefined order. This section discusses the operation of these interrupts.

Additional information regarding internal interrupts can be found in Section 12.3.7, *DMA and Interrupts*, on page 12-64; Section 12.1.8, *Timer Interrupts* on page 12-13; and Section 12.2.11, *Serial-Port Interrupt Sources*, on page 12-34.

7.6.1 TMS320C30 and TMS320C31 Interrupt Vector Table

’C30 ’C31

Table 7–4 and Table 7–5 contain the interrupt vectors. In the microprocessor mode of the ’C30 and the ’C31 (Table 7–4) and the microcomputer mode of the ’C31 (Table 7–5), the interrupt vectors contain the addresses of interrupt service routines that should start executing when an interrupt occurs. On the other hand, in the microcomputer/boot-loader mode of the ’C31, the interrupt vector contains a branch instruction to the start of the interrupt service routine.

Table 7–4. Reset, Interrupt, and Trap-Vector Locations for the TMS320C30/ TMS320C31 Microprocessor Mode

’C30 ’C31

Address	Name	Function
00h	RESET	External reset signal input
01h	INT0	External interrupt on the $\overline{\text{INT0}}$ pin
02h	INT1	External interrupt on the $\overline{\text{INT1}}$ pin
03h	INT2	External interrupt on the $\overline{\text{INT2}}$ pin
04h	INT3	External interrupt on the $\overline{\text{INT3}}$ pin
05h	XINT0	Internal interrupt generated when serial port 0 transmit buffer is empty
06h	RINT0	Internal interrupt generated when serial port 0 transmit buffer is full
07h	XINT1[†]	Internal interrupt generated when serial port 1 transmit buffer is empty
08h	RINT1[†]	Internal interrupt generated when serial port 1 transmit buffer is full
09h	TINT0	Internal interrupt generated by timer0
0Ah	TINT1	Internal interrupt generated by timer1
0Bh	DINT	Internal interrupt generated by DMA controller
0Ch	Reserved	
•	•	
•	•	
•	•	
1Fh	Reserved	
20h	TRAP 0	Internal interrupt generated by TRAP 0 instruction
	•	
	•	
	•	
3Bh	TRAP 27	Internal interrupt generated by TRAP 27 instruction
3Ch	TRAP 28 (reserved)	
3Dh	TRAP 29 (reserved)	
3Eh	TRAP 30 (reserved)	
3Fh	TRAP 31 (reserved)	

[†] Reserved on ’C31

Table 7–5. Reset, Interrupt, and Trap-Branch Locations for the TMS320C31 Microcomputer Boot Mode

’C31

Address	Name	Function
809FC1	INT0	External reset signal input
809FC2	INT1	External interrupt on the $\overline{\text{INT0}}$ pin
809FC3	INT2	External interrupt on the $\overline{\text{INT1}}$ pin
809FC4	INT3	External interrupt on the $\overline{\text{INT2}}$ pin
809FC5	XINT0	External interrupt on the $\overline{\text{INT3}}$ pin
809FC6	RINT0	Internal interrupt generated when serial port 0 transmit buffer is empty
809FC7	XINT1 (Reserved)	
809FC8	RINT1 (Reserved)	
809FC9	TINT0	Internal interrupt generated by timer0
809FCA	TINT1	Internal interrupt generated by timer1
809FCB	DINT	Internal interrupt generated by DMA controller
809FCC–809FDF	Reserved	
809FE0	TRAP0	Internal interrupt generated by TRAP 0 instruction
809FE1	TRAP1	Internal interrupt generated by TRAP 1 instruction
•	•	
•	•	
•	•	
809FFB	TRAP27	Internal interrupt generated by TRAP 27 instruction
809FFC–809FFF	Reserved	

7.6.2 TMS320C32 Interrupt Vector Table

Similarly to the rest of the ’C3x device family, the ’C32’s reset vector location remains at address 0. On the other hand, the interrupt and trap vectors are relocatable. This is achieved by a new bit field in the CPU interrupt flag register called the interrupt-trap table pointer (ITTP), shown in Figure 3–11 on page 3-15. The ITTP bit field dictates the starting location (base) of the interrupt-trap-vector table. This base address is formed by left-shifting the value of the ITTP bit field by eight bits. This shifted value is called the effective base address and is referenced as EA[ITTP], as shown in Figure 7–4. Therefore, the location of an interrupt or trap vector is given by the addition of the effective base address formed by the ITTP bit field (EA[ITTP]) and the offset of the interrupt or trap vector in the interrupt-trap-vector table, as shown in Table 7–6. For example, if the ITTP contains the value 100h, the serial-port transmit interrupt vector will be located at 10005h. Note that the vectors stored in the interrupt-trap-vector table are the addresses of the start of the respective interrupt and trap routines. Furthermore, the interrupt-trap-vector table must lie on a 256-word boundary, since the eight LSBs of the effective base address of the interrupt-trap-vector table are 0.

Figure 7–4. Effective Base Address of the Interrupt-Trap-Vector Table

EA[ITTP] =

23 8	7 0
Bits 31–16 of the CPU interrupt flag register	00000000

Table 7–6. Interrupt and Trap-Vector Locations for the TMS320C32

Address	Name	Function
EA[ITTP] + 00h	Reserved	
EA[ITTP] + 01h	INT0	External interrupt on the $\overline{\text{INT0}}$ pin
EA[ITTP] + 02h	INT1	External interrupt on the $\overline{\text{INT1}}$ pin
EA[ITTP] + 03h	INT2	External interrupt on the $\overline{\text{INT2}}$ pin
EA[ITTP] + 04h	INT3	External interrupt on the $\overline{\text{INT3}}$ pin
EA[ITTP] + 05h	XINT0	Internal interrupt generated when serial port 0 transmit buffer is empty
EA[ITTP] + 06h	RINT0	Internal interrupt generated when serial port 0 transmit buffer is full
EA[ITTP] + 07h	Reserved	
EA[ITTP] + 08h	Reserved	
EA[ITTP] + 09h	TINT0	Internal interrupt generated by timer0
EA[ITTP] + 0Ah	TINT1	Internal interrupt generated by timer1
EA[ITTP] + 0Bh	DINT0	Internal interrupt generated by DMA channel 0
EA[ITTP] + 0Ch	DINT1	Internal interrupt generated by DMA channel 1
EA[ITTP] + 0Dh	Reserved	
EA[ITTP] + 1Fh	Reserved	
EA[ITTP] + 20h	TRAP 0	Internal interrupt generated by TRAP 0 instruction
	•	
	•	
	•	
EA[ITTP] + 3Bh	TRAP 27	Internal interrupt generated by TRAP 27 instruction
EA[ITTP] + 3Ch	TRAP 28 (reserved)	
EA[ITTP] + 3Dh	TRAP 29 (reserved)	
EA[ITTP] + 3Eh	TRAP 30 (reserved)	
EA[ITTP] + 3Fh	TRAP 31 (reserved)	

7.6.3 Interrupt Prioritization

When two interrupts occur in the same clock cycle or when two previously received interrupts are waiting to be serviced, one interrupt is serviced before the other. The CPU handles this prioritization by servicing the interrupt with the least priority. The priority of interrupts is handled by the CPU according to the interrupt vector table. Priority is set according to position in the table—those with displacements closest to the base address of the table are higher in priority. Table 7–7 shows the priorities assigned to the reset and interrupt vectors.

Table 7–7. Reset and Interrupt Vector Priorities

Reset or Interrupt	Vector Location	Priority	Function
$\overline{\text{RESET}}$	0h	0	External reset signal input on the RESET pin
$\overline{\text{INT0}}$	1h	1	External interrupt on the INT0 pin
$\overline{\text{INT1}}$	2h	2	External interrupt on the INT1 pin
$\overline{\text{INT2}}$	3h	3	External interrupt on the INT2 pin
$\overline{\text{INT3}}$	4h	4	External interrupt on the INT3 pin
XINT0	5h	5	Internal interrupt generated when serial-port 0 transmit buffer is empty
RINT0	6h	6	Internal interrupt generated when serial-port 0 receive buffer is full
XINT1[†]	7h	7	Internal interrupt generated when serial-port 1 transmit buffer is empty
RINT1[†]	8h	8	Internal interrupt generated when serial-port 1 receive buffer is full
TINT0	9h	9	Internal interrupt generated by timer0
TINT1	0Ah	10	Internal interrupt generated by timer1
DINT/ DINT0	0Bh	11	Internal interrupt generated by DMA channel 0
DINT1[‡]	0Ch	12	Internal interrupt generated by DMA channel 1

[†] Reserved on ’C31 and ’C32
[‡] Present on ’C32 only

7.6.4 CPU Interrupt Control Bits

Three CPU registers contain bits that control interrupt operation:

❑ Status (ST) register

The CPU global interrupt-enable bit (GIE) located in the CPU status register (ST) controls all maskable CPU interrupts. When this bit is set to 1, the CPU responds to an enabled interrupt. When this bit is cleared to 0, all CPU interrupts are disabled. see Section 3.1.7 on page 3-5 for more information.

❑ CPU/DMA interrupt-enable (IE) register

This register individually enables/disables CPU, DMA external, serial port, and timer interrupts. See Section 3.1.8 on page 3-9 for more information.

❑ CPU interrupt flag (IF) register

This register contains interrupt flag bits that indicate the corresponding interrupt is set. See Section 3.1.9 on page 3-11 for more information.

7.6.5 Interrupt Flag Register Behavior

When an external interrupt occurs, the corresponding bit of the IF register is set to 1. When the CPU or DMA controller processes this interrupt, the corresponding interrupt flag bit is cleared by the internal interrupt acknowledge signal. However, for level-triggered interrupts, if $\overline{\text{INT}n}$ is still low when the interrupt acknowledge signal occurs, the interrupt flag bit is cleared for only one cycle and then set again, because $\overline{\text{INT}n}$ is still low. Depending on when the IF register is read, it is also possible that this bit may be 0 even though $\overline{\text{INT}n}$ is 0. When the ’C3x is reset, 0 is written to the interrupt flag register, clearing all pending interrupts.

The interrupt flag register bits can be read from and written to under software control. Writing a 1 to an IF register bit sets the associated interrupt flag to 1. Similarly, writing a 0 resets the corresponding interrupt flag to 0. In this way, all interrupts may be triggered and/or cleared through software. Since the interrupt flags may be read, the interrupt pins may be polled in software when an interrupt-driven interface is not required.

Internal interrupts operate in a similar manner. In the IF register, the bit corresponding to an internal interrupt can be read from and written to through software. Writing a 1 sets the interrupt latch; writing a 0 clears it. All internal interrupts are one H1/H3 cycle in length. If any previous bit value of the IF register needs to be preserved, a modification to IF register should be performed with logic operations (AND, OR, etc.) directly to IF.

Figure 7–5. IF Register Modification

Correct	Incorrect
LDI @MASK, R0	LDI IF, R1
AND R0, IF	AND @MASK, R1
	LDI R1, IF

Note: IF Register Load Priority

If a load of the IF register occurs simultaneously with a set or reset of a flag by an interrupt pulse, the loading of the flag has higher priority and overwrites the IF register value.

7.6.6 Interrupt Processing

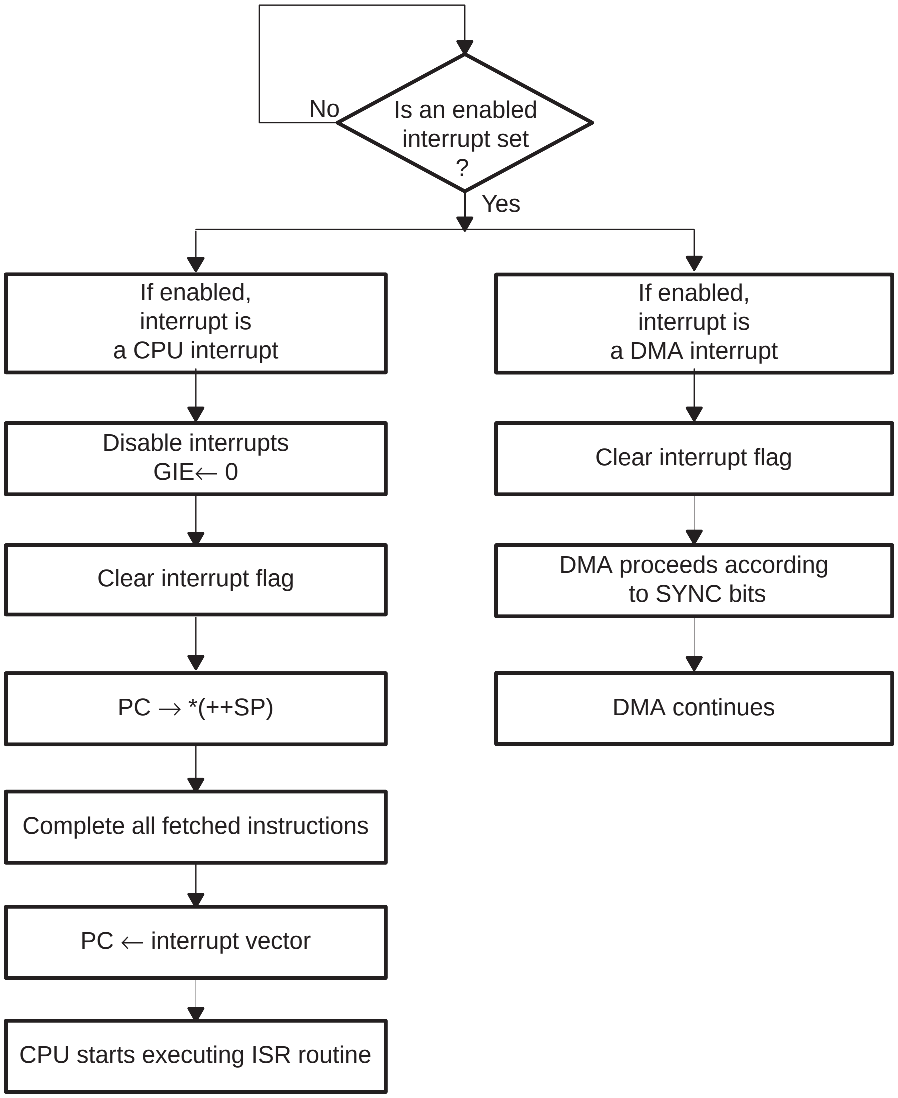
The ’C3x allows the CPU and DMA coprocessor to respond to and process interrupts in parallel. Figure 7–6 on page 7-34 shows interrupt processing flow; for the exact sequence, see Table 7–8 on page 7-36.

For a CPU interrupt to occur, at least two conditions must be met:

- All interrupts must be enabled globally by setting the GIE bit to 0 in the status register.
- The interrupt must be enabled by setting the corresponding bit in the IF register.

In the CPU interrupt processing cycle (left side of Figure 7–6), the corresponding interrupt flag in the IF register is cleared, and interrupts are globally disabled (GIE = 0). The CPU completes all fetched instructions. The current PC is pushed to the top of the stack. The interrupt vector is then fetched and loaded into the PC, and the CPU starts executing the first instruction in the interrupt service routine (ISR).

Figure 7–6. CPU Interrupt Processing


Is an enabled interrupt set ?
No
Yes
If enabled, interrupt is a CPU interrupt
If enabled, interrupt is a DMA interrupt
Disable interrupts GIE← 0
Clear interrupt flag
Clear interrupt flag
DMA proceeds according to SYNC bits
PC → *(++SP)
DMA continues
Complete all fetched instructions
PC ← interrupt vector
CPU starts executing ISR routine

Note: CPU and DMA Interrupts

CPU interrupts are acknowledged (responded to by the CPU) on instruction fetch boundaries only. If instruction fetches are halted because of pipeline conflicts or execution of RPTS loops, CPU interrupts are not acknowledged until the next instruction fetch.

If you wish to make the interrupt service routine interruptible, you can set the GIE bit to 1 after entering the ISR.

The interrupt acknowledge (IACK) instruction can be used to signal externally that an interrupt has been serviced. If external memory is specified in the operand, IACK drives the $\overline{\text{IACK}}$ pin and performs a dummy read. The read is performed from the address specified by the IACK instruction operand. IACK is typically placed in the early portion of an ISR. However, depending on your application, it may be better to place it at the end of the ISR or not at all.

Note the following:

- Interrupts are disabled during an RPTS and during a delayed branch (until the three instructions following a delayed branch are completed). Interrupts are held until after the branch.
- When an interrupt occurs, instructions currently in the decode and read phases continue regular execution, unlike an instruction in the fetch phase:
  - If the interrupt occurs in the first cycle of the fetch of an instruction, the fetched instruction is discarded (not executed), and the address of that instruction is pushed to the top of the system stack.
  - If the interrupt occurs after first cycle of the fetch (in the case of a multi-cycle fetch due to wait states), that instruction is executed, and the address of the next instruction to be fetched is pushed to the top of the system stack.

7.6.7 CPU Interrupt Latency

CPU interrupt latency, defined as the time from the acknowledgement of the interrupt to the execution of the first ISR instruction, is at least eight cycles. This is explained in Table 7–8 on page 7-36, where the interrupt is treated as an instruction. It assumed that all of the instructions are single-cycle instructions.

Table 7–8. Interrupt Latency

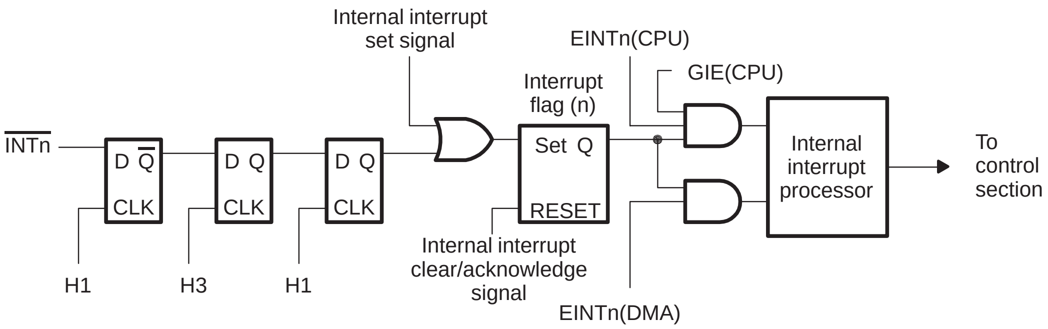
Cycle	Description	Fetch	Decode	Read	Execute
1	Recognize interrupt in single-cycle fetched (prog a + 1) instruction	prog a + 1	prog a	prog a–1	prog a–2
2	Clear GIE bit. Clear interrupt flag	—	interrupt	prog a	prog a–1
3	Read the interrupt vector table	—	—	interrupt	prog a
4	Store return address to stack	—	—	—	interrupt
5	Pipeline begins to fill with ISR instruction	isr1	—	—	—
6	Pipeline continues to fill with ISR instruction	isr2	isr1	—	—
7	Pipeline continues to fill with ISR instruction	isr3	isr2	isr1	—
8	Execute first instruction of interrupt service routine	isr4	isr3	isr2	isr1

7.6.8 External Interrupts

The four external maskable interrupt pins $\overline{\text{INT0}}$–$\overline{\text{INT3}}$ are enabled at the IF register (Section 3.1.9, *CPU Interrupt Flag (IF) Register*, on page 3-11) and are synchronized internally. They are sampled on the falling edge of H1 and passed through a series of H1/H3 latches internally. These latches require the interrupt signal to be held low for at least one H1/H3 clock cycle to be recognized by the ‘C3x. Once synchronized, the interrupt input sets the corresponding interrupt flag register (IF) bit if the interrupt is active.

Figure 7–7 shows a functional diagram of the logic used to implement external interrupt inputs.

Figure 7–7. Interrupt Logic Functional Diagram



These interrupts are prioritized by the selection of one over the other if both come on the same clock cycle ($\overline{\text{INT0}}$ the highest, $\overline{\text{INT1}}$ next, etc.). When an interrupt is taken, the status register ST(GIE) bit is reset to 0, disabling any other incoming interrupt. This prevents any other interrupt ($\overline{\text{INT0}}$ – $\overline{\text{INT3}}$) from assuming program control until the ST(GIE) bit is set back to 1. On a return from an interrupt routine, the RETI and RETI*cond* instructions set the ST(GIE) bit to 1.

On the ‘C30 and ‘C31, external interrupts are level triggered. On the ‘C32, external interrupts are edge or level triggered, depending on the INT config bit field of the status register.

For an edge-triggered interrupt to be detected by the ’C32 the external pin must transition from 1 to 0. And then, it needs to be held low for at least one H1/H3 cycle (but it could be held low longer).

For a level-triggered interrupt to be detected by the ’C3x, the external pin must be held low for between one and two cycles (1 ≤ low-pulse width ≤ 2). If the interrupt is held low for more than two cycles, more than one interrupt might be recognized. There is no need to provide an edge in this case.

7.7 DMA Interrupts

Interrupts can also trigger DMA read and write operations. This is called DMA synchronization. The DMA interrupt processing cycle is similar to that of the CPU. After the pertinent interrupt flag is cleared, the DMA coprocessor proceeds according to the status of the SYNC bits in the DMA coprocessor global-control register.

If the interrupt in the CPU/DMA interrupt-enable (IE) register is enabled, the interrupt controller automatically latches the interrupt and saves it for future DMA use. The interrupt controller latches the interrupt, clears the flag in the IF register, and informs the data that an interrupt has occurred. The DMA then proceeds with the transfer according to the previously configured CPU/DMA priority. Even if the DMA has not been started, the interrupt latch occurs, and the flag is cleared, except when the start bits in the DMA control register have the reset value 00_2 in START bits. DMA reset clears the interrupt internal latch.

7.7.1 DMA Interrupt Control Bits

Two registers contain bits used to control DMA interrupt operation:

- ❑ CPU/DMA interrupt-enable register (IE). All DMA interrupts are controlled by the most significant 16 bits in the IE register and by the SYNC bits of the DMA channel control registers (see Section 12.3.3, *DMA Registers*, on page 12-51). The DMA interrupts are not dependent upon ST(GIE) and are local to the DMA.
- ❑ The DMA channel control register. Each DMA coprocessor channel uses a channel control register to determine its mode of operation. This register is shown in Section 12.3.3.

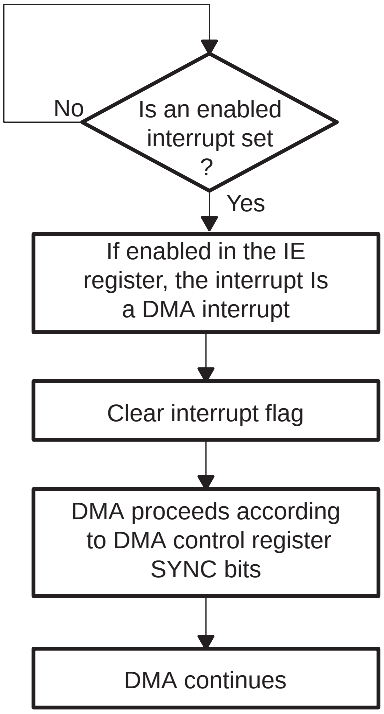
The IE is broken into several subfields that determine which interrupts can be used to control the synchronization for each DMA channel. For example, the bits in each of these fields allow you to select whether a DMA channel is synchronized to a port, a timer, or an external interrupt pin. Note that the ’C32 has two DMA channels while the ’C30 and ’C31 have a single DMA channel.

See Section 3.1.8, *CPU/DMA Interrupt-Enable Register (IE)*, on page 3-9, for a description of the IE.

7.7.2 DMA Interrupt Processing

Figure 7–8 shows the general flow of interrupt processing by the DMA coprocessor.

Figure 7–8. DMA Interrupt Processing


No
Is an enabled interrupt set ?
Yes
If enabled in the IE register, the interrupt Is a DMA interrupt
Clear interrupt flag
DMA proceeds according to DMA control register SYNC bits
DMA continues

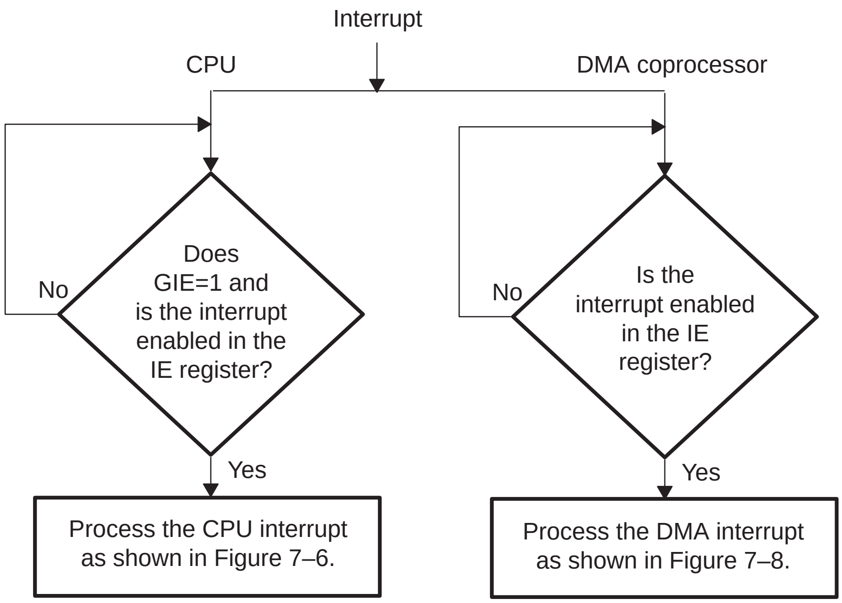
For more information about DMA interrupts, see Section 12.3.7, *DMA Interrupts* on page 12-64.

7.7.3 CPU/DMA Interaction

If the DMA is not using interrupts for synchronization of transfers, it is not affected by the processing of the CPU interrupts. Detected interrupts are responded to by the CPU and DMA on instruction fetch boundaries only. Since instruction fetches are halted due to pipeline conflicts or when executing instructions in an RPTS loop, interrupts are not responded to until instruction fetching continues. It is therefore possible to interrupt the CPU and DMA simultaneously with the same or different interrupts and, in effect, synchronize their activities. For example, it may be necessary to cause a high-priority DMA transfer that avoids bus conflicts with the CPU (that is, a transfer that makes the DMA higher priority than the CPU). This may be accomplished by using an interrupt that causes the CPU to trap to an interrupt routine that contains an IDLE instruction. Then, if the same interrupt is used to synchronize DMA transfers, the DMA transfer counter can be used to generate an interrupt and thus return control to the CPU following the DMA transfer.

Since the DMA and CPU share the same set of interrupt flags, the DMA may clear an interrupt flag before the CPU can respond to it. For example, if the CPU interrupts are disabled, the DMA can respond to interrupts and thus clear the associated interrupt flags. Figure 7–9 shows the sequence of events in the interrupt processing for both the CPU and DMA controllers.

Figure 7–9. Parallel CPU and DMA Interrupt Processing


Interrupt
CPU
DMA coprocessor
No
Does GIE=1 and is the interrupt enabled in the IE register?
Yes
Process the CPU interrupt as shown in Figure 7–6.
No
Is the interrupt enabled in the IE register?
Yes
Process the DMA interrupt as shown in Figure 7–8.

7.7.4 TMS320C3x Interrupt Considerations

Give careful consideration to ’C3x interrupts, especially if you make modifications to the status register when the global interrupt-enable (GIE) bit is set. This can result in the GIE bit being erroneously set or reset as described in the following paragraphs.

The GIE bit field of the status register is set to 0 (reset) by an interrupt. If a load of the status register occurs simultaneously with a CPU interrupt pulse trying to reset GIE, GIE will be reset.

Also, resetting GIE by an interrupt or TRAP instruction can cause a processing error if any code, following within two cycles of the interrupt recognition, attempts to read or modify the status register. For example, if the status register is being pushed onto the stack, it will be stored incorrectly if an interrupt was acknowledged two cycles before the store instruction.

When an interrupt signal is recognized, the ’C3x continues executing the instructions already in the *read* and *decode* phases in the pipeline. However, because the interrupt is acknowledged, the GIE bit is reset to 0, and the store instruction already in the pipeline will store the wrong status register value.

For example, if the program is like this:

```
                        ...
                        NOP
interrupt recognized ––>LDI   @V_ADDR,AR1
                        MPYI  *AR1, R0
                        PUSH  ST
                        ...
                        POP   ST
                        ...
```

the PUSH ST instruction will save the ST contents in memory, which includes GIE = 0. Since the device is expected to have GIE = 1, the POP ST instruction will put the wrong status register value into the ST (see Table 7–9).

Table 7–9. Pipeline Operation with PUSH ST

Cycle	Description	Fetch	Decode	Read	Execute
1		NOP			
2		LDI	NOP		
3		MPYI	LDI	NOP	
4	Read location V_ADDR	PUSH	MPYI	LDI	NOP
5	Load AR1; recognize interrupt	–	PUSH	MPYI	LDI
6	Clear GIE bit; clear interrupt flag; read SP		Interrupt	PUSH	MPYI
7	Read interrupt vector table; save ST in stack			Interrupt	PUSH
8	Store return address on stack				Interrupt

The following example shows setting the GIE bit by a load instruction that is immediately followed by an interrupt:

```
                        ...             ; GIE = 1
                        LDI 02000h, ST ; GIE = 0
interrupt recognized ––>MPYI  *AR1, R0 ;
                        ADD   *AR0, R1
```

In this example, the load of the status register or interrupt-flag register overwrites the reset of the GIE bit by the interrupt (see Table 7–10).

Table 7–10. Pipeline Operation with Load Followed by Interrupt

Cycle	Description	Fetch	Decode	Read	Execute
1		LDI			
2	Interrupt recognized	–	LDI		
3	Interrupt resets GIE bit, clears interrupt flag, reads SP		interrupt	LDI	
4	GIE set by load instruction; interrupt vector table read and ST saved on stack			interrupt	LDI
5	Store return address on stack				interrupt
6	Fetch first instruction of ISR with GIE = 1	ISR			

A similar situation may occur if the GIE bit = 1 and an instruction executes that is intended to modify the other status bits and leave the GIE bit set. In the above example, this erroneous setting would occur if the interrupt were recognized two cycles before the POP ST instruction. In that case, the interrupt would clear the GIE bit, but the execution of the POP instruction would set the GIE bit. Since the interrupt has been recognized, the interrupt service routine will be entered with interrupts enabled, rather than disabled as expected.

One solution is to use an instruction that is uninterruptible such as RPTS as follows to set the GIE:

```
RPTS   0
AND    2000h, ST    ; Set GIE=1
```

Use the following to reset the GIE:

```
RPTS   0
AND    0DFFFh, ST   ; Set GIE=0
```

Another alternative incorporates the following code fragment, which protects against modifying or saving the status register by disabling interrupts through the interrupt-enable register:

```
PUSH IE         ; Save IE register   • Added instructions to
LDI  0, IE      ; Clear IE register    avoid pipeline problems
NOP             ;                    • 2 NOPs or useful instructions
NOP             ;
AND  0DFFFh, ST ; Set GIE = 0        • Instruction that reads or
POP  IE         ;                      writes to ST register.
                ;                      Added instruction
                ;                      to avoid pipeline
                ;                      problems.
```

In summary, the next three instructions immediately following an instruction that clears the GIE bit might be interrupted. Also, the next three instructions immediately following an instruction that sets the GIE bit might not be interrupted even if there is a pending interrupt (see Example 7–15). Similarly, the next three instructions immediately following an instruction that clears an interrupt-enable mask might be interrupted. Furthermore, the next three instructions immediately following an instruction that sets an interrupt flag might be executed before the interrupt occurs.

Example 7–15. Pending Interrupt

```
LDI    0h, ST    ; set GIE = 0
LDI    1h, R1
LDI    2h, R2
MPYI   *AR1, R0  ; interrupts still enabled
ADDI   *AR1,R1   ; interrupts disabled here
```

7.7.5 TMS320C30 Interrupt Considerations

The ’C30 silicon revisions earlier than 4.0 have two unique exceptions to the interrupt operation. This does not apply to ’C30 silicon revision 4.0 or greater, any ’C31 silicon, or any ’C32 silicon.

On ’C30 silicon revisions earlier than 4.0:

- The status register global interrupt-enable (GIE) bit may be erroneously reset to 0 (disabled setting) if all of the following conditions are true:

  - A conditional trap instruction (TRAP*cond*) has been fetched
  - The condition for the trap is false.
  - A pipeline conflict has occurred, resulting in a delay in the decode or read phase of the instruction.

  During the decode phase of a conditional trap, interrupts are temporarily disabled to ensure that the trap executes before a subsequent interrupt. If a pipeline conflict occurs and causes a delay in execution of the conditional trap, the interrupt disabled condition may become the last known condition of the GIE bit. If the trap condition is *false*, interrupts are permanently disabled until the GIE bit is intentionally set. The condition is not present when the trap condition is *true,* because normal operation of the instruction causes the GIE to be reset, and standard coding practice sets the GIE to 1 before the trap routine is exited. Several instruction sequences that cause pipeline conflicts have been found:

  -
    ```
    LDI           mem,SP
    TRAPcond      n
    ```

  -
    ```
    LDI           mem,SP
    NOP
    TRAPcond      n
    ```

  -
    ```
    STI           SP,mem
    TRAPcond      n
    ```

  -
    ```
    STI           Rx,*ARy
    LDI           *ARx,Ry
    ||LDI         *ARz,Rw
    TRAPcond      n
    ```

  Other similar conditions may also cause a delay in the execution. The following solution is recommended to avoid or rectify the problem:

Insert two NOP instructions immediately before the TRAP*cond* instruction. One NOP is insufficient in some cases, as illustrated in the second bulleted item, above. This eliminates the opportunity for any pipeline conflicts in the immediately preceding instructions and enables the conditional trap instruction to execute without delays.

- Asynchronous accesses to the interrupt flag register (IF) can cause the 'C30 silicon revision prior to 4.0 to fail to recognize and service an interrupt. This may occur when an interrupt is generated and is ready to be latched into the IF register on the same cycle that the IF is being written to by the CPU. Note that logic operations (AND, OR, XOR) may write to the IF register.

  The logic of 'C30 silicon revision earlier than 4.0 currently gives the CPU write priority; consequently, the asserted interrupt might be lost. This is true if the asserted interrupt was generated internally (for example, a direct memory access (DMA) interrupt). This situation arises as a result of a decision to poll certain interrupts or a desire to clear pending interrupts due to a long pulse width. In the case of a long pulse width, the interrupt may be generated after the CPU responds to the interrupt and attempts to automatically clear it by the interrupt vector process.

  The recommended solution is to avoid using the interrupt polling technique, and to design the external interrupt inputs to have pulse widths between 1 and 2 instruction cycles. The alternative to strict polling is to periodically enable and disable the interrupts that would be polled, allowing the normal interrupt vectoring to take place; this automatically clears the interrupt flag without affecting other interrupts. If you must clear a pending interrupt, you should use a memory location to indicate that the interrupt is invalid. The interrupt service routine can read that location, clear it (if the pending interrupt is invalid), and return immediately. The following code fragments show how to handle a dummy interrupt due to a long interrupt pulse:

```
ISR_n:       PUSH      ST                   ;
             PUSH      DP                   ; Save registers
             PUSH      R0                   ;
             LDI       0, DP                ; Clear Data-page Pointer
             LDI       @DUMMY_INT, R0       ; If DUMMY_INT is 0 or positive,
             BNN       ISR_n_START          ; go to ISR_n_START
             STI       DP, @DUMMY_INT       ; Set DUMMY_INT = 0
             POP       R0                   ;
             POP       DP                   ;
             POP       ST                   ; Housekeeping, return from interrupt
             RETI                           ;

ISR_n_START: .
             .                              ; Normal interrupt service routine
             .                              ; Code goes here
             LDI       INT_Fn, R0           ;
             AND       IF, R0               ; If ones in IF reg match
             BNZ       ISR_n_END            ; INT_Fn, exit ISR
             LDI       0, DP                ; Otherwise clear
             LDI       0FFFFh, R0           ; DP and set
             STI       R0, @DUMMY_INT       ; DUMMY_INT negative & exit
ISR_n_END:
             POP       R0                   ;
             POP       DP                   ; Exit ISR
             POP       ST                   ;
             RETI                           ;
```

7.8 Traps

A trap is the equivalent of a software-triggered interrupt. In the ’C3x, traps and interrupts are treated identically, except in the way in which they are triggered.

7.8.1 Initialization of Traps and Interrupts

Traps and interrupts are triggered differently in the ’C3x:

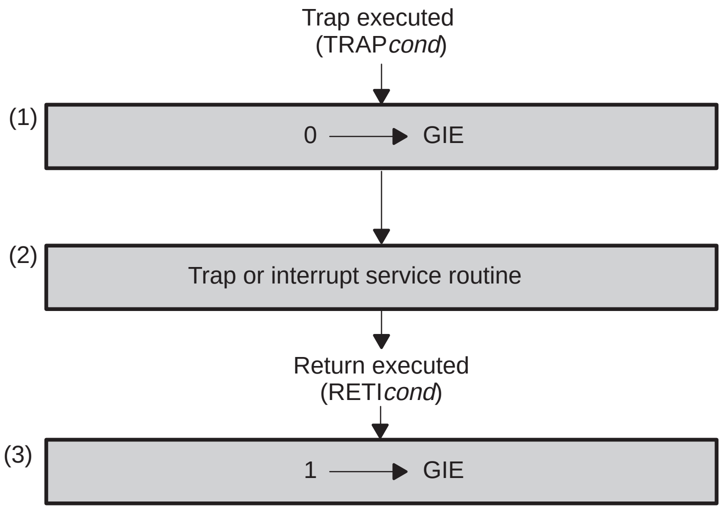
- **Traps** are always triggered by a software mechanism, by the TRAP*cond* (conditional trap) instructions.
- **Interrupts** are always triggered by hardware events (for example, by external interrupts, DMA interrupts, or serial-port interrupts).

The GIE bit in the ST register and the mask bits in the IE do not apply to traps.

7.8.2 Operation of Traps

Figure 7–10 shows the general flow of traps which is similar to interrupts.

Figure 7–10. Flow of Traps



The **RETI*cond*** instruction manipulates the status flags as shown in block (3) in Figure 7–10. RETI*cond* provides a return from a trap or interrupt.

The ’C3x supports 32 different traps. When a TRAP*cond n* instruction is executed, the ’C3x jumps to the address stored in the memory location pointed to by the corresponding trap-vector table pointer. The location of the trap-vector table is shown in Table 7–4 on page 7-27 (’C30/’C31 microprocessor mode), Table 7–5 (’C31 microcomputer boot mode) on page 7-28, and Table 7–6 on page 7-30 for the ’C32.

7.9 Power Management Modes

The following ’C3x devices have been enhanced by the addition of two power-down modes: IDLE2 and LOPOWER:

- ’C30 silicon version 7.0 or greater
- ’LC31
- ’C31 silicon revision 5.0 or greater
- ’C32

7.9.1 IDLE2 Power-Down Mode

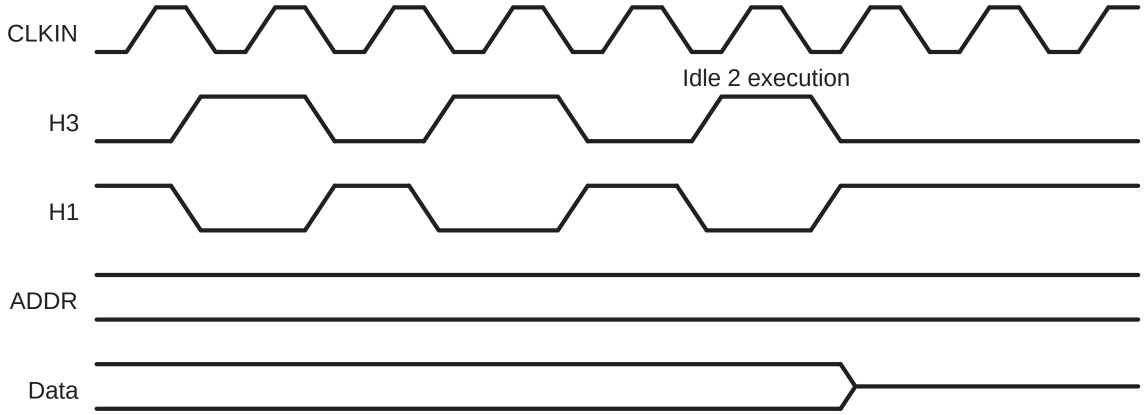
The H1 instruction clock is held high until one of the four external interrupts is asserted. In IDLE2 mode, the ’C3x devices supporting these modes behave as follows:

- No instructions are executed.
- The CPU, peripherals, and internal memory retain their previous states.
- The external bus output pins are idle:
  - The address lines remain in their previous states.
  - The data lines are in the high-impedance state.
  - The output control signals are in their inactive state.
  - If a multicycle read or write does not preceed the IDLE2 opcode, that access will be forzen onto the bus until IDLE2 is exited. This can be advantageous for low power applications since the bus is frozen in an active state. That is, the device pins are not floating, and therefore do not require pullup or pulldowns.
- When the device is in the functional (nonemulation) mode, the clocks stop with H1 high and H3 low (see Figure 7–11).
- The devices remain in IDLE2 until one of the four external interrupts (INT3–INT0) is asserted for at least one H1 cycle. When one of the four interrupts is asserted, the clocks start after a delay of one H1 cycle. When the clocks restart, they may be in the opposite phase (that is, H1 may be high if H3 was high before the clocks were stopped; H3 may be low if H1 was previously low). The H1 and H3 clocks remain 180 degrees out of phase with each other (see Figure 7–12).
- During IDLE2 operations, the CPU recognizes one of the four external interrupts if it is asserted for more than one H1 cycle. To avoid generating multiple false interrupts in level-triggered mode, the interrupt must be asserted for fewer than three H1 cycles.

- ❑ The interrupt service routine (ISR) must have been set up before placing the device in IDLE2 mode, because the instruction following the IDLE2 instruction is not executed until the RETI (return from interrupt) instruction is executed.
- ❑ When the device is in emulation mode, the H1 and H3 clocks continue to run normally and the CPU operates as if an IDLE instruction was executed. The clocks continue to run for correct operation of the emulator.

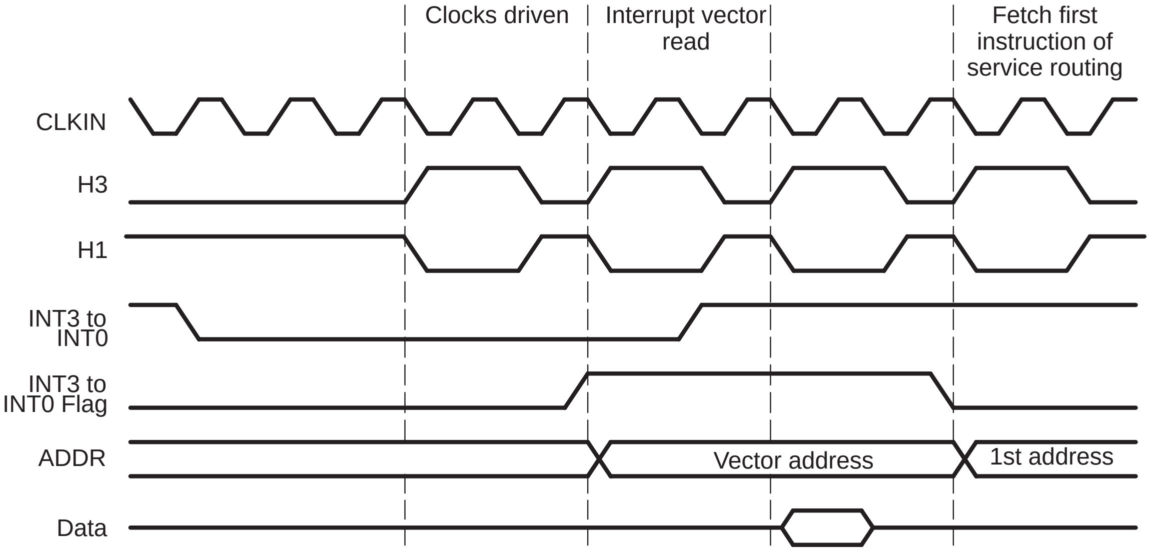
> CAUTION
>
> **Delayed Branch**
>
> **For correct device operation, the three instructions following a delayed branch should not include either IDLE or IDLE2 instructions.**

Figure 7–11. IDLE2 Timing



]

Figure 7–12. Interrupt Response Timing After IDLE2 Operation



7.9.2 LOPOWER

In the LOPOWER (low-power) mode, the CPU continues to execute instructions, and the DMA can continue to perform transfers, but at a reduced clock rate of CLKIN frequency divided by 16.

A ’C31 with a CLKIN frequency of 32 MHz performs identically to a 2 MHz ’C31 with an instruction cycle time of 1,000 ns.

During the read phase of the . . .	**The ’C31 and ’C32 . . .**
LOPOWER instruction (Figure 7–13)	Slow to 1/16 of full-speed operation.
MAXSPEED instruction (Figure 7–14)	Resume full-speed operation.

Figure 7–13. LOPOWER Timing

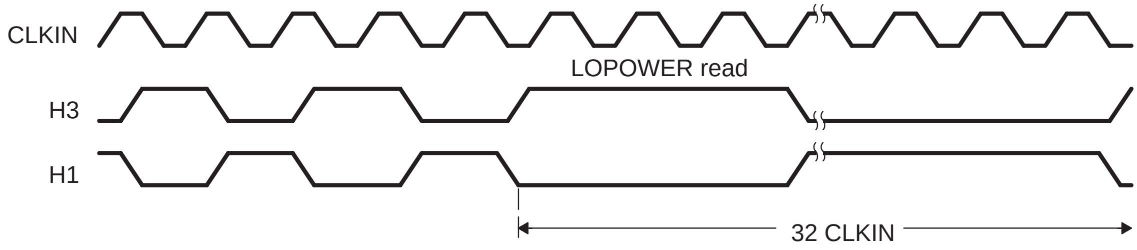
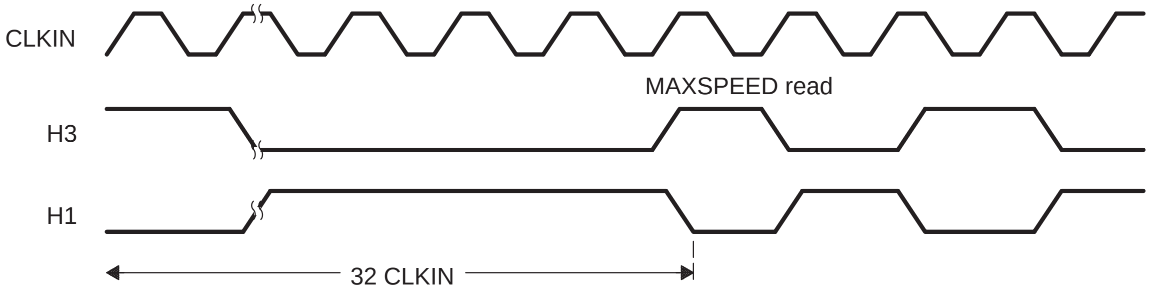


Figure 7–14. MAXSPEED Timing



Chapter 8

Pipeline Operation

Two characteristics of the'C3x that contribute to its high performance are:

- ❑ Pipelining
- ❑ Concurrent I/O and CPU operation

The following four functional units control 'C3x operation:

- ❑ Fetch
- ❑ Decode
- ❑ Read
- ❑ Execute

Pipelining is the overlapping or parallel operations of the fetch, decode, read, and execute levels of a basic instruction.

The DMA controller decreases pipeline interference and enhances the CPU's computational throughput by performing input/output operations.

Topic **Page**

8.1 Pipeline Structure

The following list describes the four major units of the ‘C3x pipeline structure and their functions:

Fetch unit (F) Fetches the instruction words from memory and updates the program counter (PC).

Decode unit (D) Decodes the instruction word and performs address generation. Also, the decode unit controls modification of the AR*n* registers in the indirect addressing mode and of the stack pointer when PUSH to/POP from the stack occurs.

Read unit (R) If required, reads the operands from memory.

Execute unit (E) If required, reads the operands from the register file, performs the necessary operation, and writes results to the register file. If required, results of previous operations are written to memory.

All instruction executions perform these four basic functions: fetch, decode, read, and execute. Figure 8–1 illustrates these four levels of the pipeline structure. The levels are indexed according to instruction and execution cycle. In the figure, perfect overlap in the pipeline, where all four units operate in parallel, occurs at cycle (*m*). Levels about to be executed are at *m* +1, and those just previously executed are at *m*–1. The ‘C3x pipeline controller supports a high-speed processing rate of one execution per cycle. It also manages pipeline conflicts so that they are transparent to you. You do not need to take any special precautions to ensure correct operation.

Figure 8–1. TMS320C3x Pipeline Structure

CYCLE	Fetch	Decode	Read	Execute	
m–3	W	–	–	–	
m–2	X	W	–	–	
m–1	Y	X	W	–	
m	Z	Y	X	W	← **Perfect overlap**
m+1	–	Z	Y	X	
m+2	–	–	Z	Y	
m+3	–	–	–	Z	

Note: W, X, Y, Z = Instruction representations

’C30 ’C31

For ‘C30 and ‘C31, priorities from highest to lowest have been assigned to each of the functional units of the pipeline and to the DMA controller as follows:

- Execute (highest)
- Read
- Decode
- Fetch
- DMA (lowest)

Despite the DMA controller’s low priority, you can minimize or even eliminate conflicts with the CPU through suitable data structuring because the DMA controller has its own data and address buses.

’C32

In the ‘C32, the DMA has configurable priorities. Therefore, priorities from highest to lowest have been assigned to each of the functioned units of the pipeline and to the DMA controller as follows:

- DMA (if configured with highest priority)
- Execute
- Read
- Decode
- Fetch
- DMA (if configured with lowest priority)

A pipeline conflict occurs when an instruction is being processed, and is ready to pass to the next higher pipeline level while that level is not ready to accept a new input. In this case, the lower priority unit waits until the higher priority unit completes executing the current function.

8.2 Pipeline Conflicts

Pipeline conflicts in the ’C3x can be grouped into the following categories:

Branch conflicts Branch conflicts involve most of those instructions or operations that read and/or modify the PC.

Register conflicts Register conflicts involve delays that can occur when reading from, or writing to, registers that are used for address generation.

Memory conflicts Memory conflicts occur when the internal units of the ’C3x compete for memory resources.

Each of these three types, including examples, is discussed in the following subsections. In these examples, when data is refetched or an operation is repeated, the symbol representing the stage of the pipeline is appended with a number. For example, if a fetch is performed again, the instruction mnemonic is repeated. When an access is detained for multiple cycles because the unit is not ready, the symbol $\overline{\text{RDY}}$ indicates that a unit is not ready and RDY indicates that a unit is ready. If the particular unit does not perform a function, the *nop* label is placed in that stage of the pipeline.

8.2.1 Branch Conflicts

The first class of pipeline conflicts occurs with standard (nondelayed) branches, that is, BR, B*cond*, DB*cond*, CALL, IDLE, RPTB, RPTS, RETI*cond*, RETS*cond*, interrupts, and reset. Conflicts arise with these instructions and operations because, during their execution, the pipeline is used only for the completion of the operation; other information fetched into the pipeline is discarded or refetched, or the pipeline is inactive. This is referred to as flushing the pipeline. Flushing the pipeline is necessary in these cases to ensure that portions of succeeding instructions do not inadvertently get partially executed. TRAP *cond* and CALL*cond* are classified differently from the other types of branches and are considered later.

Example 8–1 shows the code and pipeline operation for a standard branch.

Note: Dummy Fetch

In this example, one dummy fetch (an MPYF instruction) is performed before the branch is decoded. After the branch address is available, a new fetch (an OR instruction) is performed.

Example 8–1. Standard Branch

```
        BR     THREE     ; Unconditional branch
        MPYF             ; Not executed
        ADD              ; Not executed
        SUBF             ; Not executed
        AND              ; Not executed
 .
 .
 .
 THREE  OR               ; Fetched after BR is taken
        STI
 .
 .
 .
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	BR	–	–	–
n+1	MPYF	BR	–	–
n+1	(nop)	(nop)	BR	–
n+1	(nop)	(nop)	(nop)	BR
3	OR	(nop)	(nop)	(nop)
	STI	OR	(nop)	(nop)

Fetch held for new PC value

3 → PC

Note:

Both RPTS and RPTB flush the pipeline, allowing the RS, RE, and RC registers to be loaded at the proper time. If these registers are loaded without the use of RPTS or RPTB, no flushing of the pipeline occurs. Thus, RS, RE, and RC can be used as general-purpose 32-bit registers without pipeline conflicts. When RPTB is nested because of nested interrupts, it may be necessary to load and store these registers directly while using the repeat modes. Since up to four instructions can be fetched before entering the repeat mode, you should follow loads by a branch to flush the pipeline. If the RC is changing when an instruction is loading it, the direct load takes priority over the modification made by the repeat mode logic.

Delayed branches are implemented to ensure the fetching of the next three instructions. The delayed branches include BRD, B*cond*D, and DB*cond*D. Example 8–2 shows the code and pipeline operation for a delayed branch.

Example 8–2. Delayed Branch

```
       BRD   THREE     ; Unconditional delayed branch
       MPYF            ; Executed
       ADD             ; Executed
       SUBF            ; Executed
       AND             ; Not executed
       .
       .
       .
THREE  MPYF            ; Fetched after SUBF is fetched
   .
   .
   .
```

Pipeline Operation

PC	**Fetch**	**Decode**	**Read**	**Execute**
n	BRD	–	–	–
n+1	MPYF	BRD	–	–
n+2	ADDF	MPYF	BRD	–
n+3	SUBF	ADDF	MPYF	BRD
3	MPYF	SUBF	ADDF	MPYF

No execute delay

3 ⟶ PC

8.2.2 Register Conflicts

Register conflicts involve reading or writing registers used for addressing. These conflicts occur when the pertinent register is not ready to be used. Some conditions under which you can avoid register conflicts are discussed in Section 8.3 on page 8-19.

The registers comprise the following three functional groups:

Group 1 This group includes auxiliary registers (AR0–AR7), index registers (IR0, IR1), and block-size register (BK).

Group 2 This group includes the data-page pointer (DP).

Group 3 This group includes the system-stack pointer (SP).

If an instruction writes to one of these three groups, *the decode unit cannot use any register within that particular group until the write is complete, that is, until the instruction execution is completed*. In Example 8–3, an auxiliary register is loaded, and a different auxiliary register is used on the next instruction. Since the decode stage needs the result of the write to the auxiliary register, the decode of this second instruction is delayed two cycles. Every time the decode is delayed, a refetch of the program word is performed; the ADDF is fetched three times. Since these are actual refetches, they can cause not only conflicts with the DMA controller but also cache hits and misses.

A post-/preincrement/decrement of an AR register in an instruction is not considered a write to a register. A write is in the form of an LDF, LDI, LDII, or DB instruction.

Example 8–3. Write to an AR Followed by an AR for Address Generation

```
        LDI    7,AR2     ; 7 → AR2
  NEXT  MPYF   *AR2,R0   ; Decode delayed 2 cycles
        ADDF
        FLOAT
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	LDI	–	–	–
n+1	MPYF	LDI	–	–
n+2	ADDF	MPYF	LDI	–
n+2	ADDF	MPYF	(nop)	LDI 7,AR2
n+2	ADDF	MPYF	(nop)	(nop)
n+3	FLOAT	ADDF	MPYF	(nop)

Decode/address generation held until AR write is completed

ARs written

The case for reads of these groups is similar to the cases for writes. If an instruction must read a member of one of these groups, the use of that particular group by the decode for the following instruction is delayed until the read is complete. The registers are read at the start of the execute cycle and require only a one-cycle delay of the following decode. For four registers (IR0, IR1, BK, or DP), there is no delay. For all other registers, including the SP, the delay occurs.

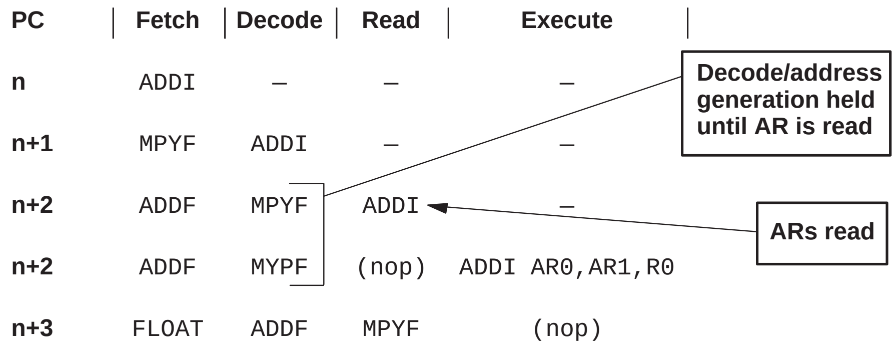
Note that an address generation through the use of an AR register (*AR*n*, *++AR*n*, *–AR*n*, etc.) in an instruction is not considered a read.

In Example 8–4, two auxiliary registers are added together, with the result going to an extended-precision register. The next instruction uses a different auxiliary register as an address register.

Example 8–4. A Read of ARs Followed by ARs for Address Generation

```
      ADDI   AR0,AR1,R1   ; AR0+AR1 → R1
NEXT  MPYF   *++AR2,R0    ; Decode delayed one cycle
      ADDF
      FLOAT
```

Pipeline Operation



PC	Fetch	Decode	Read	Execute
n	ADDI	–	–	–
n+1	MPYF	ADDI	–	–
n+2	ADDF	MPYF	ADDI	–
n+2	ADDF	MYPF	(nop)	ADDI AR0,AR1,R0
n+3	FLOAT	ADDF	MPYF	(nop)

Decode/address generation held until AR is read

ARs read

Note:

Loop counter auxiliary registers for the decrement and branch (DBR) instructions are regarded in the same way as they are for addressing. The operation shown in Example 8–3 and Example 8–4 also can occur for this instruction.

8.2.3 Memory Conflicts

Memory conflicts can occur when the memory bandwidth of a physical memory space is exceeded. For example, RAM blocks 0 and 1 and the ROM block can support only two accesses per cycle. The external interface can support only one access per cycle. Section 8.4, *Memory Access for Maximum Performance*, on page 8-22 contains some conditions under which you can avoid memory conflicts.

Memory pipeline conflicts consist of the following four types:

Program wait A program fetch is prevented from beginning.

Program fetch Incomplete A program fetch has begun but is not yet complete.

Execute only An instruction sequence requires three CPU data accesses in a single cycle.

Hold everything A primary or expansion bus operation must complete before another one can proceed.

These four types of memory conflicts are illustrated in examples and discussed in the paragraphs that follow.

8.2.3.1 Program Wait

Two conditions can prevent the program fetch from beginning:

- The start of a CPU data access when:
  - Two CPU data accesses are made to an internal RAM or ROM block, and a program fetch from the same block is necessary.
  - One of the external ports is starting a CPU data access, and a program fetch from the same port is necessary.
- A multicycle CPU data access or DMA data access over the external bus is needed.

Example 8–5 illustrates a program wait until a CPU data access completes. In this case, *AR0 and *AR1 are both pointing to data in RAM block 0, and the MPYF instruction will be fetched from RAM block 0. This results in the conflict shown in Example 8–5. Because more than two accesses can be made to RAM block 0 in a single cycle, the program fetch cannot begin and must wait until the CPU data accesses are complete.

Example 8–5. Program Wait Until CPU Data Access Completes

```
ADDF3  *AR0,*AR1,R0
FIX
MPYF
ADDF3
NEGB
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	ADDF3	–	–	–
n+1	FIX	ADDF3	–	–
n+2	(wait)	FIX	ADDF3	–
n+2	MPYF	(nop)	FIX	ADDF3
n+3	ADDF3	MPYF	(nop)	FIX
n+4	NEGB	ADDF3	MPYF	(nop)

Fetch held until data access completes

Data accessed

Example 8–6 shows a program wait due to a multicycle data-data access or a multicycle DMA access. The ADDF, MPYF, and SUBF are fetched from some portion in memory other than the external port the DMA requires. The DMA begins a multicycle access. The program fetch corresponding to the CALL is made to the same external port that the DMA is using.

Either of two cases may produce this situation:

❑ One of the following two memory boundaries is crossed:

- From internal memory to external memory
- From one external port to another

❑ Code that has been cached is executed, and the instruction prior to the ADDF is one of the following (conditional or unconditional):

- A delayed branch instruction
- A delayed decrement and branch instruction

Even though the DMA has the lowest priority on ’C30 and ’C31 or when configured as such in the ’C32, multicycle access cannot be aborted. The program fetch must wait until the DMA access completes.

Example 8–6. Program Wait Due to Multicycle Access

```
       ADDF   ; code in internal memory
       MPY    ; code in internal memory
       SUBF   ; code in internal memory
       CALL   ; code in external memory
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute	
n	ADDF	–	–	–	
n+1	MPYF	ADDF	–	–	
n+2	SUBF	MPYF	ADDF	–	**2-cycle DMA access**
n+3	(wait)	SUBF	MPYF	ADDF	
n+3	CALL	(nop)	SUBF	MPYF	
n+4	–	CALL	(nop)	SUBF	

8.2.3.2 Program Fetch Incomplete

A program fetch incomplete occurs when an instruction fetch takes more than one cycle to complete because of wait states. In Example 8–7, the MPYF and ADDF are fetched from memory that supports single-cycle accesses. The SUBF is fetched from memory requiring one wait state. One example that demonstrates this conflict is a fetch across a bank boundary on the primary port. See Section 9.5 on page 9-12.

Example 8–7. Multicycle Program Memory Fetches

Pipeline Operation

PC	Fetch	Decode	Read	Execute	
n	`MPYF`	–	–	–	
n+1	`ADDF`	`MPYF`	–	–	
n+2 $\overline{\text{RDY}}$	`SUBF`	`ADDF`	`MPYF`	–	**1 wait state required**
n+2 RDY	`SUBF`	`(nop)`	`ADDF`	`MPYF`	
n+3	`ADDI`	`SUBF`	`(nop)`	`ADDF`	

Note: PC = program counter

8.2.3.3 Execute Only

The execute-only type of memory pipeline conflict occurs when performing an interlocked load or when a sequence of instructions requires three CPU data accesses in a single cycle. There are two cases in which this occurs:

- An instruction performs a store and is followed by an instruction that performs two memory reads.
- An instruction performs two stores and is followed by an instruction that performs at least one memory read.
- An interlocked load (LDII or LDFI) instruction is performed, and XF1 = 1.

The first case is shown in Example 8–8. Since this sequence requires three data memory accesses and only two are available, only the execute phase of the pipeline is allowed to proceed. The dual reads required by the LDF **||** LDF are delayed one cycle. In this case, a refetch of the next instruction can occur.

Example 8–8. Single Store Followed by Two Reads

```
      STFR  0,*AR1    ; R0 → *AR1
      LDF   *AR2,R1   ; *AR2 → R1 in parallel with
   || LDF   *AR3,R2   ; *AR3 → R2
```

Pipeline Operation

PC	**Fetch**	**Decode**	**Read**	**Execute**
n	STF	–	–	–
n+1	LDF‖LDF	STF	–	–
n+2	W	LDF‖LDF	STF	–
n+3	X	W	LDF‖LDF	STF
n+4	X	W	LDF‖LDF	(nop)
n+4	Y	X	W	LDF‖LDF

Write must complete before the two reads can complete

2 reads performed

Note: W, X, Y = Instruction representations

Example 8–9 shows a parallel store followed by a single load or read. Since two parallel stores are required, the next CPU data-memory read must wait one cycle before beginning. One program-memory refetch can occur.

Example 8–9. Parallel Store Followed by Single Read

```
        STF    R0,*AR0   ;  R0 → *AR0 in parallel with
     || STF    R2,*AR1   ;  R2 → *AR1
        ADDF   @SUM,R1   ;  R1 + @SUM → R1
        IACK
        ASH
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	STF||STF	–	–	–
n+1	ADDF	STF||STF	–	–
n+2	IACK	ADDF	STF||STF	–
n+3	ASH	IACK	ADDF	STF||STF
n+4	ASH	IACK	ADDF	(nop)
n+4	–	ASH	IACK	ADDF

Read must wait until the writes are completed

Writes performed

The final case involves an interlocked load (LDII or LDFI) instruction and XF1 = 1. Since the interlocked loads use the XF1 pin as an acknowledge that the read can complete, the loads might need to extend the read cycle, as shown in Example 8–10. A program refetch can occur.

Example 8–10. Interlocked Load

```
NOT    R1,R0
LDII   300h,AR
2
ADDI   *AR2,R2
CMPI   R0,R2
```

Pipeline Operation

PC	XF1	Fetch	Decode	Read	Execute
n	1	NOT	–	–	—
n+1	1	LDII	NOT	–	—
n+2	1	ADDI	LDII	NOT	—
n+3	1	CMPI	ADDI	LDII	NOT
n+3	1	–	CMPI	ADDI	LDII
n+4	0	–	CMPI	ADDI	LDII

XF1 = 1, read must wait

XF1 = 0, read operation is complete

8.2.3.4 Hold Everything

Three situations result in hold-everything memory pipeline conflicts:

- ❑ A CPU data load or store cannot be performed because an external port is busy.
- ❑ An external load takes more than one cycle.
- ❑ Conditional calls and traps, which take one more cycle than conditional branches, are processed.

The first type of hold-everything conflict occurs when one of the external ports is busy because an access has started, but is not complete. In Example 8–11, the first store is a 2-cycle store. The CPU writes the data to an external port. The port control then takes two cycles to complete the data-data write. The LDF is a read over the same external port. Since the store is not complete, the CPU continues to attempt LDF until the port is available.

Example 8–11. Busy External Port

```
                         STF    R0,@DMA1
                         LDF    @DMA2,R0
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	STF	–	–	–
n+1	LDF	STF	–	–
n+2	W	LDF	STF	–
n+2	W	LDF	(nop)	STF
n+2	W	LDF	(nop)	(nop)
n+3	X	W	LDF	(nop)
n+4	Y	X	W	LDF

2-cycle external bus write access

Note: W, X, Y = Instruction representations

The second type of hold-everything conflict involves multicycle data reads. The read has begun and continues until completed. In Example 8–12, the LDF is performed from an external memory that requires several cycles to access.

Example 8–12. Multicycle Data Reads

```
LDF    @DMA,R0
```

Pipeline Operation

PC	**Fetch**	**Decode**	**Read**	**Execute**	
n	LDF	–	–	–	
n+1	I	LDF	–	–	
n+2	J	I	LDF	–	**2-cycle external bus read access**
n+3	K(dummy)	I	LDF	–	
n+3	K_2	J	I	LDF	

Note: I, J, K = Instruction representations

The final type of hold-everything conflict deals with conditional calls (CALL*cond*) and traps (TRAP*cond*), which are different from other branch instructions. Whereas other branch instructions are conditional loads, the conditional calls and traps are conditional stores, which take one more cycle to complete than conditional branches (see Example 8–13). The added cycle pushes the return address after the call condition is evaluated.

Example 8–13. Conditional Calls and Traps

Pipeline Operation

PC	**Fetch**	**Decode**	**Read**	**Execute**
n	`CALL`*`cond`*	–	–	–
n+1	`I`	`CALL`*`cond`*	–	–
n+1	`(nop)`	`(nop)`	`CALL`*`cond`*	–
n+1	`(nop)`	`(nop)`	`(nop)`	`CALL`*`cond`*
n+1	`(nop)`	`(nop)`	`(nop)`	`CALL`*`cond`*
n+2/CALLaddr	`I`	`(nop)`	`(nop)`	`(nop)`

PC store cycle

Note: I = Instruction representation

8.3 Resolving Register Conflicts

If the auxiliary registers (AR7–AR0), the index registers (IR1–IR0), data-page pointer (DP), or stack pointer (SP) are accessed for any reason other than address generation, pipeline conflicts associated with the next memory access can occur. The pipeline conflicts and delays are presented in Section 8.2 on page 8-4.

Example 8–14, Example 8–15, and Example 8–16 demonstrate some common uses of these registers that do not produce a conflict or ways that you can avoid the conflict.

Example 8–14. Address Generation Update of an AR Followed by an AR for Address Generation

```
LDF    7.0,R0    ; 7.0 → R0
MPYF   *++AR0(IR1),R0
ADDF   *AR2,R0
FIX
MPYF
ADDF
```

Pipeline Operation

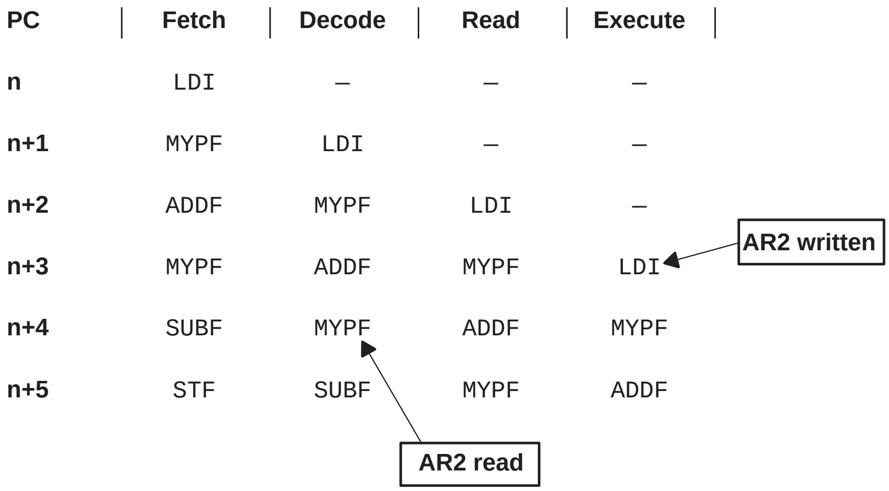
PC	Fetch	Decode	Read	Execute	
n	LDF	–	–	–	
n+1	MYPF	LDF	–	–	
n+2	ADDF	MYPF	LDF	–	
n+3	FIX	ADDF	MYPF	LDF	**ARs read**
n+4	MPYF	FIX	ADDF	MYPF	
n+5	ADDF	MYPF	FIX	ADDF	

Note: W, X, Y, Z = Instruction representations

Example 8–15. Write to an AR Followed by an AR for Address Generation Without a Pipeline Conflict

```
LDI    @TABLE,AR2
MPYF   @VALUE,R1
ADDF   R2,R1
MPYF   *AR2++,R1
SUBF
STF
```

Pipeline Operation



PC	Fetch	Decode	Read	Execute
n	LDI	—	—	—
n+1	MYPF	LDI	—	—
n+2	ADDF	MYPF	LDI	—
n+3	MYPF	ADDF	MYPF	LDI
n+4	SUBF	MYPF	ADDF	MYPF
n+5	STF	SUBF	MYPF	ADDF

AR2 written

AR2 read

Example 8–16. Write to DP Followed by a Direct Memory Read Without a Pipeline Conflict

```
LDP    TABLE_ADDR
POP    R0
LDF    *–AR3(2),R1
LDI    @TABLE_ADDR,AR0
PUSHF  R6
PUSH   R4
```

Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	LDP	–	–	–
n+1	POP	LDP	–	–
n+2	LDF	POP	LDP	–
n+3	LDI	LDF	POP	LDP
n+4	PUSHF	LDI	LDF	POP
n+5	PUSH	PUSHF	LDI	LDF

DP written

DP read

8.4 Memory Access for Maximum Performance

If program fetches and data accesses are performed so that the resources being used cannot provide the necessary bandwidth, the pipeline is stalled until the data accesses are complete. Certain configurations of program fetch and data accesses yield conditions under which the ’C3x can achieve maximum throughput.

Table 8–1 shows how many accesses can be performed from the different memory spaces when it is necessary to do a program fetch and a single data access and still achieve maximum performance (one cycle). Four cases achieve 1-cycle maximization.

Table 8–1. One Program Fetch and One Data Access for Maximum Performance

Case No.	Primary Bus Accesses	Accesses From Dual Access Internal Memory	Expansion Bus[†] or Peripheral Accesses
1	1	1	—
2	1	—	1
3	—	2 from any combination of internal memory	—
4	—	1	1

[†] The expansion bus is available only on the ’C30.

Table 8–2 shows how many accesses can be performed from the different memory spaces when it is necessary to do a program fetch and two data accesses and still achieve maximum performance (one cycle). Six conditions achieve this maximization.

Table 8–2. One Program Fetch and Two Data Accesses for Maximum Performance

Case No.	Primary Bus Accesses	Accesses From Dual-Access Internal Memory	Expansion[†] Or Peripheral Bus Accesses
1	1	2 from any combination of internal memory	—
2	1 program	1 data	1 data
3	1 data	1 data	1 program
4	1 data	1 program, 1 data	1 DMA
5	—	2 from same internal memory block and 1 from a different internal memory block	—
6	—	3 from different internal memory blocks	—
7	—	2 from any combination of internal memory	1
8	1 program	2 data	1 DMA
9	1 DMA	2 data	1 program

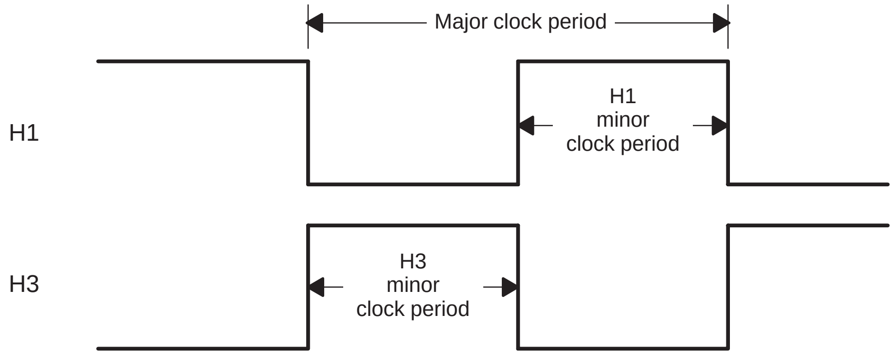
[†] The expansion bus is available only on the ’C30.

8.5 Clocking Memory Accesses

This section discusses the role of internal clock phases (H1 and H3) and how the ’C3x handles multiple-memory accesses. The previous section discusses the interaction between sequences of instructions; this section discusses the flow of data on an individual instruction basis.

Each major clock period of 33.3 ns is composed of two minor clock periods of 16.67 ns, labeled H3 and H1. The active clock period for H3 and H1 is the time when that signal is high. See Figure 8–2.

Figure 8–2. Minor Clock Periods



The precise operation of memory reads and writes can be defined according to these minor clock periods. The types of memory operations that can occur are program fetches, data loads and stores, and DMA accesses.

8.5.1 Program Fetches

Internal program fetches are always performed during H3 unless a single data store must occur at the same time due to another instruction in the pipeline. In that case, the program fetch occurs during H1 and the data store occurs during H3.

External program fetches always start at the beginning of H3 with the address being presented on the external bus. At the end of H1, the fetches are completed with the latching of the instruction word.

8.5.2 Data Loads and Stores

Four types of instructions perform loads, memory reads, and stores:

- ❑ 2-operand instructions
- ❑ 3-operand instructions
- ❑ Multiplier/ALU operation with store instructions
- ❑ Parallel multiply and add instructions

See Chapter 6, *Addressing Modes*, for more information.

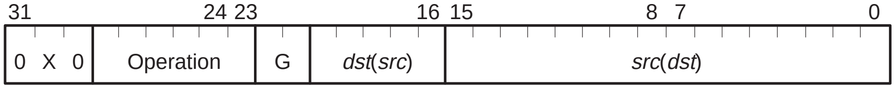
As discussed in Chapter 7, the number of bus cycles for external memory accesses differs in some cases from the number of CPU execution cycles. For external reads, the number of bus cycles and CPU execution cycles is identical. For external writes, there are always at least two bus cycles, but unless there is a port-access conflict, there is only one CPU execution cycle. In the following examples, any difference in the number of bus cycles and CPU cycles is noted.

8.5.2.1 2-Operand Instruction Memory Accesses

All instructions whose bits 31–29 are 000 or 010 (see Figure 8–3) are 2-operand instructions. In the case of a data read, bits 15–0 represent the *src* operand. Internal data reads are always performed during H1. External data reads always start at the beginning of H3 with the address presented on the external bus; they complete with the latching of the data word at the end of H1.

In the case of a data store, bits 15–0 represent the *dst* operand. Internal data stores are performed during H3. External data stores always start at the beginning of H3 with the address and data being presented on the external bus.

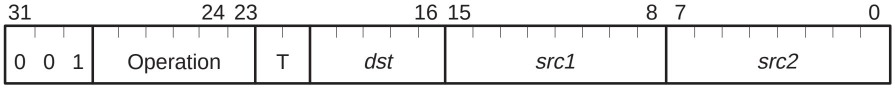
Figure 8–3. 2-Operand Instruction Word



8.5.2.2 3-Operand Instruction Memory Reads

All instructions whose bits 31–29 are 001 (see Figure 8–4) are 3-operand instructions. The source operands, *src1* and *src2*, come from either registers or memory. When one or more of the source operands are from memory, these instructions are always memory reads.

Figure 8–4. 3-Operand Instruction Word



If only one of the source operands is from memory (either *src1* or *src2*) and is located in internal memory, the data is read during H1. If the single memory source operand is in external memory, the read starts at the beginning of H3, with the address presented on the external bus, and completes with the latching of the data word at the end of H1.

If both source operands are to be fetched from memory, then memory reads can occur in several ways:

❑ If both operands are located in internal memory, the *src1* read is performed during H3 and the *src2* read during H1, completing two memory reads in a single cycle.

❑ If *src1* is in internal memory and *src2* is in external memory, the *src2* access begins at the start of H3 and latches at the end of H1. At the same time, the *src1* access to internal memory is performed during H3. Again, two memory reads are completed in a single cycle.

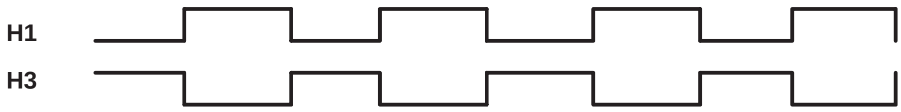
❑ If *src1* is in external memory and *src2* is in internal memory, two cycles are necessary to complete the two reads. In the first cycle, both operands are addressed. Since *src1* takes an entire cycle to be read and latched from external memory, the internal operation on *src2* cannot be completed until the second cycle. Ordering the operands so that *src1* is located internally is necessary to achieve single-cycle execution.

❑ If *src1* and *src2* are both from external memory, two cycles are required to complete the two reads. In the first cycle, the *src1* access is performed and loaded on the next H3; in the second cycle, the *src2* access is performed and loaded on that cycle’s H1.

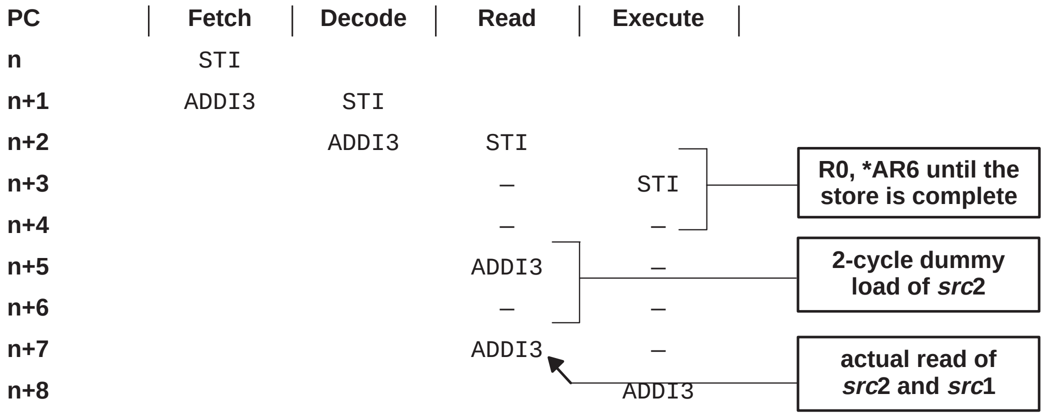
If *src2* is in external memory and *src1* is in on-chip or external memory and is immediately preceded by a single store instruction to external memory, a dummy *src2* read can occur between the execution of the store instruction and the *src2* read, regardless of which memory space is accessed ($\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$, or $\overline{\text{IOSTRB}}$). The dummy read can cause an externally interfaced FIFO address pointer to be incremented prematurely, thereby causing the loss of FIFO data. Example 8–17 illustrates how the dummy read can occur. Example 8–18 offers an alternative code segment that suppresses the dummy read. In the alternative code segment, the dummy read is eliminated by swapping the order of the source operands.

Example 8–17. Dummy sr2 Read

```
STI       R0,*AR6         ; AR6 points to MSTRB space
ADDI3     *AR1,*AR3,R0    ; AR3 points to on-chip RAM (src1)
                          ; AR1 points to MSTRB space (src2)
```



Pipeline Operation



PC	Fetch	Decode	Read	Execute
n	STI			
n+1	ADDI3	STI		
n+2		ADDI3	STI	
n+3			–	STI
n+4			–	–
n+5			ADDI3	–
n+6			–	–
n+7			ADDI3	–
n+8				ADDI3

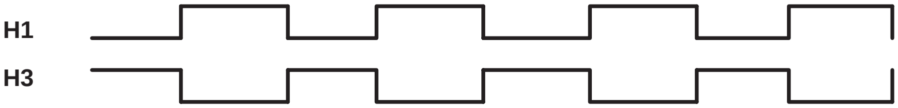
R0, *AR6 until the store is complete
2-cycle dummy load of *src*2
actual read of *src*2 and *src*1

Two cycles are required for the $\overline{\text{MSTRB}}$ store. Two additional cycles are required for the dummy $\overline{\text{MSTRB}}$ read of *AR3 (because a read follows a write). One cycle is required for an actual $\overline{\text{MSTRB}}$ read of *AR3.

Example 8–18. Operand Swapping Alternative

Switch the operands of the 3-operand instruction so that the internal read is performed first.

```
STI      R0,*AR6         ; AR6 points to MSTRB space
ADDI3    *AR3,*AR1,R0    ; AR3 points to on-chip RAM (src2)
                         ; AR1 points to MSTRB space (src1)
```



Pipeline Operation

PC	Fetch	Decode	Read	Execute
n	STI			
n+1	ADDI3	STI		
n+2		ADDI3	STI	
n+3			–	STI
n+4			–	–
n+5			ADDI3	–
n+6			–	–
n+7			–	ADDI3
n+8				ADDI3

2-cycle store

The read of *src*2 cannot start until the store is complete

2-cycle read of *src*1 and *src*2

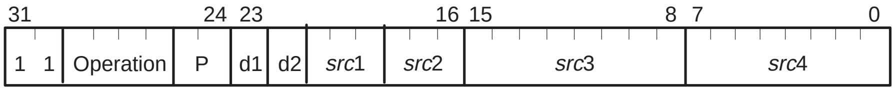
8.5.2.3 Operations with Parallel Stores

The next class of instructions includes every instruction that has a store in parallel with another instruction. Bits 31 and 30 for these instructions are equal to 1 1.

The instruction word format for operations that perform a multiply or ALU operation in parallel with a store is shown in Figure 8–5. If the store operation to *dst2* is external or internal, it is performed during H3. Two bus cycles are required for external stores, but only one CPU cycle is necessary to complete the write.

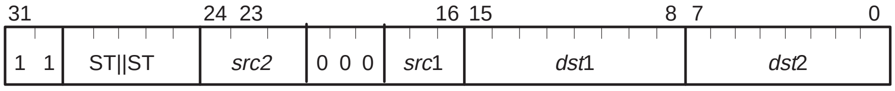
If the memory read operation is external, it starts at the beginning of H3 and latches at the end of H1. If the memory read operation is internal, it is performed during H1. Note that memory reads are performed by the CPU during the read (R) phase of the pipeline, and stores are performed during the execute (E) phase.

Figure 8–5. Multiply or CPU Operation With a Parallel Store



The instruction word format for instructions that have parallel stores to memory is shown in Figure 8–6. If both destination operands, *dst1* and *dst2*, are located in internal memory, *dst1* is stored during H3 and *dst2* during H1, thus completing two memory stores in a single cycle.

Figure 8–6. Two Parallel Stores



- ❑ If *dst1* is in external memory and *dst2* is in internal memory, the *dst1* store begins at the start of H3. The *dst2* store to internal memory is performed during H1. Two bus cycles are required for the external store, but only one CPU cycle is necessary to complete the write. Again, two memory stores are completed in a single cycle.
- ❑ If *dst1* is in internal memory and *dst2* is in external memory, an additional bus cycle is necessary to complete the *dst2* store. Only one CPU cycle is necessary to complete the write, but the port access requires three bus cycles. In the first cycle, the internal *dst1* store is performed during H3, and *dst2* is written to the port during H1. During the next cycle, the *dst2* store is performed on the external bus, beginning in H3, and executes as normal through the following cycle.

- If *dst1* and *dst2* are both written to external memory, a single CPU cycle is still all that is necessary to complete the stores. In this case, four bus cycles are required.

  1) In the first cycle, both *dst1* and *dst2* are written to the port, and the external-bus access for *dst1* begins.

  a) The store for *dst1* is completed on the second cycle.

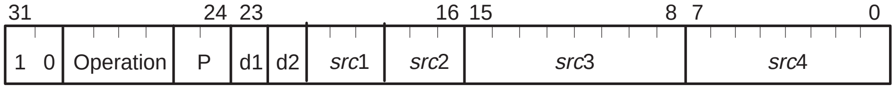
  b) The store for *dst2* begins on the third external-bus cycle.

  c) The store for *dst2* is completed on the fourth external-bus cycle.

8.5.2.4 Parallel Multiplies and Adds

Memory addressing for parallel multiplies and adds is similar to that for 3-operand instructions. The parallel multiplies and adds include all instructions with bits 31–30 = 10 (see Figure 8–7).

Figure 8–7. Parallel Multiplies and Adds



For these operations, *src3* and *src4* are both located in memory. If both operands are located in internal memory, *src3* is performed during H3, and *src4* is performed during H1, thus completing two memory reads in a single cycle.

- If *src3* is in internal memory and *src4* is in external memory, the *src4* access begins at the start of H3 and latches at the end of H1. At the same time, the *src3* access to internal memory is performed during H3. Again, two memory reads are completed in a single cycle.

- If *src3* is in external memory and *src4* is in internal memory, two cycles are necessary to complete the two reads. In the first cycle, the internal *src4* access is performed. During the H3 of the next cycle, the *src3* access is performed.

- If *src3* and *src4* are both from external memory, two cycles are necessary to complete the two reads. In the first cycle, the *src3* access is performed; in the second cycle, the *src4* access is performed.

Chapter 9

TMS320C30 and TMS320C31 External-Memory Interface

This chapter describes the ’C30 and ’C31 external-memory interface. See Chapter 10, *Enhanced External-Memory Interface*, for detailed information on the ’C32 external bus operation.

’C30

Memories and external peripheral devices are accessible through two external interfaces on the ’C30:

- ❑ Primary bus
- ❑ Expansion bus

’C31

On the ’C31, one bus, the primary bus, is available to access external memories and peripheral devices. You can control wait-state generation, permitting access to slower memories and peripherals, by manipulating memory-mapped control registers associated with the interfaces and by using an external input signal.

Topic **Page**

9.1 Overview

The ’C30 provides two external interfaces: the primary bus and the expansion bus. The TMS320C31 provides one external interface: the primary bus. The primary bus consists of a 32-bit data bus, a 24-bit address bus, and a set of control signals. The expansion bus consists of a 32-bit data bus, a 13-bit address bus, and a set of control signals. Each interface has the following features:

- Separate configurations controlled by memory-mapped external interface control registers
- Hold request and acknowledge signal for putting the external memory interface signals in high impedance mode and preventing the processor from accessing the external bus
- Selectable wait state that can be controlled through software, hardware, or combination of software and hardware
- Unified memory space for data, program, and I/O access

9.2 Memory Interface Signals

This section describes the differences between the ’C30 and ’C31 memory interface signals.

9.2.1 TMS320C30 Memory Interface Signals

The TMS320C30 has two sets of control signals as follows:

- Primary bus control signals: $\overline{\text{STRB}}$, R/$\overline{\text{W}}$, $\overline{\text{HOLD}}$, $\overline{\text{HOLDA}}$, $\overline{\text{RDY}}$

  Table 9–1 lists and describes the signals.

- Expansion bus control signals: $\overline{\text{MSTRB}}$, $\overline{\text{IOSTRB}}$, XR/$\overline{\text{W}}$, $\overline{\text{XRDY}}$

  Table 9–2 lists and describes the expansion bus control signals.

Access is determined by an active strobe signal ($\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$, or $\overline{\text{IOSTRB}}$). When a primary bus access is performed, $\overline{\text{STRB}}$ is low. The expansion bus of the ’C30 supports two types of accesses:

- Memory access signaled by $\overline{\text{MSTRB}}$ low. The timing for an $\overline{\text{MSTRB}}$ access is the same as that of the $\overline{\text{STRB}}$ access on the primary bus.

- External peripheral device access is signaled by $\overline{\text{IOSTRB}}$ low.

Each of the buses (primary and expansion) has an associated control register. These registers are memory-mapped as shown in Figure 9–1.

9.2.2 TMS320C31 Memory Interface Signals

The TMS320C31 has one set of control signals:

- Primary bus control signals: $\overline{\text{STRB}}$, R/$\overline{\text{W}}$, $\overline{\text{HOLD}}$, $\overline{\text{HOLDA}}$, $\overline{\text{RDY}}$

$\overline{\text{STRB}}$ is low when an external bus access is performed. The primary bus control register controls its behavior (see Section 9.3).

Table 9–1. Primary Bus Interface Signals

’C30 ’C31

Signal	Type[†]	Description	Value After Reset	Idle Status
$\overline{\text{STRB}}$	O/Z	Primary interface access strobe	1	1
R/$\overline{\text{W}}$	O/Z	Specifies memory read (active high) or write (active low) mode	1	1
$\overline{\text{HOLD}}$	I	Hold external memory interface	NA[‡]	Ignored
$\overline{\text{HOLDA}}$	O/Z	Hold acknowledge for external memory interface	1	1
$\overline{\text{RDY}}$	I	Indicates external primary interface is ready to be accessed	NA[‡]	Ignored
A (23–0)	O/Z	Primary address bus. When the primary bus address lines are not in high-impedance state due to $\overline{\text{HOLD}}$ signal, they keep in the last external primary bus access.	HI	Address of last external bus access
D (31–0)	I/O/Z	Primary data bus. These signals go to high-impedance between write accesses.	HIZ	HIZ

[†] I Input
O Output
Z High impedance

[‡] NA means not affected.

Table 9–2. Expansion Bus Interface Signals

’C30

Signal	Type[†]	Description	Value After Reset	Idle Status
$\overline{\text{MSTRB}}$	O/Z	Expansion bus memory access strobe	1	1
$\overline{\text{IOSTRB}}$	O/Z	Expansion bus peripheral-access strobe	1	1
XR/$\overline{\text{W}}$	O/Z	Specifies memory (active high) or write (active low) mode	1	1
$\overline{\text{XRDY}}$	I	Indicates external expansion interface is ready to be accessed	NA[‡]	Ignored
XA (12–0)	O	Expansion address bus. When the expansion bus address lines are not in high-impedance state due to $\overline{\text{HOLD}}$ signal, they keep the last external expansion bus access.	HI	Address of last external expansion bus access
XD (31–0)	I/O/Z	Expansion data bus. These signals go to high-impedance between write accesses.	HIZ	HIZ

[†] I Input
O Output
Z High impedance

[‡] NA means not affected.

Figure 9–1. Memory-Mapped External Interface Control Registers

’C30 ’C31

Peripheral Address	
808060h	Expansion-bus control (’C30 only)
808061h	Reserved
808062h	Reserved
808063h	Reserved
808064h	Primary-bus control (’C30, ’C31)
808065h	Reserved
808066h	Reserved
808067h	Reserved
808068h	Reserved
808069h	Reserved
80806Ah	Reserved
80806Bh	Reserved
80806Ch	Reserved
80806Dh	Reserved
80806Fh	Reserved

9.3 Memory Interface Control Registers

Two memory interface control registers, the primary-bus control register and the expansion-bus control register, are described in this section.

9.3.1 Primary-Bus Control Register

The primary bus control register is a 32-bit register that contains the control bits for the primary bus (see Figure 9–2). Table 9–3 describes the register bits with the bit names and functions.

Figure 9–2. Primary-Bus Control Register

’C30 ’C31

31–16	15–13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	BNKCMP					WTCNT			SWW		HIZ	NOHOLD	HOLDST
		R/W					R/W			R/W		R/W	R/W	R

Notes: 1) xx = reserved bit, read as 0
2) R = read, W = write

> **Note:**
>
> After changing the bit fields of the primary-bus control register, up to three instructions are fetched before the primary bus is reconfigured because the configuration change is performed in the execute stage of the pipeline.

Table 9–3. Primary-Bus Control Register Bits

Abbreviation	Reset Value	Name	Description
HOLDST	0	Hold status bit	This bit signals whether the port is being held (HOLDST = 1) or is not being held (HOLDST = 0). This status bit is valid whether the port has been held through hardware or software.
NOHOLD	0	Port hold signal	NOHOLD allows or disallows the port to be held by an external HOLD signal. When NOHOLD = 1, the 'C3x takes over the external bus and controls it, regardless of serviced or pending requests by external devices. No hold acknowledge (HOLDA) is asserted when a HOLD signal is received. It is asserted if an internal hold is generated (HIZ = 1).
HIZ	0	Internal hold	When set (HIZ = 1), the port is put in hold mode. This is equivalent to the external HOLD signal. By forcing a high-impedance condition, the 'C3x can relinquish the external-memory port through software. HOLDA goes low when the port is placed in the high-impedance state.
SWW	11	Software wait mode	In conjunction with WTCNT, this 2-bit field defines the mode of wait-state generation. (See Table 9–5.)
WTCNT	111	Software wait mode	This three-bit field specifies the number of cycles to use when in software wait mode for the generation of internal wait states. The range is 0 (WTCNT = 0 0 0) to 7 (WTCNT=1 1 1) H1/H3 cycles. (See Section 9.4.)
BNKCMP	10000	Bank compare	This 5-bit field specifies the number of MSBs of the address to be used to define the bank size. (See Table 9–6.)

9.3.2 Expansion-Bus Control Register

The expansion-bus control register is a 32-bit register that contains control bits for the expansion bus (see Figure 9–3 and Table 9–4).

Figure 9–3. Expansion-Bus Control Register

’C30

31–16	15–12	11–8	7	6	5	4	3	2	1	0
xx	xx	xx	WTCNT			SWW		xx	xx	xx
			R/W			R/W				

Notes: 1) xx = reserved bit, read as 0
2) R = read, W = write

Table 9–4. Expansion-Bus Control Register Bits

Abbreviation	Reset Value	Name	Description
SWW	11	Software wait mode	In conjunction with the WTCNT, 2-bit field defines the mode of wait-state generation. (See Table 9–5.)
WTCNT	111	Software wait mode	This 3-bit field specifies the number of cycles to use when in software wait mode for the generation of internal wait state. The range is 0 (WTCNT = 0 0 0) to 7 (WTCNT = 1) H1/H3 cycles. (See Section 9.4.)

Note:

After changing the bit fields of the expansion-bus control register, up to three instructions are fetched before the expansion bus is reconfigured because the configuration change is performed in the execute stage of the pipeline.

9.4 Programmable Wait States

The ’C3x has its own internal software-configurable ready-generation capability for each strobe. This software wait-state generator is controlled by configuring two bit fields in the primary or expansion bus interface control registers.

Use the WTCNT field to specify the number of software wait-states to generate and use the SWW field to select one of the following four modes of wait-state generation:

- External $\overline{\text{RDY}}$ wait states are generated solely by the external $\overline{\text{RDY}}$ line ignoring software wait states.
- WTCNT-generated $\overline{\text{RDY}}_{\text{wtcnt}}$ wait states are generated solely by the software wait-state generator ignoring external $\overline{\text{RDY}}$ signals.
- Logical-AND of $\overline{\text{RDY}}$ and $\overline{\text{RDY}}_{\text{wtcnt}}$ wait states are generated with a logical AND of internal and external ready signals. Both signals must occur.
- Logical-OR of $\overline{\text{RDY}}$ and $\overline{\text{RDY}}_{\text{wtcnt}}$ wait states are generated with a logical OR of internal and external ready signals. Either signal can generate the ready signal.

The four modes are used to generate the internal ready signal, $\overline{\text{RDY}}_{\text{int}}$, that controls accesses. As long as $\overline{\text{RDY}}_{\text{int}} = 1$, the current external access is delayed. When $\overline{\text{RDY}}_{\text{int}} = 0$, the current access completes. Since the use of programmable wait states for both external interfaces is identical, only the primary bus interface is described in the following paragraphs.

$\overline{\text{RDY}}_{\text{wtcnt}}$ is an internally-generated ready signal. When an external access is begun, the value in WTCNT is loaded into a counter. WTCNT can be any value from 0 through 7. The counter is decremented every H1/H3 clock cycle until it becomes 0. Once the counter is set to 0, it remains set to 0 until the next access. While the counter is nonzero, $\overline{\text{RDY}}_{\text{wtcnt}} = 1$. While the counter is 0, $\overline{\text{RDY}}_{\text{wtcnt}} = 0$.

Table 9–5 shows the truth table for each value of SWW and the different combinations of $\overline{\text{RDY}}$, $\overline{\text{RDY}}_{\text{wtcnt}}$, and $\overline{\text{RDY}}_{\text{int}}$.

Note:

At reset, the ’C3x is programmed with seven wait states for each external memory access. These wait states are inserted to ensure the system can function with slow memories. To maximize system performance when accessing external memories, you need to decrease the number of wait states.

After changing the wait states, up to three instructions are fetched before the change in the wait state occurs.

Table 9–5. Wait-State Generation

	Inputs		Output	
SWW Bit Field	**/RDYext**	**/RDYwtcnt**	**/RDYint**	**Functional Description**
00	0	x	0	Wait until external RDY is signaled
	1	x	1	
01	x	0	0	Wait until internal wait state generator counts down to 0
	x	1	1	
10	0	0	0	Wait until first signal: external RDY or the internal wait state generator (logical OR)
	0	1	0	
	1	0	0	
	1	1	1	
11	0	0	0	Wait until both external RDY is signaled and wait state generator counts down to 0 (logical AND)
	0	1	1	
	1	0	1	
	1	1	1	

9.5 Programmable Bank Switching

Programmable bank switching allows you to switch between external memory banks without having to insert wait states externally due to memories that require several cycles to turn off. Bank switching is implemented on the primary bus only.

The size of a bank is determined by the number of bits specified by the BNKCMP field of the primary bus control register. For example, if BNKCMP = 16, the 16 MSBs of the address are used to define a bank (see Figure 9–4). Since addresses are 24 bits, the bank size is specified by the eight LSBs, yielding a bank size of 256 words. If BNKCMP ≥ 16, only the 16 MSBs are compared. Bank sizes from $2^8 = 256$ to $2^{24} = 16M$ are allowed. Table 9–6 summarizes the relationship between BNKCMP, the address bits used to define a bank, and the resulting bank size.

Figure 9–4. BNKCMP Example

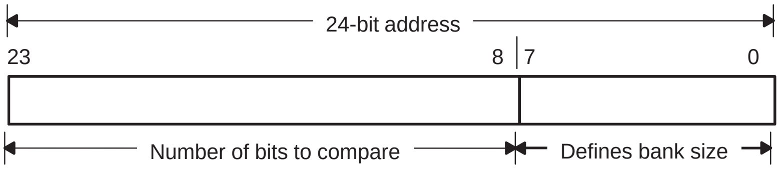


Table 9–6. BNKCMP and Bank Size

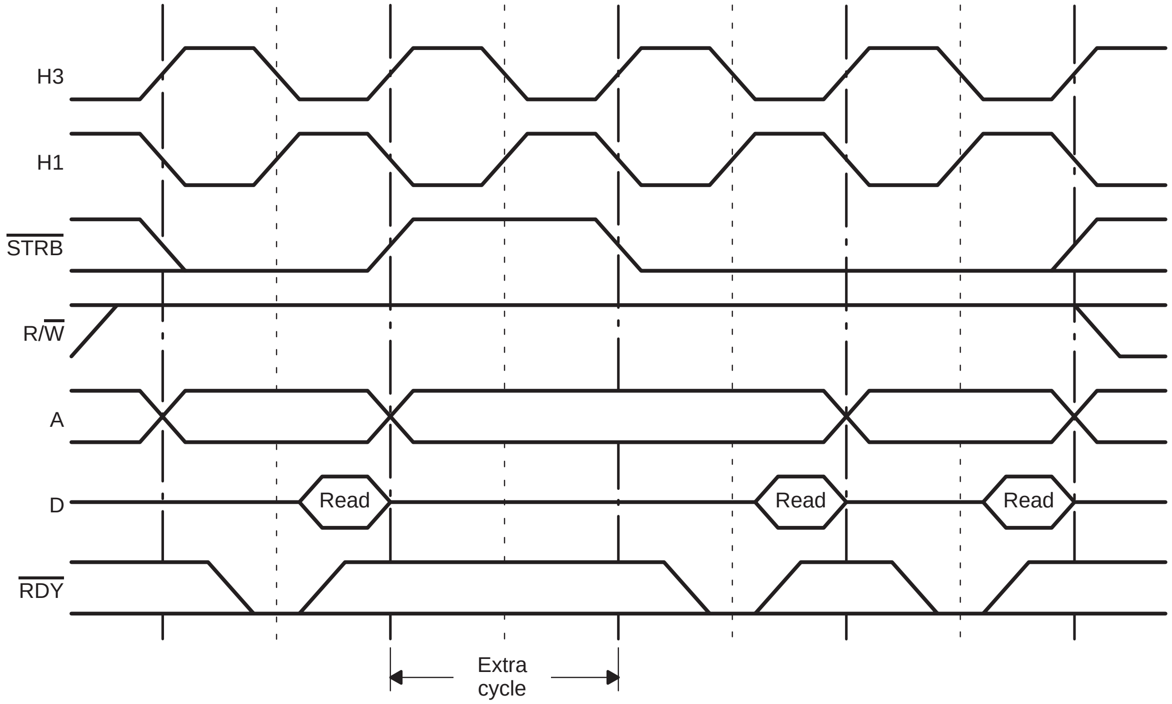
BNKCMP	MSBs Defining a Bank	Bank Size (32-Bit Words)
00000	None	2^{24} = 16M
00001	23	2^{23} = 8M
00010	23–22	2^{22} = 4M
00011	23–21	2^{21} = 2M
00100	23–20	2^{20} = 1M
00101	23–19	2^{19} = 512K
00110	23–18	2^{18} = 256K
00111	23–17	2^{17} = 128K
01000	23–16	2^{16} = 64K
01001	23–15	2^{15} = 32K
01010	23–14	2^{14} = 16K
01011	23–13	2^{13} = 8K
01100	23–12	2^{12} = 4K
01101	23–11	2^{11} = 2K
01110	23–10	2^{10} = 1K
01111	23–9	2^{9} = 512
10000	23–8	2^{8} = 256
10001–11111	Reserved	Undefined

The ’C3x has an internal register that contains the MSBs (as defined by the BNKCMP field) of the last address used for a read or write over the primary interface. At reset, the register bits are set to 0. If the MSBs of the address being used for the current primary interface read do not match those contained in this internal register, a read cycle is not asserted for one H1/H3 clock cycle. During this extra clock cycle, the address bus switches over to the new address, but $\overline{\text{STRB}}$ is inactive (high). The contents of the internal register are replaced with the MSBs being used for the current read of the current address. If the MSBs of the address being used for the current read match the bits in the register, a normal read cycle takes place.

If repeated reads are performed from the same memory bank, no extra cycles are inserted. When a read is performed from a different memory bank, an extra cycle is inserted. This feature can be disabled by setting BNKCMP to 0. The insertion of the extra cycle occurs only when a read is performed. The changing of the MSBs in the internal register occurs for all reads and writes over the primary interface.

Figure 9–5 shows the addition of an inactive cycle when switches between banks of memory occur.

Figure 9–5. Bank-Switching Example



Note:

After changing BNKCMP, up to three instructions are fetched before the change in the bank size occurs.

9.6 External Memory Interface Timing

This section discusses functional timing of operations on the primary bus and the expansion bus, the two independent parallel buses or the ’C3x devices.

The parallel buses implement three mutually exclusive address spaces distinguished through the use of three separate control signals: $\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$, and $\overline{\text{IOSTRB}}$. The $\overline{\text{STRB}}$ signal controls accesses on the primary bus, and the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals control accesses on the expansion bus. Since the two buses are independent, you can make two accesses in parallel.

With the exception of bank switching and the external $\overline{\text{HOLD}}$ function (discussed later in this section), timing of primary bus cycles and $\overline{\text{MSTRB}}$ expansion bus cycles are identical and are discussed collectively. The abbreviation ($\overline{\text{M)STRB}}$ is used in references that pertain equally to $\overline{\text{STRB}}$ and $\overline{\text{MSTRB}}$. Similarly, (X)R/$\overline{\text{W}}$, (X)A, (X)D, and $\overline{\text{(X)RDY}}$ are used to symbolize the equivalent primary and expansion bus signals. The $\overline{\text{IOSTRB}}$ expansion bus cycles are timed differently and are discussed independently.

9.6.1 Primary-Bus Cycles

All bus cycles comprise integral numbers of H1 clock cycles. One H1 cycle is defined to be from one falling edge of H1 to the next falling edge of H1. For full-speed (zero wait-state) accesses, writes require two H1 cycles and reads require one cycle; however, if the read follows a write, the read requires two cycles.This applies to both the primary bus and the $\overline{\text{MSTRB}}$ expansion bus access.

Note: Posted Write

The data written to external memory by CPU or DMA is “latched” into the bus logic, allowing the CPU to continue with internal operation. Consequently, writes to external memory effectively require only one cycle if no accesses to that interface are in progress. However, if the next DMA or CPU access is to the same external bus, the DMA or CPU waits and the write is considered a 2-cycle operation. This is normally referred to as posted-write.

The following discussions pertain to zero wait-state accesses unless otherwise specified.

The $\overline{\text{(M)STRB}}$ signal is low for the active portion of both reads and writes. The active portion lasts one H1 cycle. Additionally, before and after the active portion ($\overline{\text{(M)STRB}}$ low) of writes only, there is a transition cycle of H1. This transition cycle consists of the following sequence:

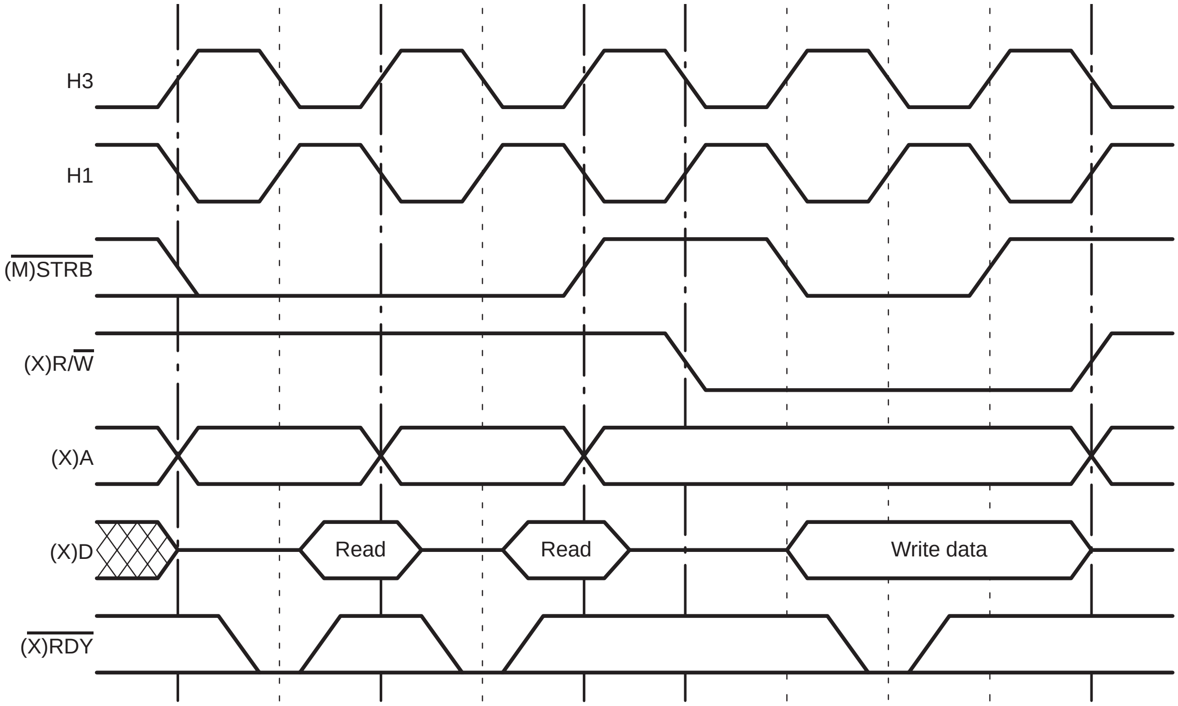
1) $\overline{\text{(M)STRB}}$ is high.

2) If required, (X)R/$\overline{\text{W}}$ changes state on H1 rising.

3) If required, address changes on H1 rising if the previous H1 cycle was the active portion of a write. If the previous H1 cycle was a read, address changes on the next H1 falling.

Figure 9–6 illustrates a read-read-write sequence for $\overline{\text{(M)STRB}}$ active and no wait states. The data is read as late in the cycle as possible to allow maximum access time from address valid. Although external writes require two cycles, internally (from the perspective of the CPU and DMA) they require only one cycle if no accesses to that interface are in progress. In the typical timing for all external interfaces, the (X)R/$\overline{\text{W}}$ strobe does not change until $\overline{\text{(M)STRB}}$ or $\overline{\text{IOSTRB}}$ goes inactive.

Figure 9–6. Read-Read-Write for $\overline{\text{(M)STRB}}$ = 0



Note: (x) $\overline{\text{RDY}}$ is sampled low on rising edge of H1. Data is read next falling edge of H1.

Note: Back-to-Back Read Operations

$\overline{\text{(M)STRB}}$ remains low during back-to-back read operations.

Figure 9–7 illustrates a write-write-read sequence for $\overline{\text{(M)STRB}}$ active and no wait states. The address and data written are held valid approximately one-half cycle after $\overline{\text{(M)STRB}}$ changes.

Figure 9–7. Write-Write-Read for $\overline{\text{(M)STRB}}$ = 0

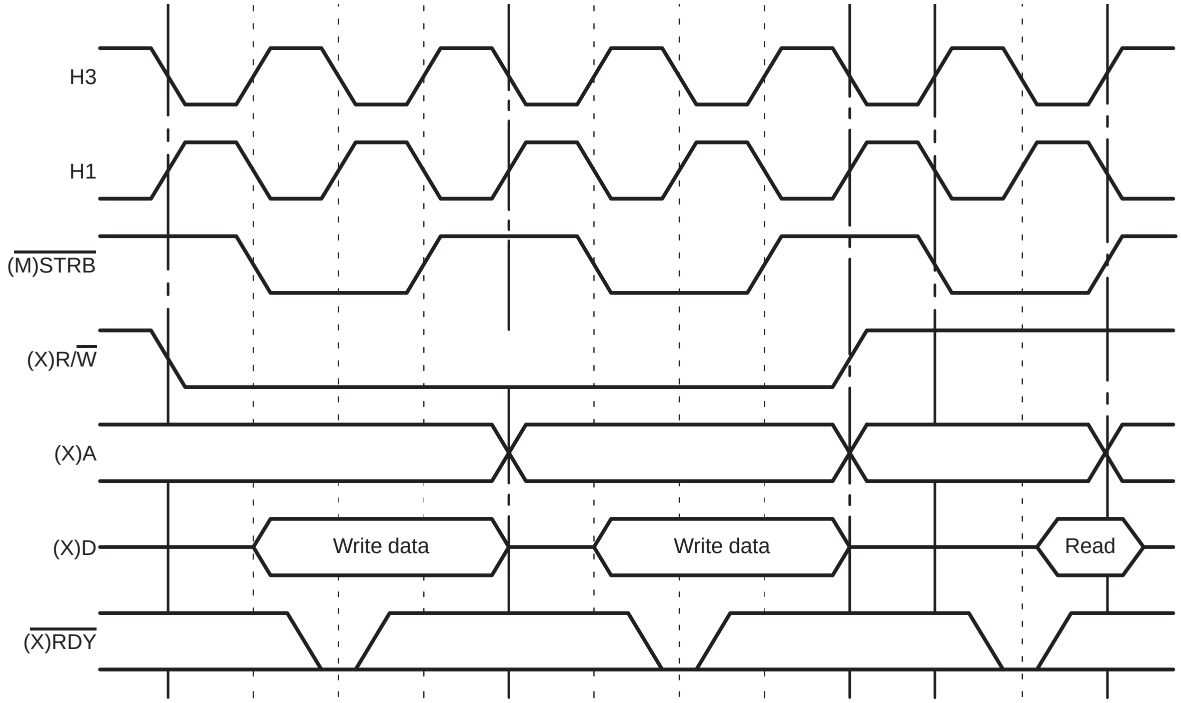

H3
H1
$\overline{\text{(M)STRB}}$
(X)R/$\overline{\text{W}}$
(X)A
(X)D
Write data
Write data
Read
$\overline{\text{(X)RDY}}$

Figure 9–8 illustrates a read cycle with one wait state. Since $\overline{\text{(X)RDY}}$ = 1, the read cycle is extended. $\overline{\text{(M)STRB}}$, (X)R/$\overline{\text{W}}$, and (X)A are also extended one cycle. The next time $\overline{\text{(X)RDY}}$ is sampled, it is 0.

Figure 9–8. Use of Wait States for Read for $\overline{\text{(M)STRB}}$ = 0

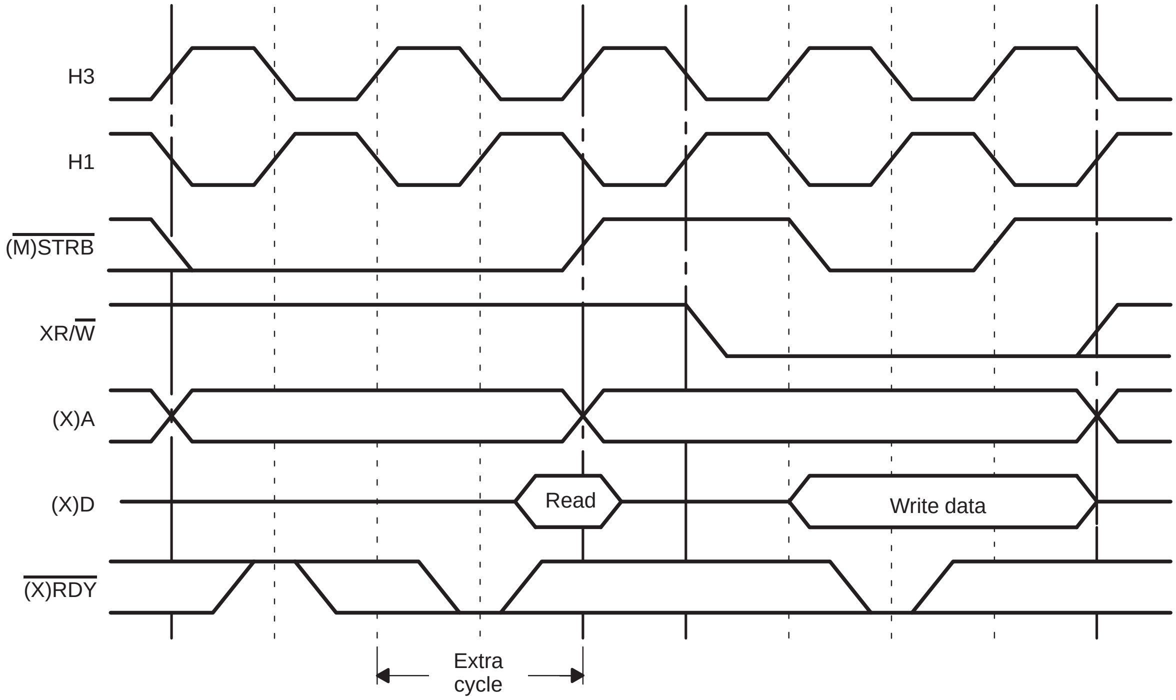
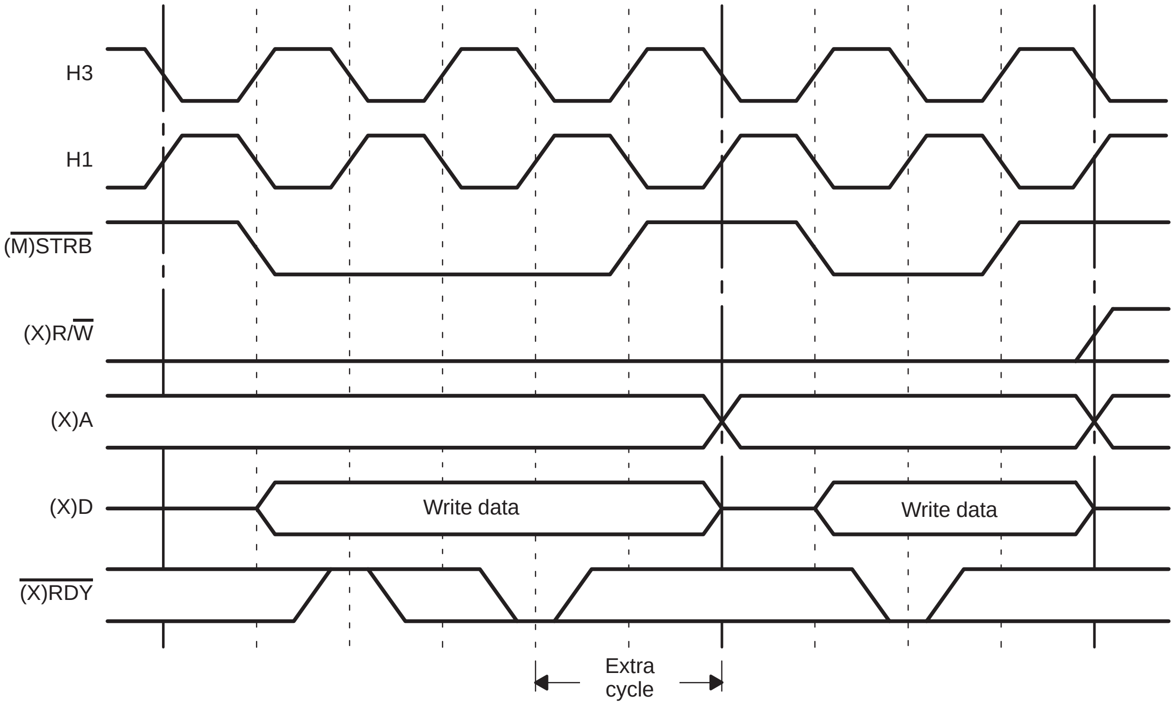


Figure 9–9 illustrates a write cycle with one wait state. Since initially $\overline{\text{(X)RDY}}$ = 1, the write cycle is extended. $\overline{\text{(M)STRB}}$, (X)R/$\overline{\text{W}}$, and (X)A are extended one cycle. The next time $\overline{\text{(X)RDY}}$ is sampled, it is 0.

Figure 9–9. Use of Wait States for Write for $\overline{\text{(M)STRB}}$ = 0



9.6.2 Expansion-Bus I/O Cycles

In contrast to primary bus and $\overline{\text{MSTRB}}$ cycles, $\overline{\text{IOSTRB}}$ reads and writes are both two cycles in duration (with no wait states) and exhibit the same timing. During these cycles, address always changes on the falling edge of H1, and $\overline{\text{IOSTRB}}$ is low from the rising edge of the first H1 cycle to the rising edge of the second H1 cycle. The $\overline{\text{IOSTRB}}$ signal always goes inactive (high) between cycles, and XR/$\overline{\text{W}}$ is high for reads and low for writes.

Figure 9–10 illustrates read and write cycles when $\overline{\text{IOSTRB}}$ is active and there are no wait states. For $\overline{\text{IOSTRB}}$ accesses, reads and writes require a minimum of two cycles. Some off-chip peripherals might change their status bits when read or written to. Therefore, it is important to maintain valid addresses when communicating with these peripherals. For reads and writes when $\overline{\text{IOSTRB}}$ is active, $\overline{\text{IOSTRB}}$ is completely framed by the address.

Figure 9–10. Read and Write for $\overline{\text{IOSTRB}}$ = 0

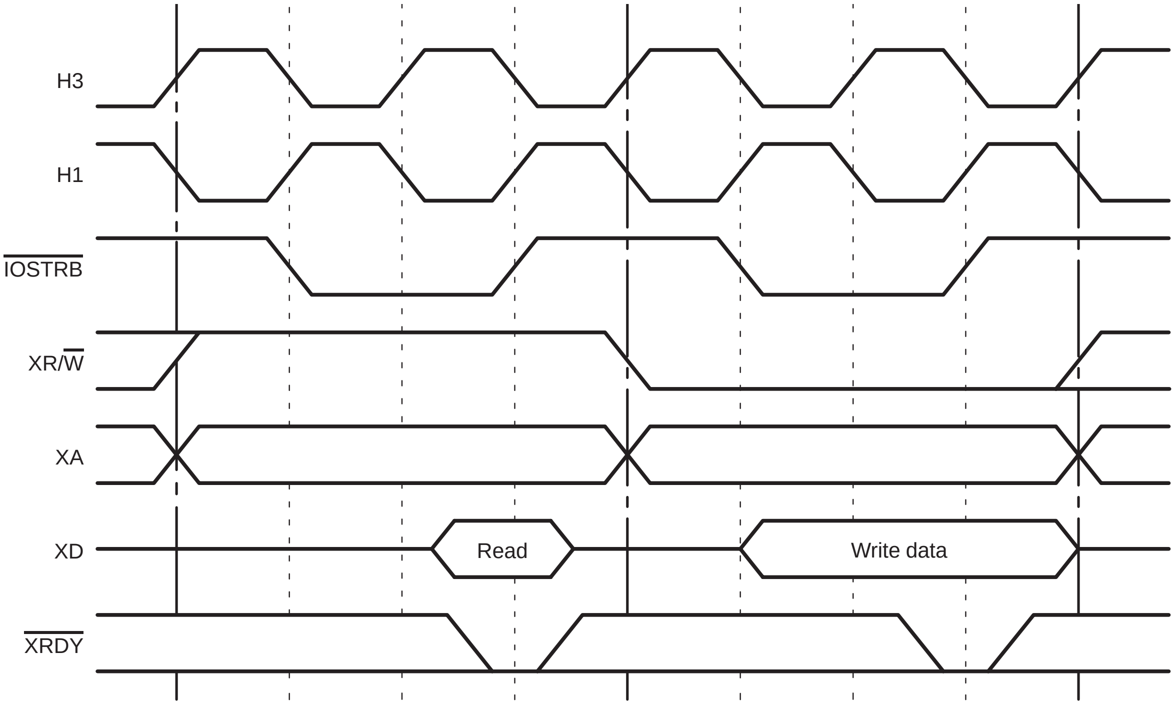


Figure 9–11 illustrates a read with one wait state when $\overline{\text{IOSTRB}}$ is active, and Figure 9–12 illustrates a write with one wait state when $\overline{\text{IOSTRB}}$ is active. For each wait state added, $\overline{\text{IOSTRB}}$, XR/$\overline{\text{W}}$, and XA are extended one clock cycle. Writes hold the data on the bus one additional cycle. The sampling of $\overline{\text{XRDY}}$ is repeated each cycle.

Figure 9–11. Read With One Wait State for $\overline{\text{IOSTRB}}$ = 0

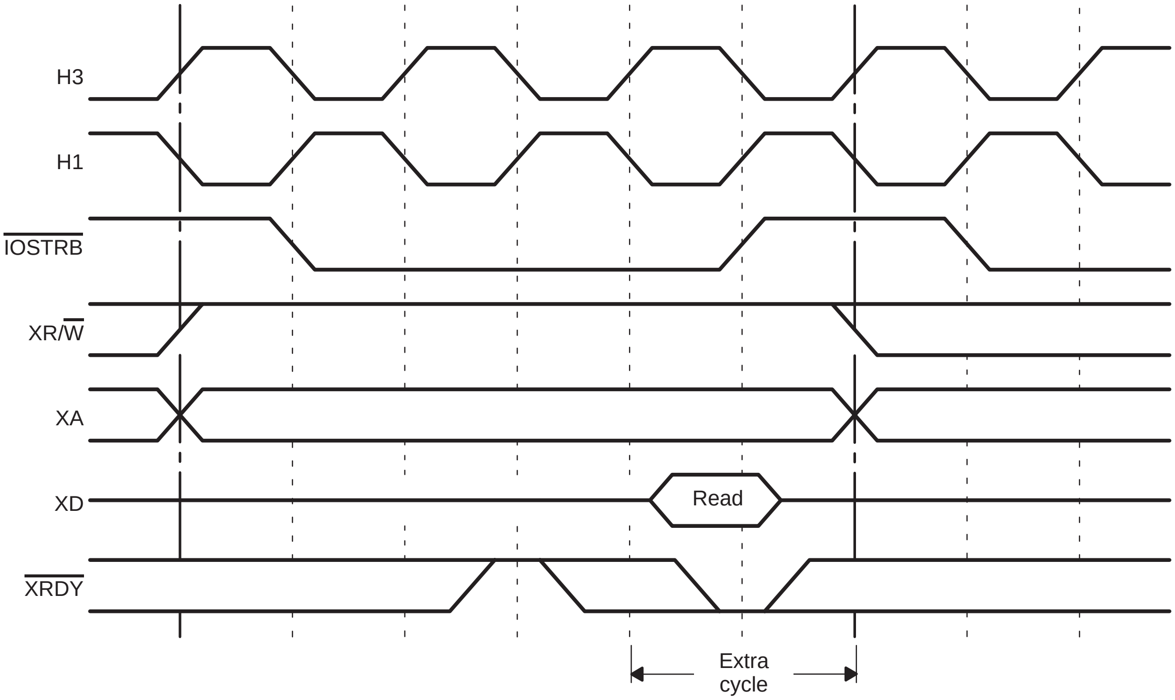


Figure 9–12. Write With One Wait State for $\overline{\text{IOSTRB}}$ = 0

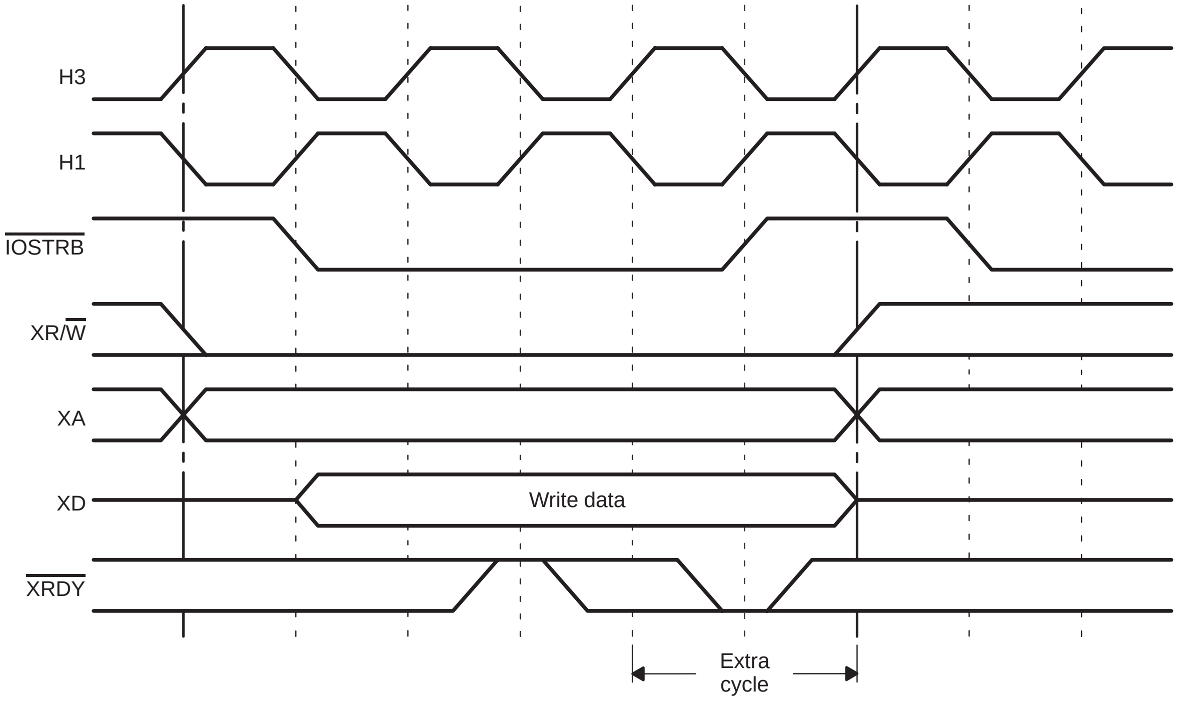


Figure 9–13 through Figure 9–23 illustrate the various transitions between memory reads and writes, and I/O writes over the expansion bus.

Figure 9–13. Memory Read and I/O Write for Expansion Bus

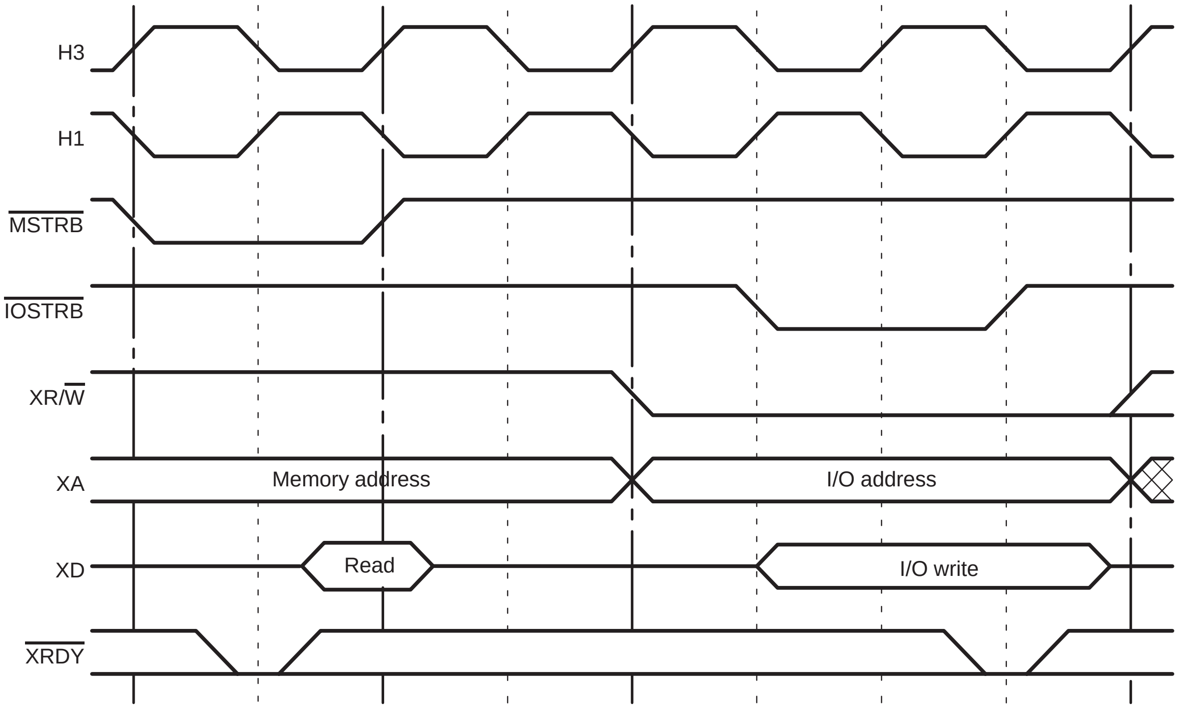


Figure 9–14. Memory Read and I/O Read for Expansion Bus

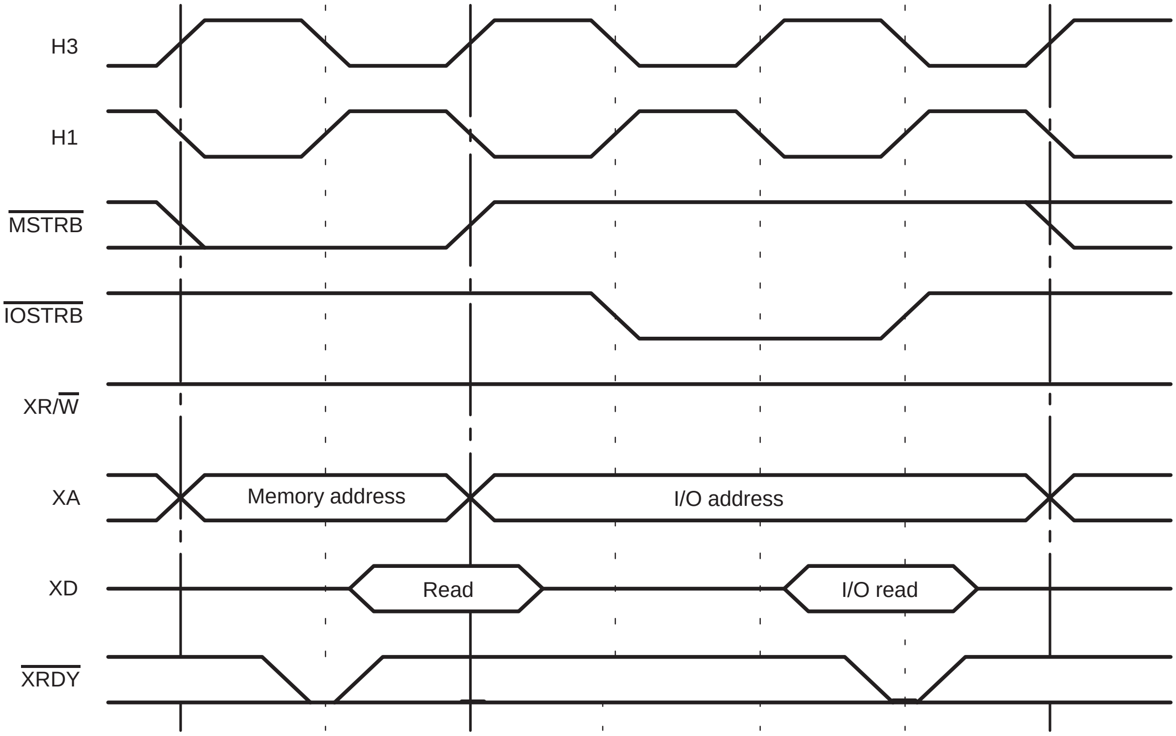


Figure 9–15. Memory Write and I/O Write for Expansion Bus

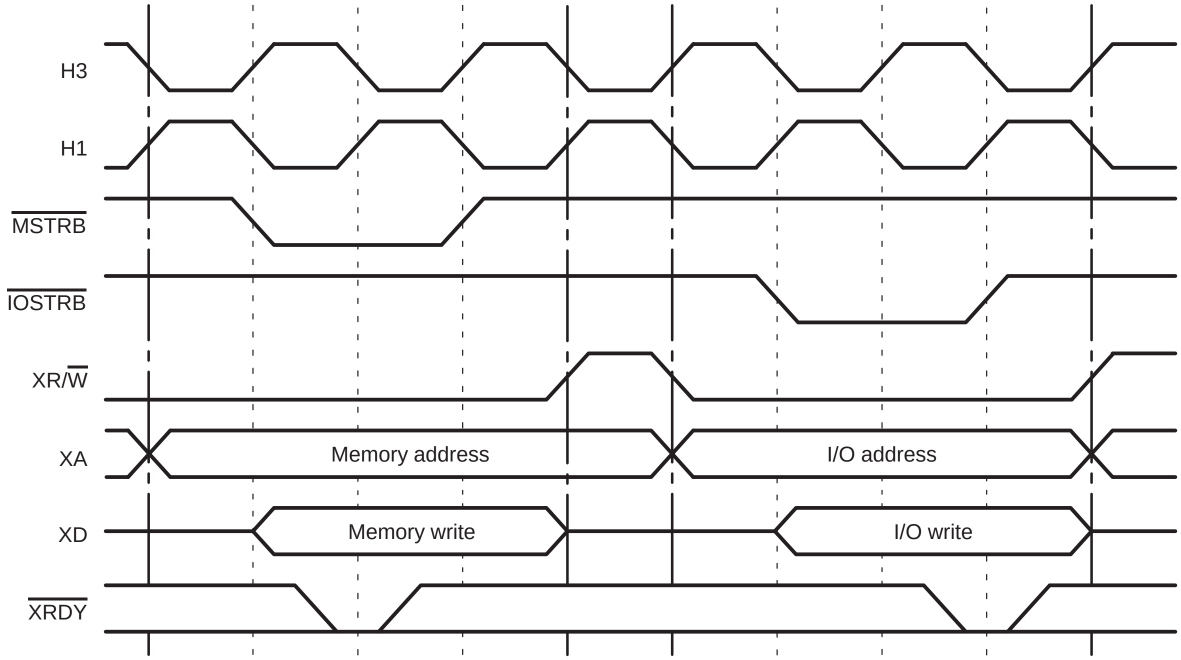


Figure 9–16. Memory Write and I/O Read for Expansion Bus

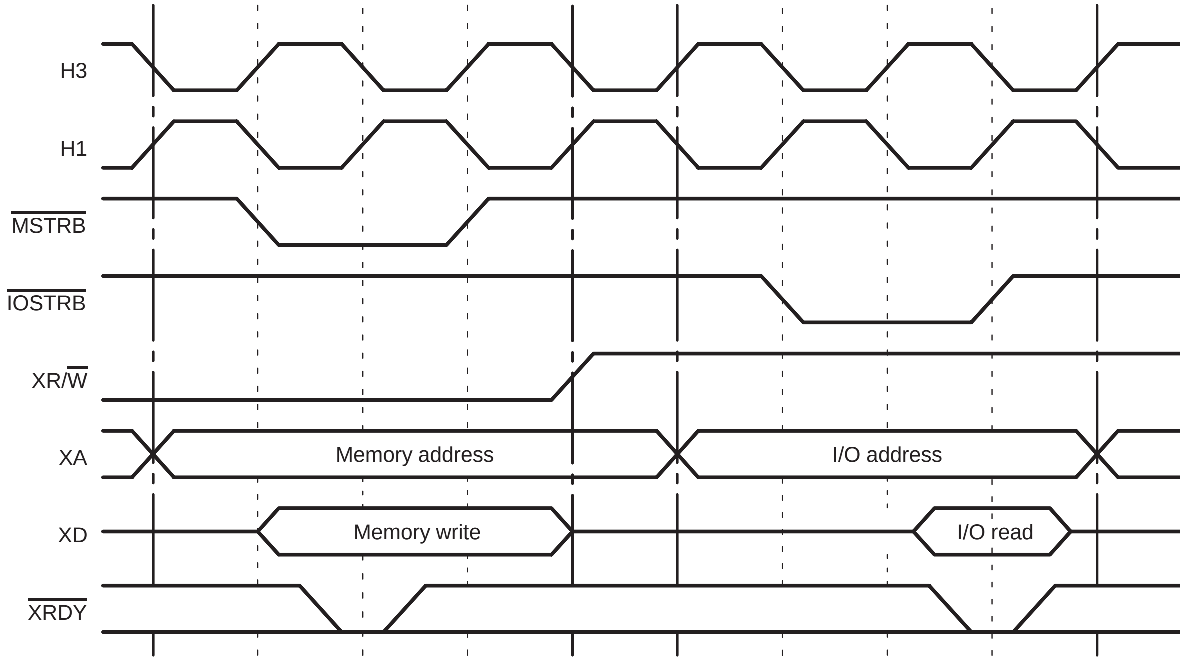


Figure 9–17. I/O Write and Memory Write for Expansion Bus

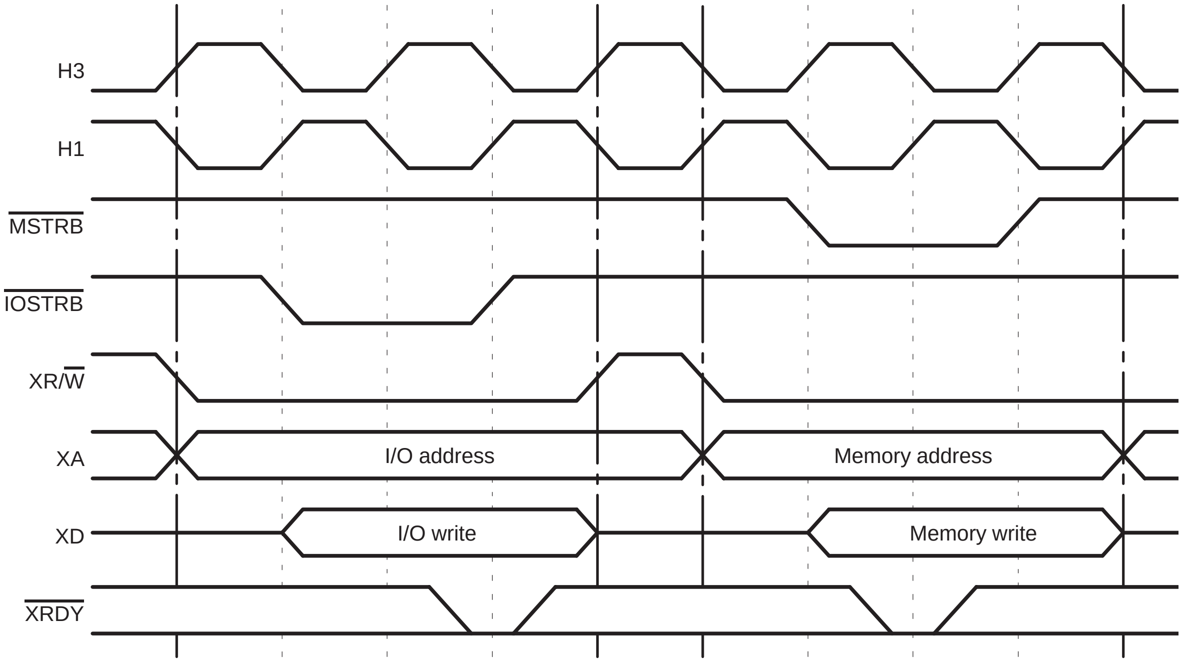


Figure 9–18. I/O Write and Memory Read for Expansion Bus

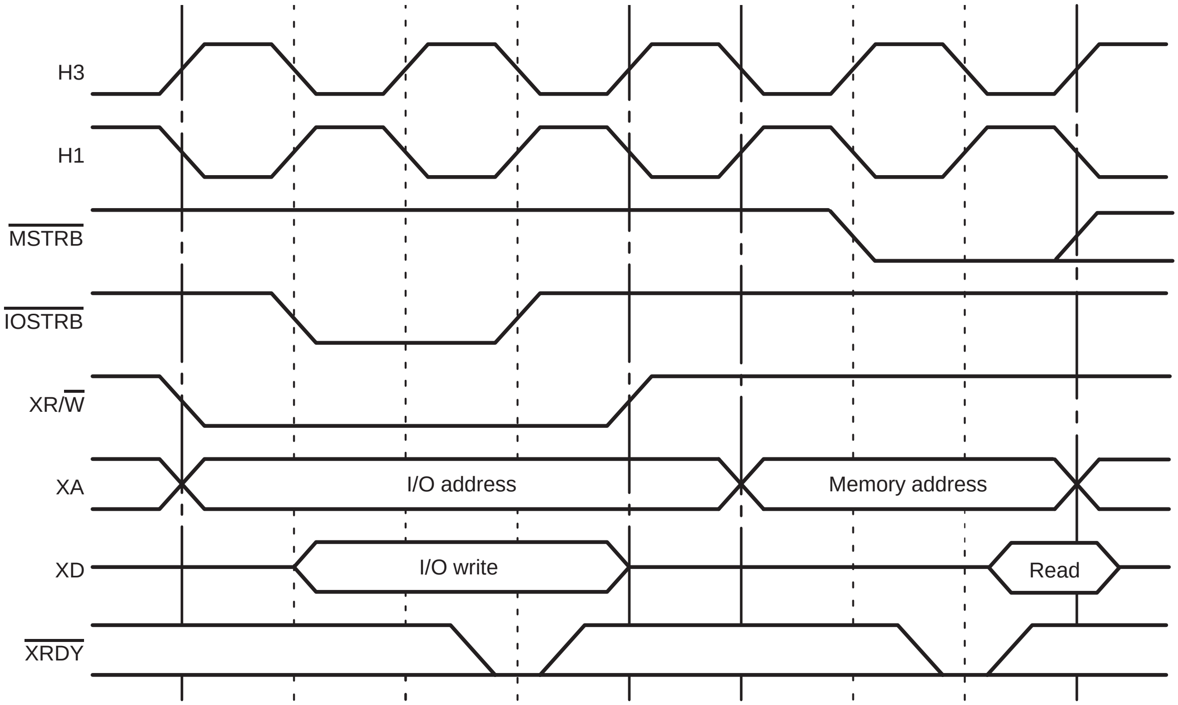


Figure 9–19. I/O Read and Memory Write for Expansion Bus

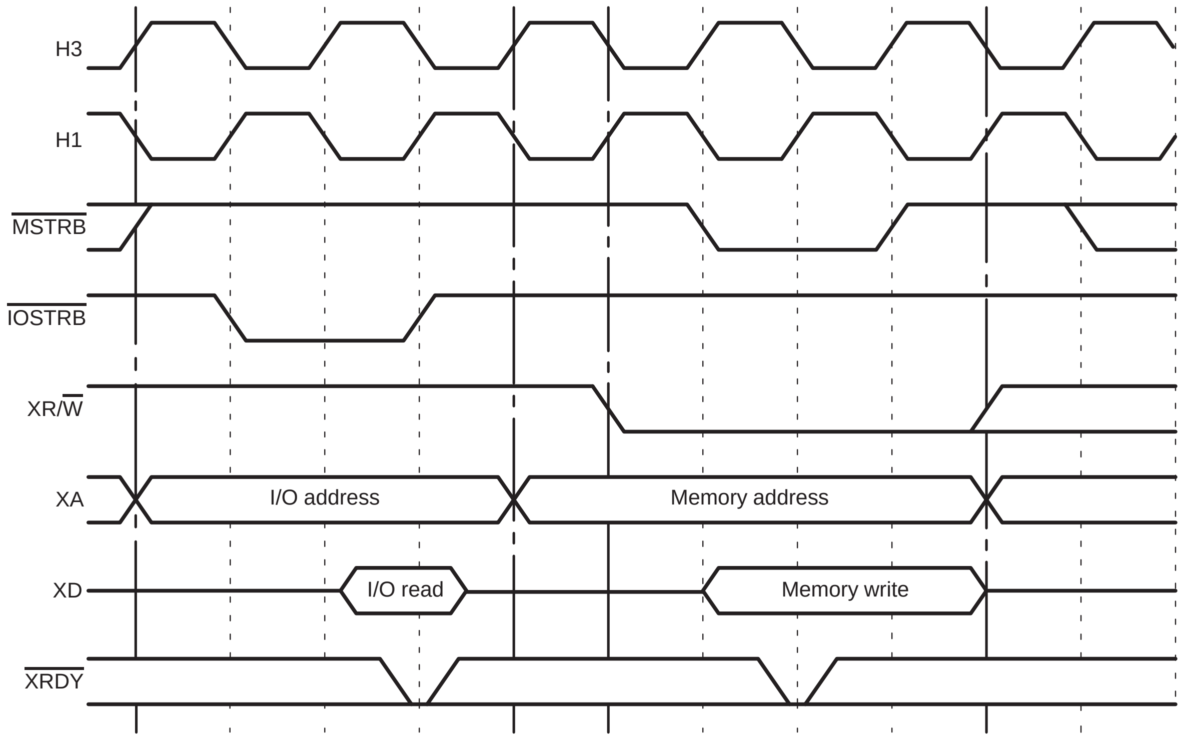


Figure 9–20. I/O Read and Memory Read for Expansion Bus

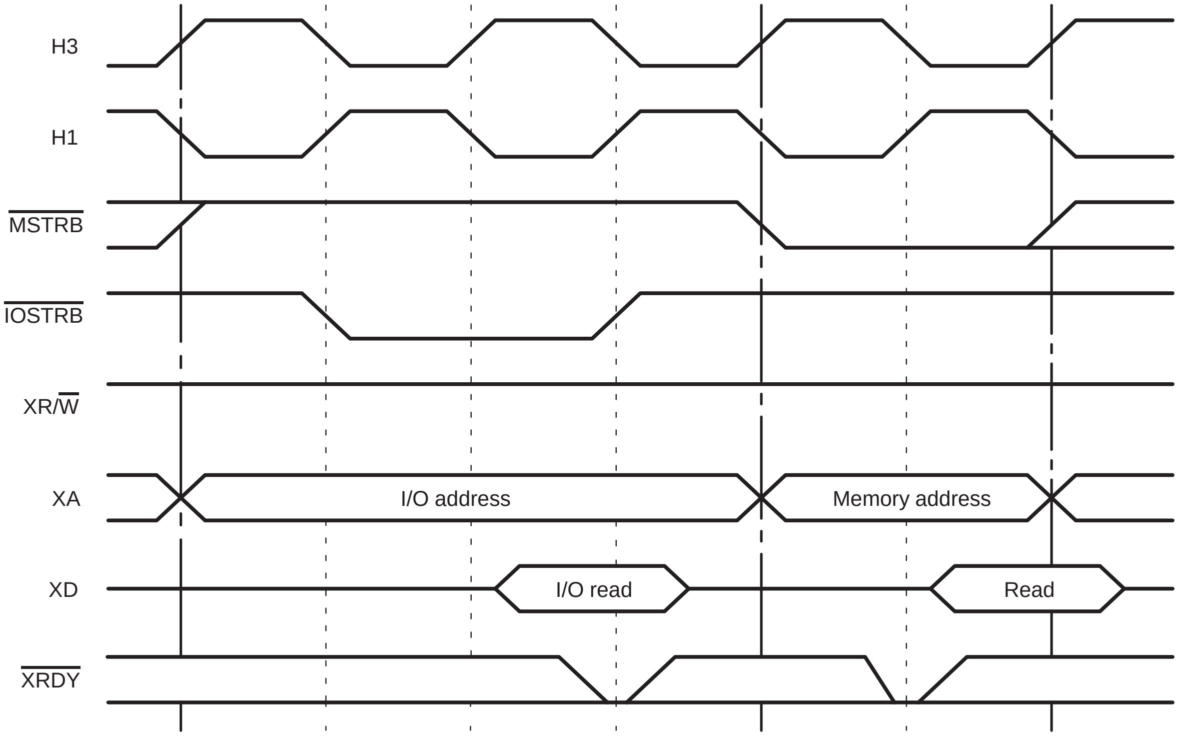


Figure 9–21. I/O Write and I/O Read for Expansion Bus

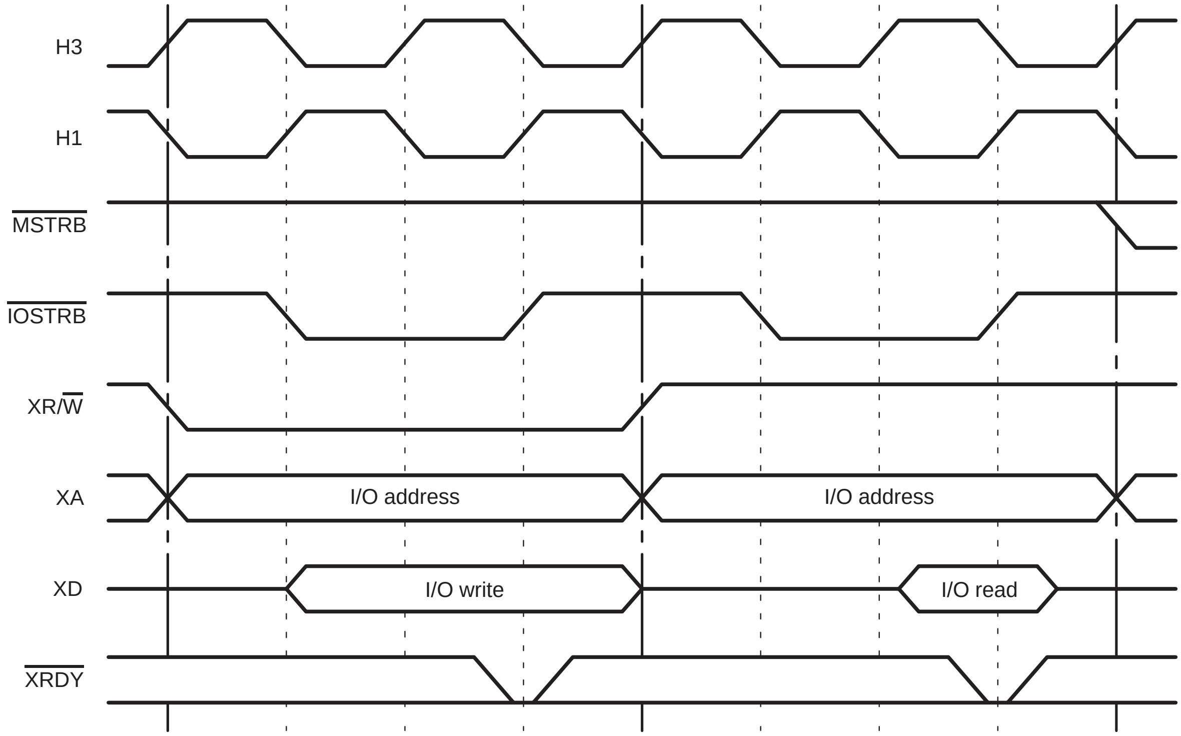


Figure 9–22. I/O Write and I/O Write for Expansion Bus

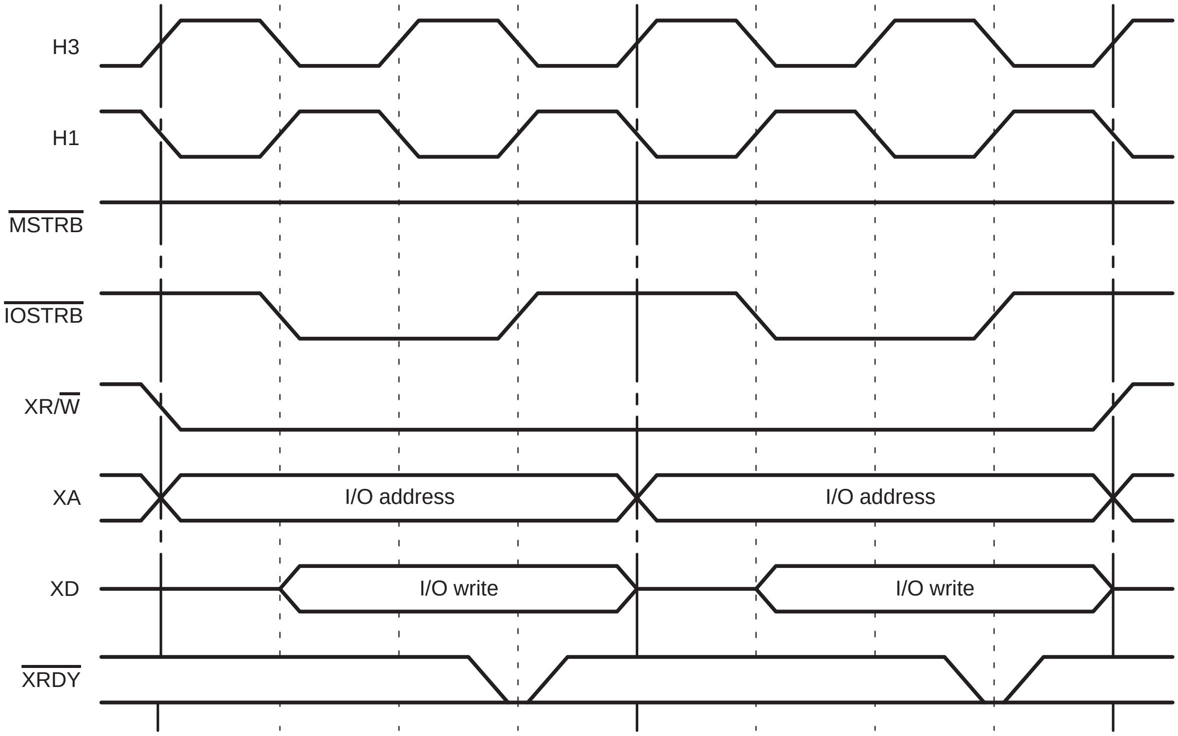


Figure 9–23. I/O Read and I/O Read for Expansion Bus

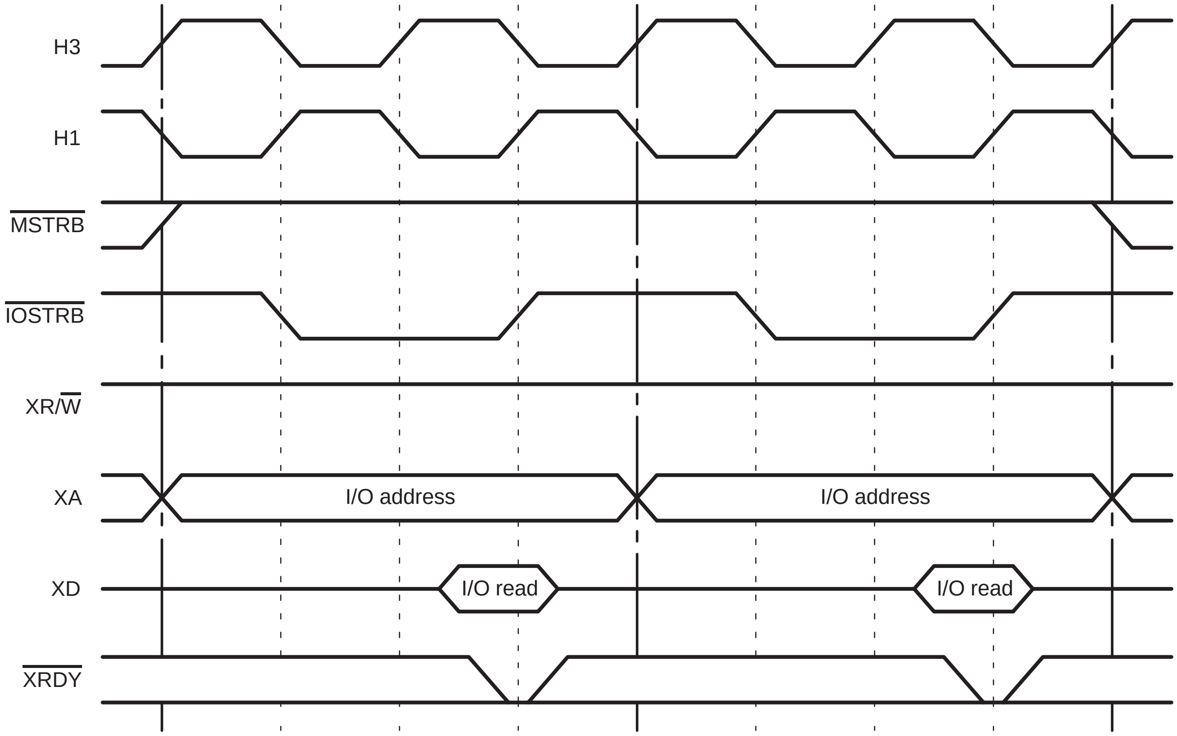


Figure 9–24 and Figure 9–25 illustrate the signal states when a bus is inactive (after an IOSTRB or (M)STRB access, respectively). The strobes (STRB, MSTRB and IOSTRB) and (X)R/W) go to 1. The address is driven with last external bus access, and the ready signal (XRDY or RDY) is ignored.

Figure 9–24. Inactive Bus States for IOSTRB

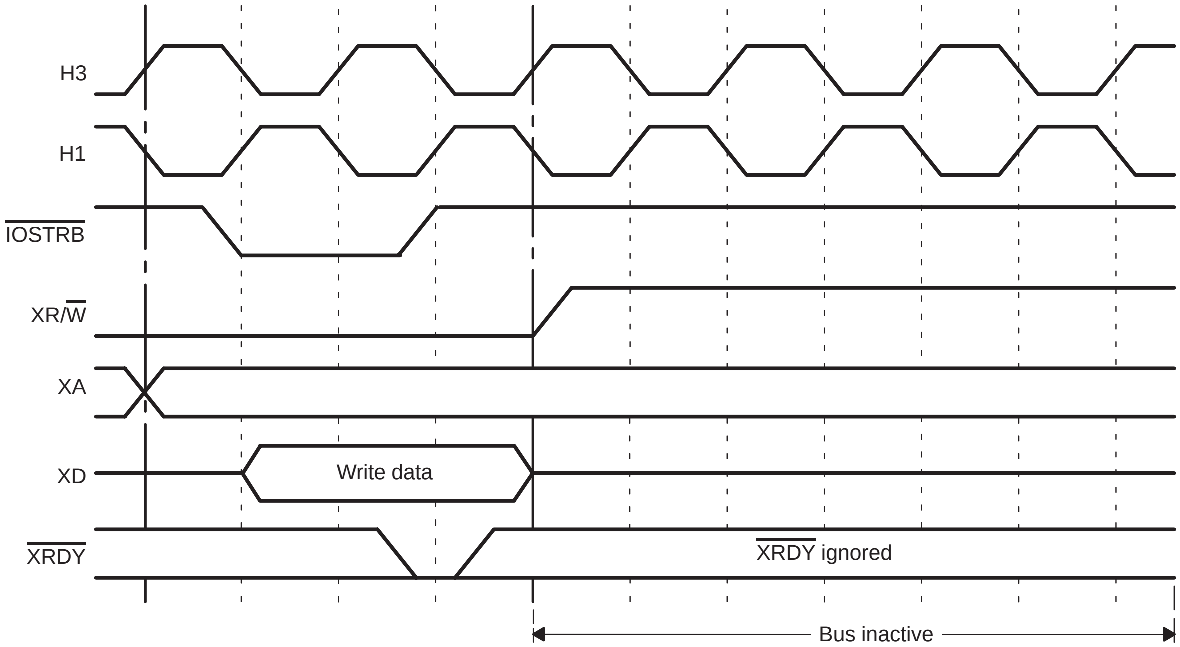
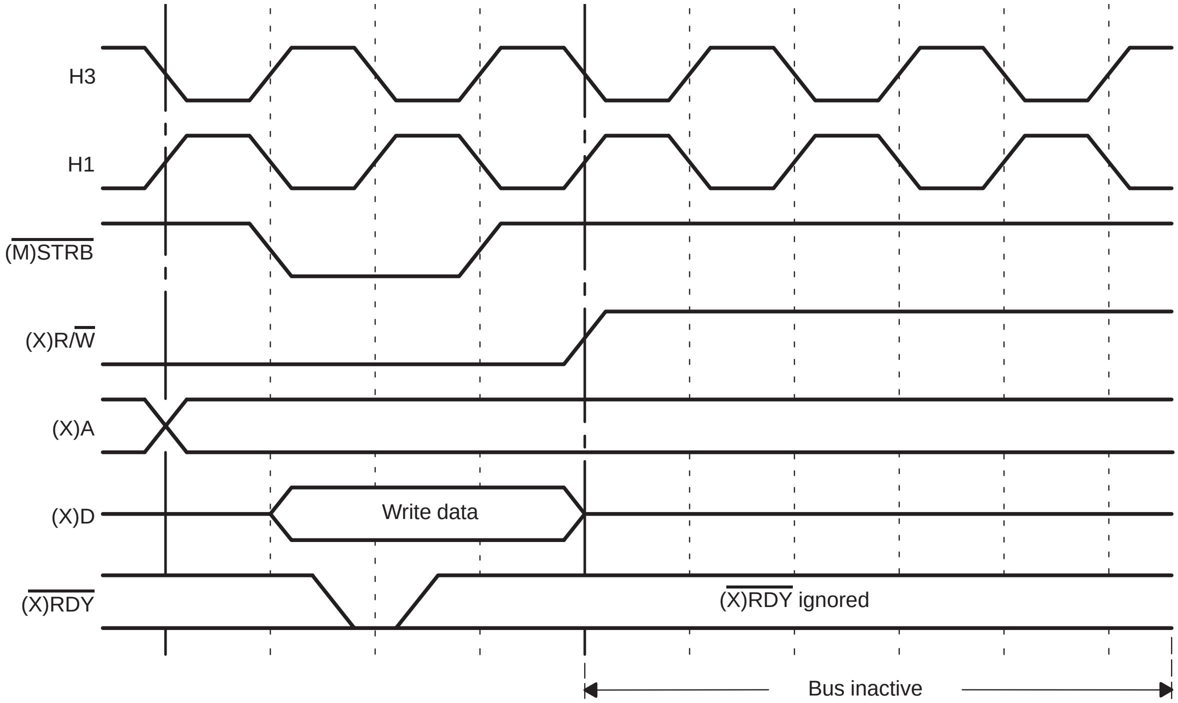


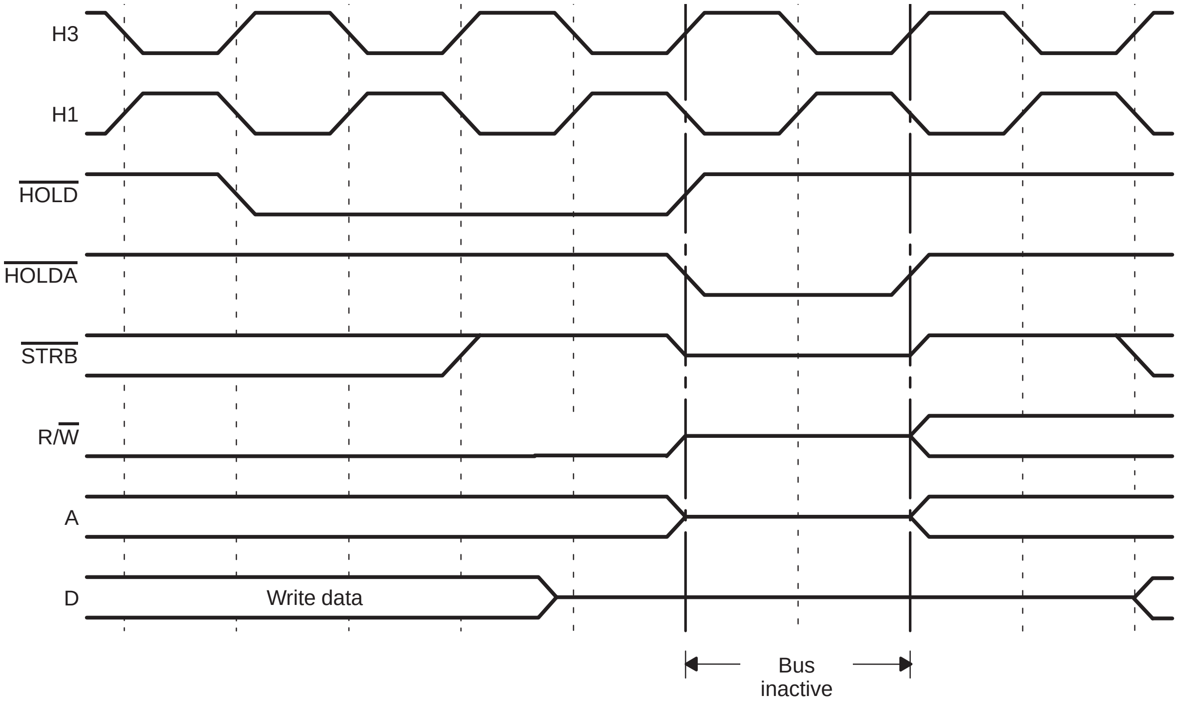
Figure 9–25. Inactive Bus States for $\overline{\text{STRB}}$ and $\overline{\text{MSTRB}}$



9.6.3 Hold Cycles

Figure 9–26 illustrates the timing for $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$. $\overline{\text{HOLD}}$ is an external asynchronous input. There is a minimum of one cycle delay from the time when the processor recognizes $\overline{\text{HOLD}}$ = 0 until $\overline{\text{HOLDA}}$ = 0. When $\overline{\text{HOLDA}}$ = 0, the address, data buses, and associated strobes are placed in a high-impedance state. All accesses occurring over an interface are completed before a hold is acknowledged.

Figure 9–26. $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ Timing



Chapter 10

TMS320C32 Enhanced External Memory Interface

’C32

The ’C32 external memory interface provides greater flexibility by improving the ’C3x core with several new features. This chapter describes these features and enhancements in detail.

Topic Page

10.1 TMS320C32 Memory Features

The ’C32 external memory interface includes the following features:

- One external pin, PRGW, configures the external-program-memory width to 16 or 32 bits.
- Two sets of memory strobes ($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$) and one $\overline{\text{IOSTRB}}$ allow zero glue-logic interface to two banks of memory and one bank of external peripherals.
- Separate bus control registers for each strobe-control wait-state generation, external memory width, and data-type size.
- Each memory STRB handles 8-, 16- or 32-bit external data accesses (reads and writes) to 8-, 16-, or 32-bit-wide memory.
- Multiprocessor support through the $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ signals, is valid for all the STRBs.

10.2 TMS320C32 Memory Overview

The following sections describe examples, control register setups, and restrictions necessary to fully understand the operation and functionality of the external memory interface.

10.2.1 External Memory Interface Overview

The ’C32 memory interface accesses external memory through one 24-bit address and one 32-bit data bus that is shared by three mutually-exclusive strobes ($\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$). Depending on the address accessed, the ’C32 activates one of these strobes according to the memory map shown in Figure 4–3 on page 4-8.

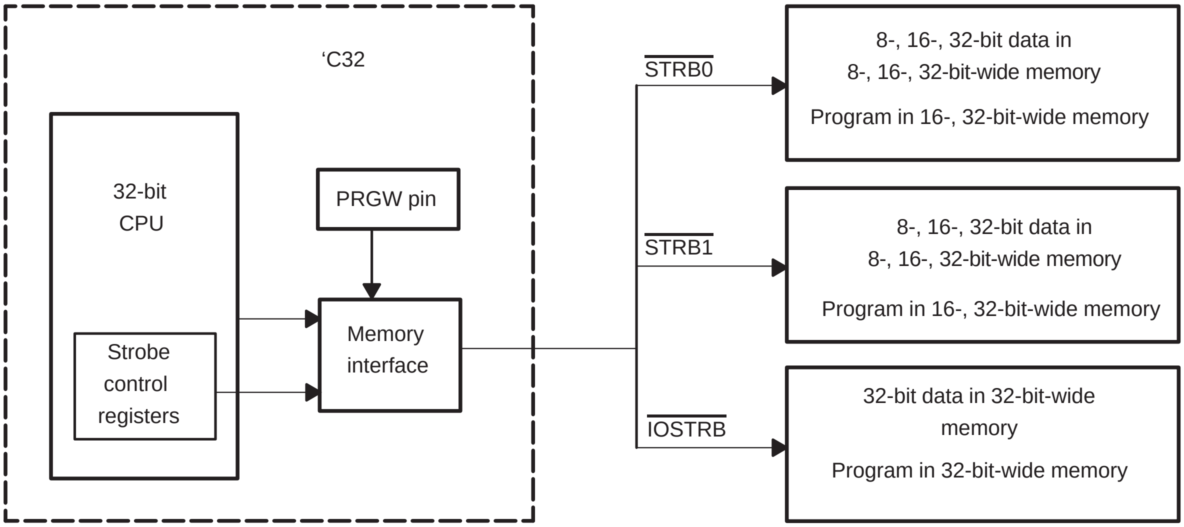
$\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ can access 8-, 16-, or 32-bit data from 8-, 16-, or 32-bit wide memory. This is accomplished by four signals in each strobe: $\overline{\text{STRBx\_B3}}$/A$_{–1}$, $\overline{\text{STRBx\_B2}}$/A$_{–2}$, $\overline{\text{STRBxB1}}$, and $\overline{\text{STRBx\_B0}}$. These signals serve as byte-enable pins to access one byte, half word, or a full word from the external memory. The first two signals also serve as additional address pins to perform two or four consecutive accesses in 8-bit or 16-bit-wide external memory. The ’C32 controls the behavior of these pins through the data size and memory width bit fields in the corresponding strobe control register, as follows:

- ❑ Memory width (default value dependent on PRGW pin level)
  - ■ 8-bit-wide memory
    - ■ $\overline{\text{STRBx\_B3}}$/A$_{–1}$ and $\overline{\text{STRBx\_B2}}$/A$_{–2}$ as address pins
    - ■ $\overline{\text{STRBx\_B0}}$ as byte-enable/chip-select signal
    - ■ $\overline{\text{STRBx\_B1}}$ unused
  - ■ 16-bit-wide memory
    - ■ $\overline{\text{STRBx\_B3}}$/A$_{–1}$ as address pin
    - ■ $\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$ as byte-enable signal
    - ■ $\overline{\text{STRBx\_B2}}$ unused
  - ■ 32-bit-wide memory
    - ■ $\overline{\text{STRBx\_B3}}$, $\overline{\text{STRBx\_B2}}$, $\overline{\text{STRBx\_B1}}$, and $\overline{\text{STRBx\_B0}}$ as byte-enable signals
- ❑ Data size
  - ■ 8-bit data, physical address = logical address shift right by 2
  - ■ 16-bit data, physical address = logical address shift right by 1
  - ■ 32-bit data, physical address = logical address

$\overline{\text{IOSTRB}}$ can access 32-bit data from 32-bit wide memory. It does not have the flexibility of $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ since it is composed of a single signal: $\overline{\text{IOSTRB}}$. $\overline{\text{IOSTRB}}$ bus cycles are different from those of $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ and are discussed in Section 10.10. This timing difference accomodates slower I/O peripherals.

The ’C32 memory interface parallel bus implements three mutually-exclusive address spaces distinguished via three separate control signals as shown in Figure 10–1. $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ support 8-, 16-, or 32-bit data access in 8-, 16-, 32-bit-wide external memory and 16-, 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, hence, the memory interface packs and unpacks the data accessed accordingly.

Figure 10–1. Memory Address Spaces


‘C32
32-bit
CPU
Strobe
control
registers
PRGW pin
Memory
interface
$\overline{\text{STRB0}}$
$\overline{\text{STRB1}}$
$\overline{\text{IOSTRB}}$
8-, 16-, 32-bit data in
8-, 16-, 32-bit-wide memory
Program in 16-, 32-bit-wide memory
8-, 16-, 32-bit data in
8-, 16-, 32-bit-wide memory
Program in 16-, 32-bit-wide memory
32-bit data in 32-bit-wide
memory
Program in 32-bit-wide memory

10.2.2 Program Memory Access

The ’C32 supports program execution from 16- or 32-bit external memory width. The PRGW pin configures the width of the external program memory. When this pin is pulled high, the ’C32 executes from 16-bit wide memory. When this 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. During the first cycle the lower 16 bits of the instruction are fetched. During the second cycle, the upper 16 bits are fetched and concatenated with the lower 16 bits. 32-bit memory fetches are identical to those of the ’C30 and ’C31.

The PRGW status bit field of the CPU status (ST) register reflects the setting of the PRGW pin. Figure 10–2 depicts all the bit fields of the CPU status (ST) register.

Figure 10–2. Status Register

31–16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	PRGW status	INT config	GIE	CC	CE	CF	xx	RM	OVM	LUF	LV	UF	N	Z	V	C
	R	R/W	R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) xx = reserved bit, read as 0

The status of the PRGW pin also affects the reset value of the physical memory width bit fields of the $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ bus-control registers. The physical memory width is set to 32-bit memory width if the PRGW pin is logic low after the device reset. The physical memory width is set to 16-bit memory width if the PRGW pin is logic high after the device reset (see Section 10.3 for more information).

CAUTION

The cycle before and the cycle after changing the PRGW should not perform a program fetch over the external memory interface.

10.2.3 Data Memory Access

The ’C32 can load and store 8-, 16-, or 32-bit data quantities from and into memory. Because the CPU has a 32-bit architecture, the device internally handles all 8-, 16-, or 32-bit data quantities as a 32-bit value. Hence, the external memory interface handles the conversion between 8- and 16-bit data quantities to the internal 32-bit representation. The external memory interface also handles the storage of 32-, 16-, or 8-bit data quantities into 32-, 16-, or 8-bit wide memories.

10.2.3.1 8-, 16-, or 32-Bit Integers Data Types

The ’C32 supports 8-, 16- or 32-bit integer data quantities. When 8- or 16-bit integers are read from external memory, the value is loaded into the LSBs of the register with the MSBs sign-extended or zero-filled. The polarity of the sign ext/zero-fill bit field of the corresponding STRB control register controls the sign extension or zero fill (see paragraphs 10.3.1.1 and 10.3.1.2). The 32-bit integer data access is identical to that of the ’C30 and ’C31.

10.2.3.2 16- or 32-Bit Floating-Point Data Types

The ’C32 supports 16- or 32-bit floating point data. For 16-bit floating-point reads, the eight MSBs are the signed exponent and the eight LSBs are the signed mantissa (see Section 5.3.2, *’C32 Short Floating-Point Format for External 16-Bit Data*, on page 5-6). When a 16-bit floating-point value is loaded into a 40-bit register, the external memory interface zero fills the least significant 24 bits of the register. When a 16-bit floating-point value is used as a 32-bit on-chip input operand, the external memory interface zero fills the 16 LSBs of the 32-bit input operand. The 32-bit floating-point data access is identical to that of the ’C30 and ’C31.

10.3 Configuration

To access 8-, 16-, or 32-bit data (types) from 8-, 16-, or 32-bit wide memory, the memory interface of the ’C32 device uses either strobe $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ with four pins each. These pins serve as byte-enable and/or additional-address pins. In conjunction with a shifted version of the internal address presented to the external address, the ’C32 can select a single byte from one external memory location or combine up to four bytes from contiguous memory locations. The behavior of these pins is controlled by the external memory width and the data type size. The selected data size also determines the amount of internal-to-physical address shift. You can assign these values to the ’C32 memory interface through bit fields in the bus control registers.

10.3.1 External Interface Control Registers

The following sections describe the bus control registers used to manipulate the byte addressability features of the ’C32. Figure 10–3 shows the external interface control memory map.

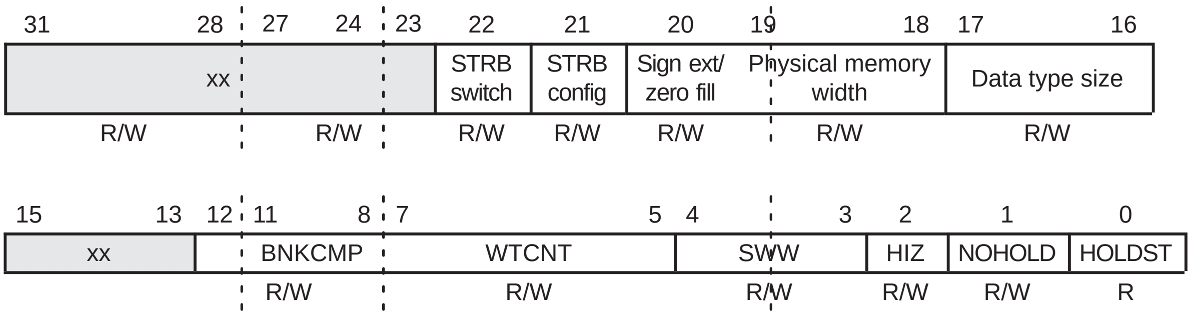
Figure 10–3. Memory-Mapped External Interface Control Registers

Address	**Register**
808060h	$\overline{\text{IOSTRB}}$ control
808061h	Reserved
808062h	Reserved
808063h	Reserved
808064h	$\overline{\text{STRB0}}$ control
808065h	Reserved
808066h	Reserved
808067h	Reserved
808068h	$\overline{\text{STRB1}}$ control
808069h	Reserved
	. . .
80806Fh	Reserved

10.3.1.1 STRB0 Control Register

The STRB0 control register (Figure 10–4) is a 32-bit register that contains the control bits for the portion of the external bus memory space that is mapped to STRB0. The following table lists the register bits with the bit names and functions. At the system reset, 0F10F8h is written to the STRB0 control register if the PRGW pin is logic low and 0710F8h is written to the STRB0 control register if the PRGW pin is logic high.

Figure 10–4. STRB0 Control Register



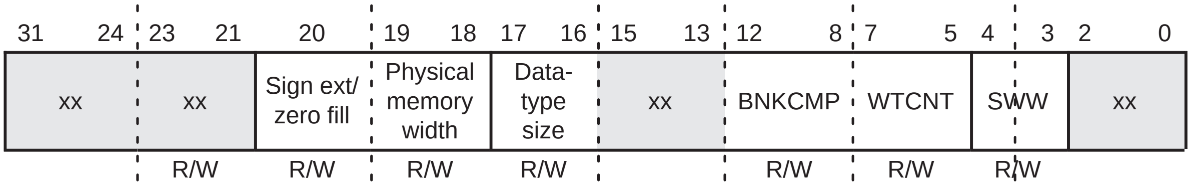
Notes: 1) R = read, W = write

2) xx = reserved, read as 0

10.3.1.2 STRB1 Control Register

The STRB1 control register (Figure 10–5) is a 32-bit register that contains the control bits for the portion of the external bus memory space that is mapped to STRB1. Figure 10–5 shows the register bits with their names and functions. At system reset, 0F10F8h is written to the STRB1 control register if the PRGW pin is logic low and 0710F8h is written to the STRB1 control register if the PRGW pin is logic high.

Figure 10–5. STRB1 Control Register



Notes: 1) R = read, W = write

2) xx = reserved, read as 0

CAUTION

The instruction immediately preceding a change in the data-size or memory-width bit fields should not perform a multicycle store. Do not follow a change in the data-size or memory-width bit fields with a store instruction. Also, do not perform a load in the next two instructions following a change in the data-size or memory-width bit fields

10.3.1.3 IOSTRB Control Register

The IOSTRB control register (Figure 10–6) is a 32-bit register that contains the control bits for the portion of the external bus memory space that is mapped to IOSTRB. Unlike the STRB0 and STRB1, there is no byte-enable signal for the IOSTRB. The data access through the IOSTRB is always 32-bit. The following table lists the register bits with the bit names and functions. At the system reset, 0F8h is written to the IOSTRB control register. The IOSTRB timing is identical to the 'C30 IOSTRB timing.

Figure 10–6. IOSTRB Control Register

31 16	15 12	11 8	7 5	4 3	2 0
xx	xx	xx	WTCNT	SWW	xx
			R/W	R/W	

Notes: 1) R = read, W = write

2) xx = reserved, read as 0

Note:

After changing the bit fields of the IOSTRB control register, up to three instructions are fetched before the IOSTRB bus is reconfigured.

Table 10–1 describes the bits in the $\overline{\text{STRBO}}$, $\overline{\text{STRB1}}$, and the $\overline{\text{IOSTRB}}$ control registers.

Table 10–1. $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$ Control Register Bits

Abbreviation	**Reset Value**	**Name**	**Description**
HOLDST	0	Hold status bit	This bit signals whether the port is being held (HOLDST = 1), or is not being held (HOLDST = 1). This status bit is valid whether the port has been held through hardware or software. *($\overline{\text{STRB0}}$ control register only)*
NOHOLD	0	Port hold signal	NOHOLD allows or disallows the port to be held by an external $\overline{\text{HOLD}}$ signal. When NOHOLD = 1, the ’C3x takes over the external bus and controls it, regardless of serviced or pending requests by external devices. No hold acknowledge ($\overline{\text{HOLDA}}$) is asserted when a $\overline{\text{HOLD}}$ is received. However, it is asserted if an internal hold is generated (HIZ = 1). *($\overline{\text{STRB0}}$ control register only)*
HIZ	0	Internal hold	When set (HIZ = 1), the port is put in hold mode. This is equivalent to the external $\overline{\text{HOLD}}$ signal. By forcing the high-impedance condition, the ’C3x can relinquish the external memory port through software. $\overline{\text{HOLDA}}$ goes low when the port is placed in the high impendance state. *($\overline{\text{STRB0}}$ control register only)*
SWW	11	Software wait mode	In conjunction with WTCNT, this 2-bit field defines the mode of wait-state generation.
WTCNT	111	Software wait mode	This 3-bit field specifies the number of cycles to use when in the software wait mode for the generation of internal wait state. The range is 0 (WTCNT = 0 0 0) to 7 (WTCNT = 111) H1/H3 cycles.
BNKCMP	10000	Bank compare	This 5-bit field specifies the number of MSBs of the address to be used to define the bank size. *($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers only)*
Data type size	11	($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers only)	Indicates the size of the data type written in memory.

Bit 17	**Bit 16**	**Data Type Size**
0	0	8 bit
0	1	16 bit
1	0	Reserved
1	1	32 bit

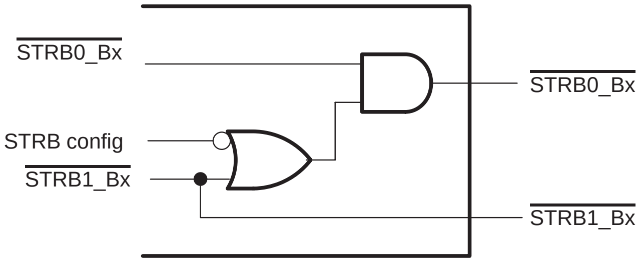
Table 10–1. $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$ Control Register Bits (Continued)

<table>
<tr><th>Abbreviation</th><th>Reset Value</th><th>Name</th><th>Description</th></tr>
<tr><td>Physical memory width</td><td>01
or
11</td><td>($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers only)</td><td>Indicates the size of the physical memory connected to the device. The “reset” value depends on the status of the PRGW pin. If the PRGW pin is logic low, the memory width is configured to 32 bits (= 11_2). If the PRGW pin is logic high, the physical memory width is configured to 16 bits (= 01_2). This field can have the following values:
<table>
<tr><th>Bit 17</th><th>Bit 16</th><th>Data Type Size</th></tr>
<tr><td>0</td><td>0</td><td>8 bit</td></tr>
<tr><td>0</td><td>1</td><td>16 bit (reset value if PRGW = 1)</td></tr>
<tr><td>1</td><td>0</td><td>Reserved</td></tr>
<tr><td>1</td><td>1</td><td>32 bit (reset value if PRGW = 0)</td></tr>
</table>
Setting the physical memory width field of the STRB0 or STRB1 control registers changes the functionality of the STRB0 or STRB1 signals.
❑ When the physical memory width field is configured to 32 bits, the corresponding STRBx_B0–STRBx_B3 signals are configured as byte-enable pins (see Figure 10–10 on page 10-20).
❑ When the physical memory width field is configured to 16 bits, the corresponding STRBx_B3/A–1 signal is configured as an address pin while the STRBx_B0 and STRBx_B1 signals are configured as byte-enable pins (see Figure 10–14 on page 10-26).
❑ When the physical memory width field is configured to 8 bits, the STRBx_B3/A–3 and STRBx_B2/A–2 signals are configured as byte-enable pins (see Figure 10–18 on page 10-32).
Once an STRBx_Bx signal is configured as an address pin, it is active for any external memory access (STRB0, STRB1, IOSTRB, or external fetch).</td></tr>
</table>

Table 10–1. STRB0, STRB1, and IOSTRB Control Register Bits (Continued)

Abbreviation	Reset Value	Name	Description
Sign ext/ zero-fill	0	(STRB0 and STRB1 control registers only)	Selects the method of converting 8- and 16-bit integer data into 32-bit integer data when transferring data from external memory to an internal register or memory location. This field can have the following values: **Bit 20** **Physical Memory Width** 0 8- or 16-bit integer reads are sign-extended to 32 bits (reset value). 1 The MSBs of 8- or 16-bit integer reads are zero-filled to make the number 32 bits.
STRB config	0	STRB configuration	Activates the STRB0 Bx signals when accessing data from STRB0 or STRB1 memory spaces. This mode is useful when accessing a single external memory bank that stores two different data types, each mapped to a different STRB. This field can have the following values. **Bit 21** **Physical Memory Width** 0 STRB0_Bx signals are active for locations 0h–7FFFFFh and 880000h–8FFFFFh. STRB1_Bx signals are active for locations 900000h–FFFFFFh (reset value). 1 STRB0_Bx signals are active for locations 0h–7FFFFFh, 880000h–8FFFFFh and 900000h–FFFFFFh. STRB1_Bx signals are active for locations 900000h–FFFFFFh. A functional representation of this configuration is shown in Figure 10–7 on page 10-13.
STRB switch	0	(STRB0 control register only)	Defines whether a single cycle is inserted between back-to-back reads when crossing STRB0 or STRB1 to STRB1 to STRB0 boundaries (switching STRBs). The extra cycle toggles the strobe signal during back-to-back reads. Otherwise, the strobe remains low during back-to-back reads. This field has the following values: **Bit 22** **Physical Memory Width** 0 Does not insert a single cycle between back-to-back reads that switch from STRB0 to STRB1 or vice-versa (reset value). 1 Inserts a single cycle between back-to-back reads that switch from STRB0 to STRB1 or vice-versa (reset value).

Figure 10–7. STRB Configuration


$\overline{\text{STRB0\_Bx}}$
STRB config
$\overline{\text{STRB1\_Bx}}$
$\overline{\text{STRB0\_Bx}}$
$\overline{\text{STRB1\_Bx}}$

10.3.2 Using Physical Memory Width and Data-Type Size Fields

Consider a ’C32 connected to two banks of external memory. In this configuration, one bank is mapped to $\overline{\text{STRB0}}$ while the other bank is mapped to $\overline{\text{STRB1}}$. The $\overline{\text{STRB0}}$ bank of memory is 32 bits wide and stores 32-bit data types. The $\overline{\text{STRB1}}$ bank of memory is 16 bits wide and stores 16-bit data types. You can transfer these configurations to the ’C32 by setting the physical memory width and data-type size fields of the respective $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ control registers. You also must clear the STRB config bit field to 0 since the banks are separate memories. Note that ‘C32 address pins $A_{23}A_{22}A_{21}...A_1A_0$ are connected to the $\overline{\text{STRB0}}$ memory bank address pins $A_{23}A_{22}A_{21}...A_1A_0$. But ’C32 address pins $A_{22}A_{21}...A_1A_0$ A_{-1} are connected to the $\overline{\text{STRB1}}$ memory-bank address pins $A_{23}A_{22}A_{21}...A_1A_0$.

Executing the following code on this device results in the data-access sequence shown in Table 10–2.

```
1)  LDI    4000h, AR1  ; AR1 = 4000h
2)  LDI    *AR1++, R2  ; R2 = *4000h and AR1 = AR1 + 1
3)  ADDI   *AR1++, R2  ; R2 = R2 + *4001h and AR1 = AR1 + 1
4)  ADDI   *AR1++, R2  ; R2 = R2 + *4002h and AR1 = AR1 + 1
5)  ADDI   *AR1++, R2  ; R2 = R2 + *4003h and AR1 = AR1 + 1
6)  LDI     900h, AR2  ; AR2 = 900h
7)  LSH     12, AR2    ; AR2 = 900000h
8)  LDI    *AR2++, R3  ; R3 = *900000h and AR2 = AR2 + 1
9)  ADDI    *AR2, R3   ; R3 = R3 + 900001h
```

By setting the bit fields of the $\overline{\text{STRB0}}$ bus control register with a physical-memory width of 32 bits and a data type size of 32 bits, the external address referring to the $\overline{\text{STRB0}}$ location is identical to the internal address used by the ‘C32 CPU. Alternatively, setting the bit fields of the $\overline{\text{STRB1}}$ bus control register with a physical memory width of 16-bit and a data-type size of 16-bit, the address presented by the ‘C32 external pins is the internal address shifted right by one bit with A_{23} driving A_{23} and A_{22}. Since the $\overline{\text{STRB1}}$ memory-bank address pins $A_{23}A_{22}A_{21}...A_{1}A_{0}$ are connected to the ‘C32 address pins $A_{22}A_{21}...A_{1}A_{0}A_{-1}$, the address seen by the $\overline{\text{STRB1}}$ memory bank is identical to the ‘C32 CPU internal address.

Table 10–2. Data-Access Sequence for a Memory Configuration with Two Banks

Instruction #	Internal Address Bus	External Address Pins	Active Strobe Byte Enable	Accessed Data Pins
(2)	4000h	4000h	$\overline{\text{STRB0\_B0/B1/B2/B3}}$	$D_{31–0}$ 4000h
(3)	4001h	4001h	$\overline{\text{STRB0\_B0/B1/B2/B3}}$	$D_{31–0}$ 4001h
(4)	4002h	4002h	$\overline{\text{STRB0\_B0/B1/B2/B3}}$	$D_{31–0}$ 4002h
(5)	4003h	4003h	$\overline{\text{STRB0\_B0/B1/B2/B3}}$	$D_{31–0}$ 4003h
(8)	900000h	C80000h	$\overline{\text{STRB1\_B0/B1}}$ and $\overline{\text{STRB1\_B3}}/A_{-1} = 0$	$D_{15–0}$ 900000h
(9)	900001h	C80001h	$\overline{\text{STRB1\_B0/B1}}$ and $\overline{\text{STRB1\_B3}}/A_{-1} = 1$	$D_{15–0}$ 900001h

The ability of the ‘C32 device to select a single byte from a single external memory location or combinations of bytes from several contiguous memory locations dictates that the internal address seen by the CPU correspond to a shifted version of the address presented to the external pins. The ’C32 external memory interface handles this conversion automatically as long as you configure the bus control register to match the external memory configuration present in your hardware implementation.

As seen in Figure 2–8 on page 2-20, ’C32 handles nine different memory access cases. The following sections discuss these cases in detail.

10.4 Programmable Wait States

The 'C3x has its own internal software-configurable ready-generation capability for each strobe. This software wait-state generator is controlled by configuring two fields in the primary or expansion bus interface control registers. Use the WTCNT field to specify the number of software wait states to generate and use the SWW field to select one of the following four modes of wait-state generation:

- External $\overline{\text{RDY}}$. Wait states are generated solely by the external $\overline{\text{RDY}}$ line ignoring software wait states.
- WTCNT-generated $\overline{\text{RDY}}$wtcnt. Wait states generated solely by the software wait-state generator ignoring external $\overline{\text{RDY}}$ signals.
- Logical-AND of $\overline{\text{RDY}}$ and $\overline{\text{RDY}}$wtcnt. Wait states generated with a logical AND of internal and external ready signals. Both signals must occur.
- Logical-OR of $\overline{\text{RDY}}$ and $\overline{\text{RDY}}$wtcnt. Wait states are generated with a logical OR of internal and external ready signals. Either signal can generate ready.

The four modes are used to generate the internal ready signal, $\overline{\text{RDY}}_{\text{int}}$, that controls accesses. As long as $\overline{\text{RDY}}_{\text{int}} = 1$, the current external access is delayed. When $\overline{\text{RDY}}_{\text{int}} = 0$, the current access completes. Since the use of programmable wait states for both external interfaces is identical, only the primary bus interface is described in the following paragraphs.

$\overline{\text{RDY}}_{\text{wtcnt}}$ is an internally generated ready signal. When an external access is begun, the value in WTCNT is loaded into a counter. WTCNT can be any value from 0 through 7. The counter is decremented every H1/H3 clock cycle until it becomes 0. Once the counter is set to 0, it remains set to 0 until the next access. While the counter is nonzero, $\overline{\text{RDY}}_{\text{wtcnt}} = 1$. While the counter is 0, $\overline{\text{RDY}}_{\text{wtcnt}} = 0$.

Table 10–3 shows the truth table for each value of SWW and the different combinations of $\overline{\text{RDY}}$, $\overline{\text{RDY}}_{\text{wtcnt}}$, and $\overline{\text{RDY}}_{\text{int}}$.

Note:

At reset, the 'C3x is programmed with seven wait states for each external memory access. These wait states are inserted to ensure the system can function with slow memories. To maximize system performance when accessing external memories, you need to decrease the number of wait states.

After changing wait states, up to three instructions will be fetched before the change in the wait-state occurs.

Table 10–3. Wait-State Generation

SWW Bit Field	Inputs		Output	Functional Description
	/RDYext	/RDYwtcnt	/RDYint	
00	0	x	0	Wait until external RDY is signaled
	1	x	1	
01	x	0	0	Wait until internal wait state generator counts down to 0
	x	1	1	
10	0	0	0	Wait until first signal: external RDY or the internal wait state generator (logical OR)
	0	1	0	
	1	0	0	
	1	1	1	
11	0	0	0	Wait until both external RDY is signaled and wait state generator counts down to 0 (logical AND)
	0	1	1	
	1	0	1	
	1	1	1	

10.5 Programmable Bank Switching

Programmable bank switching allows you to switch between external memory banks without having to insert wait states externally due to memories that require several cycles to turn off. Bank switching is implemented on STRB0 and STRB1 only.

The size of a bank is determined by the number of bits specified to be examined on the BNKCMP field of the primary bus control register. For example, if BNKCMP = 16, the 16 MSBs of the address are used to define a bank (see Figure 9–4). Since addresses are 24 bits, the bank size is specified by the eight LSBs, yielding a bank size of 256 words. If BNKCMP ≥ 16, only the 16 MSBs are compared. Bank sizes from 2^8 = 256 to 2^{24} = 16M are allowed. Table 9–6 summarizes the relationship between BNKCMP, the address bits used to define a bank, and the resulting bank size.

Figure 10–8. BNKCMP Example

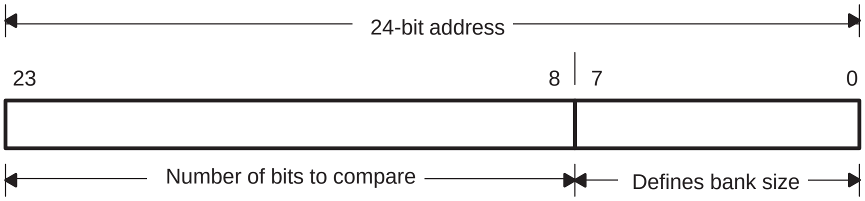


Table 10–4. BNKCMP and Bank Size

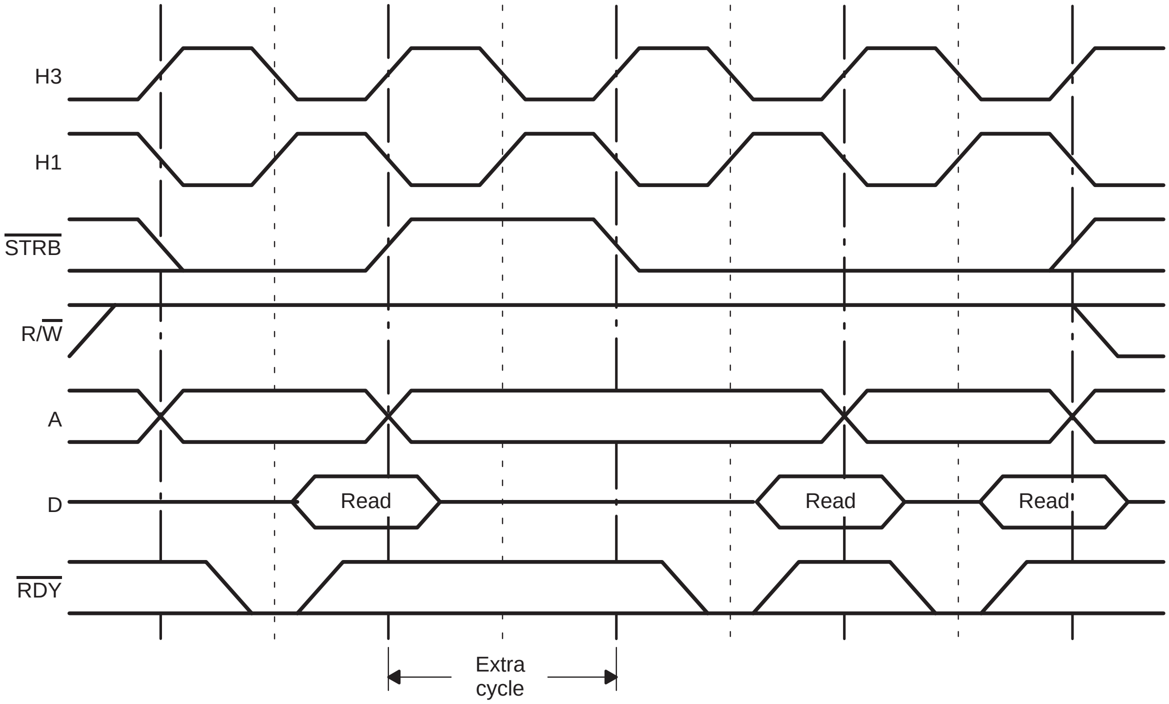
BNKCMP	MSBs Defining a Bank	Bank Size (32-Bit Words)
00000	None	2^{24} = 16M
00001	23	2^{23} = 8M
00010	23–22	2^{22} = 4M
00011	23–21	2^{21} = 2M
00100	23–20	2^{20} = 1M
00101	23–19	2^{19} = 512K
00110	23–18	2^{18} = 256K
00111	23–17	2^{17} = 128K
01000	23–16	2^{16} = 64K
01001	23–15	2^{15} = 32K
01010	23–14	2^{14} = 16K
01011	23–13	2^{13} = 8K
01100	23–12	2^{12} = 4K
01101	23–11	2^{11} = 2K
01110	23–10	2^{10} = 1K
01111	23–9	2^{9} = 512
10000	23–8	2^{8} = 256
10001–11111	Reserved	Undefined

The ’C3x has an internal register that contains the MSBs (as defined by the BNKCMP field) of the last address used for a read or write over the primary interface. At reset, the register bits are set to 0. If the MSBs of the address being used for the current primary interface read do not match those contained in this internal register, a read cycle is not asserted for one H1/H3 clock cycle. During this extra clock cycle, the address bus switches over to the new address, but $\overline{\text{STRB}}$ is inactive (high). The contents of the internal register are replaced with the MSBs being used for the current read of the current address. If the MSBs of the address being used for the current read match the bits in the register, a normal read cycle takes place.

If repeated reads are performed from the same memory bank, no extra cycles are inserted. When a read is performed from a different memory bank, memory conflicts are avoided by the insertion of an extra cycle. This feature can be disabled by setting BNKCMP to 0. The insertion of the extra cycle occurs only when a read is performed. The changing of the MSBs in the internal register occurs for all reads and writes over the primary interface.

Figure 9–5 shows the addition of an inactive cycle when switches between banks of memory occur.

Figure 10–9. Bank-Switching Example



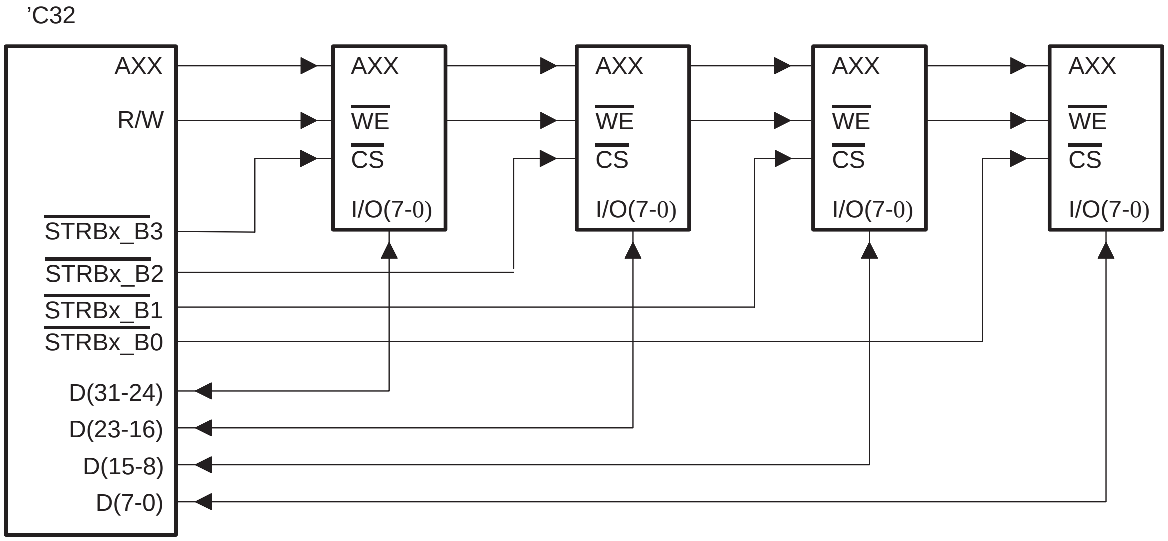
Note:

After changing BNKCMP, up to three instructions are fetched before the change in bank size occurs.

10.6 32-Bit-Wide Memory Interface

The ’C32 memory interface to 32-bit-wide external memory uses $\overline{\text{STRBx\_B3}}$ through $\overline{\text{STRBx\_B0}}$ pins as strobe-byte-enable pins as shown in Figure 10–10. In this manner, the ’C32 can read from, or write to, a single 32-, 16-, or 8-bit value from the external 32-bit-wide memory.

Figure 10–10. TMS320C32 External Memory Interface for 32-Bit SRAMs



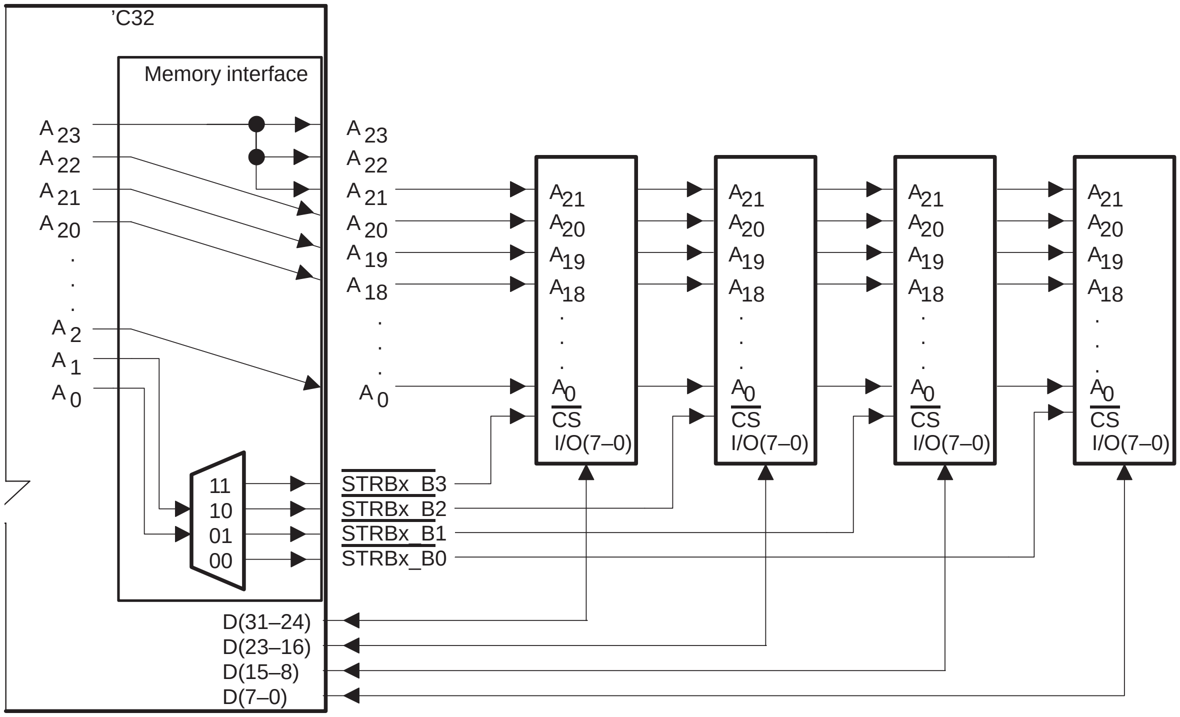
Case 1: 32-Bit-Wide Memory With 8-Bit Data-Type Size

When the data-type size is 8 bits, the ’C32 shifts the internal address two bits to the right before presenting it to the external-address pins. In this shift, the memory interface copies the value of the internal-address A_{23} to the external-address pins A_{23}, A_{22}, and A_{21}. The memory interface activates the $\overline{\text{STRBx\_B3}}$ through $\overline{\text{STRBx\_B0}}$ pins according to the value of the internal address bits A_1 and A_0 as shown in Table 10–5. Figure 10–11 shows a functional diagram of the memory interface for 32-bit-wide memory with an 8-bit data-type size.

Table 10–5. Strobe Byte-Enable for 32-Bit-Wide Memory With 8-Bit Data-Type Size

Internal A_1	Internal A_0	Active Strobe Byte Enable
0	0	$\overline{\text{STRBx\_B0}}$
0	1	$\overline{\text{STRBx\_B1}}$
1	0	$\overline{\text{STRBx\_B2}}$
1	1	$\overline{\text{STRBx\_B3}}$

Figure 10–11. Functional Diagram for 8-Bit Data-Type Size and 32-Bit External-Memory Width



For example, reading from or writing to memory locations 904000h to 904004h involves the pins listed in Table 10–6.

Table 10–6. Example of 8-Bit Data-Type Size

Internal Address Bus	External Address Pins	Active Strobe Byte Enable	Accessed Data Pins
904000h	E41000h	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$
904001h	E41000h	$\overline{\text{STRB1\_B1}}$	$D_{15–8}$
904002h	E41000h	$\overline{\text{STRB1\_B2}}$	$D_{23–16}$
904003h	E41000h	$\overline{\text{STRB1\_B3}}$	$D_{31–24}$
904004h	E41001h	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$

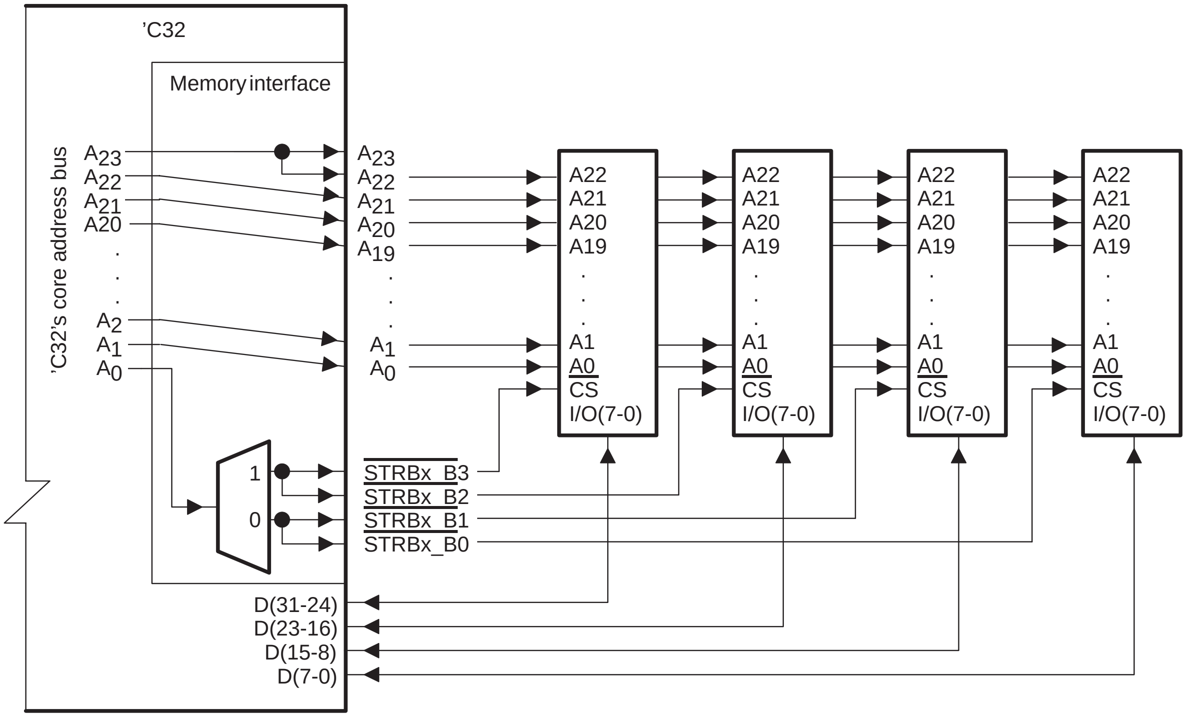
Case 2: 32-Bit-Wide Memory With 16-Bit Data-Type Size

When the data-type size is 16 bits, the ’C32 shifts the internal address one bit to the right before presenting it to the external-address pins. In this shift, the memory interface copies the value of the internal-address A_{23} to the external-address pins A_{23} and A_{22}. Also, the memory interface activates the $\overline{\text{STRBX-B3}}$ through $\overline{\text{STRBx\_B0}}$ pins according to the value of the internal address bit A_0 as shown in Table 10–7. Figure 10–12 shows a functional diagram of the memory interface for 32-bit-wide memory with 16-bit data-type size.

Table 10–7. Strobe Byte-Enable for 32-Bit-Wide Memory With 16-Bit Data-Type Size

Internal A_0	Active Strobe Byte Enable
0	$\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$
1	$\overline{\text{STRBx\_B3}}$ and $\overline{\text{STRBx\_B2}}$

Figure 10–12. Functional Diagram for 16-Bit Data-Type Size and 32-Bit External-Memory Width



For example, reading or writing to memory locations 904000h to 904004h involves the pins listed in Table 10–8.

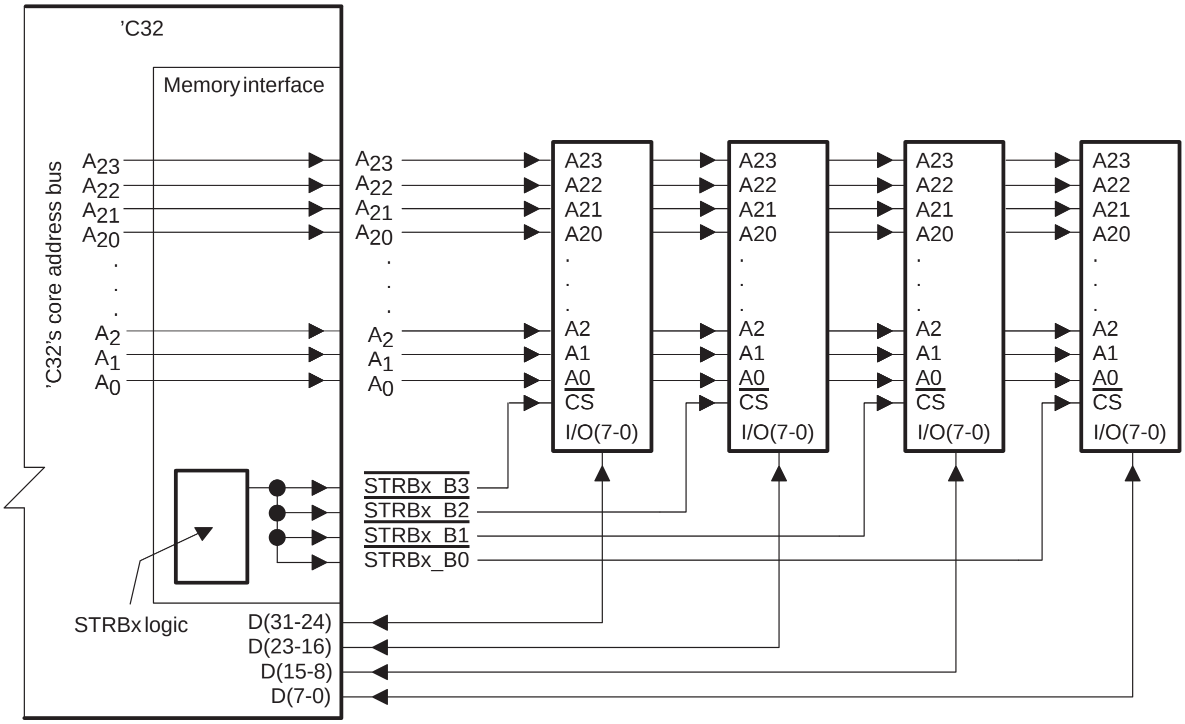
Table 10–8. Example of 16-Bit Data-Type Size and 32-Bit-Wide External Memory

Internal Address Bus	**External Address Pins**	**Active Strobe Byte Enable**	**Accessed Data Pins**
904000h	C82000h	$\overline{\text{STRB1\_B1}}$ and $\overline{\text{STRB1\_B0}}$	D_{15-0}
904001h	C82000h	$\overline{\text{STRB1\_B3}}$ and $\overline{\text{STRB1\_B2}}$	D_{31-16}
904002h	C82001h	$\overline{\text{STRB1\_B1}}$ and $\overline{\text{STRB1\_B0}}$	D_{15-0}
904003h	C82001h	$\overline{\text{STRB1\_B3}}$ and $\overline{\text{STRB1\_B2}}$	D_{31-16}
904004h	C82002h	$\overline{\text{STRB1\_B1}}$ and $\overline{\text{STRB1\_B0}}$	D_{15-0}

Case 3: 32-Bit-Wide Memory With 32-Bit Data-Type Size

When the data size is 32 bits, the ’C32 does not shift the internal address before presenting it to the external address pins. In this case, the memory interface copies the value of the internal address bus to the respective external-address pins. Also, the memory interface activates $\overline{\text{STRBx\_B3}}$ through $\overline{\text{STRBx\_B0}}$ pins during accesses. Figure 10–13 shows a functional diagram of the memory interface for 32-bit-wide memory with 32-bit data size.

Figure 10–13. Functional Diagram for 32-Bit Data Size and 32-Bit External-Memory Width



For example, reading or writing to memory locations 904000h to 904004h involves the pins listed in Table 10–9.

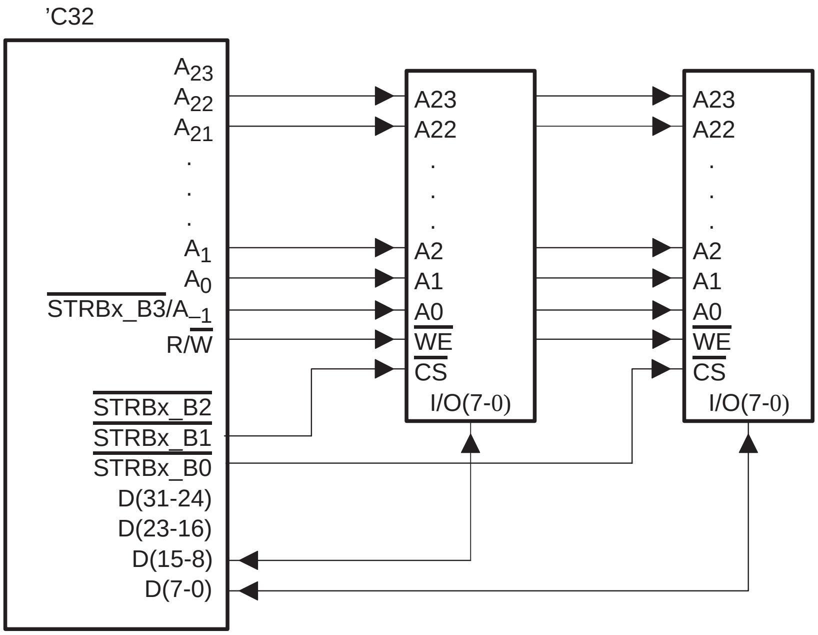
Table 10–9. Example of 32-Bit-Wide Memory With 32-Bit Data-Type Size

Internal Address Bus	**External Address Pins**	**Active Strobe Byte Enable**	**Accessed Data Pins**
904000h	904000h	$\overline{\text{STRB1\_B0}}$, $\overline{\text{STRB1\_B1}}$, $\overline{\text{STRB1\_B2}}$, and $\overline{\text{STRB1\_B3}}$	$D_{31–0}$
904001h	904001h	$\overline{\text{STRB1\_B0}}$, $\overline{\text{STRB1\_B1}}$, $\overline{\text{STRB1\_B2}}$, and $\overline{\text{STRB1\_B3}}$	$D_{31–0}$
904002h	904002h	$\overline{\text{STRB1\_B0}}$, $\overline{\text{STRB1\_B1}}$, $\overline{\text{STRB1\_B2}}$, and $\overline{\text{STRB1\_B3}}$	$D_{31–0}$
904003h	904003h	$\overline{\text{STRB1\_B0}}$, $\overline{\text{STRB1\_B1}}$, $\overline{\text{STRB1\_B2}}$, and $\overline{\text{STRB1\_B3}}$	$D_{31–0}$
904004h	904004h	$\overline{\text{STRB1\_B0}}$, $\overline{\text{STRB1\_B1}}$, $\overline{\text{STRB1\_B2}}$, and $\overline{\text{STRB1\_B3}}$	$D_{31–0}$

10.7 16-Bit-Wide Memory Interface

The ’C32 memory interface to 16-bit-wide external memory uses $\overline{\text{STRBx\_B3}}$ pin as an additional address pin, $A_{–1}$, while using $\overline{\text{STRBx\_B0}}$ and $\overline{\text{STRBx\_B1}}$ as strobe byte-enable pins as shown in Figure 10–14. Note that the external-memory address pins are connected to the ’C32 address pins $A_{22}A_{21}...A_1A_0A_{–1}$. In this manner, the ’C32 can read/write a single 32-, 16-, or 8-bit value from the external 16-bit-wide memory.

Figure 10–14. External-Memory Interface for 16-Bit SRAMs



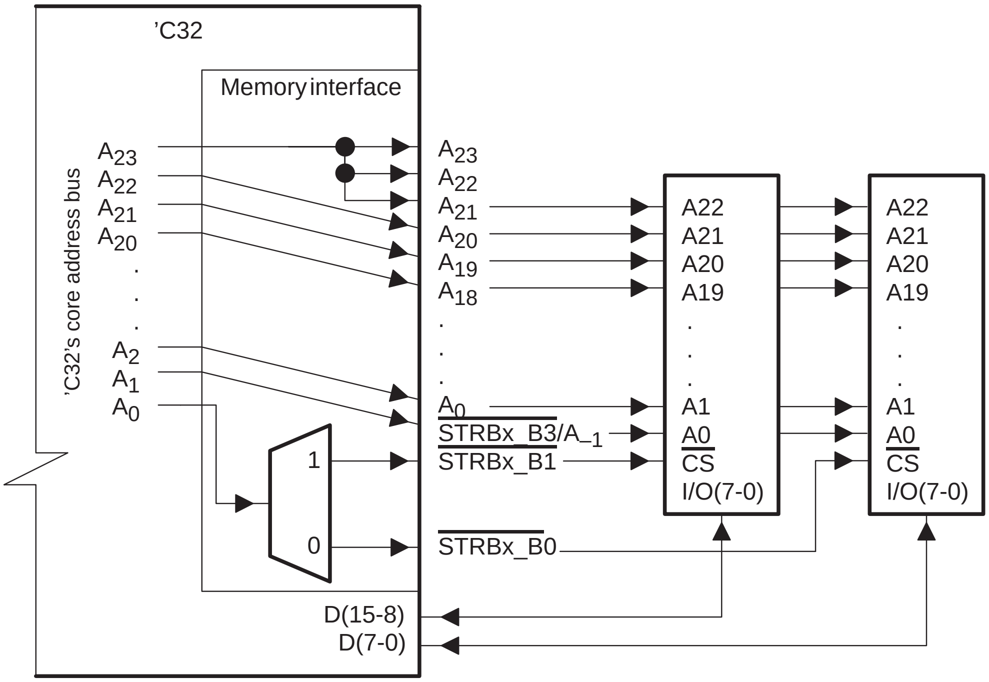
Case 4: 16-Bit-Wide Memory With 8-Bit Data-Type Size

When the data type size is 8 bits, the ’C32 shifts the internal address two bits to the right before presenting it to the external-address pins. In this shift, the memory interface copies the value of the internal-address A_{23} to the external-address pins A_{23}, A_{22}, and A_{21}. The memory interface also copies the value of the internal-address A_1 to the external $\overline{\text{STRBx\_B3}}$/$A_{–1}$ pin. Furthermore, the memory interface activates the $\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$ pins according to the value of the internal address bit A_0 as shown in Table 10–10. Figure 10–15 shows a functional diagram of the memory interface for 16-bit-wide memory with 8-bit data-type size.

Table 10–10. Strobe-Byte Enable Behavior for 16-Bit-Wide Memory with 8-Bit Data-Type Size

Internal A_0	Active Strobe Byte Enable
0	$\overline{\text{STRBx\_B0}}$
1	$\overline{\text{STRBx\_B1}}$

Figure 10–15. Functional Diagram for 8-Bit Data-Type Size and 16-Bit External-Memory Width



For example, reading or writing to memory locations 4000h to 4004h involves the pins listed in Table 10–11.

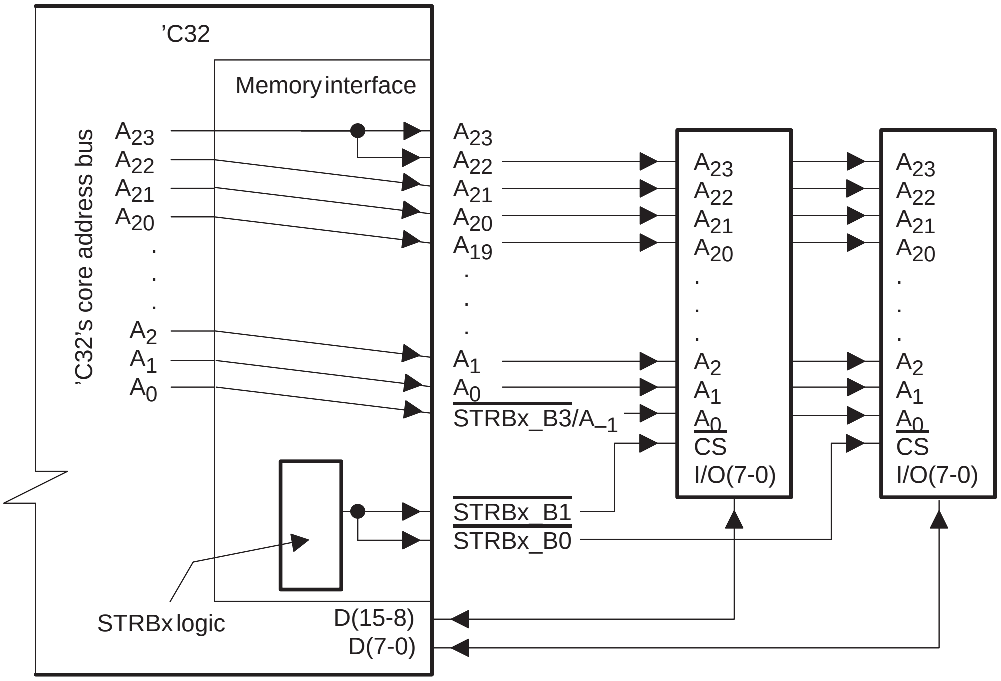
Table 10–11. Example of 8-Bit Data-Type Size and 16-Bit-Wide External Memory

Internal Address Bus	External Address Pins	$\overline{\text{STRB0\_B3}}/A_{-1}$	Active Strobe Byte Enable	Accessed Data Pins
4000h	1000h	0	$\overline{\text{STRB0\_B0}}$	D_{7-0}
4001h	1000h	0	$\overline{\text{STRB0\_B1}}$	D_{15-8}
4002h	1000h	1	$\overline{\text{STRB0\_B0}}$	D_{7-0}
4003h	1000h	1	$\overline{\text{STRB0\_B1}}$	D_{15-8}
4004h	1001h	0	$\overline{\text{STRB0\_B0}}$	D_{7-0}

Case 5: 16-Bit-Wide Memory With 16-Bit Data-Type Size

When the data-type size is 16 bits, the 'C32 shifts the internal address one bit to the right before presenting it to the external address pins. In this shift, the memory interface copies the value of the internal-address A_{23} to the external-address pins A_{23} and A_{22}. Also, the memory interface copies the value of the internal-address A_1 to the external $\overline{\text{STRBx\_B3}}/A_{-1}$ pin. Moreover, the memory interface activates the $\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$ during accesses. Figure 10–16 shows a functional diagram of the memory interface for 16-bit-wide memory with 16-bit data-type size.

Figure 10–16. Functional Diagram for 16-Bit Data-Type Size and 16-Bit External-Memory Width



For example, reading or writing to memory locations 4000h to 4004h involves the pins listed in Table 10–12.

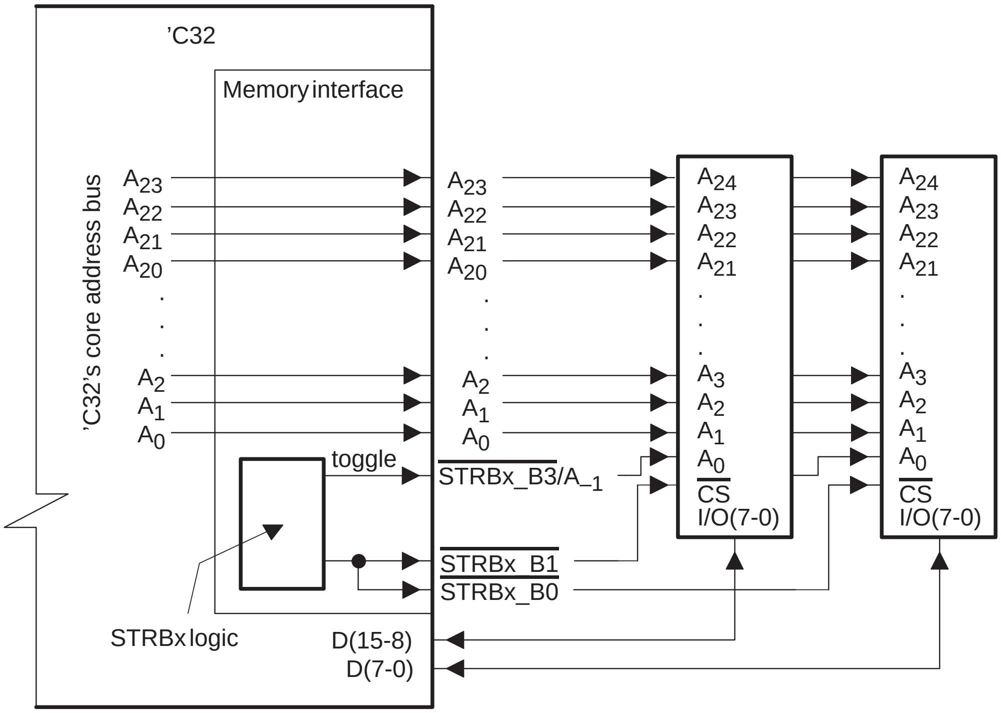
Table 10–12. Example of 16-Bit-Wide Memory With 16-Bit Data-Type Size

Internal Address Bus	External Address Pins	$\overline{\text{STRB0\_B3}}/A_{-1}$	Active Strobe Byte Enable	Accessed Data Pins
4000h	2000h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	$D_{15–0}$
4001h	2000h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	$D_{15–0}$
4002h	2001h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	$D_{15–0}$
4003h	2001h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	$D_{15–0}$
4004h	2002h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	$D_{15–0}$

Case 6: 16-Bit-Wide Memory with 32-Bit Data-Type Size

When the data type size is 32 bits, the 'C32 does not shift the internal address before presenting it to the external address pins. In this case, the memory interface copies the value of the internal address bus to the respective external address pins. The memory interface also toggles $\overline{\text{STRBx\_B3}}$/$A_{-1}$ twice to perform two 16-bit memory accesses. In the consecutive memory accesses, the memory interface activates $\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$. In summary, the memory interface seems to add one wait state to the 32-bit data access. Figure 10–17 depicts a functional diagram of the memory interface for 16-bit wide memory with 32-bit data type size.

Figure 10–17. Functional Diagram for 32-Bit Data-Type Size and 16-Bit External-Memory Width



For example, reading or writing to memory locations 4000h to 4004h involves the pins listed in Table 10–13.

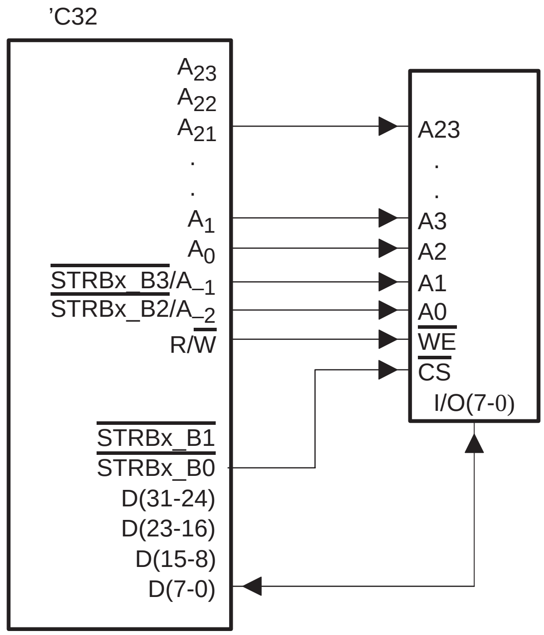
Table 10–13. Example of 16-Bit-Wide Memory With 32-Bit Data-Type Size

Internal Address Bus	External Address Pins	$\overline{\text{STRB0\_B3}}/A_{-1}$	Active Strobe Byte Enable	Accessed Data Pins
4000h	4000h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
	4000h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
4001h	4001h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
	4001h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
4002h	4002h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
	4002h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
4003h	4003h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
	4003h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
4004h	4004h	0	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}
	4004h	1	$\overline{\text{STRB0\_B0}}$ and $\overline{\text{STRB0\_B1}}$	D_{15-0}

10.8 8-Bit-Wide Memory Interface

’C32 memory interface to an 8-bit wide external memory uses $\overline{\text{STRBx\_B3}}$ and $\overline{\text{STRBx\_B2}}$ pins as additional address pins, A_{-1} and A_{-2}, respectively, while using $\overline{\text{STRBx\_B0}}$ as strobe byte-enable pin as shown in Figure 10–18. The external-memory address pins are connected to the ’C32’s address pins $A_{21}A_{20}...A_1A_0A_{-1}A_{-2}$. In this manner, the ’C32 can read/write a single 32-, 16-, or 8-bit value from the external 8-bit-wide memory.

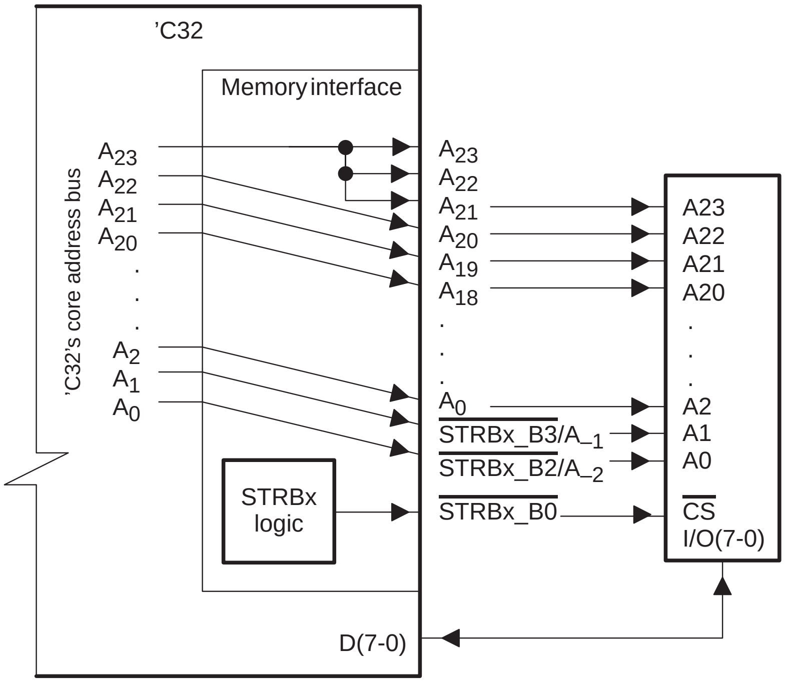
Figure 10–18. External Memory Interface for 8-Bit SRAMs



Case 7: 8-Bit-Wide Memory With 8-Bit Data-Type Size

Similarly to case 4, the ’C32 shifts the internal address two bits to the right before presenting it to the external-address pins when the data type is 8-bit. As in case 4, the memory interface copies the value of the internal-address A_{23} to the external-address pins A_{23}, A_{22}, and A_{21}. But in case 7, the memory interface also copies the value of the internal-address A_1 to the external $\overline{\text{STRBx\_B3}}/A_{-1}$ pin and the value of A_0 to the external $\overline{\text{STRBx\_B2}}/A_{-2}$. Moreover, the memory interface only activates the $\overline{\text{STRBx\_B0}}$ pin during the external memory access. Figure 10–19 shows a functional diagram of the memory interface for 8-bit-wide memory with an 8-bit data-type size.

Figure 10–19. Functional Diagram for 8-Bit Data-Type Size and 8-Bit External-Memory Width



For example, reading or writing to memory locations A04000h to A04004h involves the pins listed in Table 10–14.

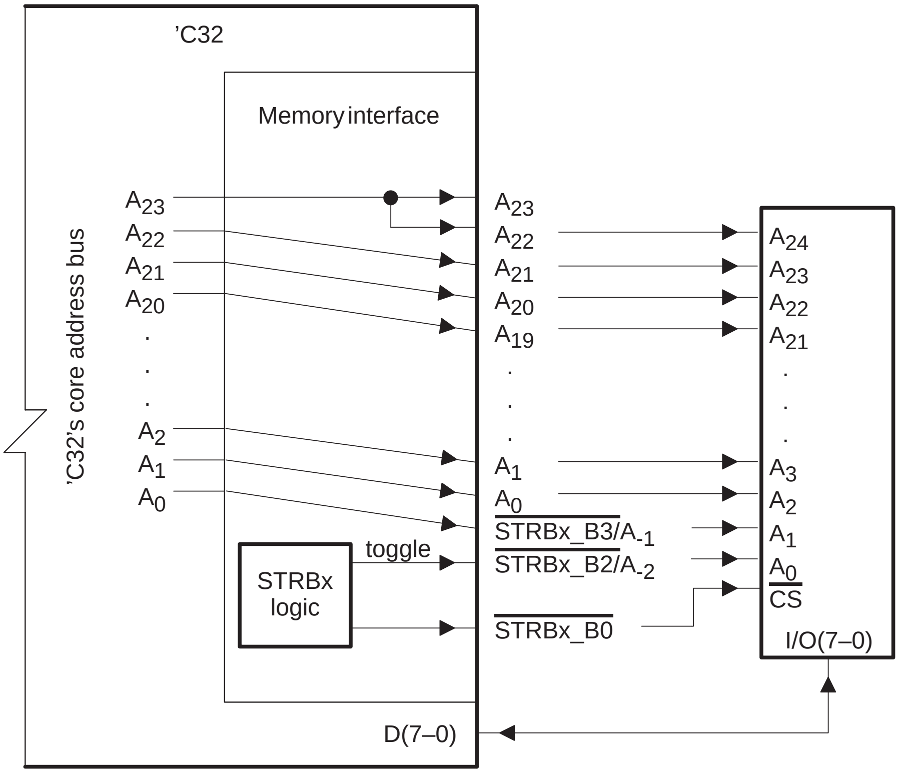
Table 10–14. Example of 8-Bit-Wide Memory With 8-Bit Data-Type Size

Internal Address Bus	External Address Pins	$\overline{\text{STRB0\_B3}}/A_{-1}$	$\overline{\text{STRB0\_B3}}/A_{-2}$	Active Strobe Byte Enable	Accessed Data Pins
A04000h	E81000h	0	0	$\overline{\text{STRB1\_B0}}$	D_{7-0}
A04001h	E81000h	0	1	$\overline{\text{STRB1\_B0}}$	D_{7-0}
A04002h	E81000h	1	0	$\overline{\text{STRB1\_B0}}$	D_{7-0}
A04003h	E81000h	1	1	$\overline{\text{STRB1\_B0}}$	D_{7-0}
A04004h	E81001h	0	0	$\overline{\text{STRB1\_B0}}$	D_{7-0}

Case 8: 8-Bit Wide Memory With 16-Bit Data-Type Size

When the data-type size is 16 bits, the ‘C32 shifts the internal address one bit to the right before presenting it to the external-address pins. In this shift, the memory interface copies the value of the internal-address A_{23} to the external-address pins A_{23} and A_{22}. Also, the memory interface copies the value of the internal-address A_0 to the external $\overline{\text{STRBx\_B3}}/A_{–1}$ pin. Furthermore, the memory interface toggles $\overline{\text{STRBx\_B2}}/A_{–2}$ twice to perform two 8-bit memory accesses. Moreover, the memory interface activates the $\overline{\text{STRBx\_B0}}$ during accesses. In summary, the memory interface adds one wait state to the 16-bit data access. Figure 10–20 shows a functional diagram of the memory interface for 8-bit-wide memory with 16-bit data-type size.

Figure 10–20. Functional Diagram for 16-Bit Data-Type Size and 8-Bit External-Memory Width



For example, reading or writing to memory locations A04000h to A04002h involves the pins listed in Table 10–15.

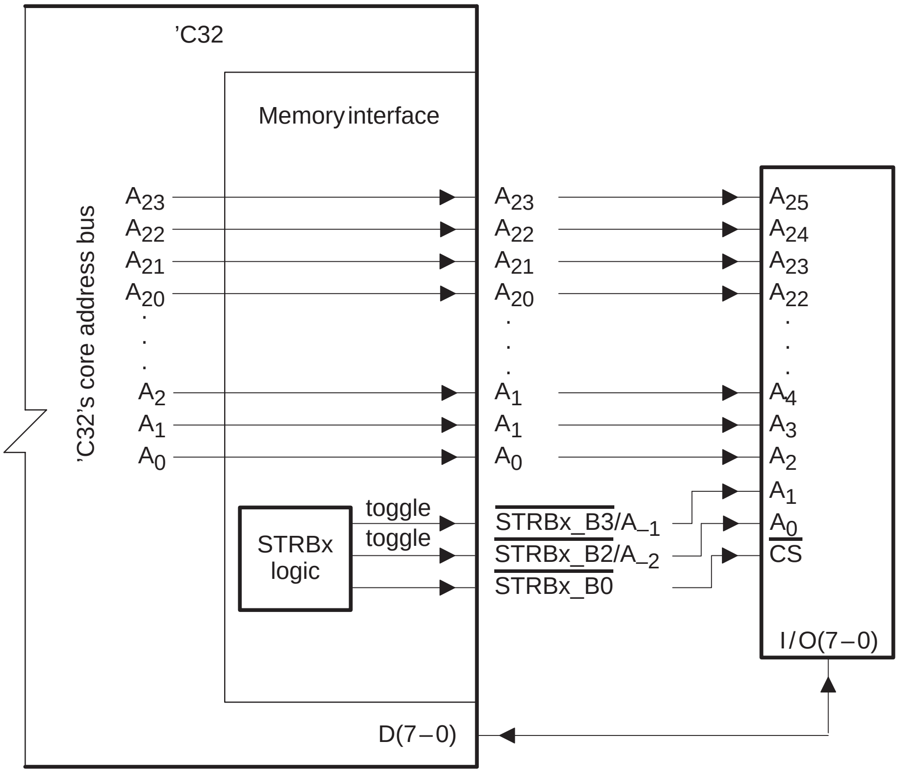
Table 10–15. Example of 8-Bit-Wide Memory With 16-Bit Data-Type Size

Internal Address Bus	External Address Pins	$\overline{\text{STRB0\_B3}}/\text{A}_{-1}$	$\overline{\text{STRB0\_B3}}/\text{A}_{-2}$	Active Strobe Byte Enable	Accessed Data Pins
A04000h	D02000h	0	0	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$
	D02000h	0	1	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$
A04001h	D02001h	1	0	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$
	D02001h	1	1	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$
A04002h	D02002h	0	0	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$
	D02002h	0	1	$\overline{\text{STRB1\_B0}}$	$D_{7–0}$

Case 9: 8-Bit-Wide Memory With 32-Bit Data-Type Size

When the data-type size is 32 bits, the ‘C32 does not shift the internal address before presenting it to the external-address pins. In this case, the memory interface copies the value of the internal-address bus to the respective external-address pins. The memory interface also toggles $\overline{\text{STRBx\_B3}}/\text{A}_{-1}$ and $\overline{\text{STRBx\_B2}}/\text{A}_{-2}$ to perform four 8-bit memory accesses. In the consecutive memory accesses, the memory interface activates $\overline{\text{STRBx\_B0}}$. In summary, the memory interface adds three wait states to the 32-bit data access. Figure 10–21 shows a functional diagram of the memory interface for 8-bit-wide memory with 32-bit data-type size.

Figure 10–21. Functional Diagram for 32-Bit Data-Type Size and 8-Bit External-Memory Width



For example, reading or writing to memory locations A04000h to A04001h involves the pins listed in Table 10–16.

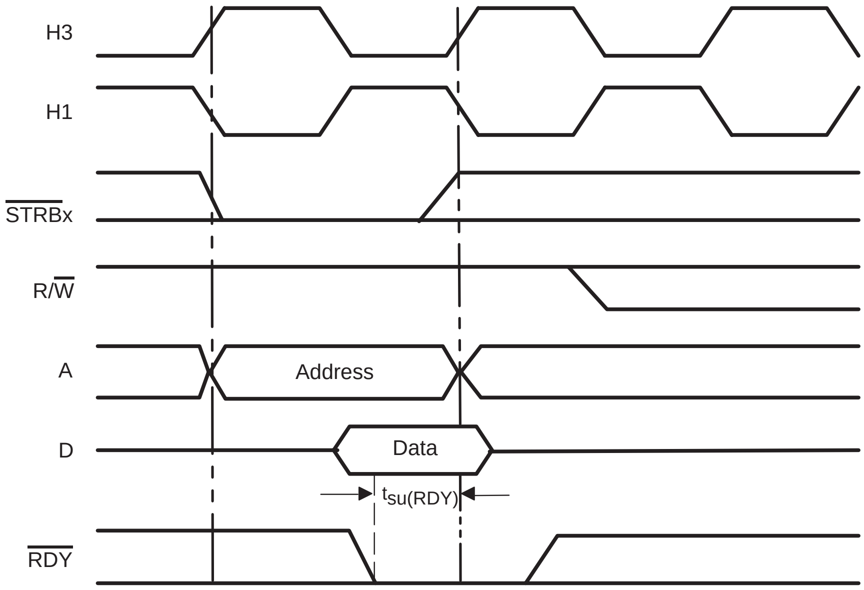
Table 10–16. Example of 32-Bit Data-Type Size and 8-Bit-Wide Memory

Internal Address Bus	External Address Pins	$\overline{\mathrm{STRB0\_B3}}/A_{-1}$	$\overline{\mathrm{STRB0\_B3}}/A_{-2}$	Active Strobe Byte Enable	Accessed Data Pins
A04000h	A04000h	0	0	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
	A04000h	0	1	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
	A04000h	1	0	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
	A04000h	1	1	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
A04001h	A04001h	0	0	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
	A04001h	0	1	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
	A04001h	1	0	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$
	A04001h	1	1	$\overline{\mathrm{STRB1\_B0}}$	$D_{7–0}$

10.9 External Ready Timing Improvement

The ready ($\overline{\text{RDY}}$) timing should relate to the H1 low signal as shown in Figure 10–22. This is equivalent to the ’C4x ready timing, which increases the time between valid address and the sampling of $\overline{\text{RDY}}$. This facilitates the memory hardware interface by allowing a longer address decode-circuit response time to generate a ready signal.

Figure 10–22. $\overline{\text{RDY}}$ Timing for Memory Read



CAUTION

Do not change the $\overline{\text{RDY}}$ signal during its setup time [$t_{su(RDY)}$].

10.10 Bus Timing

This section discusses functional timing of operations on the external memory bus. Detailed timing specifications are contained in the *TMS320C32 Data Sheet*. The timing of $\overline{\mathrm{STRB0}}$ and $\overline{\mathrm{STRB1}}$ bus cycles is identical and discussed in subsection 10.10.1. The abbreviation $\overline{\mathrm{STRBx}}$ is used in references that pertain equally to $\overline{\mathrm{STRB0}}$ and $\overline{\mathrm{STRB1}}$. The $\overline{\mathrm{IOSTRB}}$ bus cycles are timed differently and are discussed in subsection 10.10.2.

10.10.1 $\overline{\mathrm{STRB0}}$ and $\overline{\mathrm{STRB1}}$ Bus Cycles

All bus cycles comprise integral numbers of H1 clock cycles. One H1 cycle is defined from one falling edge of H1 to the next falling edge of H1. For full speed (zero wait-state) accesses on $\overline{\mathrm{STRB0}}$ and $\overline{\mathrm{STRB1}}$, writes take two H1 cycles and reads take one cycle. However, if the read immediately follows a write, the read takes two cycles. Writes to internal memory take one cycle if no other accesses to that interface are in progress. The following discussion pertains to zero wait-state accesses, unless otherwise specified.

The $\overline{\mathrm{STRBx}}$ signal is low for the active portion of both reads and writes (one H1 cycle). Additionally, before and after the active portions of writes only ($\overline{\mathrm{STRBx}}$ low), there is a transition of one H1 cycle. During this transition cycle the following might occur:

- $\overline{\mathrm{STRBx}}$ is high.
- If required, R/$\overline{\mathrm{W}}$ changes state on the rising edge of H1.
- If required, address changes on the rising edge of H1 if the previous H1 cycle performed a write. If the previous H1 cycle performed a read, address changes on the falling edge of H1.

Figure 10–23 illustrates a zero wait-state read-read-write sequence for $\overline{\mathrm{STRBx}}$ active. The data is read as late in the cycle as possible to allow for the maximum access time from address valid. Although external writes take two cycles, writes to internal memory take one cycle if no other accesses to that interface are in progress. Similar to typical external interfaces, the R/$\overline{\mathrm{W}}$ signal does not change until $\overline{\mathrm{STRB0}}$ and $\overline{\mathrm{STRB1}}$ are deactivated.

Figure 10–23. Read-Read-Write Sequence for $\overline{\text{STRBx}}$ Active

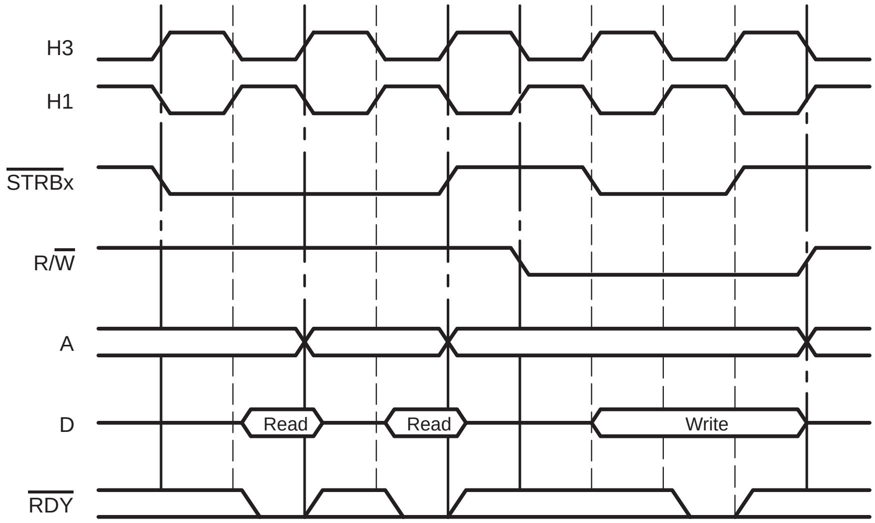


Figure 10–24 shows a zero wait-state write-write-read sequence for $\overline{\text{STRBx}}$ active. During back-to-back writes, the data is valid when $\overline{\text{STRBx}}$ changes for the first write, but for subsequent writes the data is valid when the address changes.

Figure 10–24. Write-Write-Read Sequence for $\overline{\text{STRBx}}$ Active

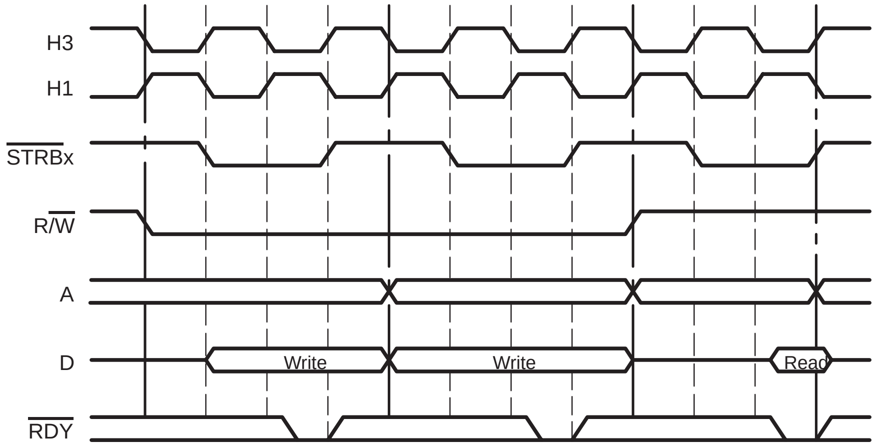


Figure 10–25 shows a one wait-state read sequence and Figure 10–26 shows the write sequence for $\overline{\text{STRB}}$x active. On the first H1 cycle, $\overline{\text{RDY}}$ is high; therefore, the read or write sequence is extended for one extra cycle. On the second H1 cycle, $\overline{\text{RDY}}$ is low and the read or write sequence is terminated.

Figure 10–25. One Wait-State Read Sequence for $\overline{\text{STRBx}}$ Active

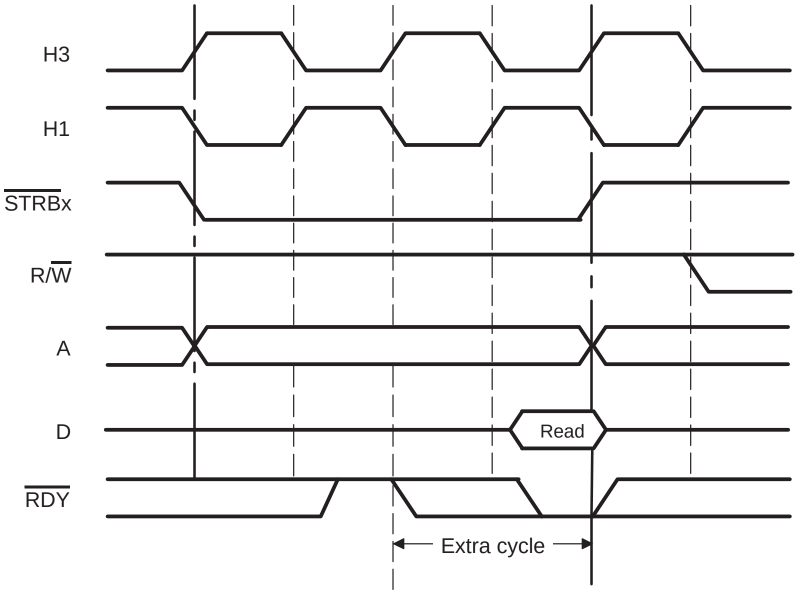
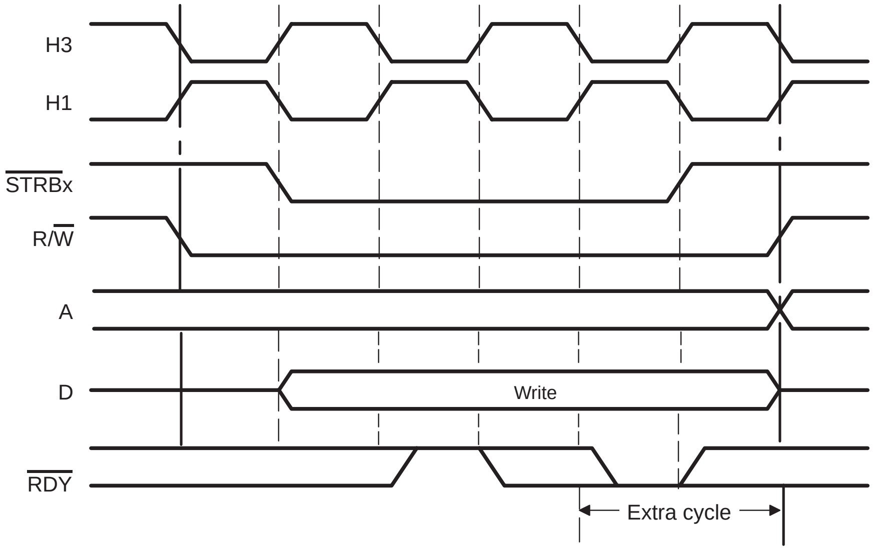


Figure 10–26. One Wait-State Write Sequence for $\overline{\text{STRBx}}$ Active



10.10.2 $\overline{\text{IOSTRB}}$ Bus Cycles

In contrast to $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ bus cycles, $\overline{\text{IOSTRB}}$ full speed (zero wait-state) reads and writes consume two H1 cycles. During these cycles, the $\overline{\text{IOSTRB}}$ signal is low from the rising edge of the first H1 cycle to the rising edge of the second H1 cycle. Also, the address changes on the falling edge of the first H1 cycle and R/$\overline{\text{W}}$ changes state on the falling edge of H1. This provides a valid address to peripherals that may change their status bits when read or written while $\overline{\text{IOSTRB}}$ is active. Moreover, the $\overline{\text{IOSTRB}}$ signal is high between $\overline{\text{IOSTRB}}$ read and write cycles.

Figure 10–27 illustrates a zero wait-state read and write sequence for $\overline{\text{IOSTRB}}$ active. During writes, the data is valid when $\overline{\text{IOSTRB}}$ changes.

Figure 10–27. Zero Wait-State Read and Write Sequence for $\overline{\text{IOSTRB}}$ Active

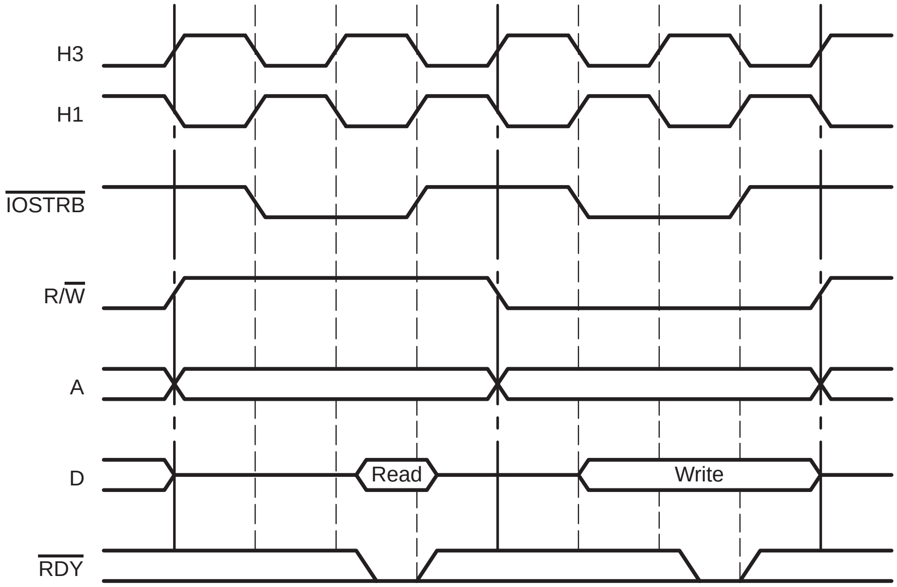


Figure 10–28 depicts a one wait-state read sequence for $\overline{\text{IOSTRB}}$ active. Figure 10–29 shows a one wait-state write sequence for $\overline{\text{IOSTRB}}$ active. For each wait-state added, $\overline{\text{IOSTRB}}$, R/$\overline{\text{W}}$, and A are extended for one extra clock cycle. Writes hold the data on the bus for one extra clock cycle. $\overline{\text{RDY}}$ is sampled on each extra cycle and the sequence is terminated when $\overline{\text{RDY}}$ is low.

Figure 10–28. One Wait-State Read Sequence for $\overline{\text{IOSTRB}}$ Active

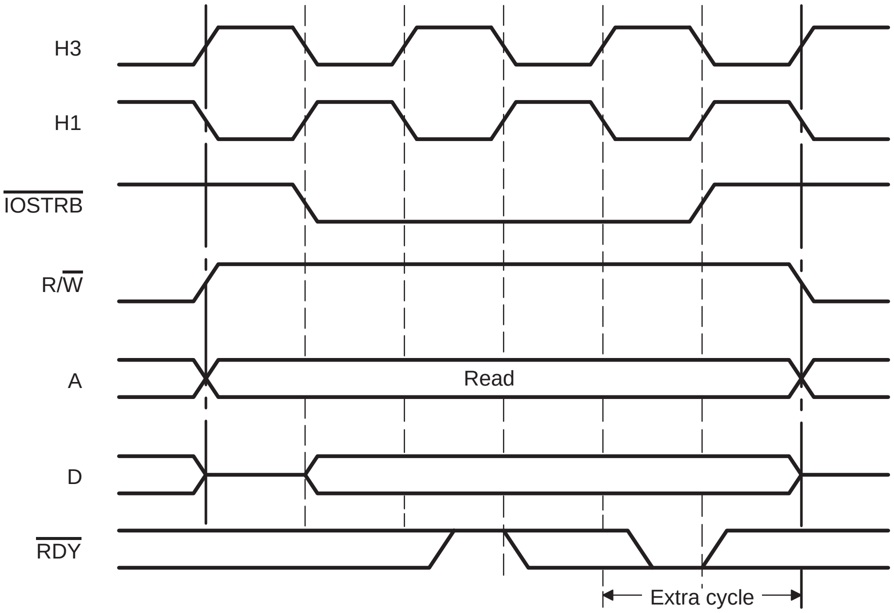


Figure 10–29. One Wait-State Write Sequence for $\overline{\text{IOSTRB}}$ Active

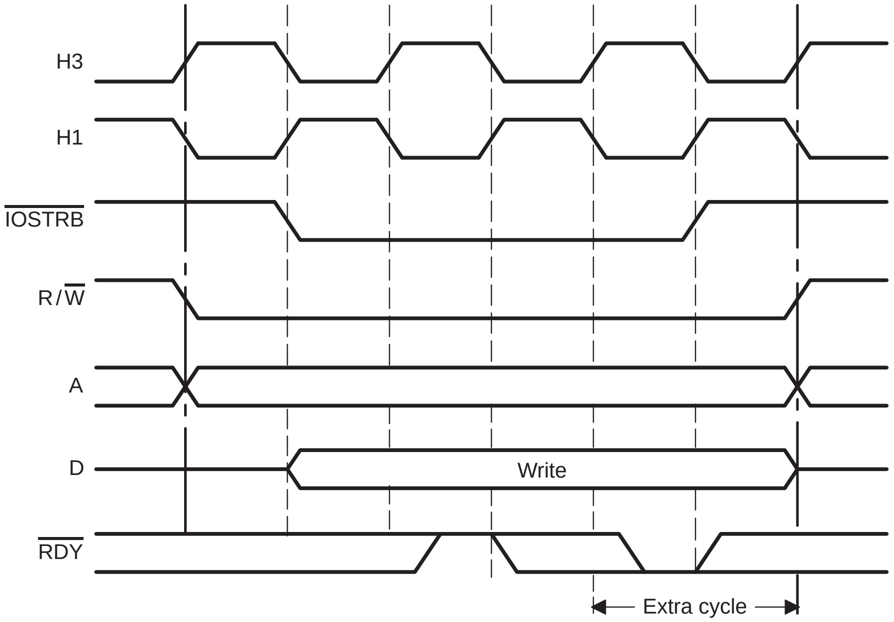


Figure 10–30 and Figure 10–31 illustrate the transitions between $\overline{\text{STRBx}}$ reads and $\overline{\text{IOSTRB}}$ writes and reads, respectively. In these transitions, the address changes on the falling edge of the H1 cycle.

Figure 10–30. $\overline{\text{STRBx}}$ Read and $\overline{\text{IOSTRB}}$ Write

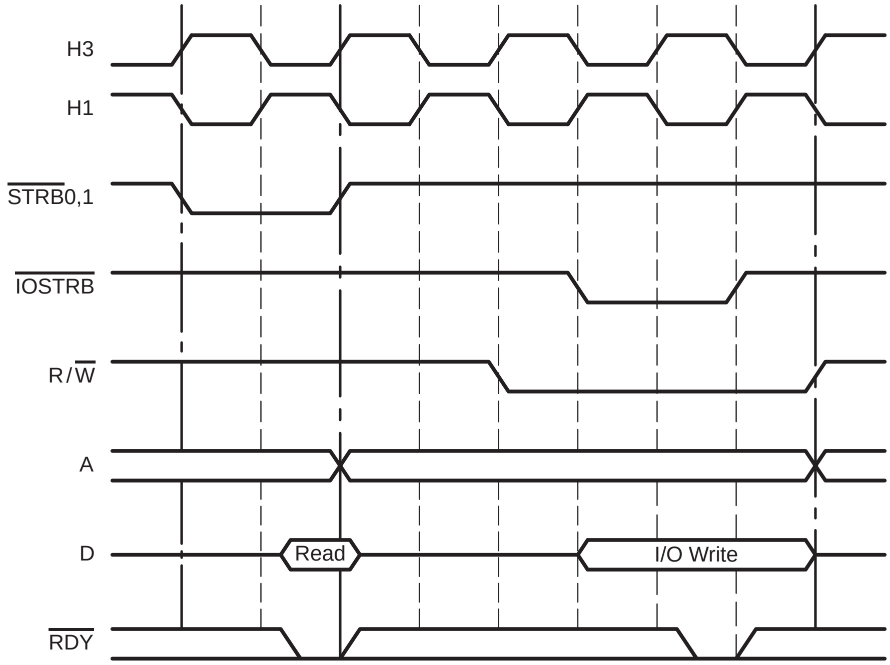

H3
H1
$\overline{\text{STRB}}$0,1
$\overline{\text{IOSTRB}}$
R / $\overline{\text{W}}$
A
D
Read
I/O Write
$\overline{\text{RDY}}$

Figure 10–31. $\overline{\text{STRBx}}$ Read and $\overline{\text{IOSTRB}}$ Read

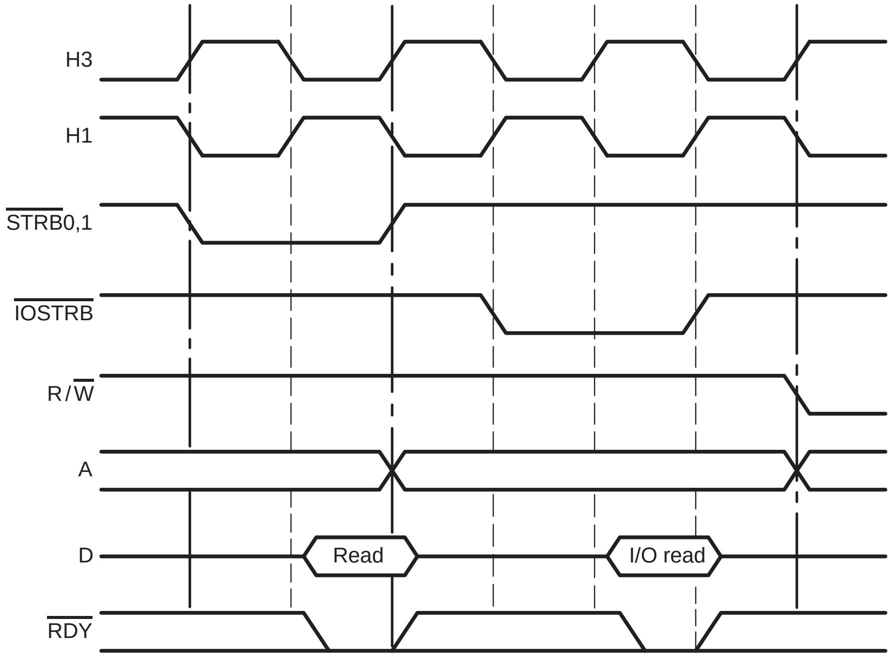

H3
H1
$\overline{\text{STRB}}$0,1
$\overline{\text{IOSTRB}}$
R / $\overline{\text{W}}$
A
D
Read
I/O read
$\overline{\text{RDY}}$

Figure 10–32 and Figure 10–33 illustrate the transitions between $\overline{\text{STRBx}}$ writes and $\overline{\text{IOSTRB}}$ writes and reads, respectively. In these transitions, the address changes on the falling edge of the H3 cycle.

Figure 10–32. $\overline{\text{STRBx}}$ Write and $\overline{\text{IOSTRB}}$ Write

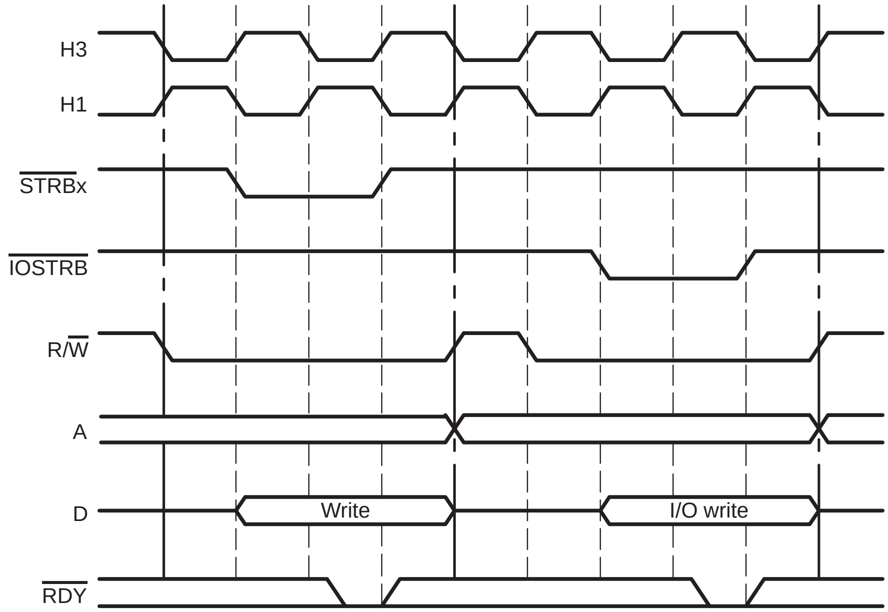


Figure 10–33. $\overline{\text{STRBx}}$ Write and $\overline{\text{IOSTRB}}$ Read

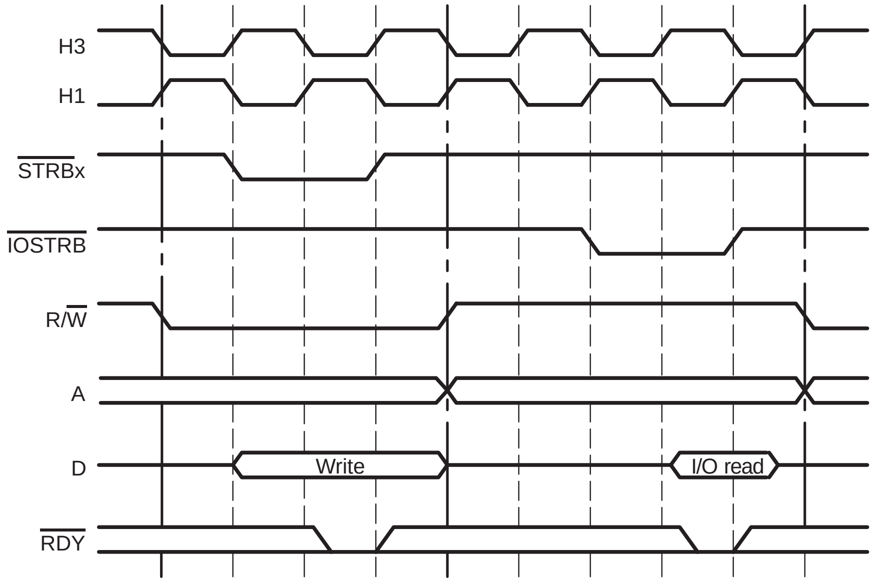


Figure 10–34 through Figure 10–37 show the transitions between $\overline{\text{IOSTRB}}$ writes/reads and $\overline{\text{STRBx}}$ writes/reads. In these transitions, the address changes on the rising edge of the H3 cycle.

Figure 10–34. $\overline{\text{IOSTRB}}$ Write and $\overline{\text{STRBx}}$ Write

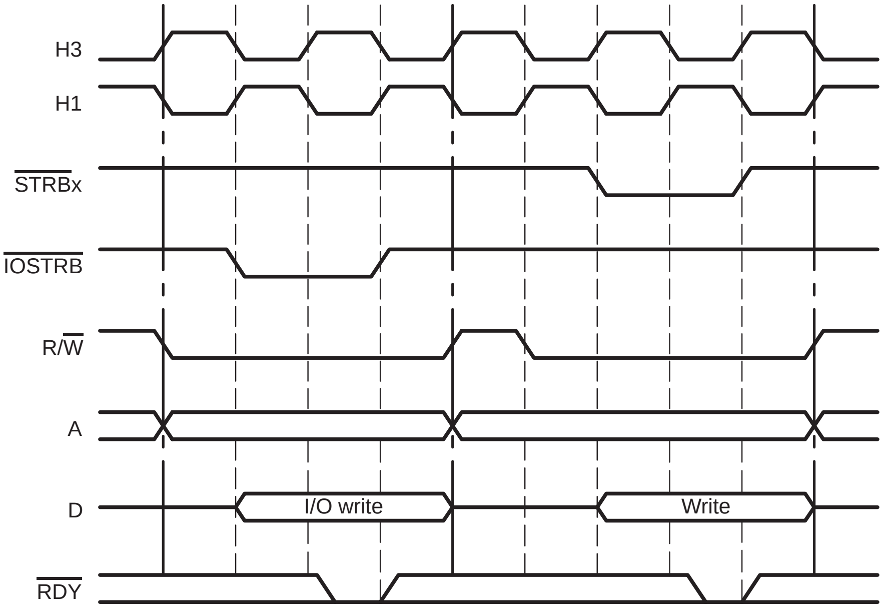


Figure 10–35. $\overline{\text{IOSTRB}}$ Write and $\overline{\text{STRBx}}$ Read

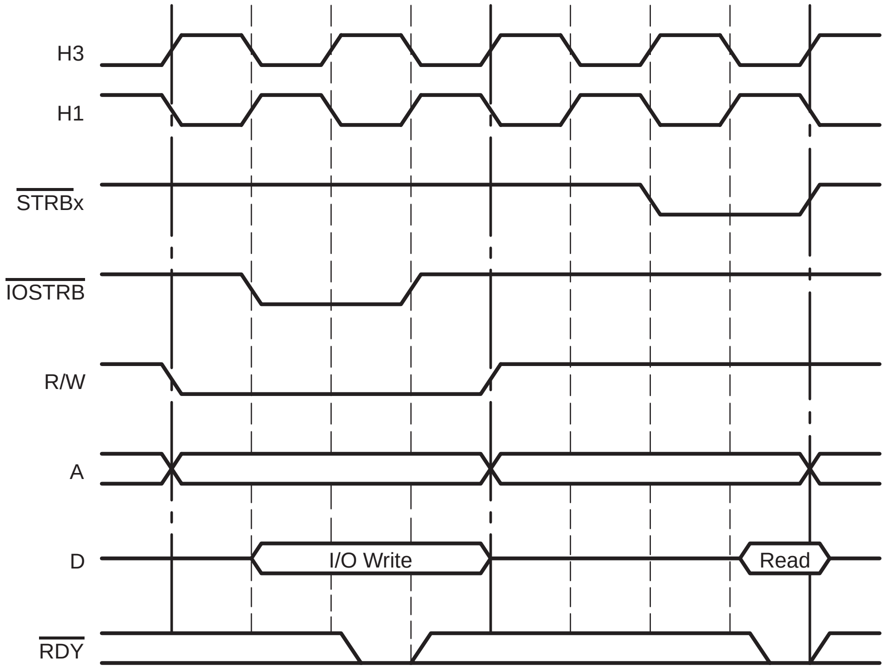


Figure 10–36. $\overline{\text{IOSTRB}}$ Read and $\overline{\text{STRBx}}$ Write

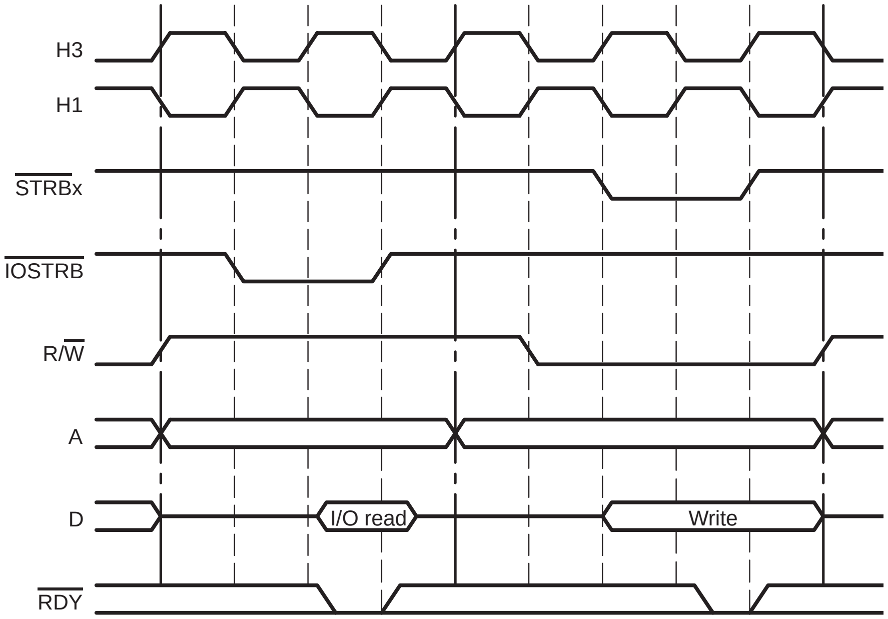


Figure 10–37. $\overline{\text{IOSTRB}}$ Read and $\overline{\text{STRBx}}$ Read

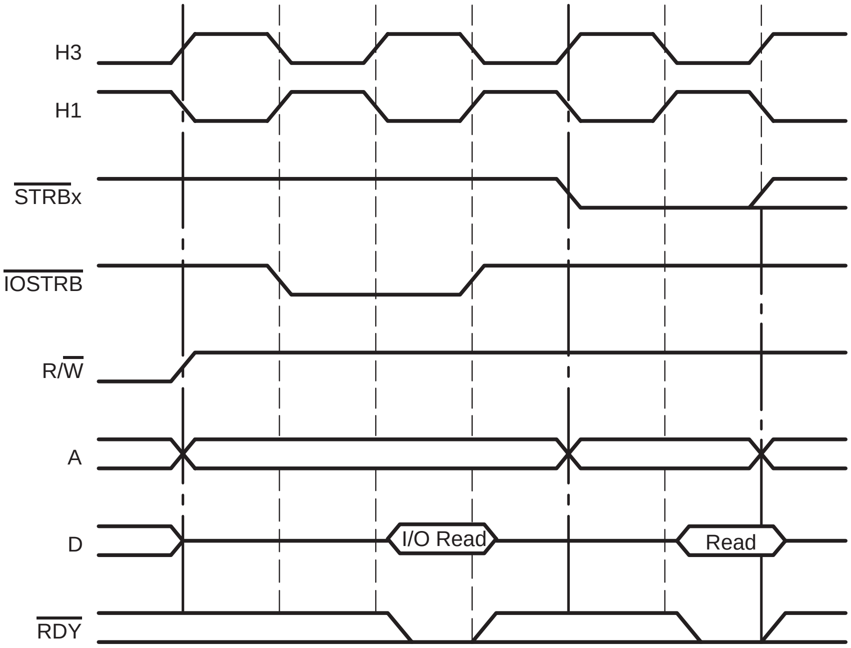


Figure 10–38 through Figure 10–40 illustrate the transitions between reads and writes.

Figure 10–38. $\overline{\text{IOSTRB}}$ Write and Read

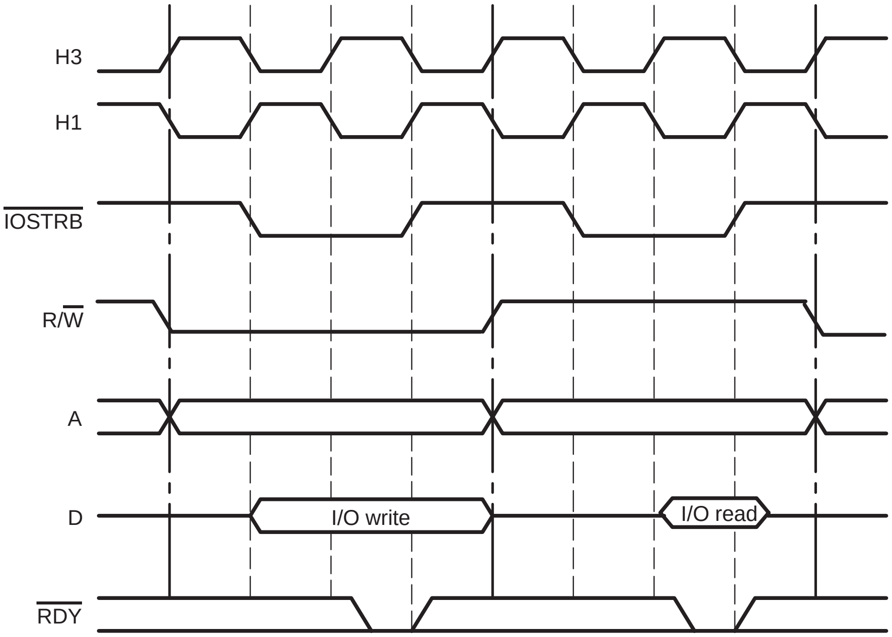


Figure 10–39. $\overline{\text{IOSTRB}}$ Write and Write

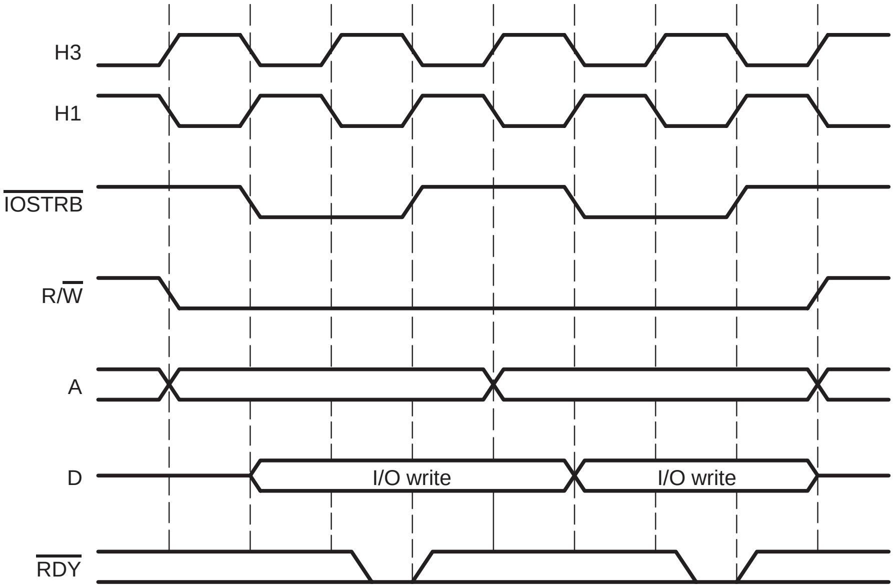
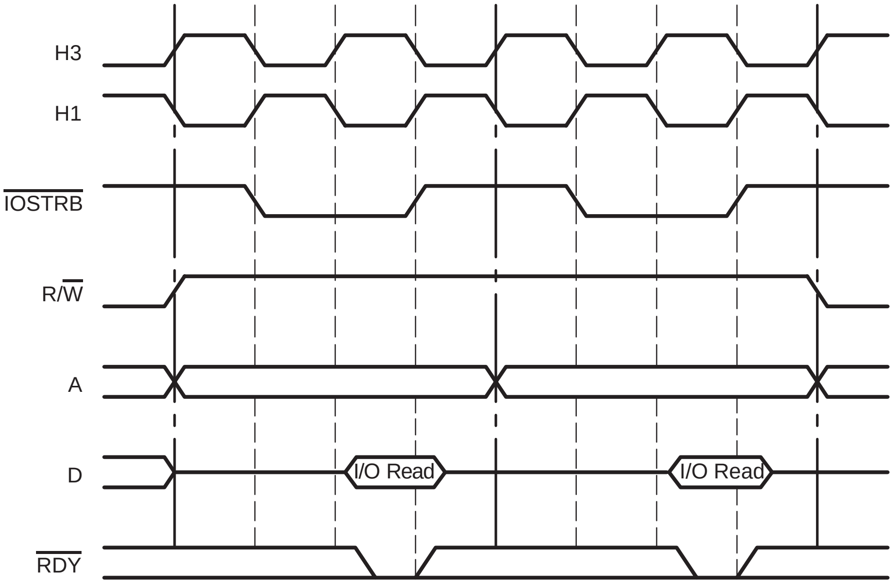


Figure 10–40. $\overline{\text{IOSTRB}}$ Read and Read



10.10.3 Inactive Bus States

Figure 10–41 and Figure 10–42 show the signal states when a bus becomes inactive after an $\overline{\text{IOSTRB}}$ or $\overline{\text{STRBx}}$, respectively. The strobes ($\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, $\overline{\text{IOSTRB}}$, and R/$\overline{\text{W}}$) are deasserted going to a high level. The address bus preserves the last value and the ready signal ($\overline{\text{RDY}}$) is ignored.

Figure 10–41. Inactive Bus States Following $\overline{\text{IOSTRB}}$ Bus Cycle

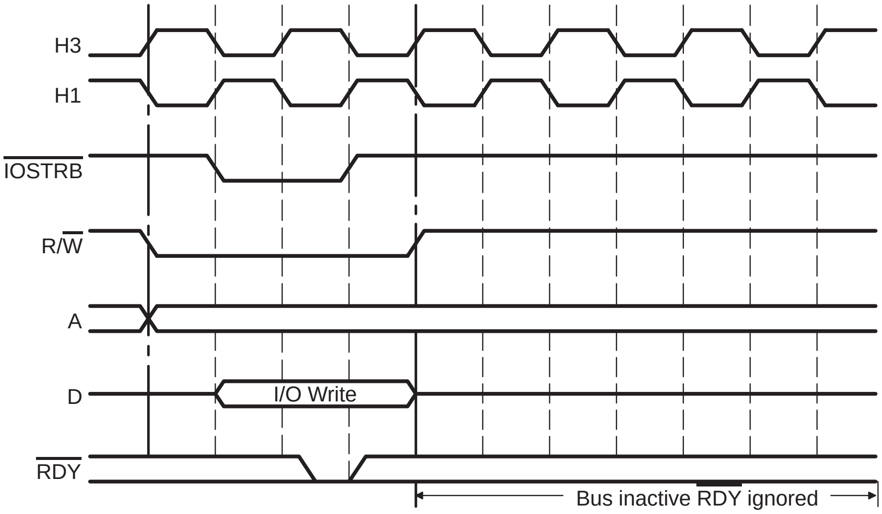
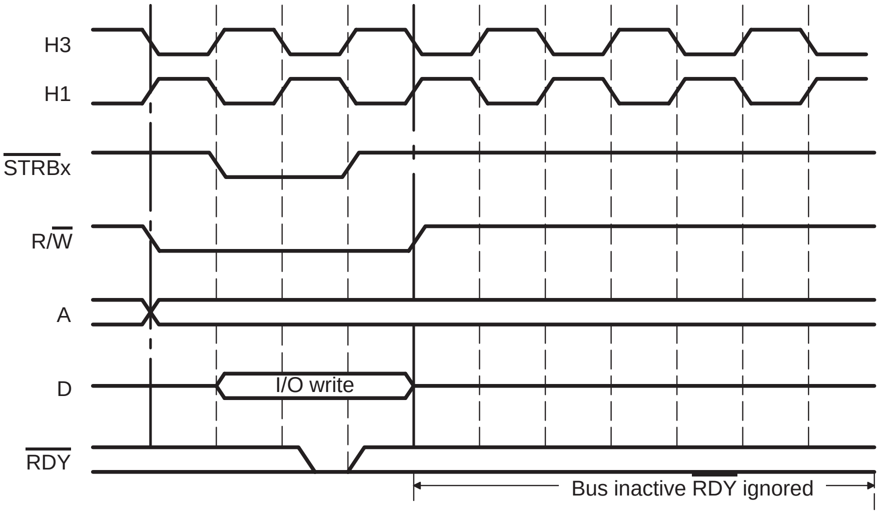

H3
H1
$\overline{\text{IOSTRB}}$
R/$\overline{\text{W}}$
A
D
I/O Write
$\overline{\text{RDY}}$
Bus inactive $\overline{\text{RDY}}$ ignored

Figure 10–42. Inactive Bus States Following $\overline{\text{STRBx}}$ Bus Cycle



Chapter 11

Using the TMS320C31 and TMS320C32 Boot Loaders

The ’C31 and ’C32 have on-chip boot loaders that can load and execute programs received from a host processor, standard memory devices (including EPROM), or via serial port.

Topic **Page**

11.1 TMS320C31 Boot Loader

This section describes how to use the ’C31 microcomputer/boot loader (MCBL/$\overline{\text{MP}}$) function. This feature is unique to the ’C31 and ’C32, and is not available on the ’C30 devices.

11.1.1 TMS320C31 Boot-Loader Description

The boot loader lets you load and execute programs that are received from a host processor, inexpensive EPROMs, or other standard memory devices. The programs to be loaded reside in one of three memory-mapped areas identified as Boot 1, Boot 2, and Boot 3 (see the shaded areas of Figure 4–2 on page 4-6), or they are received by means of the serial port.

The boot loader supports user-definable byte, half-word, and word-data formats, as well as 32-bit fixed-burst loads from the ’C31 serial port. See Section 12.2, *Serial Ports*, on page 12-15 for a detailed description of the serial-port operation.

The boot-loader code starts at location 0x45 in the on-chip ROM. The source code is supplied in Appendix B.

11.1.2 TMS320C31 Boot-Loader Mode Selection

The ’C31 boot loader functions as a memory boot loader or a serial-port boot loader. The boot-loader function is selected by resetting the processor while driving the MCBL/$\overline{\text{MP}}$ pin high. Use interrupt pins $\overline{\text{INT3}}$ – $\overline{\text{INT0}}$ to select the boot-load operation. Figure 11–1 shows the flow of this operation, which depends on the mode selected (external memory or serial boot).

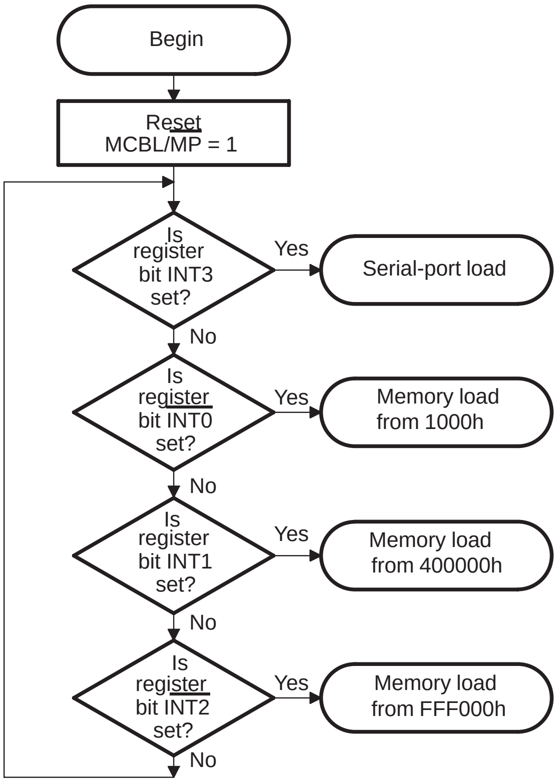
- The memory boot loader supports user-definable byte, half-word, and full-word data formats, allowing the flexibility to load a source program from memories having widths of 8-, 16-, or 32 bits. The source program must reside in one of three memory locations as listed in Table 11–1. Figure 11–2 shows the memory boot-loader flow.

- The serial-port boot loader supports 32-bit fixed-burst transfers, with externally generated serial-port clock and frame-sync signals. The format of the incoming data stream is similar to that of the memory boot loader, except the source memory width and memory configuration word are omitted. Figure 11–3 shows the serial-port boot-loader flow.

Table 11–1. Boot-Loader Mode Selection

$\overline{\mathrm{INT0}}$	$\overline{\mathrm{INT1}}$	$\overline{\mathrm{INT2}}$	$\overline{\mathrm{INT3}}$	Loader Mode	Memory Addresses
0	1	1	1	External memory	Boot 1 address 0x001000
1	0	1	1	External memory	Boot 2 address 0x400000
1	1	0	1	External memory	Boot 3 address 0xFFF000
1	1	1	0	32-bit serial	Serial port 0

Figure 11–1. TMS320C31 Boot-Loader Mode-Selection Flowchart



11.1.3 TMS320C31 Boot-Loading Sequence

The following is the sequence of events that occur during the boot load of a source program. Table 11–2 shows the structure of the source program.

1) Select the boot loader by resetting the ’C31 while driving the MCBL/$\overline{\text{MP}}$ pin high and the corresponding $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pin low. The MCBL/$\overline{\text{MP}}$ must stay high during boot loading, but can be changed anytime after boot loading has terminated. No reset is necessary when changing the $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pin, as long as the ’C31 is not accessing the overlapping memory (0h–FFFh) during this transition (see Section 11.1.6). The $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pin can be driven low anytime after deasserting the $\overline{\text{RESET}}$ pin (driven low and then high).

2) The status of the interrupt flag (IF) register’s $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ bit fields dictate the boot-loading mode. The bits are polled in the order described in the flow chart in Figure 11–1.

3) If only the IF register’s $\overline{\text{INT3}}$ bit field is set, the boot loader configures the serial port for 32-bit fixed-burst mode reads with an externally generated serial-port clock and FSR. Then, it proceeds to boot load the source program from the serial port. The transferred data-bit order supplied to the serial port must begin with the most significant bit (MSB) and end with the least significant bit (LSB). Figure 11–3 depicts the boot-loader serial-port flow.

4) Otherwise, the boot loader attempts a memory boot load. Figure 11–2 shows the boot-loader memory flow. If the IF register’s $\overline{\text{INT0}}$ bit field is set, the source program is loaded from memory location 1000h. If the IF register’s $\overline{\text{INT1}}$ bit field is set, the source program is loaded from memory location 400000h. If the IF register’s $\overline{\text{INT2}}$ bit field is set, the source program is loaded from memory location FFF000h.

   The memory boot-load source program has a header indicating the boot memory width and memory configuration control word. This word is copied into the $\overline{\text{STRB}}$ control register to configure the external primary bus interface.

5) After reading the header, the boot loader copies the source-program blocks. The source-program blocks have two entries preceding the source-program-block data. The first entry in the source-program block indicates the size of the block. A block size of zero signals the end of the source program code. The second entry indicates the address where the block is to be loaded. The boot loader cannot load the source program to any memory address below 1000h, unless the address decode logic is remapped.

6) The boot loader branches to the destination address of the first source block loaded and begins program execution.

Figure 11–2. Boot-Loader Memory-Load Flowchart

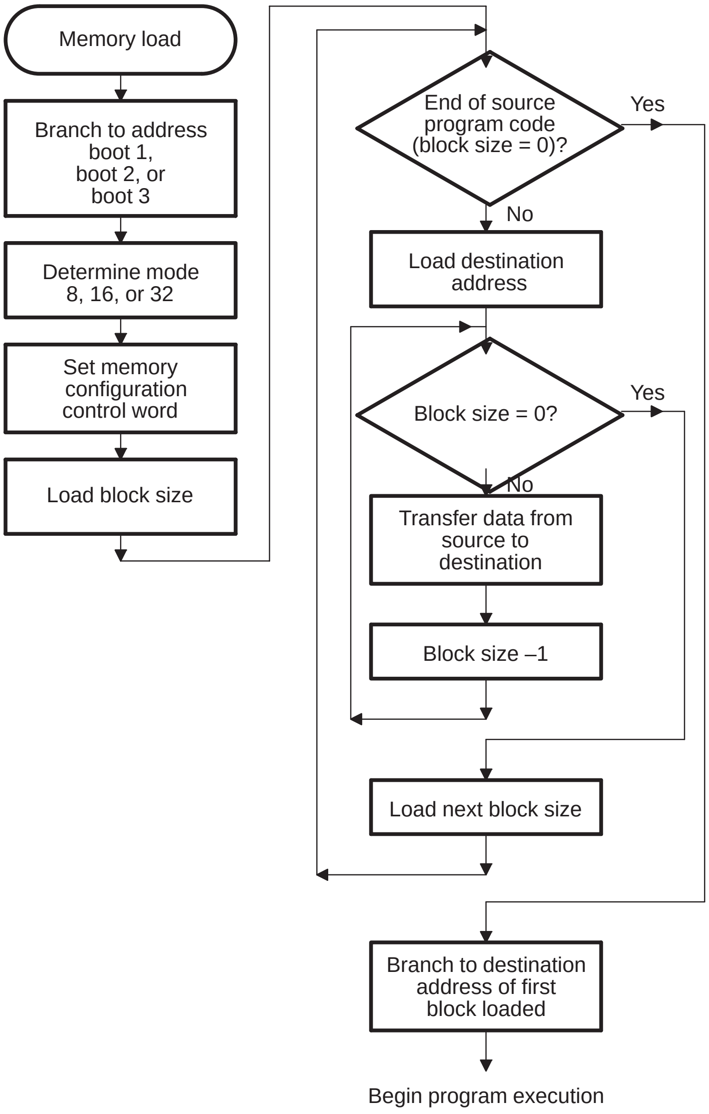
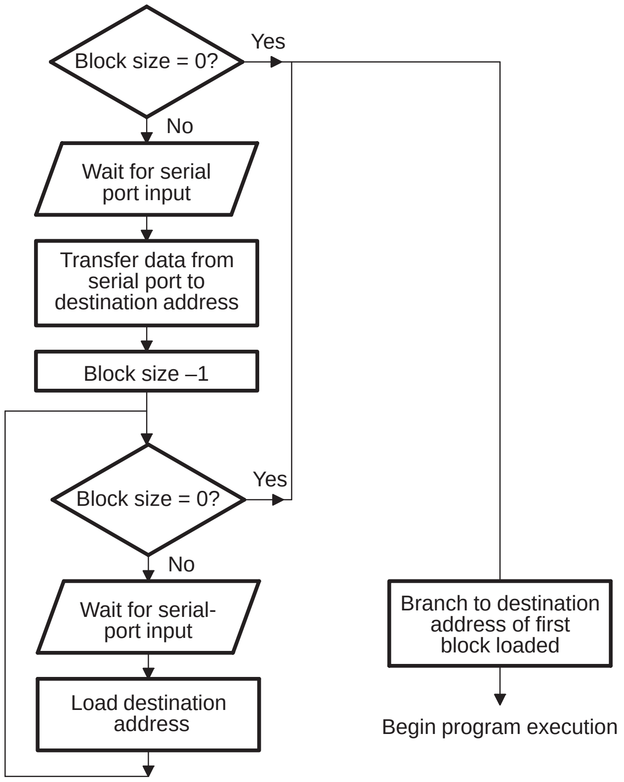

Memory load
Branch to address boot 1, boot 2, or boot 3
Determine mode 8, 16, or 32
Set memory configuration control word
Load block size
End of source program code (block size = 0)?
Yes
No
Load destination address
Block size = 0?
Yes
No
Transfer data from source to destination
Block size –1
Load next block size
Branch to destination address of first block loaded
Begin program execution

Figure 11–3. Boot-Loader Serial-Port Load-Mode Flowchart


Block size = 0?
Yes
No
Wait for serial port input
Transfer data from serial port to destination address
Block size –1
Block size = 0?
Yes
No
Wait for serial-port input
Load destination address
Branch to destination address of first block loaded
Begin program execution

11.1.4 TMS320C31 Boot Data Stream Structure

Table 11–2 shows the data stream structure. The data stream is composed of a header of 1 (serial-port load) or 2 (memory load) words and one or more blocks of source data. The boot loader uses this header to determine the physical memory width where the source program resides (memory load) and to configure the primary bus interface before source program boot load. The blocks of source data have two entries in addition to the raw data. The first entry in this block indicates the size of the block. The second entry in this block indicates the memory address where the boot loader copies this source block. Words 5 through *n* of the shaded entries in Table 11–2 contain the source data for the first block.

Table 11–2. Source Data Stream Structure

Word†	Content	Valid Data Entries
1	Memory width (8, 16, or 32 bits) where source program resides	8h, 10h, or 20h, respectively
2	Value to set the $\overline{\text{STRB}}$ control register	See subsection 10.7
3	Size of first data block. The block size is the number of 32-bit words in the data block. A 0 in this entry signifies the end of the source data stream	$0 \le \text{size} \le 2^{24}$
4	Destination address to load the first block	A valid ’C31 24-bit address
5	First word of first block	A ’C31 valid instruction or any 32-bit wide data value (LSB first)
. . .	. . .	. . .
n	Last word of first block	A ’C31 valid instruction or any 32-bit wide data value
. . .	. . .	. . .
m	Size of last data block. The block size is the number of 32-bit words in the data block. If the next word following this block is not 0, another block is loaded.	$0 \le \text{size} \le 2^{24}$
m + 1	Destination address to load the last block	A valid ’C31 24-bit address
m + 2	First word of last block	A ’C31 valid instruction or any 32-bit wide data value (LSB first)
. . .	. . .	. . .
j	Last word of last source block	
j + 1	Zero word. If more than one source block was read, word *j* would be the last word of the last block. Each block consists of header and data portions. The block’s header is shaded darker than the block’s data section.	0h

† Words 1 and 2 do not exist in serial-port boot load since the source program does not reside in memory.

Each source block of data can be loaded to different memory locations. Each block specifies its own size and destination address. The last source block of the data stream is appended with a zero word.

11.1.4.1 Examples of External TMS320C31 Memory Loads

Table 11–3, Table 11–4, and Table 11–5 show memory images for byte-wide, 16-bit-wide, and 32-bit-wide configured memory (see Figure 4–2 on page 4-6).

These examples assume the following:

- An $\overline{\text{INT0}}$ signal was detected after reset was deasserted (signifying an external memory load from boot 1).
- The source program header resides at memory location 0x1000 and defines the following:
  - Boot memory-type EPROMs that require two wait states and SWW = 11
  - A loader destination address at the beginning of the ’C31 internal RAM block 1
  - A single block of memory that is 0x1FF in length

Table 11–3. Byte-Wide Configured Memory

Address	Value	Comments
0x1000	0x08	Memory width = 8 bits
0x1001	0x00	
0x1002	0x00	
0x1003	0x00	
0x1004	0x58	Memory type = SWW = 11, WCNT = 2
0x1005	0x10	
0x1006	0x00	
0x1007	0x00	
0x1008	0xFF	Program block size in words = 0x1FF
0x1009	0x01	
0x100A	0x00	
0x100B	0x00	
0x100C	0x00	Program load starting address = 0x809C00
0x100D	0x9C	
0x100E	0x80	
0x100F	0x00	

Table 11–4. 16-Bit-Wide Configured Memory

Address	Value	Comments
0x1000	0x10	Memory width = 16
0x1001	0x0000	
0x1002	0x1058	Memory type = SWW = 11, WCNT = 2
0x1003	0x0000	
0x1004	0x1FF	Program block size in words = 0x1FF
0x1005	0x0000	
0x1006	0x9C00	Program load starting address = 0x809C00
0x1007	0x0080	

Table 11–5. 32-Bit-Wide Configured Memory

Address	Value	Comments
0x1000	0x00000020	Memory width = 32
0x1001	0x00001058	Memory type = SWW = 11, WCNT = 2
0x1002	0x000001FF	Program block size in words = 0x1FF
0x1003	0x00809C00	Program load starting address = 0x809C00

After reading the header, the loader transfers 0x IFF 32-bit words, beginning at a specified destination address 0x 809C00. Code blocks require the same byte and half-word ordering conventions. The loader can also load multiple code blocks at different address destinations.

After loading all code blocks, the boot loader branches to the destination address of the first block loaded and begins program execution. Consequently, the first code lock loaded is a start-up routine to access the other loaded programs.

CAUTION

It is assumed that at least one block of code is loaded when the loader is invoked. Initial loader invocation with a block size of 0x00000000 produces unpredictable results.

11.1.4.2 Serial-Port Loading

Boot loads, by way of the ’C31 serial port, are selected by driving the $\overline{\text{INT3}}$ pin active (low) following reset. The loader automatically configures the serial port for 32-bit fixed-burst-mode reads. It is interrupt-driven by the frame synchronization receive (FSR) signal. You cannot change this mode for boot loads. Your hardware must generate the serial-port clock and FSR externally.

As with memory loads, a header must precede the actual program to be loaded. However, you need only supply the block size and destination address because the loader and your hardware have predefined serial-port speed and data format (that is, skip data words 0 and 1).

The transferred data-bit order must begin with the MSB and end with the LSB.

11.1.5 Interrupt and Trap-Vector Mapping

Unlike the microprocessor mode, the microcomputer/boot-loader (MCBL) mode uses a dual-vectoring scheme to service interrupt and trap requests. Dual vectoring was implemented to ensure code compatibility with future versions of ’C3x devices.

In a dual-vectoring scheme, branch instructions to an address, rather than direct-interrupt vectoring, are used. The normal interrupt and trap vectors are defined to vector to the last 63 locations in the on-chip RAM, starting at address 809FC1h. When the loader is invoked, the interrupt vector table is remapped by the processor to the last 63 locations in RAM block 1 of the ’C31. These locations are assumed to contain branch instructions to the interrupt source routines.

CAUTION

Make sure that these locations are not inadvertently overwritten by loaded program or data values.

Table 11–6 shows the MCBL/$\overline{\text{MP}}$ mode interrupt and trap instruction memory maps.

Table 11–6. TMS320C31 Interrupt and Trap Memory Maps

Address	Description
809FC1	INT0
809FC2	INT1
809FC3	INT2
809FC4	INT3
809FC5	XINT0
809FC6	RINT0
809FC7	XINT1 (Reserved)
809FC8	RINT1 (Reserved)
809FC9	TINT0
809FCA	TINT1
809FCB	DINT0
809FCC–809FDF	Reserved
809FE0	TRAP0
809FE1	TRAP1
•	•
•	•
•	•
809FFB	TRAP27
809FFC–809FFF	Reserved

11.1.6 TMS320C31 Boot-Loader Precautions

The boot loader builds a one-word-deep stack, starting at location 809801h.

CAUTION

Avoid loading code at location 809801h.

The interrupt flags are not reset by the boot-loader function. If pending interrupts are to be avoided when interrupts are enabled, clear the IF register before enabling interrupts.

The MCBL/$\overline{\text{MP}}$ pin must remain high during the entire boot-loader execution, but it can be changed subsequently at any time. The ’C31 does not need to be reset after the MCBL/$\overline{\text{MP}}$ pin is changed. During the change, the ’C31 must not access addresses 0h–FFFh. The memory space 0h–FFFh will be mapped to external memory three clock cycles after changing the MCBL/$\overline{\text{MP}}$ pin.

11.2 TMS320C32 Boot Loader

This section describes how to use the ’C32 microcomputer/boot loader (MCBL/$\overline{\text{MP}}$) functions.

11.2.1 TMS320C32 Boot-Loader Description

The ’C32 boot loader is an enhanced version of that found in the ’C31. The boot loader can load and execute programs received from a host processor through standard memory devices (including EPROM), with and without handshake, or through the serial port. The ’C32 boot loader supports 16- and 32-bit program external memory widths, as well as 8-, 16-, and 32-bit data-type sizes and external memory widths.

The programs to be loaded reside in one of three memory-mapped areas identified as Boot 1, Boot 2, and Boot 3 (see shaded areas of Figure 4–3 on page 4-6) or they are received by means of the serial port.

The boot-loader code starts at location 0x45 in the on-chip ROM. The source code is supplied in Appendix C.

11.2.2 TMS320C32 Boot-Loader Mode Selection

The ’C32 boot loader functions as a memory boot loader, memory boot loader with handshake, or a serial-port boot loader. The boot-loader mode selection is determined by the status of the $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pins immediately following reset. Table 11–7 lists the boot-loader modes.

- ❑ The memory boot loader supports user-definable byte, half-word, and full-word data formats, allowing the flexibility to load a source program from memories having widths of 8, 16, and 32 bits with or without handshaking. The source programs to be loaded reside in one of the three memory locations: 1000h, 810000h, and 900000h (see Table 11–7).
- ❑ The memory boot-load handshaking mode uses XF0 as a data-acknowledge signal and XF1 as a data-ready signal.
- ❑ The serial-port boot loader supports 32-bit fixed-burst loads from the ’C32 serial port with an externally generated serial-port clock and frame sync signals. The format is similar to that of the memory boot loader, except that the source memory width is omitted.

Table 11–7. Boot-Loader Mode Selection

$\overline{\text{INT0}}$	$\overline{\text{INT1}}$	$\overline{\text{INT2}}$	$\overline{\text{INT3}}$	Boot Loader Mode	Source Program Location
0	1	1	1	External memory	Boot 1 address 1000h
1	0	1	1	External memory	Boot 2 address 81 0000h
1	1	0	1	External memory	Boot 3 address 90 0000h
1	1	1	0	32-bit fixed-burst serial	Serial Port
0	1	1	0	External memory with handshake	Boot 1 address 1000h, XF0 and XF1 used in handshaking
1	0	1	0	External memory with handshake	Boot 2 address 81 0000h, XF0 and XF1 used in handshaking
1	1	0	0	External memory with handshake	Boot 3 address 90 0000h, XF0 and XF1 used in handshaking

11.2.3 TMS320C32 Boot-Loading Sequence

The following is the sequence of events that occur during the boot load of a source program. Table 11–2 shows the structure of the source program.

1) Select the boot loader by resetting the ’C32 while driving the MCBL/$\overline{\text{MP}}$ pin high and the corresponding $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pins low. The MCBL/$\overline{\text{MP}}$ must stay high during boot loading, but can be changed anytime after boot loading has terminated. No reset is necessary when changing the $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pin, as long as the ’C32 is not accessing the overlapping memory (0h–FFFh) during this transition. In nonhandshake mode, one of the $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ pins can be driven low any time after deasserting the $\overline{\text{RESET}}$ pin (driven low and then high). While in handshake mode, two interrupt pins must be asserted before deasserting the $\overline{\text{RESET}}$ pin.

2) The status of the IF register’s $\overline{\text{INT3}}$–$\overline{\text{INT0}}$ bit fields dictates the boot-loading mode. The bits are polled in the order described in the flowchart in Figure 11–4.

3) If only the IF register’s $\overline{\text{INT3}}$ bit field is set, the boot loader configures the serial port for 32-bit fixed burst mode reads with an externally generated serial-port clock and FSR. Then, it proceeds to boot load the source program from the serial port. A header indicating the $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$ control registers precedes the actual program (see Table 11–2). These header values are loaded into the corresponding locations at the completion of the boot-load operation. The transferred data-bit order supplied to the serial port must begin with the most significant bit (MSB) and end with the least significant bit (LSB). Figure 11–5 depicts the boot-loader serial-port flow.

4) Otherwise, the boot loader attempts a memory boot load. Figure 11–6 shows the boot-loader memory flow. If the IF register's $\overline{\text{INT0}}$ bit field is set, the source program is loaded from memory location 1000h. If the IF register's $\overline{\text{INT1}}$ bit field is set, the source program is loaded from memory location 810000h. If the IF register's $\overline{\text{INT2}}$ bit field is set, the source program is loaded from memory location 900000h. After determining the memory location of the source program, the boot loader checks $\overline{\text{INT3}}$ bit field in the IF register. If this bit is set, all data transfers are performed with synchronous handshake. The handshake protocol uses XF0 as a data-acknowledge and XF1 as a data-ready signals. 'C32's XF0 is an output pin while the XF1 is an input pin. Figure 11–7 shows the handshake data-transfer operation.

   The data-transfer operation occurs as follows:

   a) The 'C32 boot loader waits until the host sets XF1 low to read in the data. While the 'C32 waits for XF1 to drop low, the $\overline{\text{IACK}}$ pin pulses until XF1 is low. Setting XF1 low communicates to the 'C32 that the data is valid. The $\overline{\text{IACK}}$ pulse indicates that the 'C32 is waiting for data.

   b) The boot loader sets XF0 low after reading the data value. Dropping XF0 acknowledges to the host that the data was read.

   c) The host sets XF1 high to inform the 'C32 that the data is no longer valid.

   d) The 'C32 terminates the transfer by setting XF0 high.

   The memory boot-load source program has a header indicating the boot memory width and the contents of the $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$ control registers (see Table 11–2).

5) After reading the header, the boot loader copies the source-program blocks. The source-program blocks have three entries preceding the source-program-block data. The first entry in the source-program block indicates the size of the block, the second entry indicates the address where the block is to be loaded, while the third entry contains the destination-memory strobe including a pointer that identifies the destination-memory strobe ($\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, or $\overline{\text{IOSTRB}}$) and a value that describes the strobe configuration for the memory width and data-type size. If the destination memory is internal, the third entry should contain a zero. The boot loader cannot load the source program to any memory address below 1000h, unless the address decode logic is remapped.

6) Once all the program blocks are loaded into their respective address locations with the given data-type sizes, the boot loader sets the $\overline{\text{IOSTRB}}$, $\overline{\text{STRB0}}$, and $\overline{\text{STRB1}}$ control registers to the values read at the beginning of the boot-load process.

7) The boot loader branches to the destination address of the first source block loaded and begins program execution.

Figure 11–4. TMS320C32 Boot-Loader Mode-Selection Flowchart

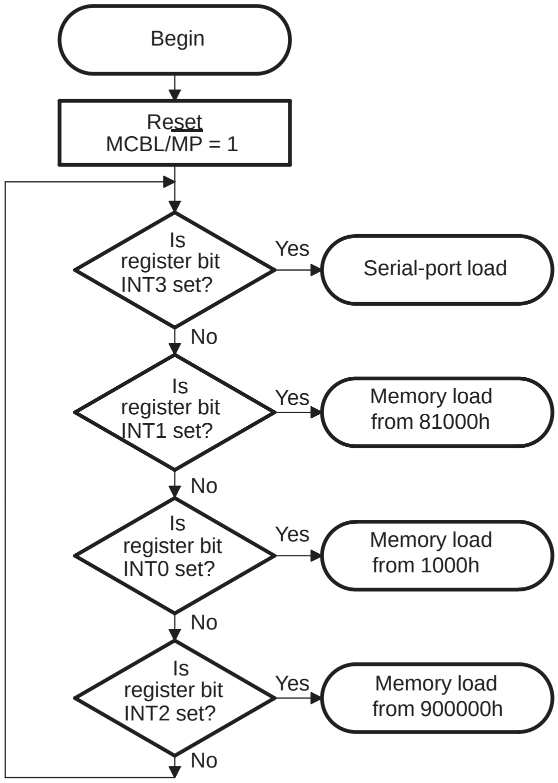


Figure 11–5. Boot-Loader Serial-Port Load Flowchart

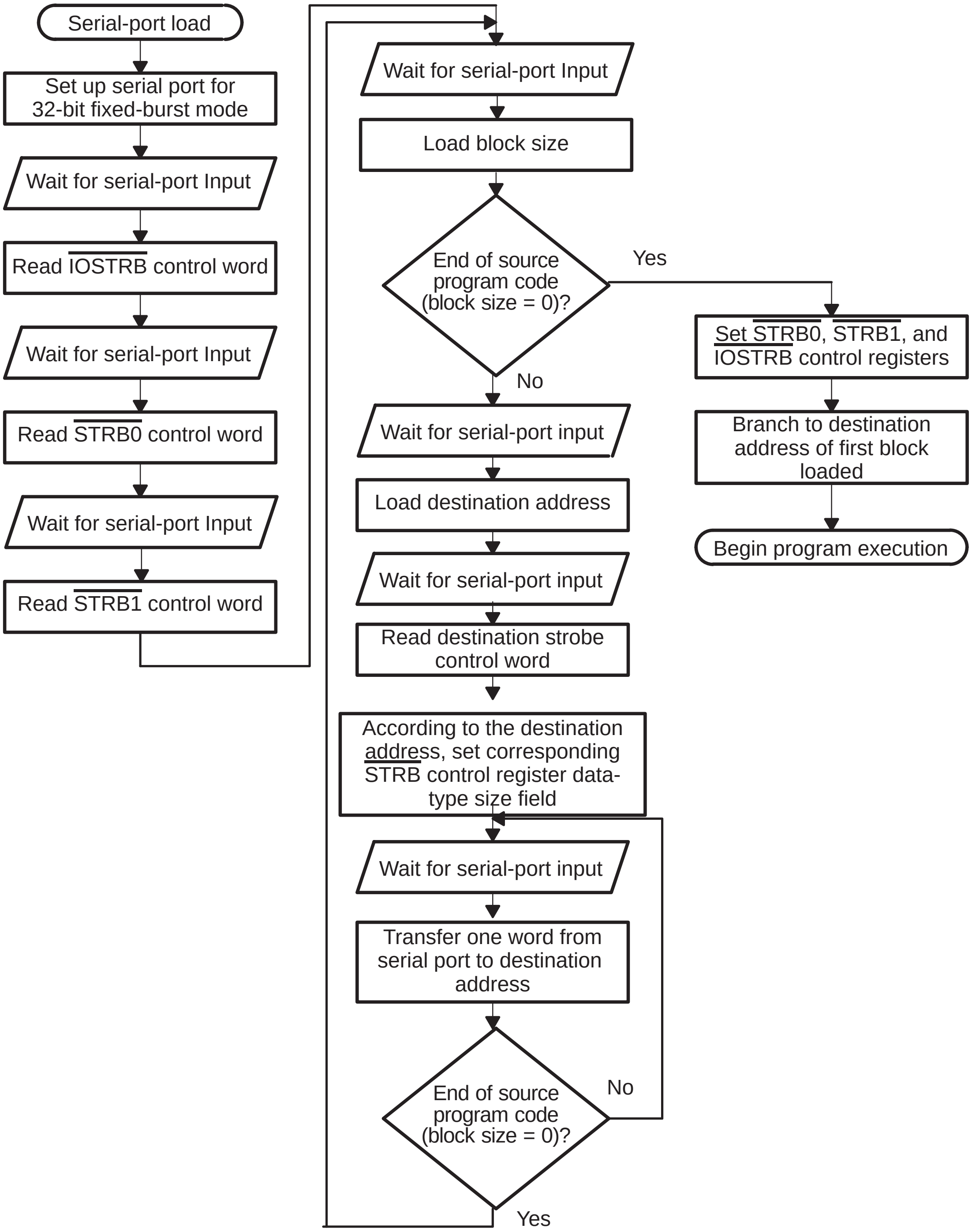

Serial-port load
Set up serial port for 32-bit fixed-burst mode
Wait for serial-port Input
Read IOSTRB control word
Wait for serial-port Input
Read STRB0 control word
Wait for serial-port Input
Read STRB1 control word
Wait for serial-port Input
Load block size
End of source program code (block size = 0)?
Yes
No
Set STRB0, STRB1, and IOSTRB control registers
Branch to destination address of first block loaded
Begin program execution
Wait for serial-port input
Load destination address
Wait for serial-port input
Read destination strobe control word
According to the destination address, set corresponding STRB control register data-type size field
Wait for serial-port input
Transfer one word from serial port to destination address
End of source program code (block size = 0)?
No
Yes

Figure 11–6. Boot-Loader Memory-Load Flowchart

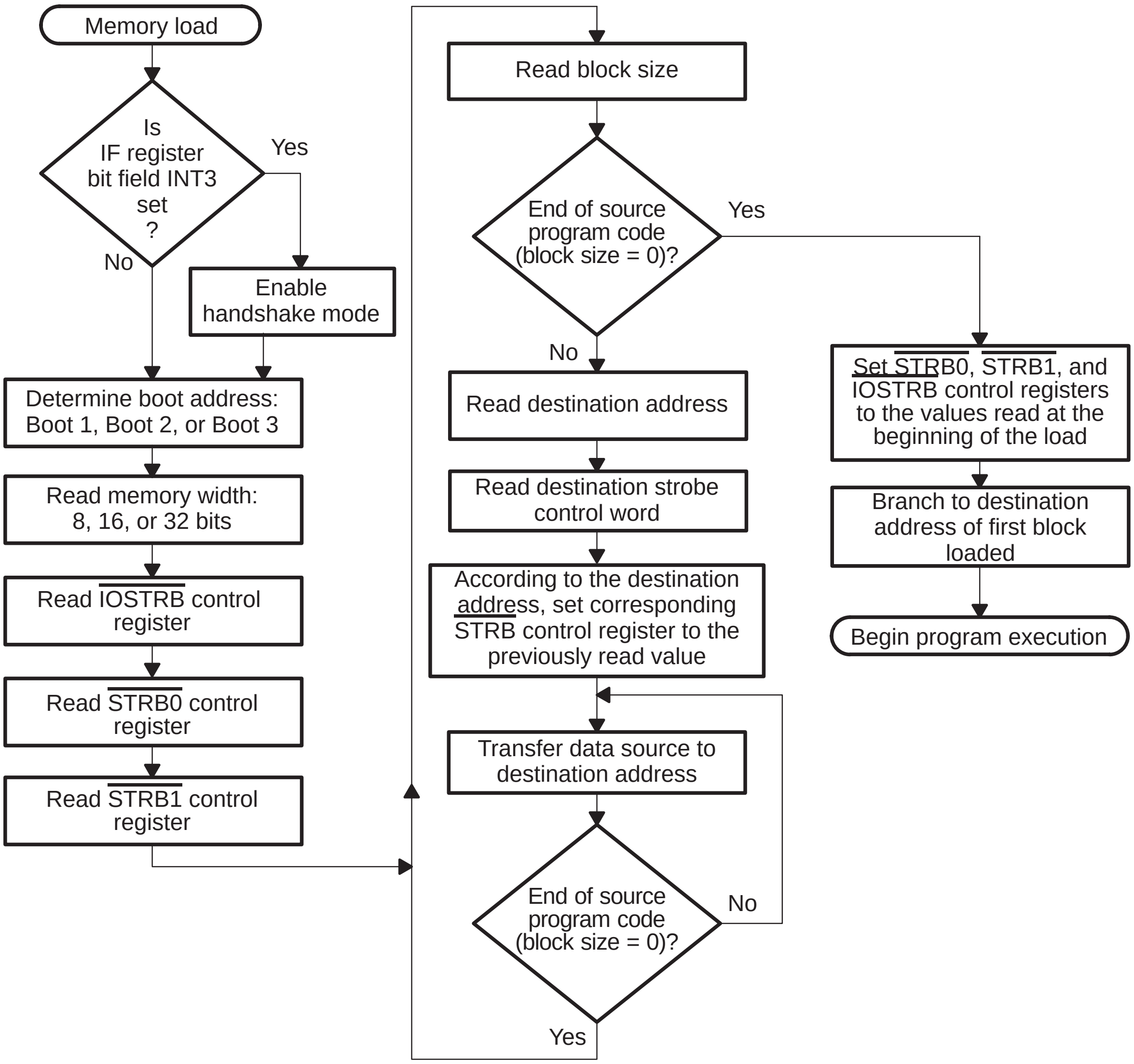
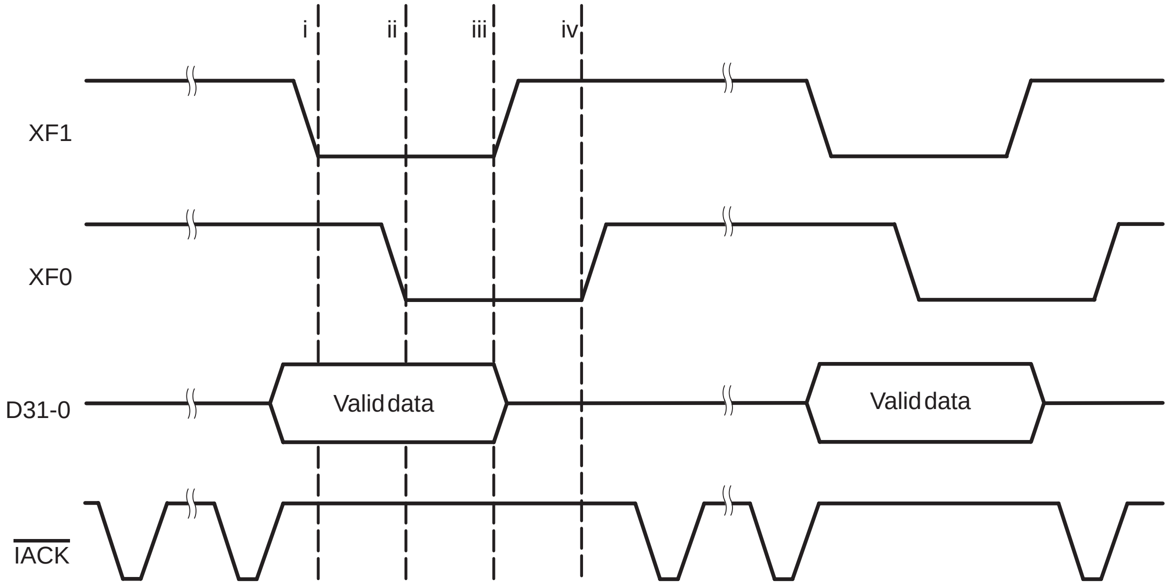

Memory load
Is IF register bit field INT3 set ?
Yes
No
Enable handshake mode
Determine boot address: Boot 1, Boot 2, or Boot 3
Read memory width: 8, 16, or 32 bits
Read IOSTRB control register
Read STRB0 control register
Read STRB1 control register
Read block size
End of source program code (block size = 0)?
Yes
No
Read destination address
Read destination strobe control word
According to the destination address, set corresponding STRB control register to the previously read value
Transfer data source to destination address
End of source program code (block size = 0)?
No
Yes
Set STRB0, STRB1, and IOSTRB control registers to the values read at the beginning of the load
Branch to destination address of first block loaded
Begin program execution

Figure 11–7. Handshake Data-Transfer Operation



11.2.4 TMS320C32 Boot Data Stream Structure

Table 11–8 shows the data stream structure. The data stream is composed of a header of three (serial-port load) or four (memory load) words and one or more blocks of source data. The boot loader uses this header to determine the physical memory width where the source program resides (memory load) and to configure the STRBs after completion of source program boot load. The blocks of source data have three entries in addition to the raw data. The first entry in this block indicates the size of the block. The second entry in this block indicates the memory address where the boot loader copies this source block. The third entry contains the destination memory strobe configuration including memory width and data-type size. This allows the boot loader to copy and store 8-, 16-, or 32-bit data values into 8-, 16-, or 32-bit wide memory. Words 8 through *n* of the shaded entries in Table 11–8 contain the source data for the first block.

Table 11–8. Source Data Stream Structure

Word[†]	Content	Valid Data Entries
1	Memory width (8, 16, or 32 bits) where source program resides	8h, 10h, or 20h, respectively
2	Value to set the $\overline{\text{IOSTRB}}$ control register at end of boot loader process	See Section 10.7 on page 10-26
3	Value to set the $\overline{\text{STRB0}}$ control register at end of boot loader process	See Section 10.3.1 on page 10-7
4	Value to set the $\overline{\text{STRB1}}$ control register at end of boot loader process	See Section 10.6 on page 10-20
5	Size of the first data block. The block size is the number of words in the data block (word length is specified by the data-type size). A 0 in this entry signifies the end of the source data stream.	$0 \le \text{size} \le 2^{24}$
6	Destination address to load the first block	A valid ’C32 24-bit address
7	First block destination memory width and data-type size in the format given in the Valid Data Entries column.	SSSSSS6*x*h[‡]
8	First word of first block	A ’C32 valid instruction or any 8-, 16-, or 32-bit wide data value
. . .	. . .	. . .
n	Last word of first block	A ’C32 valid instruction or any 8-, 16-, or 32-bit wide data value
. . .	. . .	. . .
m	Size of the last data block. The block size is the number of words in the data block (word length is specified by the data-type size). If the next word following this block is not 0, another block is loaded.	$0 \le \text{size} \le 2^{24}$
m + 1	Destination address to load the last block	A valid ’C32 24-bit address

[†] Word 1 does not exist in serial-port boot load since the source program does not reside in memory.

[‡] The *SSSSSS* hexadecimal digits refer to the lower 24 bits of the strobe control register. The *x* hexadecimal digit identifies the strobe as follows: 0 for $\overline{\text{IOSTRB}}$, 4 for $\overline{\text{STRB0}}$, and 8 for $\overline{\text{STRB1}}$. SSSSSS6xh is cleared to 0 when loading the entire field into internal memory.

Table 11–8. Source Data Stream Structure (Continued)

Word[†]	Content	Valid Data Entries
m + 2	Last block destination memory width and data-type size in the format given in the Valid Data Entries column.	*SSSSSS6x*h[‡]
m + 3	First word of last block.	A ’C32 valid instruction or any 8-, 16-, or 32-bit wide data value
.	.	.
.	.	.
.	.	.
j	Last word of last source block	
j + 1	Zero word. Note that if more than one source block was read, word *j* shown above would be the last word of the last block. Each block consists of header and data portions. The block’s header is shaded darker than the block’s data section.	0h

[†] Word 1 does not exist in serial-port boot load since the source program does not reside in memory.
[‡] The *SSSSSS* hexadecimal digits refer to the lower 24 bits of the strobe control register. The *x* hexadecimal digit identifies the strobe as follows: 0 for $\overline{\text{IOSTRB}}$, 4 for $\overline{\text{STRB0}}$, and 8 for $\overline{\text{STRB1}}$. SSSSSS6xh is cleared to 0 when loading the entire field into internal memory.

Each source block can be loaded into a different memory location. Each block specifies its own size and destination address. The last source block of the data stream is appended with a zero word. Because the ’C32’s $\overline{\text{STRB}}$s can be configured to support different external memory widths and data-type sizes, each source block specifies its data-type size. The external memory width is set when the boot loader reads the $\overline{\text{STRBx}}$ control register values in the source data stream header.

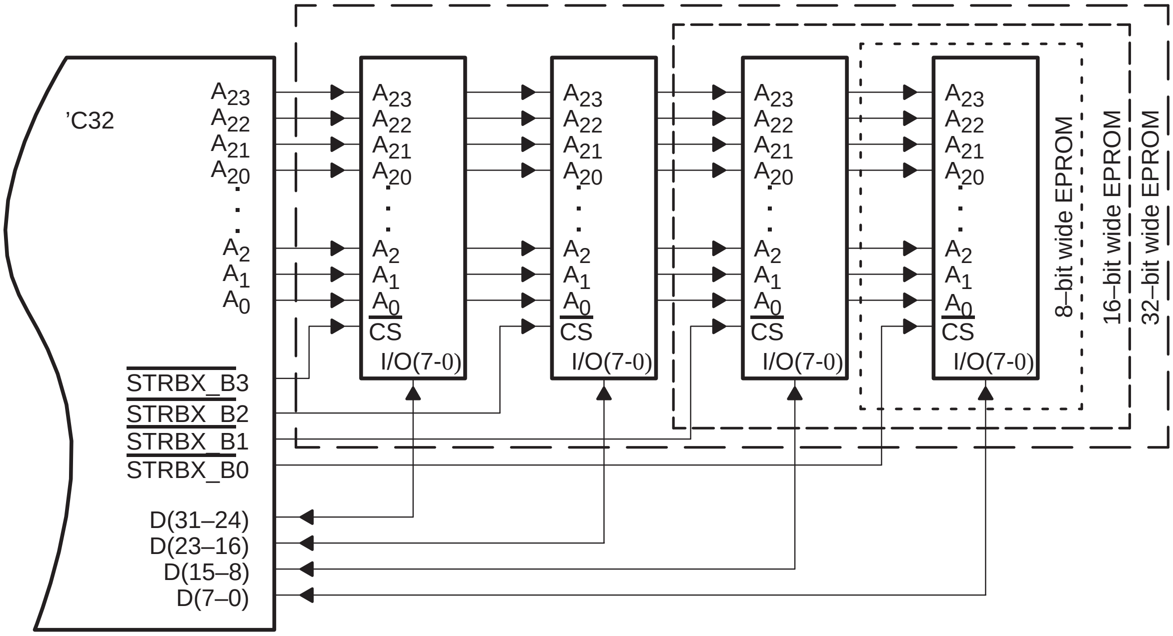
To build a ’C32 boot data stream with the HEX30 utility provided with the TMS320 floating-point code-generation tools, use the following steps:

- ❑ Compile/assemble code with **–v32** switch using v4.7 or later of the TMS320 floating-point C compiler/assembler. If the code-generation tools are invoked with **CL30** and **–z** switch, include **–v32** switch in the linker command file.
- ❑ Link as usual.
- ❑ Run Hex30 utility version 4.7 or later. The **–v32** switch used in the compiler/assembler will create a header in the COFF file, identifying it as a ’C32 for the Hex30.

11.2.5 Boot-Loader Hardware Interface

The hardware interface for the memory boot load uses the $\overline{\text{STRBX\_B3}}$ through $\overline{\text{STRBX\_B0}}$ pins as strobe byte-enable pins (see Figure 11–8). The hardware interface is **independent of the boot source memory width**. This interface is identical to the 32-bit-wide memory interface described in Case 2, in Section 10.6 on page 10-20. For 16-bit memory widths, remove the two left-most memory devices of Figure 11–8. For 8-bit memory widths, remove all but the right-most of the memory devices of Figure 11–8.

Figure 11–8. External Memory Interface for Source Data Stream Memory Boot Load



11.2.6 TMS320C32 Boot-Loader Precautions

The interrupt flags are not reset by the boot-loader function. If pending interrupts are to be avoided when interrupts are enabled, clear the IF register before enabling interrupts.

The MCBL/$\overline{\text{MP}}$ pin should remain high during the entire boot-loading execution, but it can be changed subsequently at any time. The ’C32 does not need to be reset after the MCBL/$\overline{\text{MP}}$ pin is changed. During the change, the ’C32 should not access addresses Oh–FFh. The memory space Oh–FFFh is mapped to external memory three clock cycles after changing the MCBL/$\overline{\text{MP}}$ pin.

The ’C32 boot loader uses the following peripheral memory-mapped registers as a temporary stack:

- Timer0 counter register (808024h)
- Timer0 period register (808028h)
- DMA0 source address register (808004h)
- DMA0 destination address register (808006h)
- DMA0 transfer counter register (808008h)

These memory-mapped registers are not reset by the boot-loading process. Before using these peripherals, reprogram these registers with the appropriate values.

Chapter 12

Peripherals

The ’C3x features two timers, a serial port (two serial ports for the ’C30), and an on-chip direct memory access (DMA) controller (2-channel DMA controller on the ’C32). These peripheral modules are controlled through memory-mapped registers located on the dedicated peripheral bus.

The DMA controller performs input/output operations without interfering with the operation of the CPU, making it possible to interface the ’C3x to slow, external memories and peripherals, analog-to-digital converters (A/Ds), serial ports, and so forth, without reducing the computational throughput of the CPU. The result is improved system performance and decreased system cost.

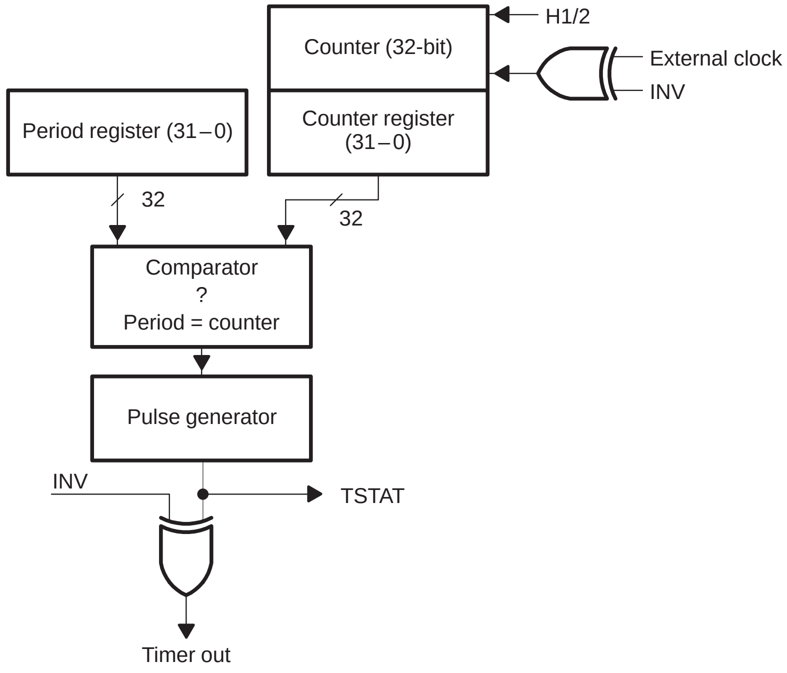
Topic **Page**

12.1 Timers

The ’C3x has two 32-bit general-purpose timer modules. Each timer has two signaling modes and internal or external clocking. You can use the timer modules to signal to the ’C3x or the external world at specified intervals or to count external events. With an internal clock, the timer can signal an external A/D converter to start a conversion, or it can interrupt the ’C3x DMA controller to begin a data transfer. The timer interrupt is one of the internal interrupts. With an external clock, the timer can count external events and interrupt the CPU after a specified number of events. Each timer has an I/O pin that you can use as an input clock to the timer, as an output clock signal, or as a general-purpose I/O pin.

Each timer consists of a 32-bit counter, a comparator, an input clock selector, a pulse generator, and supporting hardware (see Figure 12–1). A timer counts the cycles of a timer input clock with the counter register. When that counter register equals the value stored in the timer-period register, it resets the counter to 0 and produces a transition in the timer output signal. The timer input clock can be driven by either half the internal clock frequency of the ’C3x or an external clock on TCLKx pin.

Figure 12–1. Timer Block Diagram



12.1.1 Timer Pins

Each timer has one pin associated with the timer clock signal (TCLK) pin. This pin (TCK) is used as a general-purpose I/0 signal, as a timer output, or as an input for an external clock for a timer. Each timer has a TCLK pin: TCLK0 is connected to timer0, TCLK1 to timer1.

12.1.2 Timer Control Registers

Three memory-mapped registers are used by each timer:

❑ **Global-control register**

The global-control register determines the operating mode of the timer, monitors the timer status, and controls the function of the I/O pin of the timer.

❑ **Period register**

The period register specifies the timer’s signaling frequency.

❑ **Counter register**

The counter register contains the current value of the incrementing counter. You can increment the timer on the rising edge or the falling edge of the input clock. The counter is zeroed and can cause an internal interrupt whenever its value equals that in the period register. The pulse generator generates either of two types of external clock signals: pulse or clock. The memory map for the timer modules is shown in Figure 12–2.

Figure 12–2. Memory-Mapped Timer Locations

808020h	Timer0 global control†
808024h	Timer0 counter‡
808028h	Timer0 period‡
808030h	Timer1 global control†
808034h	Timer1 counter‡
808038h	Timer1 period‡

†See Section 12.1.3

‡See Section 12.1.4

12.1.3 Timer Global-Control Register

The timer global-control register is a 32-bit register that contains the global and port control bits for the timer module. Figure 12–3 shows the format of the timer global-control register. Bits 3–0 are the port control bits; bits 11–6 are the timer global-control bits. At reset, all bits are set to 0 except for DATIN (which is set to the value read on TCLK). Table 12–1 describes the timer global-control register bits, their names, and functions.

Figure 12–3. Timer Global-Control Register

31 16	15 12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	TSTAT	INV	CLKSRC	C/$\overline{P}$	$\overline{HLD}$	GO	xx	xx	DATIN	DATOUT	$\overline{I}$/O	FUNC
		R	R/W	R/W	R/W	R/W	R/W			R	R/W	R/W	R/W

Notes: 1) R = read, W = write

2) xx = reserved bit, read as 0

Table 12–1. Timer Global-Control Register Bits Summary

Abbreviation	Reset Value	Name	Description
FUNC	0	Function	Controls the function of TCLK. If FUNC = 0, TCLK is configured as a general-purpose digital I/O port. If FUNC = 1, TCLK is configured as a timer pin. See section 12.1.6 *Timer Operation Modes* on page 12-10 for a description of the relationship between FUNC and CLKSRC.
$\overline{\text{I}}$/O	0	Input/output	If FUNC = 0 and CLKSRC = 0, TCLK is configured as a general-purpose I/O pin. If $\overline{\text{I}}$/O = 0, TCLK is configured as a general-purpose input pin. If $\overline{\text{I}}$/O = 1, TCLK is configured as a general-purpose output pin.
DATOUT	0	Data output	Drives TCLK when the ’C3x is in I/O port mode. You can use DATOUT as an input to the timer.
DATIN	x[†]	Data input	Data input on TCLK or DATOUT. A write has no effect.
GO	0	Go	Resets and starts the timer counter. When GO = 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The GO bit is cleared on the same rising edge. GO = 0 has no effect on the timer.
$\overline{\text{HLD}}$	0	Counter hold signal	When this bit is 0, the counter is disabled and held in its current state. If the timer is driving TCLK, the state of TCLK is also held. The internal divide-by-2 counter is also held so that the counter can continue where it left off when $\overline{\text{HLD}}$ is set to 1. You can read and modify the timer registers while the timer is being held. $\overline{\text{RESET}}$ has priority over $\overline{\text{HLD}}$. The effect of writing to GO and HOLD is shown below.

GO	**$\overline{\text{HLD}}$**	**Result**
0	0	All timer operations are held. No reset is performed (reset value).
0	1	Timer proceeds from state before write.
1	0	All timer operations are held, including zeroing of the counter. The GO bit is not cleared until the timer is taken out of hold.
1	1	Timer resets and starts.

[†] x = 0 or 1 (set to value read on TCLK)

Table 12–1. Timer Global-Control Register Bits Summary (Continued)

	Reset		
Abbreviation	**Value**	**Name**	**Description**
C/$\overline{P}$	0	Clock/pulse mode control	When $C/\overline{P} = 1$, clock mode is chosen, and the signaling of the TSTAT flag and external output has a 50% duty cycle. When $C/\overline{P} = 0$, the status flag and external output will be active for one H1 cycle during each timer period (see Figure 12–4 on page 12-8).
CLKSRC	0	Clock source	This bit specifies the source of the timer clock. When CLKSRC = 1, an internal clock with a frequency equal to one-half of the H1 frequency is used to increment the counter. The INV bit has no effect on the internal clock source. When CLKSRC = 0, you can use an external signal from the TCLK pin to increment the counter. The external clock is synchronized internally, thus allowing external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency. This is less than f(H1)/2. See section 12.1.6, *Timer Operation Modes*, on page 12-10 for a description of the relationship between FUNC and CLKSRC.
INV	0	Inverter control bit	If an external clock source is used and INV = 1, the external clock is inverted as it goes into the counter. If the output of the pulse generator is routed to TCLK and INV = 1, the output is inverted before it goes to TCLK (see Figure 12–1 on page 12-2). If INV = 0, no inversion is performed on the input or output of the timer. The INV bit has no effect, regardless of its value, when TCLK is used in I/O port mode.
TSTAT	0	Timer status bit	This bit indicates the status of the timer. It tracks the output of the uninverted TCLK pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.

† x = 0 or 1 (set to value read on TCLK)

12.1.4 Timer-Period and Counter Registers

The 32-bit timer-period register is used to specify the frequency of the timer signaling. The timer-counter register is a 32-bit register, which is reset to 0 whenever it increments to the value of the period register. Both registers are set to 0 at reset.

Certain boundary conditions affect timer operation. These conditions are listed below:

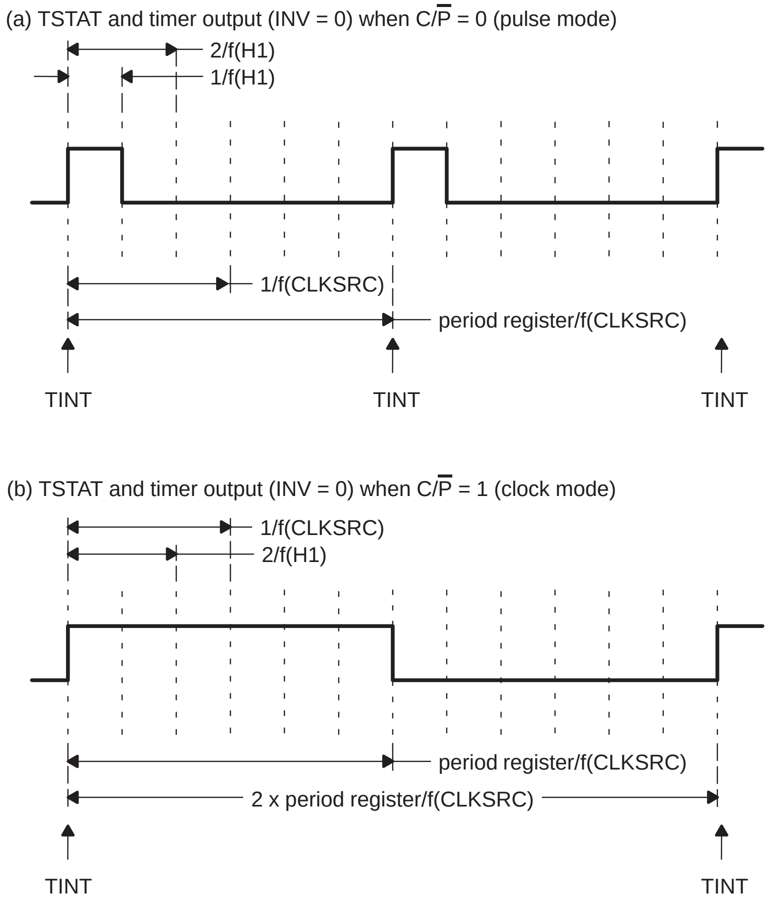
- When the period and counter registers are 0, the operation of the timer is dependent upon the $C/\overline{P}$ mode selected. In pulse mode ($C/\overline{P}$ = 0), TSTAT is set and remains set. In clock mode ($C/\overline{P}$ = 1), the width of the cycle is 2/f(H1), and the external clocks are ignored.
- When the counter register is not 0 and the period register = 0, the counter counts, rolls over to 0, and behaves as described above.
- When the counter register is set to a value greater than the period register, the counter may overflow when incremented. Once the counter reaches its maximum 32-bit value (0FFFFFFFFh), it rolls over to 0 and continues.

Writes from the peripheral bus override register updates from the counter and new status updates to the control register.

12.1.5 Timer Pulse Generation

The timer pulse generator (see Figure 12–1 on page 12-2) can generate several external signals. You can invert these signals with the INV bit. The two basic modes are pulse mode and clock mode, as shown in Figure 12–4. In both modes, an internal clock source f (timer clock) has a frequency of f(H1)/2, and an externally generated clock source f (timer clock) can have a maximum frequency of f(H1)/2.6. In pulse mode ($C/\overline{P}$ = 0), the width of the pulse is 1/f(H1).

Figure 12–4. Timer Timing


(a) TSTAT and timer output (INV = 0) when C/$\overline{P}$ = 0 (pulse mode)
2/f(H1)
1/f(H1)
1/f(CLKSRC)
period register/f(CLKSRC)
TINT
TINT
TINT
(b) TSTAT and timer output (INV = 0) when C/$\overline{P}$ = 1 (clock mode)
1/f(CLKSRC)
2/f(H1)
period register/f(CLKSRC)
2 x period register/f(CLKSRC)
TINT
TINT

The timer signaling is determined by the frequency of the timer input clock and the period register. The following equations are valid with either an internal or an external timer clock:

f(pulse mode) = f(timer clock) / period register

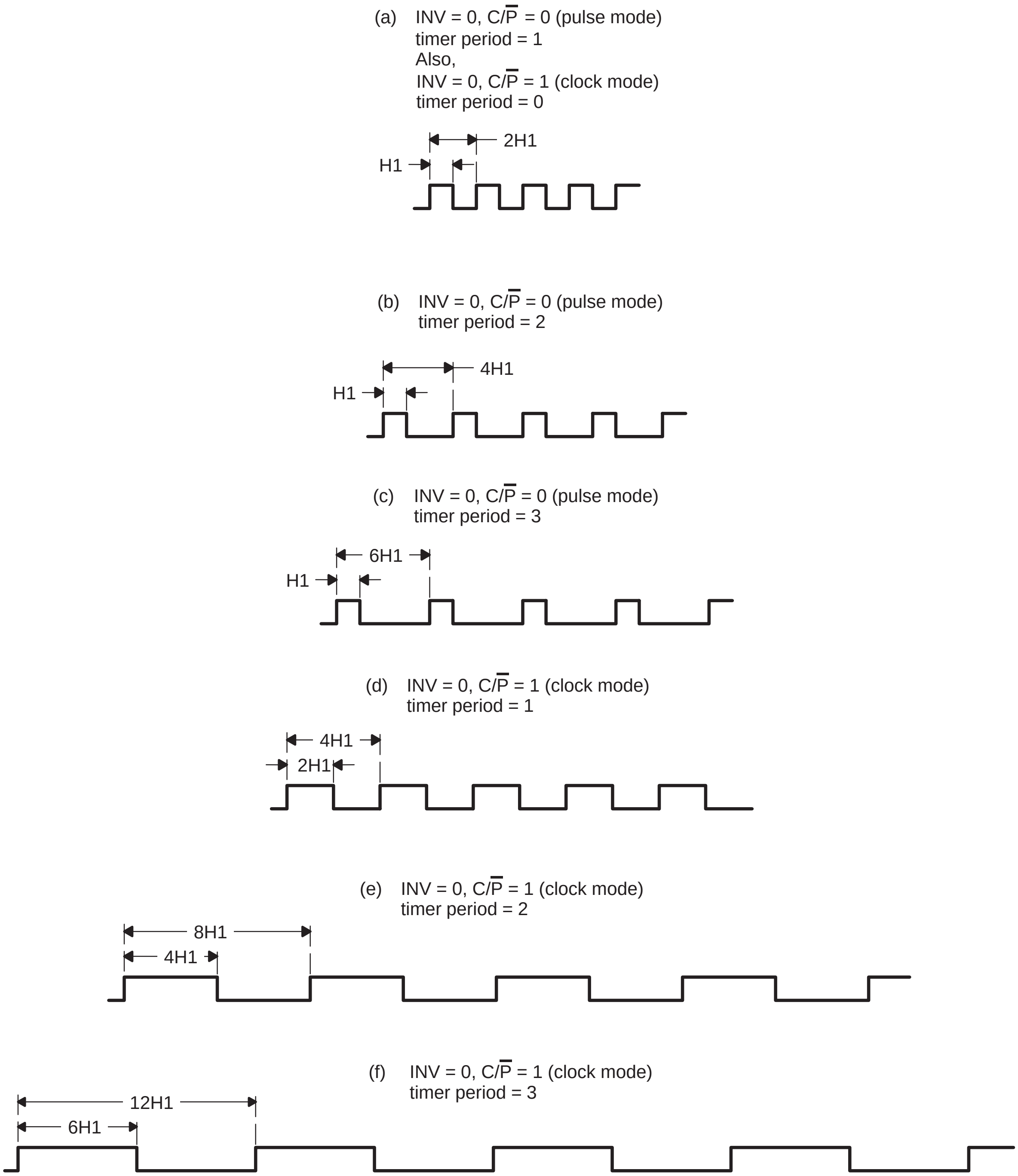
*f(clock mode) = f(timer clock) / (*2 x *period register)*

Note: Period register

If the period register equals 0, see Section 12.1.4.

Example 12–1 provides some examples of the TCLKx output when the period register is set to various values and clock or pulse mode is selected.

Example 12–1. Timer Output Generation Examples



(a) INV = 0, $C/\overline{P}$ = 0 (pulse mode)
timer period = 1
Also,
INV = 0, $C/\overline{P}$ = 1 (clock mode)
timer period = 0

(b) INV = 0, $C/\overline{P}$ = 0 (pulse mode)
timer period = 2

(c) INV = 0, $C/\overline{P}$ = 0 (pulse mode)
timer period = 3

(d) INV = 0, $C/\overline{P}$ = 1 (clock mode)
timer period = 1

(e) INV = 0, $C/\overline{P}$ = 1 (clock mode)
timer period = 2

(f) INV = 0, $C/\overline{P}$ = 1 (clock mode)
timer period = 3

12.1.6 Timer Operation Modes

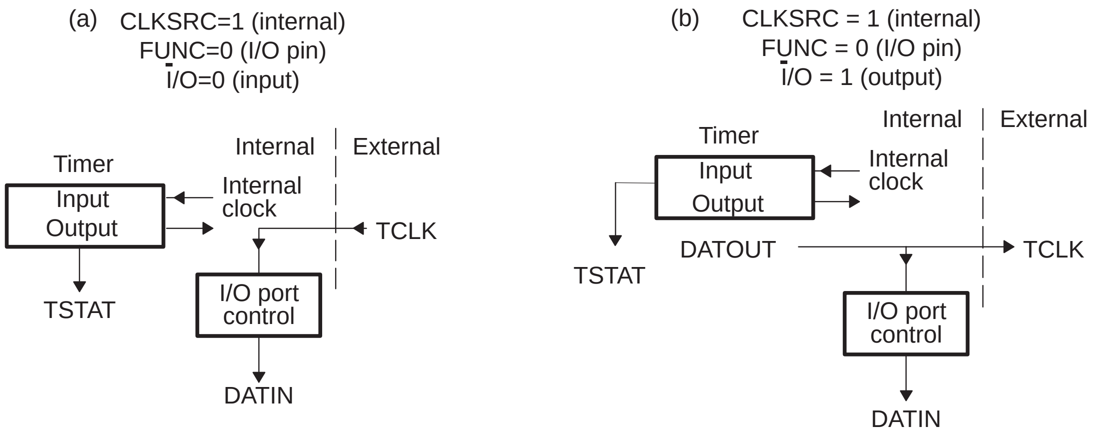
The timer can receive its input and send its output in several different modes, depending upon the setting of CLKSRC, FUNC, and Ī/O. The four timer modes of operation are defined in the following sections.

12.1.6.1 CLKSRC = 1 and FUNC = 0

If CLKSRC = 1 and FUNC = 0, the timer input comes from the internal clock. The internal clock is not affected by the INV bit in the global-control register. In this mode, TCLK is connected to the I/O port control, and you use TCLK as a general-purpose I/O pin (see Figure 12–5).

- If Ī/O = 0, TCLK is configured as a general-purpose input pin whose state you can read in DATIN. DATOUT has no effect on TCLK or DATIN. See Figure 12–5 (a).
- If Ī/O = 1, TCLK is configured as a general-purpose output pin. DATOUT is placed on TCLK and can be read in DATIN. See Figure 12–5 (b).

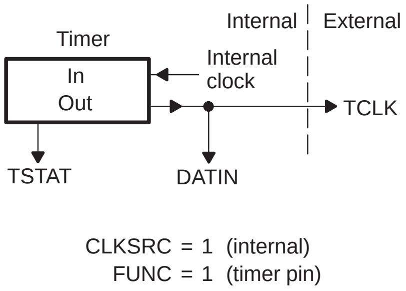
Figure 12–5. Timer Configuration with CLKSRC = 1 and FUNC = 0



12.1.6.2 CLKSRC = 1 and FUNC = 1

If CLKSRC = 1 and FUNC = 1 (see Figure 12–6), the timer input comes from the internal clock, and the timer output goes to TCLK. This value can be inverted using INV, and you can read in DATIN the value output on TCLK.

Figure 12–6. Timer Configuration with CLKSRC = 1 and FUNC = 1

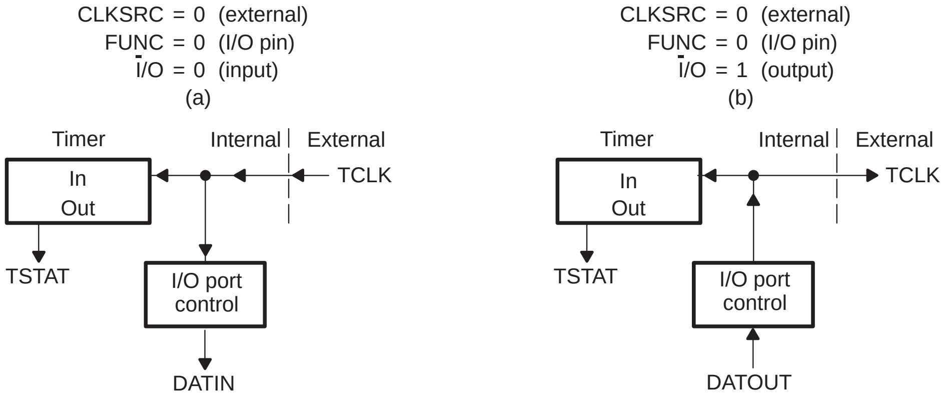


12.1.6.3 CLKSRC = 0 and FUNC = 0

If CLKSRC = 0 and FUNC = 0 (see Figure 12–7), the timer is driven according to the status of the $\overline{\text{I}}$/O bit.

- If $\overline{\text{I}}$/O = 0, the timer input comes from TCLK. This value can be inverted using INV, and you can read in DATIN the value of TCLK. See Figure 12–7 (a).
- If $\overline{\text{I}}$/O = 1, TCLK is an output pin. Then, TCLK and the timer are both driven by DATOUT. All 0-to-1 transitions of DATOUT increment the counter. INV has no effect on DATOUT. You can read in DATIN the value of DATOUT. See Figure 12–7 (b).

Figure 12–7. Timer Configuration with CLKSRC = 0 and FUNC = 0

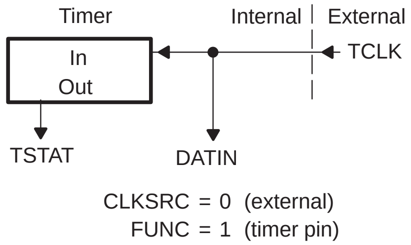


12.1.6.4 CLKSRC = 0 and FUNC = 1

If CLKSRC = 0 and FUNC = 1 (see Figure 12–8), TCLK drives the timer.

- ❑ If INV = 0, all 0-to-1 transitions of TCLK increment the counter.
- ❑ If INV = 1, all 1-to-0 transitions of TCLK increment the counter. You can read in DATIN the value of TCLK.

Figure 12–8. Timer Configuration with CLKSRC = 0 and FUNC = 1



12.1.7 Using TCLKx as General-Purpose I/O Pins

When FUNC = 0, TCLKx can be used as an I/O pin. Figure 12–9 and Figure 12–10 show how the TCLKx is connected when it is configured as a general-purpose I/O pin. In Figure 12–9, the I/O bit equals 0 and TCLK is configured as an input pin whose value can be read in the DATIN bit. In Figure 12–10, the I/O bit equals 1 and TCLK is configured as an output pin that outputs the value you wrote in the DATOUT bit.

Figure 12–9. TCLK as an Input (I/O = 0)

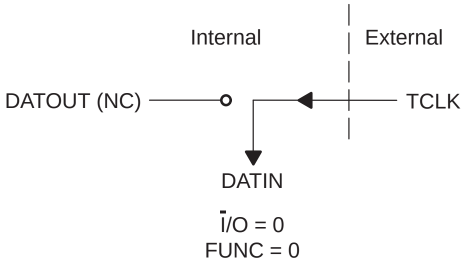
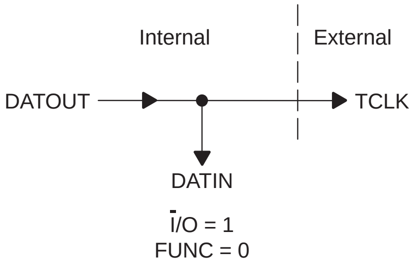


Figure 12–10. TCLK as an Output (I/O = 1)



12.1.8 Timer Interrupts

A timer interrupt is generated whenever the TSTAT bit of the timer control register changes from a 0 to a 1. The frequency of timer interrupts depends on whether the timer is set up in pulse mode or clock mode.

- ❑ In pulse mode, the interrupt frequency is determined by the following equation:

  $$f_{(interrupt)} = \frac{f_{(timer\ clock)}}{period\ register}$$

  where:

  $f_{(interrupt)}$ = *interrupt frequency*

  $f_{(timer\ clock)}$ = *timer frequency*

- ❑ In clock mode, the interrupt frequency is determined by the following equation:

  $$f_{(interrupt)} = \frac{f_{(timer\ clock)}}{2\ x\ period\ register}$$

  where:

  $f_{(interrupt)}$ = *interrupt frequency*

  $f_{(timer\ clock)}$ = *timer frequency*

The timer counter is automatically reset to 0 whenever it is equal to the value in the timer-period register. You can use the timer interrupt for either the CPU or the DMA. Interrupt-enable control for each timer, for either the CPU or the DMA, is found in the CPU/DMA interrupt-enable register. Refer to Section 3.1.8, *CPU/DMA Interrupt-Enable Register (IE)*, on page 3-9 for more information.

When a timer interrupt occurs, a change in the state of the corresponding TCLK pin is observed if FUNC = 1 and CLKSRC = 1 in the timer global-control register. The exact change in the state depends on the state of the C/$\overline{\text{P}}$ bit. In pulse mode (C/$\overline{\text{P}}$ = 0), the width of the pulse change is 1/f (H1). In clock mode (C/$\overline{\text{P}}$ = 1), the width of the pulse change is the period register divided by the frequency of the timer input clock.

12.1.9 Timer Initialization/Reconfiguration

The timers are controlled through memory-mapped registers located on the dedicated peripheral bus. The general procedure for initializing and/or reconfiguring the timers follows:

1) Halt the timer by clearing the GO/$\overline{\text{HLD}}$ bits of the timer global-control register. To do this, write a 0 to the timer global-control register. Note that the timers are halted on $\overline{\text{RESET}}$.

2) Configure the timer through the timer global-control register (with GO = $\overline{\text{HLD}}$ = 0), the timer-counter register, and timer-period register, if necessary.

3) Start the timer by setting the GO/$\overline{\text{HLD}}$ bits of the timer global-control register.

Example 12–2 shows how to set up the ‘C3x timer to generate the maximum clock frequency through the TCLKx pin.

Example 12–2. Maximum Frequency Timer Clock Setup

```
*   Maximum Frequency Timer Clock Setup
*
                 .data

Timer0           .word     808020h             ; Timer global control address
TCTRL_RST        .word     301h
TCTRL_GD         .word     3C1h
TCNT             .word     0                   ; Timer counter value
TPRD             .word     0                   ; Timer-period value
                 .text
                    .
                    .
                    .
                 LDP       Timer0
                 LD1       @Timer0,AR0         ; Load data page pointer
                 LD1       0,R0
                 ST1       R0,*AR0             ; Halt timer
                 LD1       @TCTRL_RST,R0       ; Configure timer
                 ST1       R0,*AR0
                 LD1       @TCNT,R0
                 ST1       R0,*+AR0(4)         ; Load timer counter
                 LD1       @TPRD,R0
                 ST1       R0,**+AR0(8)        ; Load timer period
                 LD1       @TCTRL_G0,R0
                 ST1       R0,*AR0             ; Start timer
```

12.2 Serial Ports

The ’C30 has two totally independent bidirectional serial ports. Both serial ports are identical, and there is a complementary set of control registers in each one. Only one serial port is available on the ’C31 and the ’C32. You can configure each serial port to transfer 8, 16, 24, or 32 bits of data per word simultaneously in both directions. The clock for each serial port can originate either internally, through the serial port timer and period registers, or externally, through a supplied clock. An internally generated clock is a divide down of the clockout frequency, f(H1). A continuous transfer mode is available, which allows the serial port to transmit and receive any number of words without new synchronization pulses.

Eight memory-mapped registers are provided for each serial port:

- Global-control register
- Two control registers for the six serial I/O pins
- Three receive/transmit timer registers
- Data-transmit register
- Data-receive register

The global-control register controls the global functions of the serial port and determines the serial-port operating mode. Two port control registers control the functions of the six serial port pins. The transmit buffer contains the next complete word to be transmitted. The receive buffer contains the last complete word received. Three additional registers are associated with the transmit/ receive sections of the serial-port timer. A serial-port block diagram is shown in Figure 12–11 on page 12-16, and the memory map of the serial ports is shown in Figure 12–12 on page 12-17.

Figure 12–11. Serial Port Block Diagram

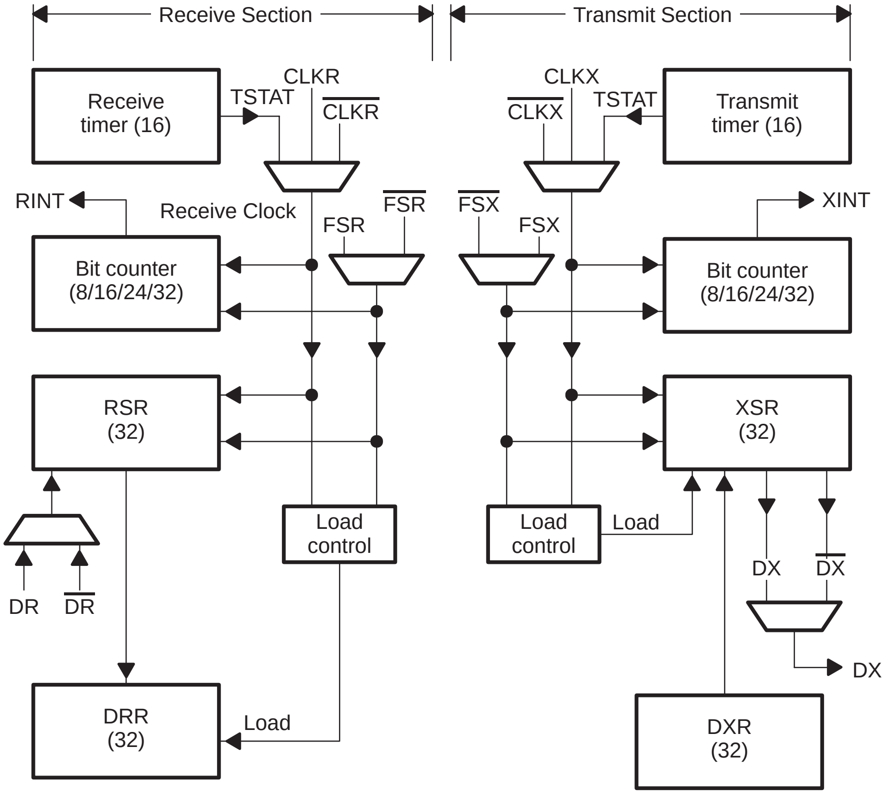


Figure 12–12. Memory-Mapped Locations for the Serial Ports

808040h	Serial-port 0 global control[†]
808042h	Serial port 0 FSX/DX/CLKX control[‡]
808043h	Serial port 0 FSR/DR/CLKR control[§]
808044h	Serial port 0 R/X timer control[¶]
808045h	Serial port 0 R/X timer counter[#]
808046h	Serial port 0 R/X timer period[‖]
808048h	Serial port 0 data transmit[☆]
80804Ch	Serial port 0 data receive[□]
808050h	Serial-port 1 global control[†]
808052h	Serial port 1 FSX/DX/CLKX control[‡]
808053h	Serial port 1 FSR/DR/CLKR control[§]
808054h	Serial port 1 R/X timer control[¶]
808055h	Serial port 1 R/X timer counter[#]
808056h	Serial port 1 R/X timer period[‖]
808058h	Serial port 1 data transmit[☆]
80805Ch	Serial port 1 data receive[□]

Note: Serial port1 locations are reserved on the ’C31 and ’C32.

[†] See Figure 12–13.
[‡] See Figure 12–14.
[§] See Figure 12–15.
[¶] See Figure 12–16.
[#] See Figure 12–17.
[‖] See Figure 12–18.
[☆] See Figure 12–19.
[□] See Figure 12–20.

12.2.1 Serial-Port Global-Control Register

The serial-port global-control register is a 32-bit register that contains the global-control bits for the serial port. The register is shown in Figure 12–13. Table 12–2 shows the register bits, bit names, and bit functions.

Figure 12–13. Serial-Port Global-Control Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	RRESET	XRESET	RINT	RTINT	XINT	XTINT	RLEN		XLEN		FSRP	FSXP
				R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DRP	DXP	CLKRP	CLKXP	RFSM	XFSM	RVAREN	XVAREN	RCLK SRCE	XCLK SRCE	HS	RSR FULL	XSR EMPTY	FSXOUT	XRDY	RRDY
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R/W	R	R

Notes: 1) R = read, W = write

2) xx = reserved bit, read as 0

Table 12–2. Serial-Port Global-Control Register Bits Summary

Abbreviation	Reset Value	Name	Description
RRDY	0	Receive ready flag	If RRDY = 1, the receive buffer has new data and is ready to be read. A three H1/H3 cycle delay occurs from the loading of DRR to RRDY = 1. The rising edge of this signal sets RINT. If RRDY = 0, the receive buffer does not have new data since the last read. RRDY = 0 at reset and after the receive buffer is read.
XRDY	1	Transmit ready flag	If XRDY = 1, the transmit buffer has written the last bit of data to the shifter and is ready for a new word. A three H1/H3 cycle delay occurs from the loading of the transmit shifter until XRDY is set to 1. The rising edge of this signal sets XINT. If XRDY = 0, the transmit buffer has not written the last bit of data to the transmit shifter and is not ready for a new word.
FSXOUT		Transmit frame sync configuration	FSXOUT = 0 configures the FSX pin as an input. FSXOUT = 1 configures the FSX pin as an output.
XSREMPTY	0	Transmit-shift register empty flag	If XSREMPTY = 0, the transmit-shift register is empty. If XSREMPTY = 1, the transmit-shift register is not empty. Reset or XRESET causes this bit to = 0.
RSRFULL	0	Receive-shift register full flag	If RSRFULL = 1, an overrun of the receiver has occurred. In continuous mode, RSRFULL is set to 1 when both RSR and DRR are full. In noncontinuous mode, RSRFULL is set to 1 when RSR and DRR are full and a new FSR is received. A read causes this bit to be set to 0. This bit can be set to 0 only by a system reset, a serial-port receive reset (RRESET = 1), or a read. When the receiver tries to set RSRFULL to 1 at the same time that the global register is read, the receiver dominates, and RSRFULL is set to 1. If RSRFULL = 0, no overrun of the receiver has occurred.

Table 12–2. Serial-Port Global-Control Register Bits Summary (Continued)

Abbreviation	Reset Value	Name	Description
HS	0	Handshake	If HS = 1, the handshake mode is enabled. If HS = 0, the handshake mode is disabled.
XCLK SRCE	0	Transmit clock source	If XCLK SRCE = 1, the internal transmit clock is used. If XCLK SRCE = 0, the external transmit clock is used.
RCLK SRCE	0	Receive clock source	If RCLK SRCE = 1, the internal receive clock is used. If RCLK SRCE = 0, the external receive clock is used.
XVAREN	0	Transmit data rate mode	Specifies a fixed or variable data rate mode when transmitting. With a fixed data rate, FSX is active for at least one XCLK cycle and then goes inactive before transmission begins. With variable data rate, FSX is active while all bits are being transmitted. When you use an external FSX and variable data rate signaling, the DX pin is driven by the transmitter when FSX is held active or when a word is being shifted out.
RVAREN	0	Receive data rate mode	Specifies a fixed or variable data rate mode when receiving. If RVAREN = 0 (fixed data rate), FSX is active for at least one RCLK cycle and then goes inactive before reception begins. If RVAREN = 1 (controlled data rate), FSX is active while all bits are being received.
XFSM	0	Transmit frame sync mode	Configures the port for continuous mode operation or standard mode operation. If XFSM = 1 (continuous mode), only the first word of a block generates a sync pulse, and the rest are transmitted continuously to the end of the block. If XFSM = 0 (standard mode), each word has an associated sync pulse.
RFSM	0	Receive frame sync mode	Configures the port for continuous mode operation or standard mode operation. If RFSM = 1 (continuous mode), only the first word of a block generates a sync pulse, and the rest are received continuously to the end of the block. If RFSM = 0 (standard mode), each word received has an associated sync pulse.
CLKXP		CLKX polarity	If CLKXP = 0, CLKX is active high. If CLKXP = 1, CLKX is active low.

Table 12–2. Serial-Port Global-Control Register Bits Summary (Continued)

Abbreviation	Reset Value	Name	Description
CLKRP	0	CLKR polarity	If CLKRP = 0, CLKR is active (high). If CLKRP = 1, CLKR is active (low).
DXP	0	DX polarity	If DXP = 0, DX is active (high). If DXP = 1, DX is active (low).
DRP	0	DR polarity	If DRP = 0, DR is active (high). If DRP = 1, DR is active (low).
FSXP	0	FSX polarity	If FSXP = 0, FSX is active (high). If FSXP = 1, FSX is active (low).
FSRP	0	FSR polarity	If FSRP = 0, FSR is active (high). If FSRP = 1, FSR is active (low).
XLEN	00	Transmit word length	These two bits define the word length of serial data transmitted. All data is assumed to be right justified in the transmit buffer when fewer than 32 bits are specified. 0 0 — 8 bits 1 0 — 24 bits 0 1 — 16 bits 1 1 — 32 bits
RLEN	00	Receive word length	These two bits define the word length of serial data received. All data is right justified in the receive buffer. 0 0 — 8 bits 1 0 — 24 bits 0 1 — 16 bits 1 1 — 32 bits
XTINT	0	Transmit timer interrupt enable	If XTINT = 0, the transmit timer interrupt is disabled. If XTINT = 1, the transmit timer interrupt is enabled.
XINT	0	Transmit interrupt enable	If XINT = 0, the transmit interrupt is disabled. If XINT = 1, the transmit interrupt is enabled. **Note:** The CPU receive flag XINT and the serial-port-to-DMA interrupt (EXINT0 in the IE register) is the OR of the enabled transmit timer interrupt and the enabled transmit interrupt.
RTINT	0	Receive timer interrupt enable	If RTINT = 0, the receive timer interrupt is disabled. If RTINT = 1, the receive timer interrupt is enabled.

Table 12–2. Serial-Port Global-Control Register Bits Summary (Continued)

	Reset		
Abbreviation	**Value**	**Name**	**Description**
RINT	0	Receive interrupt enable	If RINT = 0, the receive interrupt is disabled. If RINT = 1, the receive interrupt is enabled. **Note:** The CPU receive interrupt flag RINT and the serial-port-to-DMA interrupt (ERINT0 in the IE register) are the OR of the enabled receive timer interrupt and the enabled receive interrupt.
XRESET	0	Transmit reset	If XRESET = 0, the transmit side of the serial port is reset. To take the transmit side of the serial port out of reset, set XRESET to 1. Do not set XRESET to 1 until at least three cycles after $\overline{\text{RESET}}$ goes inactive. This applies only to system reset. Setting XRESET to 0 does not change the contents of any of the serial-port control registers. It places the transmitter in a state corresponding to the beginning of a frame of data. Resetting the transmitter generates a transmit interrupt. Reset this bit during the time the mode of the transmitter is set. You can toggle XFSM without resetting the global-control register.
RRESET	0	Receive reset	If RRESET = 0, the receive side of the serial port is reset. To take the receive side of the serial port out of reset, set RRESET to 1. Do not set RRESET to 1 until at least three cycles after RESET goes inactive. This applies only to system reset. Setting RRESET to 0 does not change the contents of any of the serial-port control registers. It places the receiver in a state corresponding to the beginning of a frame of data. Reset this bit at the same time that the mode of the receiver is set. You can toggle without resetting the global-control register.

12.2.2 FSX/DX/CLKX Port-Control Register

This 32-bit port-control register controls the function of the serial port FSX, DX, and CLKX pins. The register is shown in Figure 12–14. Table 12–3 shows the register bits, bit names, and bit functions.

Figure 12–14. FSX/DX/CLKX Port-Control Register

31–16	15–12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	FSX DATIN	FSX DATOUT	FSX $\overline{\mathrm{I}}$/O	FSX FUNC	DX DATIN	DX DATOUT	DX $\overline{\mathrm{I}}$/O	DX FUNC	CLKX DATIN	CLKX DATOUT	CLKX $\overline{\mathrm{I}}$/O	CLKX FUNC
		R	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R/W	R/W	R/W

Notes: 1) R = read, W = write.

2) xx = reserved bit, read as 0.

Table 12–3. FSX/DX/CLKX Port-Control Register Bits Summary

Abbreviation	Reset Value	Name	Description
CLKX FUNC	0	Clock transmit function	Controls the function of CLKX. If CLKX FUNC = 0, CLKX is configured as a general-purpose digital I/O port. If CLKX FUNC = 1, CLKX is configured as a serial port pin.
CLKX $\overline{\mathrm{I}}$/O	0	Clock transmit input/output mode	If CLKX $\overline{\mathrm{I}}$/O = 0, CLKX is configured as a general-purpose input pin. If CLKX $\overline{\mathrm{I}}$/O = 1, CLKX is configured as a general-purpose output pin.
CLKX DATOUT	0	Clock transmit data ouput	Data output on CLKX when configured as general-purpose output.
CLKX DATIN	x[†]	Clock transmit data input	Data input on CLKX when configured as general-purpose input. A write has no effect.
DX FUNC	0	DX function	DXFUNC controls the function of DX. If DXFUNC = 0, DX is configured as a general-purpose digital I/O port. If DXFUNC = 1, DX is configured as a serial port pin.
DX $\overline{\mathrm{I}}$/O	0	DX input/output mode	If DX $\overline{\mathrm{I}}$/O = 0, DX is configured as a general-purpose input pin. If DX $\overline{\mathrm{I}}$/O = 1, DX is configured as a general-purpose output pin.
DX DATOUT	0	DX data output	Data output on DX when configured as general-purpose output.
DX DATIN	x[†]	DX data input	Data input on DX when configured as general-purpose input. A write has no effect.

[†] x = 0 or 1

Table 12–3. FSX/DX/CLKX Port-Control Register Bits Summary (Continued)

Abbreviation	Reset Value	Name	Description
FSX FUNC	0	FSX function	Controls the function of FSX. If FSX FUNC = 0, FSX is configured as a general-purpose digital I/O port. If FSX FUNC = 1, FSX is configured as a serial port pin.
FSX $\overline{\text{I}}$/O	0	FSX input/output mode	If FSX $\overline{\text{I}}$/O = 0, FSX is configured as a general-purpose input pin. If FSX $\overline{\text{I}}$/O = 1, FSX is configured as a general-purpose output pin.
FSX DATOUT	0	FSX data output	Data output on FSX when configured as general-purpose output.
FSX DATIN	x[†]	FSX data input	Data input on FSX when configured as general-purpose input. A write has no effect.

[†] x = 0 or 1

12.2.3 FSR/DR/CLKR Port-Control Register

This 32-bit port-control register is controlled by the function of the FSR, DR, and CLKR pins. At reset, all bits are set to 0. The register is shown in Figure 12–15. Table 12–4 shows the register bits, bit names, and bit functions.

Figure 12–15. FSR/DR/CLKR Port-Control Register

31 16	15 12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	FSR DATIN	FSR DATOUT	FSR I/O	FSR FUNC	DR DATIN	DR DATOUT	DR I/O	DR FUNC	CLKR DATIN	CLKR DATOUT	CLKR I/O	CLKR FUNC
		R	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R/W	R/W	R/W

Notes: 1) R = read, W = write

2) xx = reserved bit, read as 0

Table 12–4. FSR/DR/CLKR Port-Control Register Bits Summary

Abbreviation	Reset Value	Name	Description
CLKR FUNC	0	Clock receive function	Controls the function of CLKR. If CLKR FUNC = 0, CLKR is configured as a general-purpose digital I/O port. If CLKR FUNC = 1, CLKR is configured as a serial port pin.
CLKR $\overline{\text{I}}$/O	0	Clock receive input/output mode	If CLKR $\overline{\text{I}}$/O = 0, CLKR is configured as a general-purpose input pin. If CLKR $\overline{\text{I}}$/O = 1, CLKR is configured as a general-purpose output pin.
CLKR DATOUT	0	Clock receive data output	Data output on CLKR when configured as general-purpose output.
CLKR DATIN	x†	Clock receive data input	Data input on CLKR when configured as general-purpose input. A write has no effect.
DR FUNC	0	DR function	Controls the function of DR. If DR FUNC = 0, DR is configured as a general-purpose digital I/O port. If DR FUNC = 1, DR is configured as a serial port pin.
DR $\overline{\text{I}}$/O	0	DR input/output mode	If DR $\overline{\text{I}}$/O = 0, DR is configured as a general-purpose input pin. If DR $\overline{\text{I}}$/O = 1, DR is configured as a general-purpose output pin.
DR DATOUT	0	DR data output	Data output on DR when configured as general-purpose output.
DR DATIN	x†	DR data input	Data input on DR when configured as general-purpose input. A write has no effect.
FSR FUNC	0	FSR function	FSR FUNC controls the function of FSR. If FSR FUNC = 0, FSR is configured as a general-purpose digital I/O port. If FSR FUNC = 1, FSR is configured as a serial port pin.
FSR $\overline{\text{I}}$/O	0	FSR input/output mode	If FSR $\overline{\text{I}}$/O = 0, FSR is configured as a general-purpose input pin. If FSR $\overline{\text{I}}$/O = 1, FSR is configured as a general-purpose output pin.
FSR DATOUT	0	FSR data output	Data output on FSR when configured as general-purpose output.
FSR DATIN	x†	FSR data input	Data input on FSR when configured as general-purpose input. A write has no effect.

† x = 0 or 1.

12.2.4 Receive/Transmit Timer-Control Register

A 32-bit receive/transmit timer-control register contains the control bits for the timer module. At reset, all bits are set to 0. Figure 12–16 shows the register. Bits 5–0 control the transmitter timer. Bits 11–6 control the receiver timer. The serial port receive/transmit timer function is similar to timer module operation. It can be considered a 16-bit-wide timer. Table 12–5 describes the register bits, bit names, and bit functions.

Figure 12–16. Receive/Transmit Timer-Control Register

31 16	15 12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	RSTAT	xx	RCLKSRC	RC/$\overline{\text{P}}$	$\overline{\text{RHLD}}$	RGO	XSTAT	xx	XCLKSRC	XC/$\overline{\text{P}}$	$\overline{\text{XHLD}}$	XGO
		R		R/W	R/W	R/W	R/W	R		R/W	R/W	R/W	R/W

Notes: 1) R = read, W = write

2) xx = reserved bit, read as 0

Table 12–5. Receive/Transmit Timer-Control Register Register Bits Summary

Abbreviation	Reset Value	Name	Function
XGO	0	Transmit timer counter restart	Resets and restarts the transmit timer counter. If XGO = 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The XGO bit is cleared on the same rising edge. Writing 0 to XGO has no effect on the transmit timer.
$\overline{\text{XHLD}}$	0	Transmit counter hold signal	If $\overline{\text{XHLD}}$ = 0, the counter is disabled and held in its current state. If $\overline{\text{XHLD}}$ = 1, the internal divide-by-two counter is also held so that the counter continues where it left off.
XC/$\overline{\text{P}}$	0	Transmit clock/pulse mode control	When XC/$\overline{\text{P}}$ = 1, the clock mode is chosen. The signaling of the status flag and external output has a 50 percent duty cycle. When XC/$\overline{\text{P}}$ = 0, the status flag and external output are active for one CLKOUT cycle during each timer period.

Table 12–5. Receive/Transmit Timer-Control Register Register Bits Summary (Continued)

Abbreviation	Reset Value	Name	Function
XCLKSRC	0	Transmit clock source	Specifies the source of the transmit timer clock. When XCLKSRC = 1, an internal clock with frequency equal to one-half the CLKOUT frequency is used to increment the counter. When XCLKSRC = 0, you can use an external signal from the CLKX pin to increment the counter. The external clock source is synchronized internally, thus allowing for external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency, that is, less than f(H1)/2.6.
XTSTAT	0	Transmit timer status	Indicates the status of the transmit timer. It tracks what would be the output of the uninverted CLKX pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.
RGO	0	Receive timer counter restart	Resets and starts the receive timer counter. When RGO is set to 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The RGO bit is cleared on the same rising edge. Writing 0 to RGO has no effect on the receive timer.
$\overline{\text{RHLD}}$	0	Receive counter hold signal	If $\overline{\text{RHLD}}$ = 0, the counter is disabled and held in its current state. If $\overline{\text{RHLD}}$ = 1, the internal divide-by-2 counter is also held so that the counter will continue where it left off. You can read and modify the timer registers while the timer is being held. $\overline{\text{RESET}}$ has priority over $\overline{\text{RHLD}}$.
RC/$\overline{\text{P}}$	0	Rclock/pulse mode control	When RC/$\overline{\text{P}}$ = 1, the clock mode is chosen. The signaling of the status flag and external output has a 50% duty cycle. When RC/$\overline{\text{P}}$ = 0, the status flag and external output are active for one CLKOUT cycle during each timer period.

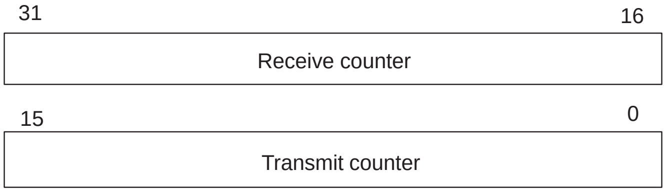
Table 12–5. Receive/Transmit Timer-Control Register Register Bits Summary (Continued)

Abbreviation	Reset Value	Name	Function
RCLKSRC	0	Receive timer clock source	Specifies the source of the receive timer clock. When RCLKSRC = 1, an internal clock with frequency equal to one-half the CLKOUT frequency is used to increment the counter. When RCLKSRC = 0, you can use an external signal from the CLKR pin to increment the counter. The external clock source is synchronized internally, allowing for external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency (that is, less than f(H1)/2.6).
RTSTAT	0	Receive timer status	Indicates the status of the receive timer. It tracks what would be the output of the uninverted CLKR pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.

12.2.5 Receive/Transmit Timer-Counter Register

The receive/transmit timer-counter register is a 32-bit register (see Figure 12–17). Bits 15–0 are the transmit timer-counter, and bits 31–16 are the receive timer-counter. Each counter is cleared to 0 whenever it increments to the value of the period register (see Section 12.2.6). *It is also set to 0 at reset.*

Figure 12–17. Receive/Transmit Timer-Counter Register

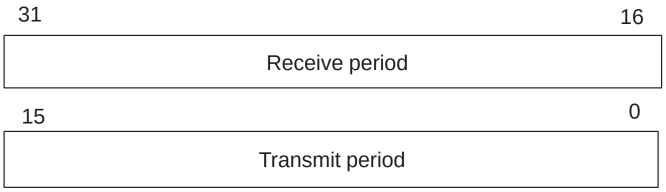


Note: All bits are read/write.

12.2.6 Receive/Transmit Timer-Period Register

The receive/transmit timer-period register is a 32-bit register (see Figure 12–18). Bits 15–0 are the timer transmit period, and bits 31–16 are the receive period. Each register specifies the period of the timer and is *cleared to 0 at reset*.

Figure 12–18. Receive/Transmit Timer-Period Register

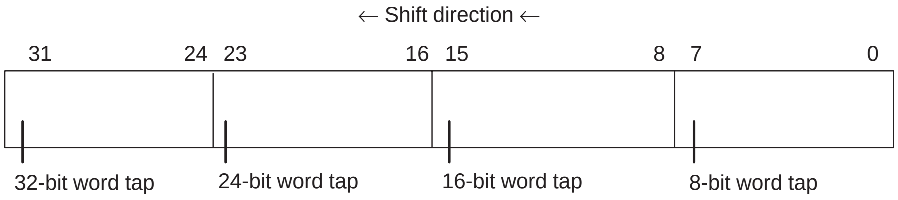


Note: All bits are read/write.

12.2.7 Data-Transmit Register

When the data-transmit register (DXR) is loaded, the transmitter loads the word into the transmit-shift register (XSR), and the bits are shifted out. The delay from a write to DXR until an FSX occurs (or can be accepted) is two CLKX cycles. The word is not loaded into the shift register until the shifter is empty. When DXR is loaded into XSR, the XRDY bit is set, specifying that the buffer is available to receive the next word. Four tap points within the transmit-shift register are used to transmit the word. These tap points correspond to the four-data word sizes and are illustrated in Figure 12–19. The shift is a left-shift (LSB to MSB) with the data shifted out of the MSB corresponding to the appropriate tap point.

Figure 12–19. Transmit Buffer Shift Operation

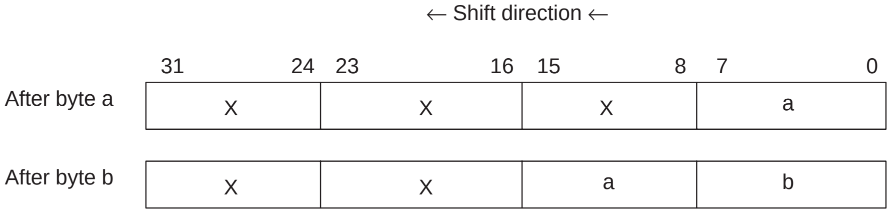


12.2.8 Data-Receive Register

When serial data is input, the receiver shifts the bits into the receive-shift register (RSR). When the specified number of bits are shifted in, the data-receive register (DRR) is loaded from RSR, and the RRDY status bit is set. The receiver is double-buffered. If the DRR has not been read and the RSR is full, the receiver is frozen. New data coming into the DR pin is ignored. The receive shifter does not write over the DRR. The DRR must be read to allow new data in the RSR to be transferred to the DRR. When a write to DRR occurs at the same time that an RSR-to-DRR transfer takes place, the RSR-to-DRR transfer has priority.

Data is shifted to the left (LSB to MSB). Figure 12–20 illustrates what happens when words less than 32 bits are shifted into the serial port. In this figure, it is assumed that an 8-bit word is being received and that the upper three bytes of the receive buffer are originally undefined. In the first portion of the figure, byte *a* has been shifted in. When byte *b* is shifted in, byte *a* is shifted to the left. When the data-receive register is read, both bytes *a* and *b* are read.

Figure 12–20. Receive Buffer Shift Operation



← Shift direction ←

	31 24	23 16	15 8	7 0
After byte a	X	X	X	a
After byte b	X	X	a	b

12.2.9 Serial-Port Operation Configurations

Several configurations are provided for the operation of the serial-port clocks and timer. The clocks for each serial port can originate either internally or externally. Figure 12–21 shows serial-port clocking in the I/O mode (CLKXFUNC = 0) when CLKX is either an input or an output. Figure 12–22 shows clocking in the serial-port mode (CLKXFUNC=1). Both figures use a transmit section for an example. The same relationship holds for a receive section.

Figure 12–21. Serial-Port Clocking in I/O Mode

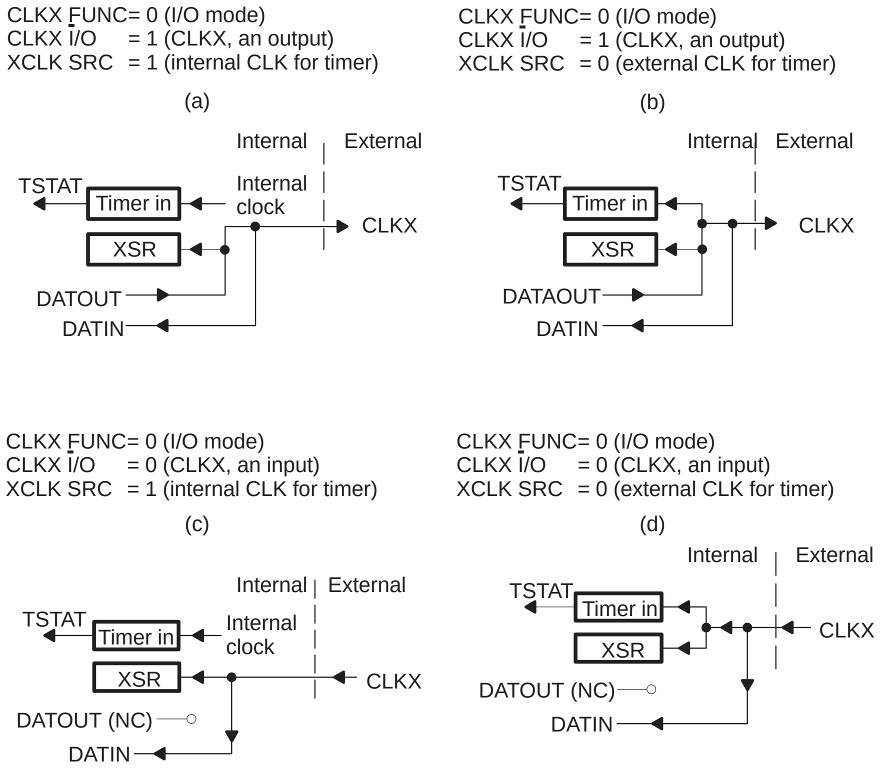
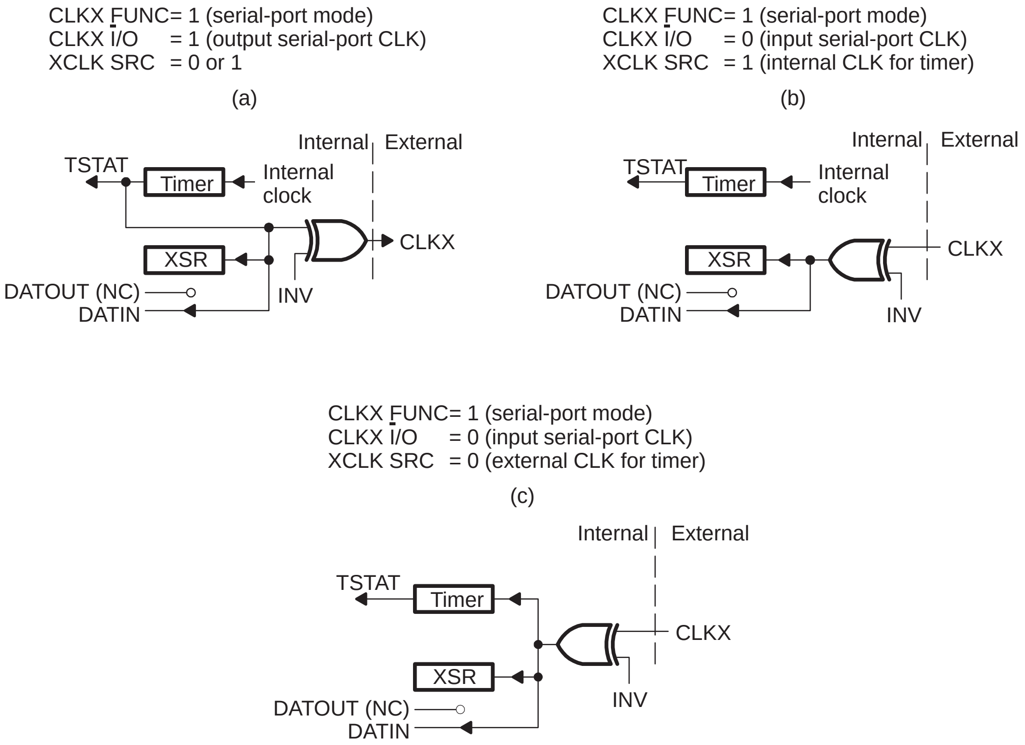


Figure 12–22. Serial-Port Clocking in Serial-Port Mode


CLKX FUNC= 1 (serial-port mode)
CLKX I/O = 1 (output serial-port CLK)
XCLK SRC = 0 or 1
(a)
Internal
External
TSTAT
Timer
Internal clock
CLKX
XSR
DATOUT (NC)
DATIN
INV
CLKX FUNC= 1 (serial-port mode)
CLKX I/O = 0 (input serial-port CLK)
XCLK SRC = 1 (internal CLK for timer)
(b)
Internal
External
TSTAT
Timer
Internal clock
CLKX
XSR
DATOUT (NC)
DATIN
INV
CLKX FUNC= 1 (serial-port mode)
CLKX I/O = 0 (input serial-port CLK)
XCLK SRC = 0 (external CLK for timer)
(c)
Internal
External
TSTAT
Timer
CLKX
XSR
DATOUT (NC)
DATIN
INV

12.2.10 Serial-Port Timing

The formula for calculating the frequency of the serial-port clock with an internally generated clock depends upon the operation mode of the serial-port timers, defined as:

f (pulse mode) = f (timer clock)/period register

f (clock mode) = f (timer clock)/(2 x period register)

An internally generated clock source f (timer clock) has a maximum frequency of f(H1)/2. An externally generated serial-port clock f (timer clock) (CLKX or CLKR) has a maximum frequency of less than f(H1)/2.6. See section 12.1.5 on page 12-7 for information on timer pulse/clock generation.

Transmit data is clocked out on the rising edge of the selected serial-port clock. Receive data is latched into the receive-shift register on the falling edge of the serial-port clock. All data is received MSB first and shifted to the left. If fewer than 32 bits are received, the data received is right-justified in the receive buffer.

The transmit ready (XRDY) signal specifies that the data-transmit register (DXR) is available to be loaded with new data. XRDY goes active as soon as the data is loaded into the transmit-shift register (XSR). The last word may still be shifting out when XRDY goes active. If DXR is loaded before the last word has completed transmission, the data bits transmitted are consecutive; that is, the LSB of the first word immediately precedes the MSB of the second, with all signaling valid as in two separate transmits. XRDY goes inactive when DXR is loaded and remains inactive until the data is loaded into the shifter.

The receive ready (RRDY) signal is active as long as a new word of data is loaded into the data-receive register and has not been read. As soon as the data is read, the RRDY bit is turned off.

When FSX is specified as an output, the activity of the signal is determined by the internal state of the serial port. If a fixed data rate is specified, FSX goes active when DXR is loaded into XSR. One serial-clock cycle later FSX turns inactive and data transmission begins. If a variable data rate is specified, the FSX pin is activated when the data transmission begins and remains active during the entire transmission of the word. Again, the data is transmitted one clock cycle after it is loaded into the data-transmit register.

An input FSX in the fixed data-rate mode must go active for at least one serial-clock cycle and then inactive to initiate the data transfer. The transmitter then sends the number of bits specified by the XLEN (bit field 19, serial-port global-control register) bits. In the variable data-rate mode, the transmitter begins sending from the time FSX goes active until the number of specified bits have been shifted out. In the variable data-rate mode, when the FSX status changes prior to all the data bits being shifted out, the transmission completes, and the DX pin is placed in a high-impedance state. An FSR input is exactly complementary to the FSX.

When using an external FSX, if DXR and XSR are empty, a write to DXR results in a DXR-to-XSR transfer. This data is held in the XSR until an FSX occurs. When the external FSX is received, the XSR begins shifting the data. If XSR is waiting for the external FSX, a write to DXR changes DXR, but a DXR-to-XSR transfer does not occur. XSR begins shifting when the external FSX is received, or when it is reset using XRESET.

12.2.10.1 Continuous Transmit and Receive Modes

When you choose continuous mode, consecutive writes do not generate or expect new sync pulse signaling. Only the first word of a block begins with an active synchronization. Thereafter, data is transmitted as long as new data is loaded into DXR before the last word has been transmitted. As soon as TXRDY is active and all of the data has been transmitted out of the shift register, the DX pin is placed in a high-impedance state, and a subsequent write to DXR initiates a new block and a new FSX.

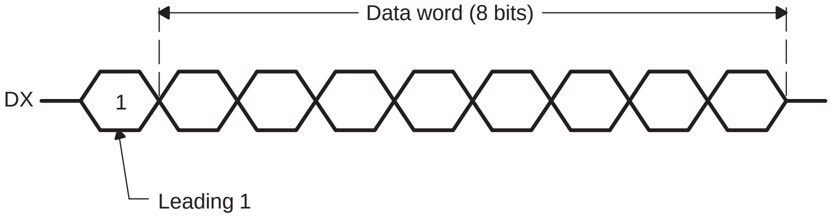
Similarly with FSR, the receiver continues shifting in new data and loading DRR. If the data-receive buffer is not read before the next word is shifted in, you will lose subsequent incoming data. You can use the RFSM bit to terminate the receive-continuous mode.

12.2.10.2 Handshake Mode

The handshake mode (HS = 1) allows for direct connection between processors. In this mode, all data words are transmitted with a leading 1 (see Figure 12–23). For example, in order to transmit an 8-bit word, the first bit sent is a 1, followed by the 8-bit data word.

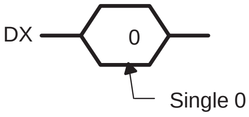
Once the serial port transmits a word in this mode, it does not transmit another word until it receives a separately transmitted 0 bit. Therefore, the 1 bit that precedes every data word is a request bit.

Figure 12–23. Data Word Format in Handshake Mode



After a serial port receives a word that has been read from the DRR (with the leading 1), the receiving serial port sends a single 0 to the transmitting serial port. The single 0 bit acts as an acknowledge bit (see Figure 12–24). This single acknowledge bit is sent every time the DRR is read, even if the DRR does not contain new data.

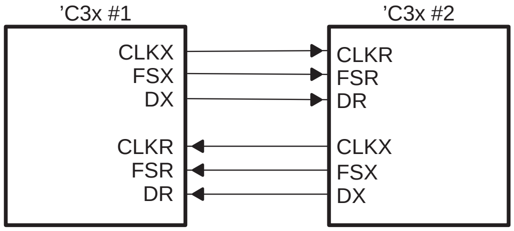
Figure 12–24. Single 0 Sent as an Acknowledge Bit



When the serial port is placed in the handshake mode, the insertion and deletion of a leading 1 for transmitted data, the sending of a 0 for acknowledgement of received data, and the waiting for this acknowledge bit are all performed automatically. Using this scheme, it is simple to connect processors with no external hardware and to guarantee secure communication. Figure 12–25 is a typical configuration.

In the handshake mode, FSX is automatically configured as an output. Continuous mode is automatically disabled. After a system reset or XRESET, the transmitter is always permitted to transmit. The transmitter and receiver must be reset when entering the handshake mode.

Figure 12–25. Direct Connection Using Handshake Mode



12.2.11 Serial-Port Interrupt Sources

A serial port has the following interrupt sources:

- **Transmit-timer interrupt.** The rising edge of XTSTAT causes a single-cycle interrupt pulse to occur. When XTINT is 0, this interrupt pulse is disabled.
- **Receive-timer interrupt.** The rising edge of RTSTAT causes a single-cycle interrupt pulse to occur. When RTINT is 0, this interrupt pulse is disabled.
- **Transmitter-interrupt.** Occurs immediately following a DXR-to-XSR transfer. The transmitter interrupt is a single-cycle pulse. When the serial-port global-control register bit XINT is 0, this interrupt pulse is disabled.
- **Receiver-interrupt.** Occurs immediately following an RSR-to-DRR transfer. The receiver interrupt is a single-cycle pulse. When the serial-port global-control register bit RINT is 0, this interrupt pulse is disabled.

The transmit-timer interrupt pulse is ORed with the transmitter interrupt pulse to create the CPU-transmit interrupt flag XINT. The receive-timer interrupt pulse is ORed with the receiver interrupt pulse to create the CPU receive-interrupt flag RINT.

12.2.12 Serial-Port Functional Operation

The following paragraphs and figures illustrate the functional timing of the various serial-port modes of operation. The timing descriptions are presented with the assumption that all signal polarities are configured to be positive (that is, CLKXP = CLKRP = DXP = DRP = FSXP = FSRP = 0). Logical timing, in situations where one or more of these polarities are inverted, is the same except with respect to the opposite polarity reference points (that is, rising vs. falling edges, etc.).

These discussions pertain to the numerous operating modes and configurations of the serial-port logic. When it is necessary to switch operating modes or change configurations of the serial port, you should do so only when $\overline{\text{XRESET}}$ or $\overline{\text{RRESET}}$ are asserted (low), as appropriate. When transmit configurations are modified, $\overline{\text{XRESET}}$ should be low, and when receive configurations are modified, $\overline{\text{RRESET}}$ should be low. When you use handshake mode, however, since the transmitter and receiver are interrelated, you should make any configuration changes with $\overline{\text{XRESET}}$ and $\overline{\text{RRESET}}$ both low.

All of the serial-port operating configurations can be classified in two categories: fixed data-rate timing and variable data-rate timing. Both categories support operation in either burst or continuous mode.

Burst-mode operation with variable data-rate timing is similar to burst-mode operation with fixed data-rate timing. With variable data-rate timing, however, FSX/R and data timing differ slightly at the beginning and end of transfers. Specifically, there are three major differences between fixed and variable data-rate timing:

- FSX/R pulses typically last for the entire transfer interval in variable data-rate timing operation, although FSR and external FSX are ignored after the first bit transferred. FSX/R pulses in fixed data-rate mode typically last only one CLKX/R cycle but can last longer.

- With variable data-rate timing, data transfer begins during the CLKX/R cycle in which FSX/R occurs. With fixed data-rate timing, data transfer begins in the CLKX/R cycle following FSX/R.

- With variable data-rate timing, frame sync inputs are ignored until the end of the last bit transferred. With fixed data-rate timing, frame sync inputs are ignored until the beginning of the last bit transferred.

The following paragraphs discuss fixed and variable data-rate operation and all of their variations.

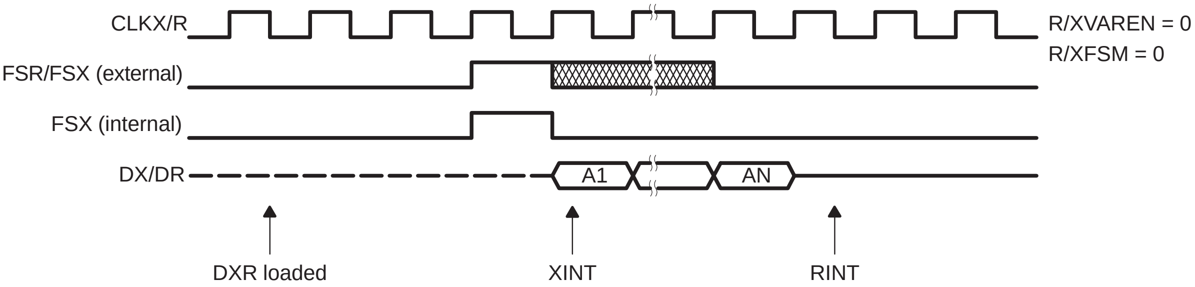
12.2.12.1 Fixed Data-Rate Timing Operation

Fixed data-rate serial-port transfers can occur in two varieties: burst mode and continuous mode. In burst mode, transfers of single words are separated by periods of inactivity on the serial port. In continuous mode, there are no gaps between successive word transfers; the first bit of a new word is transferred on the next CLKX/R pulse following the last bit of the previous word. This occurs continuously until the process is terminated. The following variations are included in fixed data-rate timing operations.

- **Fixed Burst Mode**

  In burst mode with fixed data-rate timing, FSX/FSR pulses initiate transfers, and each transfer involves a single word. With an internally generated FSX (see Figure 12–26), transmission is initiated by loading DXR. In this mode, there is a delay of approximately 2.5 CLKX cycles (depending on CLKX and H1 frequencies) from the time DXR is loaded until FSX occurs. With an external FSX, the FSX pulse initiates the transfer, and the 2.5-cycle delay effectively becomes a setup requirement for loading DXR with respect to FSX. In this case, you must load DXR no later than three CLKX cycles before FSX occurs. Once the XSR is loaded from the DXR, an XINT is generated.

Figure 12–26. Fixed Burst Mode

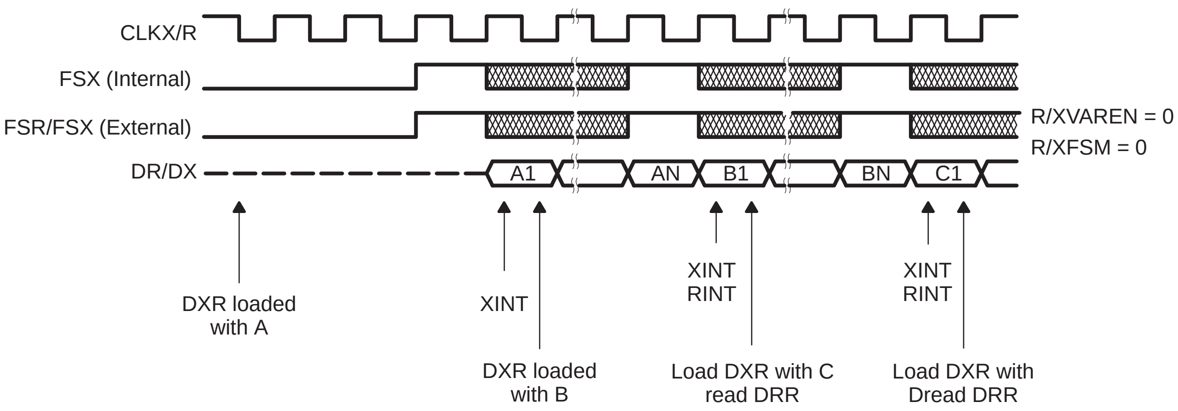


  In receive operations, once a transfer is initiated, FSR is ignored until the last bit. For burst-mode transfers, FSR must be low during the last bit, or another transfer will be initiated. After a full word has been received and transferred to the DRR, an RINT is generated.

- **Fixed Standard Mode**

  In fixed data-rate mode, you can perform continuous transfers even if R/XFSM = 0, as long as properly timed frame synchronization is provided, or as long as DXR is reloaded each cycle with an internally generated FSX (see Figure 12–27).

Figure 12–27. Fixed Standard Mode With Back-to-Back Frame Sync



For receive operations and with externally generated FSX, once transfers have begun, frame sync pulses are required only during the last bit transferred to initiate another contiguous transfer. Otherwise, frame sync inputs are ignored. Continuous transfers occur if the frame sync is held high. With an internally generated FSX, there is a delay of approximately 2.5 CLKX cycles from the time DXR is loaded until FSX occurs. This delay occurs each time DXR is loaded; therefore, during continuous transmission, the instruction that loads DXR must be executed by the *N*–3 bit for an *N*-bit transmission. Since delays due to pipelining vary, you should incorporate a conservative margin of safety in allowing for this delay.

Once the process begins, an XINT and an RINT are generated at the beginning of each transfer. The XINT indicates that the XSR has been loaded from DXR and can be used to cause DXR to be reloaded. To maintain continuous transmission in fixed rate mode with frame sync, especially with an internally generated FSX, DXR must be reloaded early in the ongoing transfer.

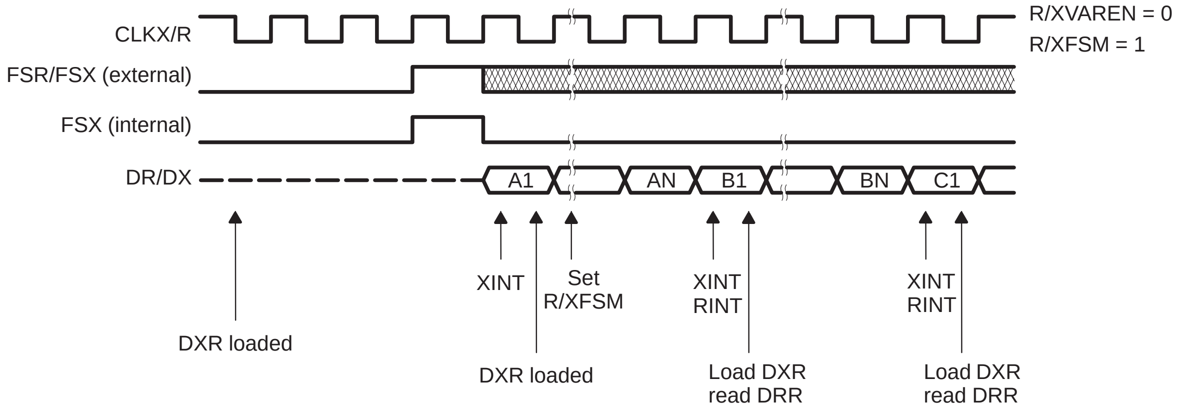
The RINT indicates that a full word has been received and transferred into the DRR; RINT indicates an appropriate time to read DRR.

Continuous transfers are terminated by discontinuing frame sync pulses or, in the case of an internally-generated FSX, not reloading DXR.

❑ **Fixed Continuous Mode**

You can accomplish continuous serial-port transfers, without the use of frame sync pulses, if R/XFSM is set to 1. In this mode, operation of the serial port is similar to continuous operation with frame sync, except that a frame sync pulse is involved only in the first word transferred, and no further frame sync pulses are used. Following the first word transferred (see Figure 12–28), no internal frame sync pulses are generated, and frame sync inputs are ignored. Additionally, you should set R/XFSM prior to or during the first word transferred; you must set R/XFSM no later than the transfer of the *N*–1 bit of the first word, except for transmit operations. For transmit operations in the fixed data-rate mode, XFSM must be set no later than the *N*–2 bit. You must clear R/XFSM no later than the *N*–1 bit to be recognized in the current cycle.

Figure 12–28. Fixed Continuous Mode Without Frame Sync

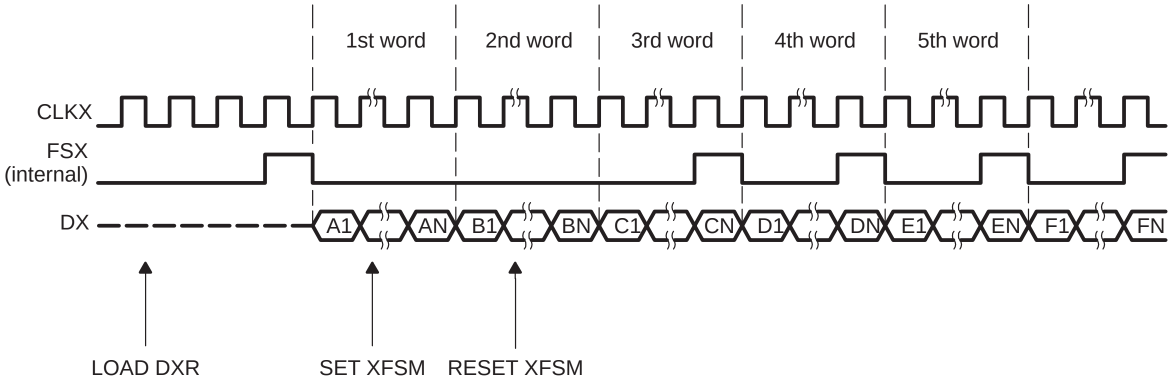


The timing of RINT and XINT and data transfers to and from DXR and DRR, respectively, are the same as in fixed data-rate standard mode with back-to-back frame syncs. This mode of operation also exhibits the same delay of 2.5 CLKX cycles after DXR is loaded before an internal FSX is generated. As in the case of continuous operation in fixed data-rate mode with frame sync, you must reload DXR no later than transmission of the *N*–3 bit.

❑ **Enabling or Disabling Frame Syncs in Fixed Mode**

When you use continuous operation in fixed data-rate mode, you can set and clear R/XFSM as desired, even during active transfers, to enable or disable the use of frame sync pulses as dictated by system requirements. Under most conditions, changing the state of R/XFSM occurs during the transfer in which the R/XFSM change was made, provided the change was made early enough in the transfer. For transmit operations with internal FSX in fixed data-rate mode, however, a 1-word delay occurs before frame sync pulse generation resumes when clearing XFSM to 0 (see Figure 12–29). In this case, one additional word is transferred before the next FSX pulse is generated. The clearing of XFSM is recognized during the transmission of the word currently being transmitted as long as XFSM is cleared no later than the *N*–1 bit. The setting of XFSM is recognized as long as XFSM is set no later than the *N*–2 bit.

Figure 12–29. Exiting Fixed Continuous Mode Without Frame Sync, FSX Internal



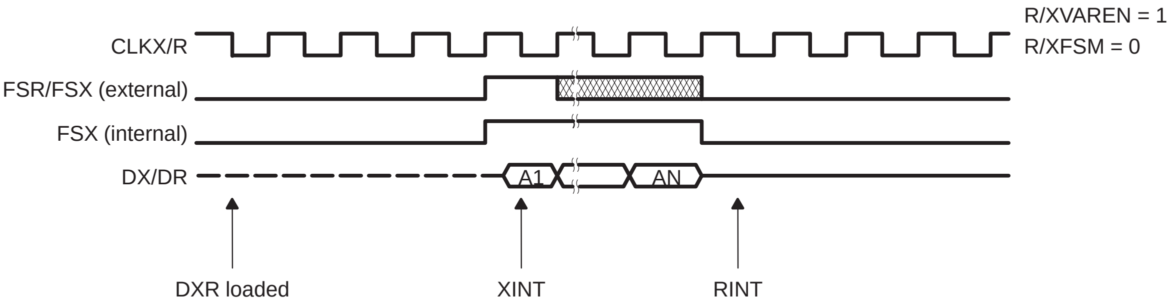
12.2.12.2 Variable Data-Rate Timing Operation

The following variations are included in variable data-rate timing operations.

❑ **Variable Burst Mode**

In burst mode with variable data-rate timing, FSX/FSR pulse lasts for the entire duration of transfer. With an internally generated FSX (see Figure 12–30), transmission is initiated by loading DXR. In this mode there is a delay of approximately 3.5 CLKX cycles (depending on CLKX and H1 freqency) from the time DXR is loaded until FSX occurs. With an external FSX, the FSX pulse initiates the transfer and the 3.5-cycle delay effectively becomes a setup requirement for loading DXR with respect to FSX. Therefore, in this case, you must load DXR no later than four CLKX cycles before FSX occurs. Once the XSR is loaded from the DXR, an XINT is generated.

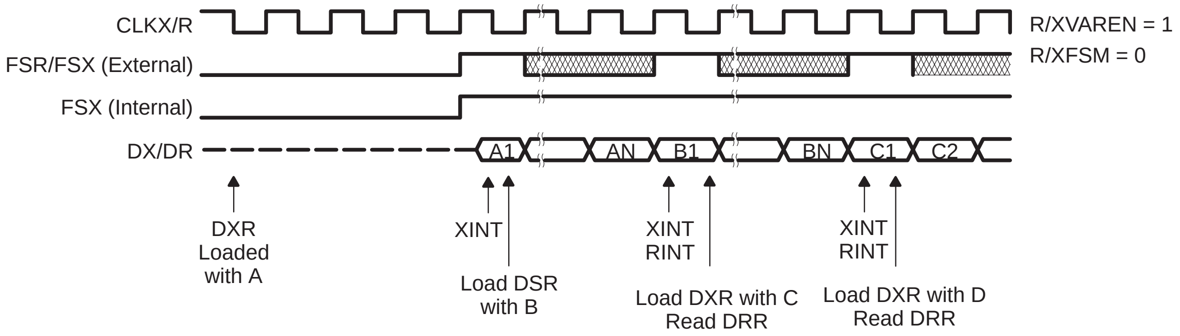
Figure 12–30. Variable Burst Mode



- **Variable Standard Mode**

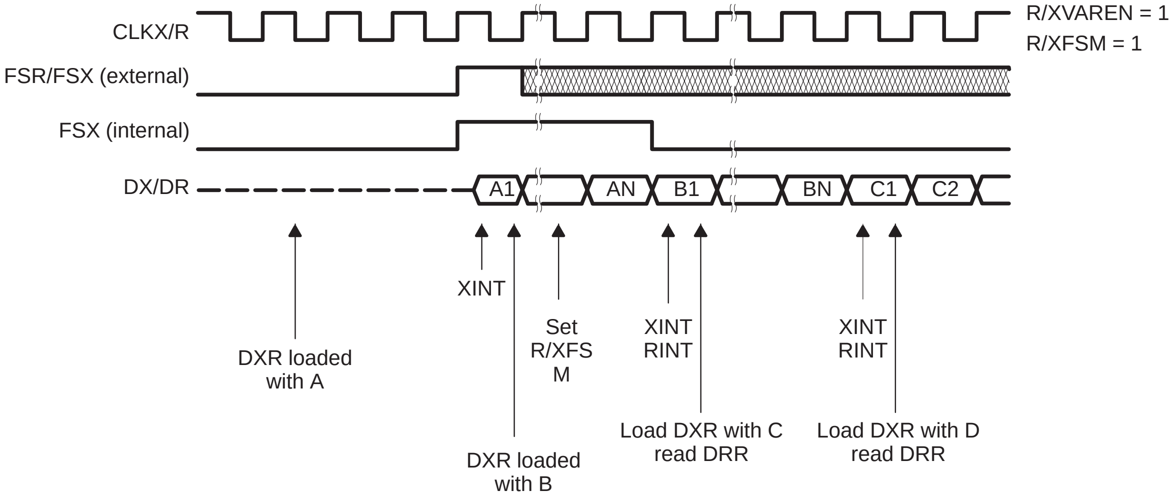
  When you transmit continuously in variable data-rate mode with frame sync, timing is the same as for fixed data-rate mode, except for the differences between these two modes as described in Section 12.2.12 *Serial-Port Functional Operation*, on page 12-35. The only other exception is that you must reload DXR no later than the *N*–4 bit to maintain continuous operation of the variable data-rate mode (see Figure 12–31); you must reload DXR no later than the *N*–3 bit to maintain continuous operation of the fixed data-rate mode.

Figure 12–31. Variable Standard Mode With Back-to-Back Frame Syncs



Continuous operation in variable data-rate mode without frame sync (see Figure 12–32) is similar to continuous operation without frame sync in fixed data-rate mode (see Figure 12–28). As with variable data-rate standard mode with back-to-back frame syncs, you must reload DXR no later than the *N*–4 bit to maintain continuous operation. Additionally, when R/XFSM is set or cleared in the variable data-rate mode, you must make the modification no later than the *N*–1 bit for the result to be affected in the current transfer.

Figure 12–32. Variable Continuous Mode Without Frame Sync



12.2.13 Serial-Port Initialization/Reconfiguration

The serial ports are controlled through memory-mapped registers on the dedicated peripheral bus. A general procedure for initializing and/or reconfiguring the serial ports follows.

1) Halt the serial port by clearing the XRESET and/or RRESET bits of the serial-port global-control register. To do this, write a 0 to the serial-port global-control register. The serial ports are halted on $\overline{\text{RESET}}$.

2) Configure the serial port via the serial-port global-control register (with XRESET = RRESET = 0) and the FSX/DX/CLKX and FSR/DR/CLKR port-control registers. If necessary, configure the receive/transmit registers; timer control (with $\overline{\text{XHLD}}$ = $\overline{\text{RHLD}}$ = 0), timer counter, and timer period. Refer to section 12.2.14 for more information.

3) Start the serial-port operation by setting the XRESET and RRESET bits of the serial-port global-control register and the $\overline{\text{XHLD}}$ and $\overline{\text{RHLD}}$ bits of the serial-port receive/transmit timer-control register, if necessary.

12.2.14 TMS320C3x Serial-Port Interface Examples

In addition to the examples presented in this section, you can find DMA/serial port initialization examples in Example 12–9 and Example 12–10 on pages 12-76 and 12-77, respectively.

12.2.14.1 Handshake Mode Example

When using the handshake mode, the transmit (FSX/DS/CLKX) and receive (FSR/DR/CLKR) signals transmit and receive data, respectively. Even if the ’C3x serial port is receiving data only with handshake mode, the transmit signals are still needed to transmit the acknowledge signal. Example 12–3 shows the serial-port register setup for the ’C3x serial-port handshake communication, as shown in Figure 12–25 on page 12-34.

Example 12–3. Serial-Port Register Setup #1

```
Global control         =  011x0x0xxxx00000000xx01100100b,
Transmit port control  =  0111h
Receive port control   =  0111h
S_port timer control   =  0Fh
S_port timer count     =  0h
S_port timer period    ≥  01h (if two C3xs have the same system clock).
```

Note: *x* = user-configurable

Since FSX is set as an output and continuous mode is disabled when handshake mode is selected, follow these steps:

1) Set the XFSM and RFSM bits to 0 and the FSXOUT bit to 1 in the global-control register.

2) Set the XRESET, RRESET, and HS bits to 1 in order to start the handshake communication.

3) Set the polarity of the serial-port pins active (high) for simplification.

4) Although the CLKX/CLKR can be set as either input or output, set the CLKX as output and the CLKR as input.

The rest of the bits are user-configurable as long as both serial ports have consistent setup.

You need the serial-port timer only if the CLKX or CLKR is configured as an output. Since only the CLKX is configured as an output, set the timer control register to 0Fh. When you use the serial-port timer, set the serial timer register to the proper value for the clock speed. The serial-port timer clock speed setup is similar to the ’C3x timer. Refer to Section 12.1, *Timers*, on page 12-2 for detailed information on timer clock generation.

The maximum clock frequency for serial transfers is f(CLKIN)/4 if the internal clock is used and f(CLKIN)/5.2 if an external clock is used. If two ’C3xs have the same system clock, the timer-period register should be set equal to or greater than 1, which makes the clock frequency equal to f(CLKIN)/8.

Example 12–4 and Example 12–5 are serial-port register setups for the above case. (Assume two ’C3xs have the same system clock.)

Example 12–4. Serial-Port Register Setup #1

```
Global control          =  0EBC0064h; 32 bits, fixed data rate, burst mode,
Transmit port control   =  0111h  ; FSX (output), CLKX (output) = F(CLKIN)/8
Receive port control    =  0111h  ; CLKR (input), handshake mode, transmit
S_port timer control    =  0Fh ; and receive interrupt is enabled.
S_port timer count      =  0h
S_port timer period     ≥  01h
```

Example 12–5. Serial-Port Register Setup #2

```
Global control          =  0C000364h; 8 bits, variable data rate, burst mode,
Transmit port control   =  0111h; FSX (output), CLKX (output) = f(CLKIN)/24
Receive port control    =  0111h  ; CLKR (input), handshake mode, transmit
S_port timer control    =  0Fh ; and receive interrupt is disabled.
S_port timer count      =  0h
S_port timer period     ≥  01h
```

Since the data has a leading 1 and the acknowledge signal is a 0 in the handshake mode, the ’C3x serial port can distinguish between the data and the acknowledge signal. Even if the ’C3x serial port receives the data before the acknowledge signal, the data is not misinterpreted as the acknowledge signal and lost. Additionally, the acknowledge signal is not generated until the data is read from the data-receive register (DRR); the ’C3x does not transmit the data and the acknowledge signal simultaneously.

12.2.14.2 CPU Transfer With Serial Port Transmit Polling Method

Example 12–6 sets up the CPU to transfer data (128 words) from an array buffer to the serial port 0 output register when the previous value stored in the serial-port output register has been sent. 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.

Example 12–6. CPU Transfer With Serial Port Transmit Polling Method

```
* TITLE: CPU TRANSFER WITH SERIAL-PORT TRANSMIT POLLING METHOD
*
         .GLOBAL START
         .DATA
SOURCE   .WORD _ARRAY
         .BSS _ARRAY,128        ; DATA ARRAY LOCATED IN .BSS SECTION
                                ; THE UNDERSCORE USED IS JUST TO MAKE IT
                                ; ACCESSIBLE FROM C (OPTIONAL)
SPORT    .WORD 808040H          ; SERIAL-PORT GLOBAL CONTROL REG ADDRESS
SPRESET  .WORD 008C0044         ; SERIAL-PORT RESET
SGCCTRL  .WORD 048C0044H        ; 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
RESET    .WORD 0H               ; SERIAL-PORT TIMER RESET VALUE
         .TEXT
START     LDP RESET             ; LOAD DATA PAGE POINTER
          ANDN 10H,IE           ; DISABLE SERIAL-PORT TRANSMIT INTERRUPT TO CPU

* SERIAL PORT INITIALIZATION
        LDI @SPORT,AR1
        LDI @RESET,R0
        LDI 4,IR0
        STI R0,*+AR1(IR0)       ; SERIAL-PORT TIMER RESET
        LDI @SPRESET,R0
        STI R0,*AR1             ; SERIAL-PORT RESET
        LDI @SXCTRL,R0          ; SERIAL-PORT TX CONTROL REG INITIALIZATON
        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

* CPU WRITES THE FIRST WORD

        LDI @SOURCE,AR0
        LDI *AR0++,R1
        STI R1,*+AR1(8)

* CPU WRITES 127 WORDS TO THE SERIAL PORT OUTPUT REG

        LDI 8,IR0
        LDI 2,R0
        LDI 126,RC
        RPTB  LOOP
WAIT    AND *AR1,R0,R2          ; WAIT UNTIL XRDY BIT = 1
        BZ  WAIT
LOOP    STI R1,*+AR1(IR0)
     || LDI *++AR0(1),R1
        BU $
        .END
```

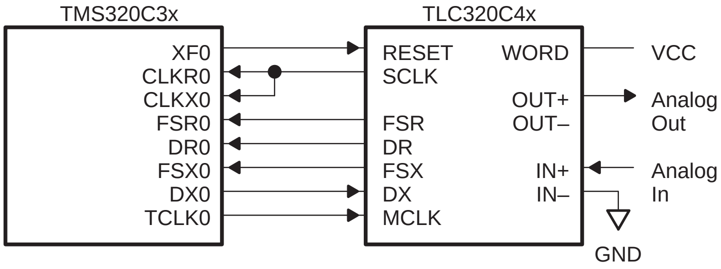
12.2.14.3 DMA Transfer With Serial Port Interrupt

Example 12–8 and Example 12–9 of Section 12.3.11 on page 12-74 use the DMA synchronized to serial port interrupts to transfer data (128 words) from an array buffer to the serial port0 output register.

12.2.14.4 Serial Analog Interface Chips Interface Example

The TLC320C4x analog interface chips (AIC) from Texas Instruments offer a zero-glue-logic interface to the 'C3x family of DSPs. The interface is shown in Figure 12–33 as an example of the 'C3x serial-port configuration and operation.

Figure 12–33. TMS320C3x Zero-Glue-Logic Interface to TLC320C4x Example



The 'C3x resets the AIC through the external pin XF0. It also generates the master clock for the AIC through the timer 0 output pin, TCLK0. (Precise selection of a sample rate may require the use of an external oscillator rather than the TCLK0 output to drive the AIC MCLK input.) In turn, the AIC generates the CLKR0 and CLKX0 shift clocks as well as the FSR0 and FSX0 frame synchronization signals.

A typical use of the AIC requires an 8-kHz sample rate of the analog signal. If the clock input frequency to the 'C3x device is 30 MHz, you should load the following values into the serial port and timer registers.

Serial Port:

Port global-control register	0E970300h
FSX/DX/CLKX port-control register	00000111h
FSR/DR/CLKR port-control register	00000111h

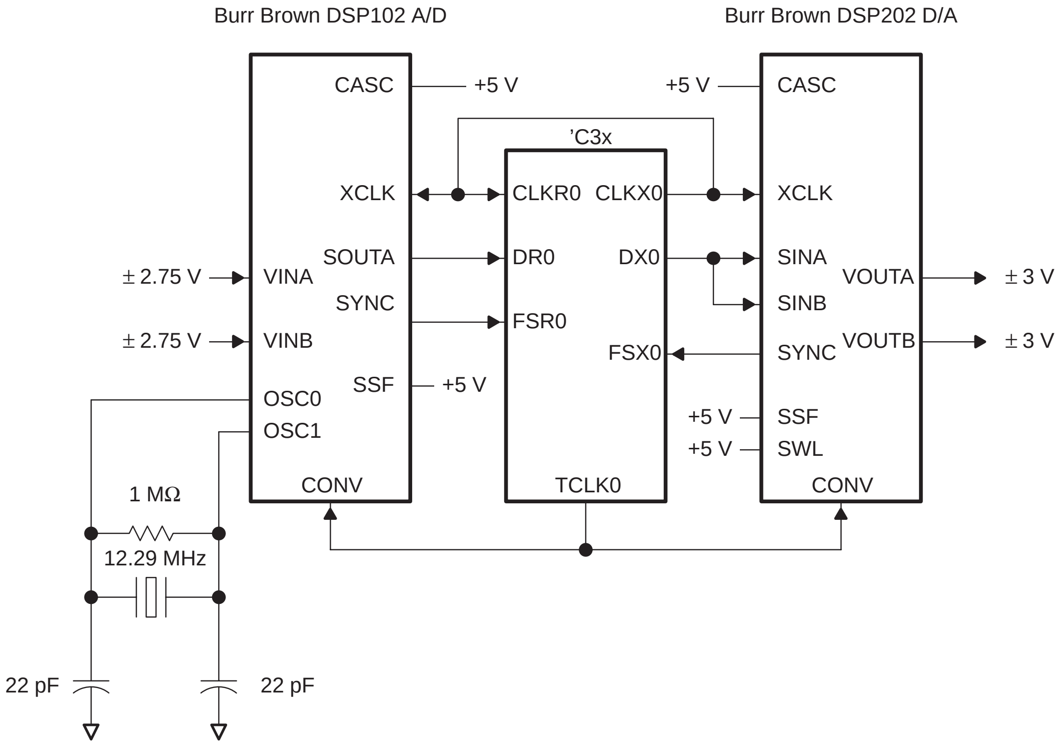
Timer:

Timer global-control register	000002C1h
Timer-period register	00000001h

12.2.14.5 Serial Analog-to-Digital (A/D) and Digital-to-Analog (D/A) Interface Example

The DSP201/2 and DSP101/2 family of D/As and A/Ds from Burr Brown also offer a zero-glue-logic interface to the ’C3x family of DSPs. The interface is shown in Example 12–7. This interface is used as an example of the ’C3x serial port configuration and operation.

Example 12–7. TMS320C3x Zero-Glue-Logic Interface to Burr Brown A/D and D/A



The DSP102 A/D is interfaced to the ’C3x serial-port receive side; the DSP202 D/A is interfaced to the transmit side. The A/Ds and D/As are hard-wired to run in cascade mode. In this mode, when the ’C3x initiates a convert command to the A/D via the TCLK0 pin, both analog inputs are converted into two 16-bit words, which are concatenated to form one 32-bit word.

1) The A/D signals the ’C3x via the A/D’s SYNC signal (connected to the FSR0 pin) that serial data is to be transmitted.

2) The 32-bit word is then serially transmitted, MSB first, out the SOUTA serial pin of the DSP102 to the DR0 pin of the ’C3x serial port.

3) The ’C3x is programmed to drive the analog interface bit clock from the CLKX0 pin of the ’C3x.

4) The bit clock drives both the A/D’s and D/A’s XCLK input.

5) The ’C3x transmit clock also acts as the input clock on the receive side of the ’C3x serial port.

6) Since the receive clock is synchronous to the internal clock of the ’C3x, the receive clock can run at full speed (that is, f(H1)/2).

Similarly, on receiving a convert command, the pipelined D/A converts the last word received from the ’C3x and signals the ’C3x via the SYNC signal (connected to the ’C3x FSX0 pin) 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 D/A as shown in Example 12–7.

The ’C3x is set up to transfer bits at the maximum rate of about 8 Mbps, with a dual-channel sample rate of about 44.1 kHz. Assuming a 32-MHz CLKIN, you can configure this standard-mode fixed-data-rate signaling interface by setting the registers as described below:

Serial Port:

Port global-control register	0EBC0040h
FSX/DX/CLKX port-control register	00000111h
FSR/DR/CLKR port-control register	00000111h
Receive/transmit timer-control register	0000000Fh

Timer:

Timer global-control register	000002C1h
Timer-period register	000000B5h

12.3 DMA Controller

The DMA controller is a programmable peripheral that transfers blocks of data to any location in the memory map without interfering with CPU operation. The ’C3x can interface to slow, external memories and peripherals without reducing throughput to the CPU. The ’C3x DMA controller features are:

- Transfers to and from anywhere in the processor’s memory map. For example, transfers can be made to and from on-chip memory, off-chip memory, and on-chip serial ports.
- One DMA channel for memory-to-memory transfers in ‘C30 and ‘C31. Two DMA channels for memory-to-memory transfers in ‘C32.
- Concurrent CPU and DMA controller operation with DMA transfers at the same rate as the CPU (supported by separate internal DMA address and data buses).
- Source and destination-address registers with auto increment/decrement.
- Synchronization of data transfers via external and internal interrupts.

12.3.1 DMA Functional Description

The DMA controller supports one (‘C30 and ‘C31) or two (‘C32) DMA channels that perform transfers to and from anywhere in the ‘C3x memory map.

Each DMA channel is controlled by four registers that are mapped in the ‘C3x peripheral address space, as shown in Figure 12–35. The major DMA registers are described in Section 12.3.3.

The DMA controller has dedicated on-chip address and data buses (see Figure 2–5 through Figure 2–7 on pages 2-14 through 2-16 for a block diagram of the peripherals of the ‘C3x). All accesses made by the DMA channels are arbitrated in the DMA controller and take place over these dedicated buses. The DMA channels transfer data in a sequential time-slice fashion, rather than simultaneously, because they share common buses.

The DMA channels can run constantly or can be triggered by external ($\overline{\text{INT}}$3–0) or internal (on-chip timers and serial ports) interrupts.

’C30 ’C31

12.3.1.1 TMS320C30 and TMS320C31 DMA Controller

The ’C30 and ’C31 have an on-chip direct memory access (DMA) controller that reduces the need for the CPU to perform input/output functions. The DMA controller can perform input/output operations without interfering with the operation of the CPU. Therefore, it is possible to interface the ’C30 and ’C31 to slow external memories and peripherals (A/Ds, serial ports, etc.) without reducing the computational throughput of the CPU. The result is improved system performance and decreased system cost.

’C32

12.3.1.2 TMS320C32 Two-Channel DMA Controller

The ’C32 has an improved DMA that supports two channels and configurable priorities. The next sections discuss the new features.

The ’C32 has a two-channel (channel 0 and channel 1) DMA instead of a one-channel DMA as in the ’C30/’C31 devices. The ’C32’s DMA functions similarly to that of the ’C30/’C31 DMA but with the addition of DMA/CPU priority scheme and inter-DMA priority mode. Although the ’C32 CPU supports both floating-point and integer data access with different data size from the external memory, the ’C32’s DMA transfer is strictly an integer data transfer. The integer data access of the ’C32 DMA is the same as the CPU integer data access — 32-bit internal and data size conversion at the external memory interface port.

12.3.2 DMA Basic Operation

If a block of data is to be transferred from one region in memory to another region in memory (as shown in Figure 12–34), the following sequence is performed:

DMA Registers Initialization

1) The source-address register of a DMA channel is loaded with the address of the memory location to read from.

2) The destination-address register of the same DMA channel is loaded with the address of the memory location to write to.

3) The transfer counter is loaded with the number of words to be transferred.

4) The DMA channel control register is loaded with the appropriate modes to synchronize the DMA controller reads and writes with interrupts.

DMA Start

5) The DMA controller is started through the DMA START field in the DMA channel control register.

Word Transfers

6) The DMA channel reads a word from the source-address register and writes it to a temporary register within the DMA channel.

7) After a read by the DMA channel, the source-address register is incremented, decremented, or unchanged depending on the INCSRC or DECSRC bit fields of DMA channel control register.

8) After the read operation completes, the DMA channel writes the temporary register value to the destination-address pointed to by the destination-address register.

9) After the destination-address has been fetched, the transfer-counter register is decremented and the destination-address register is incremented, decremented, or unchanged, depending on the INCDST or DECDST bit fields of the DMA channel control register.

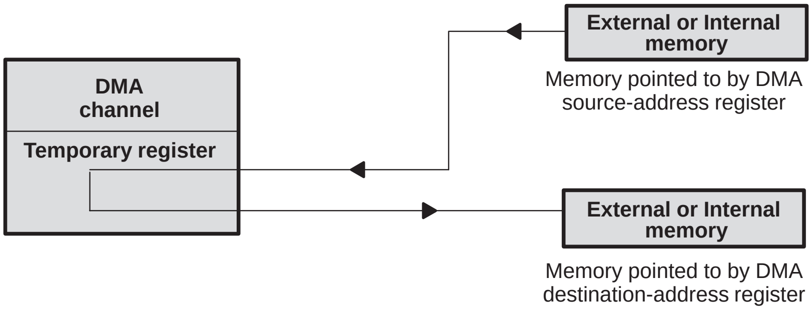
10) During every data write, the transfer counter is decremented. The block transfer terminates when the transfer counter reaches zero *and* the write of the last transfer is completed. The DMA channel sets the transfer-counter interrupt (TCINT) flag in the DMA channel control register.

After the completion of a block transfer, the DMA controller can be programmed to do several things:

- ❑ Stop until reprogrammed (TC = 1)
- ❑ Continue transferring data (TC = 0)
- ❑ Generate an interrupt to signal the CPU that the block transfer is complete (TCINT = 1)

The DMA can be stopped by setting the START bits to 00, 01, or 10. When the DMA is restarted (START = 11), it completes any pending transfer.

Figure 12–34. DMA Basic Operation


External or Internal memory
Memory pointed to by DMA source-address register
DMA channel
Temporary register
External or Internal memory
Memory pointed to by DMA destination-address register

12.3.3 DMA Registers

Each DMA channel has four registers designated as follows:

- ❑ **Control register:** contains the status and mode information about the associated DMA channel
- ❑ **Source-address register:** contains the memory address of data to be read
- ❑ **Destination-address register:** contains the memory address where data is written
- ❑ **Transfer-counter register:** contains the block size to move

After reset, the control register, the transfer counter, and the auxiliary transfer-counter registers are set to 0s and the other registers are undefined.

Figure 12–36 shows these registers for ’C30 and ’C31. Figure 12–37 shows these registers for ’C32.

The format of the DMA-channel control register is shown in Figure 12–35. The text following the figure describes the functions of each field in the register.

At reset, each DMA-channel control register is set to 0. This makes the DMA channels lower-priority than the CPU, sets up the source address and destination address to be calculated through linear addressing, and configures the DMA channel in the unified mode.

Figure 12–35. Memory-Mapped Locations for DMA Channels

Address	**Register**
808000h	DMA 0 global control
808004h	DMA 0 source address
808006h	DMA 0 destination address
808008h	DMA 0 transfer counter
808010h	DMA 1 global control†
808014h	DMA 1 source address†
808016h	DMA 1 destination address†
808018h	DMA 1 transfer counter†

† ’C32 only

12.3.3.1 DMA Global-Control Register

The global-control register controls the state in which the DMA controller operates. This register also indicates the status of the DMA, which changes every cycle. Source and destination addresses can be incremented, decremented, or synchronized using specified global-control register bits. At system reset, all bits in the DMA control register are cleared to 0. Figure 12–36 shows the global-control registers for the ’C30 and ’C31 devices. Figure 12–37 and Figure 12–38 show the global-control registers for the ’C32. Table 12–6 shows the register bits, bit names, and bit functions.

Figure 12–36. TMS320C30 and TMS320C31 DMA Global-Control Register ’C30 ’C31

31 15	14 12	11	10	9 8	7	6	5	4	3 2	1 0
xx	xx	TCINT	TC	SYNC	DECDST	INCDST	DECSRC	INCSRC	STAT	START
		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W

Notes: 1) R = read, W = write
2) xx = reserved bit, read as 0

Figure 12–37. TMS320C32 DMA0 Global-Control Register ’C32

31 15	14	13 12	11	10	9 8	7	6	5	4	3 2	1 0
xx	PRIORITY MODE	DMAO PRI	TCINT	TC	SYNC	DECDST	INCDST	DECSRC	INCSRC	STAT	START
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W

Notes: 1) R = read, W = write
2) xx = reserved bit, read as 0

Figure 12–38. TMS320C32 DMA1 Global-Control Register ’C32

31 15	14	13 12	11	10	9 8	7	6	5	4	3 2	1 0
xx	xx	DMA1 PRI	TCINT	TC	SYNC	DECDST	INCDST	DECSRC	INCSRC	STAT	START
		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W

Notes: 1) R = read, W = write
2) xx = reserved bit, read as 0

Table 12–6. DMA Global-Control Register Bits Summary

<table>
<tr><th>Abbreviation</th><th>Reset Value</th><th>Name</th><th>Description</th></tr>
<tr><td><b>START</b></td><td>00</td><td>DMA start control</td><td>Controls the state in which the DMA starts and stops. The DMA may be stopped without any loss of data.
The following table summarizes the START bits and DMA operation:
<table>
<tr><th>Bit 1</th><th>Bit 0</th><th>Function</th></tr>
<tr><td>0</td><td>0</td><td>DMA read or write cycles in progress are completed; any data read is ignored. Any pending read or write is cancelled. The DMA is reset so that when it starts, a new transaction begins; that is, a read is performed (Reset value).</td></tr>
<tr><td>0</td><td>1</td><td>If a read or write has begun, it is completed before it stops. If a read or write has not begun, no read or write is started.</td></tr>
<tr><td>1</td><td>0</td><td>If a DMA transfer has begun, the entire transfer is complete (including both read and write operations) before stopping. If a transfer has not begun, none is started.</td></tr>
<tr><td>1</td><td>1</td><td>DMA starts from reset or restarts from the previous state.</td></tr>
</table>
When the DMA completes a transfer, the START bits remain in 11 (base 2). The DMA starts when the START bits are set to 11 and one of the following conditions applies:
❑ The transfer counter is set to a value different from 0x0.
❑ The TC bit is set to 0.</td></tr>
<tr><td><b>STAT</b></td><td>00</td><td>DMA status</td><td>Indicates the status of the DMA and changes every cycle.
The following table summarizes the STAT bits and DMA status.
<table>
<tr><th>Bit 3</th><th>Bit 2</th><th>Function</th></tr>
<tr><td>0</td><td>0</td><td>The DMA is being held between DMA transfer (between a write and a read). This is the value at reset.</td></tr>
<tr><td>0</td><td>1</td><td>DMA is being held in the middle of a DMA transfer (between a read and a write).</td></tr>
<tr><td>1</td><td>0</td><td>Reserved.</td></tr>
<tr><td>1</td><td>1</td><td>DMA busy. DMA is performing a read or write or waiting for a source or destination synchronization interrupt.</td></tr>
</table></td></tr>
</table>

Table 12–6. DMA Global-Control Register Bits Summary (Continued)

<table>
<tr><th>Abbreviation</th><th>Reset Value</th><th>Name</th><th>Description</th></tr>
<tr><td><b>INCSRC</b></td><td>0</td><td>DMA source address increment mode</td><td>If INCSRC = 1, the source address is incremented after every read.</td></tr>
<tr><td><b>DECSRC</b></td><td>0</td><td>DMA source address decrement mode</td><td>If DECSRC = 1, the source address is decremented after every read.
If INCSRC = DECSRC, the source address is not modified after a read.</td></tr>
<tr><td><b>INCDST</b></td><td>0</td><td>DMA destination address increment mode</td><td>If INCDST = 1, the destination address is incremented after every write.</td></tr>
<tr><td><b>DECDST</b></td><td>0</td><td>DMA destination address decrement mode</td><td>If DECDST = 1, the destination address is decremented after every write.
If INCDST = DECDST, the destination address is not modified after a write.</td></tr>
<tr><td><b>SYNC</b></td><td>0</td><td>DMA synchronization mode</td><td>Determines the timing synchronization between the events initiating the source and destination transfers.
The following table summarizes the SYNC bits and DMA synchronization.
<table>
<tr><th>Bit 9</th><th>Bit 8</th><th>Function</th></tr>
<tr><td>0</td><td>0</td><td>No synchronization. Enabled interrupts are ignored (reset value).</td></tr>
<tr><td>0</td><td>1</td><td>Source synchronization. A read is performed when an enabled interrupt occurs.</td></tr>
<tr><td>1</td><td>0</td><td>Destination synchronization. A write is performed when an enabled interrupt occurs.</td></tr>
<tr><td>1</td><td>1</td><td>Source and destination synchronization. A read is performed when an enabled interrupt occurs. A write is then performed when the next enabled interrupt occurs.</td></tr>
</table>
</td></tr>
<tr><td><b>TC</b></td><td>0</td><td>DMA transfer mode</td><td>Affects the operation of the transfer counter.
If TC = 0, transfers are not terminated when the transfer counter becomes 0.
If TC = 1, transfers are terminated when the transfer counter becomes 0.</td></tr>
<tr><td><b>TCINT</b></td><td>0</td><td>DMA transfer counter interrupt mode</td><td>If TCINT = 1, the DMA interrupt is set when the transfer counter makes a transition to 0.
If TCINT = 0, the DMA interrupt is not set when the transfer counter makes a transition to 0.</td></tr>
</table>

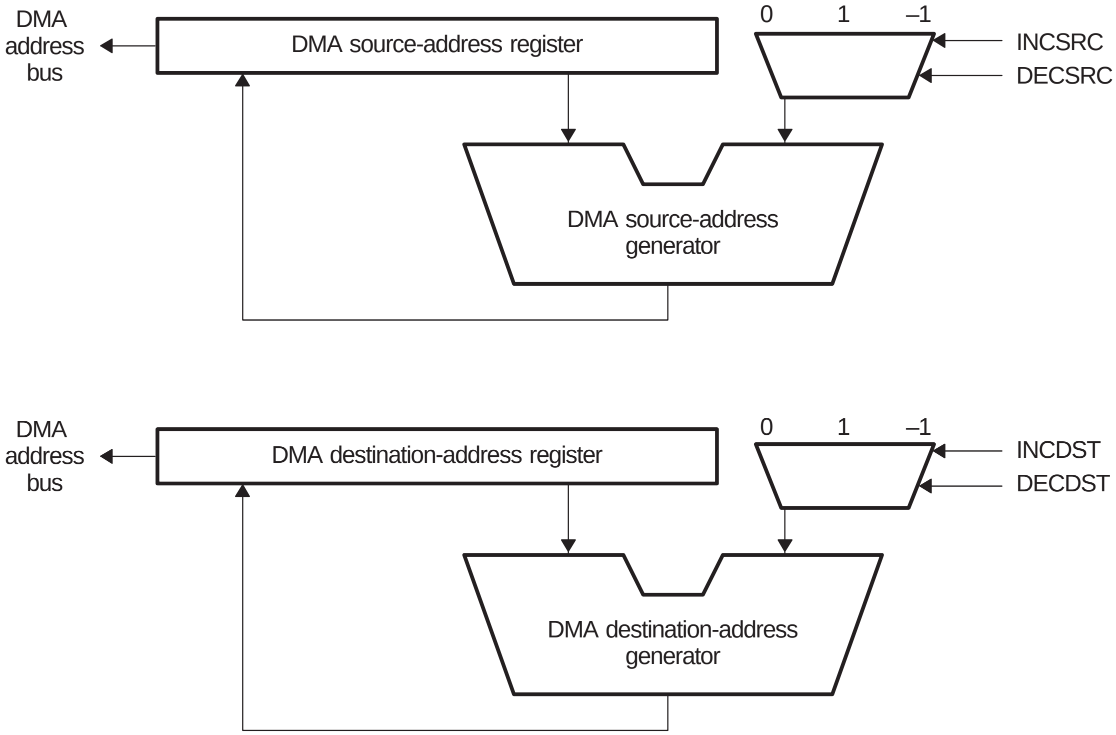
Table 12–6. DMA Global-Control Register Bits Summary (Continued)

<table>
<tr><th>Abbreviation</th><th>Reset Value</th><th>Name</th><th>Description</th></tr>
<tr><td>’C32 <b>DMA0 PRI</b></td><td>00</td><td>CPU/DMA channel 0 priority mode</td><td>(on the DMA0 control register) (’C32 only)</td></tr>
<tr><td>’C32 <b>DMA1 PRI</b></td><td>00</td><td>CPU/DMA channel 1 priority mode</td><td>(on the DMA1 control register) (‘C32 only)
Configures CPU/DMA controller priority. (See Section 12.3.6 on page 12-63).
The following table explains the DMA PRI bits and CPU/DMA priorities.
<table>
<tr><th>Bit 13</th><th>Bit 12</th><th>Function</th></tr>
<tr><td>0</td><td>0</td><td>DMA has lower priority than the CPU access. If the DMA channel and the CPU are requesting the same resource, the CPU has priority (reset value).</td></tr>
<tr><td>0</td><td>1</td><td>Reserved.</td></tr>
<tr><td>1</td><td>0</td><td>Rotating arbitration, which sets priorities be tween the CPU and DMA channel by alternating their accesses (but not exactly equally). Priority rotates between the CPU and DMA accesses when they conflict during consecutive instruction cycles.</td></tr>
<tr><td>1</td><td>1</td><td>DMA has higher priority than the CPU access. If the DMA channel and the CPU are requesting the same resource, the DMA has priority.</td></tr>
</table></td></tr>
<tr><td>’C32 <b>PRIORITY MODE</b></td><td>0</td><td>DMA channels priority mode</td><td>If PRIORITY MODE = 0, fixed priority for the two DMA channels. DMA channel 0 always has priority over DMA channel 1.
If priority mode = 1, rotating priority for the two DMA channels. DMA channel 0 has priority after the device is reset. After reset, the last channel serviced has the lowest priority. The arbitration is performed at DMA service boundaries, that is, after either a DMA read or DMA write.
See Section 12.3.5 on page 12-62 for more information.</td></tr>
</table>

12.3.3.2 Destination-Address and Source-Address Registers

The DMA destination-address and source-address registers are 24-bit registers whose contents specify destination and source addresses. As specified by control bits DECSRC, INCSRC, DECDST, and INCDST of the DMA global-control register, these registers are incremented, decremented, or remain unchanged at the end of the corresponding memory access; that is, the source register for a read and the destination register for a write (see Figure 12–39). On system reset, 0 is written to these registers.

Figure 12–39. DMA Controller Address Generation


DMA address bus
DMA source-address register
0
1
–1
INCSRC
DECSRC
DMA source-address generator
DMA address bus
DMA destination-address register
0
1
–1
INCDST
DECDST
DMA destination-address generator

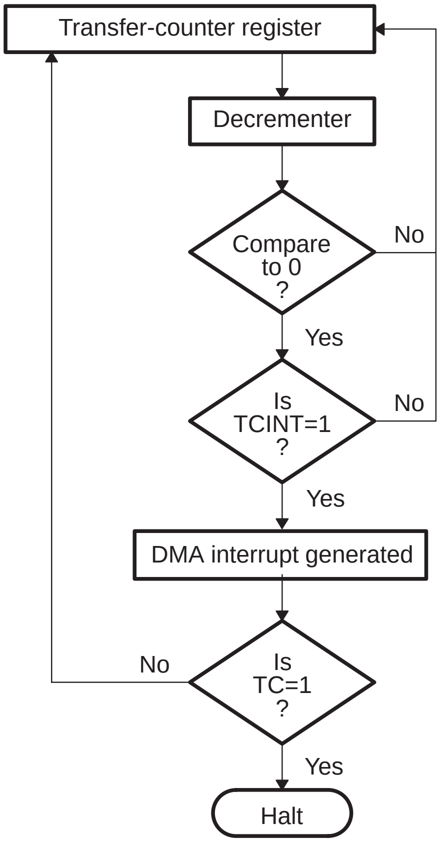
12.3.3.3 Transfer-Counter Register

The transfer-counter register is a 24-bit register that contains the number of words to be transmitted. Figure 12–40 shows the transfer-counter operation. It is controlled by a 24-bit counter that decrements at the beginning of a DMA memory write. In this way, it can control the size of a block of data transferred. The transfer-counter register is set to 0 at system reset. When the TCINT bit of the DMA global-control register is set, the transfer-counter register causes a DMA interrupt flag to be set when 0 is reached.

The counter is decremented after completing the destination-address fetch. The interrupt is generated after the transfer counter is decremented and after the completion of the write of the last transfer.

The decrementer checks whether the transfer counter equals 0 after the decrement is performed. As a result, if the counter register has a value of 1, then the DMA channel can be halted after only one transfer is performed. Thus, by setting the transfer counter to 1, the DMA channel transfers the minimum possible number of words (1 time). The value of the transfer counter is treated as an unsigned integer. Transfers can be halted when a 0 value is detected after a decrement. If the DMA controller channel is not halted after the transfer reaches zero, the counter continues decrementing below 0. Thus, by setting the transfer counter to 0, the DMA channel transfers the maximum possible number of words (100 0000h times).

Figure 12–40. Transfer-Counter Operation



12.3.4 CPU/DMA Interrupt-Enable Register

The CPU/DMA interrupt-enable register (IE) is a 32-bit register located in the CPU register file. The CPU interrupt-enable bits are in locations 10–1. The DMA interrupt-enable bits are in locations 26–16. A 1 in a CPU/DMA interrupt-enable register bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. At reset, 0 is written to this register.

Figure 12–41 shows the CPU/DMA interrupt-enable registers for the ‘C30 and ‘C31. Figure 12–42 shows the CPU/DMA interrupt-enable register for the ‘C32. Table 12–7 describes the register bits, bit names, and bit functions.

’C30 ’C31

Figure 12–41. TMS320C30 and TMS320C31 CPU/DMA Interrupt-Enable Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	EDINT (DMA)	ETINT1 (DMA)	ETINT0 (DMA)	ERINT1 (DMA)	EXINT1 (DMA)	ERINT0 (DMA)	EXINT0 (DMA)	EINT3 (DMA)	EINT2 (DMA)	EINT1 (DMA)	EINT0 (DMA)
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	xx	EDINT (CPU)	ETINT1 (CPU)	ETINT0 (CPU)	ERINT1 (CPU)	EXINT1 (CPU)	ERINT0 (CPU)	EXINT0 (CPU)	EINT3 (CPU)	EINT2 (CPU)	EINT1 (CPU)	EINT0 (CPU)
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) R = read, W = write

2) xx = reserved bit, read as 0

’C32

Figure 12–42. TMS320C32 CPU/DMA Interrupt-Enable Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
EINT3 (DMA1)	EINT2 (DMA1)	EINT1 (DMA1)	EINT0 (DMA1)	EDINT0 (DMA1)	EDINT1 (DMA0)	ETINT1 (DMA0)	ETINT0 (DMA0)	ETINT1 (DMA1)	ETINT0 (DMA1)	ERINT0 (DMA1)	EXINT0 (DMA0)	EINT3 (DMA0)	EINT2 (DMA0)	EINT1 (DMA0)	EINT0 (DMA0)
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	EDINT1 (CPU)	EDINT0 (CPU)	ETINT1 (CPU)	ETINT0 (CPU)	xx	xx	ERINT0 (CPU)	EXINT0 (CPU)	EINT3 (CPU)	EINT2 (CPU)	EINT1 (CPU)	EINT0 (CPU)
				R/W	R/W	R/W	R/W			R/W	R/W	R/W	R/W	R/W	R/W

Notes: 1) R = read, W = write

2) xx = reserved bit, read as 0

Table 12–7. CPU/DMA Interrupt-Enable Register Bits

Abbreviation	Reset Value	Description
EINT0 (CPU)	0	CPU external interrupt 0 enable
EINT1 (CPU)	0	CPU external interrupt 1 enable
EINT2 (CPU)	0	CPU external interrupt 2 enable
EINT3 (CPU)	0	CPU external interrupt 3 enable
EXINT0 (CPU)	0	CPU serial port 0 transmit interrupt enable
ERINT0 (CPU)	0	CPU serial port 0 receive interrupt enable
EXINT1 (CPU)	0	CPU serial port 1 transmit interrupt enable (’C30 only)
ERINT1 (CPU)	0	CPU serial port 1 receive interrupt enable (’C30 only)
ETINT0 (CPU)	0	CPU timer0 interrupt enable
ETINT1 (CPU)	0	CPU timer1 interrupt enable
EDINT (CPU)	0	CPU DMA controller interrupt enable (’C30 and ’C31 only)
EDINT0 (CPU)	0	CPU DMA0 controller interrupt enable (’C32 only)
EDINT1 (CPU)	0	CPU DMA1 controller interrupt enable (’C32 only)
EINT0 (DMA)	0	DMA external interrupt 0 enable (’C30 and ’C31 only)
EINT1 (DMA)	0	DMA external interrupt 1 enable (’C30 and ’C31 only)
EINT2 (DMA)	0	DMA external interrupt 2 enable (’C30 and ’C31 only)
EINT3 (DMA)	0	DMA external interrupt 3 enable (’C30 and ’C31 only)
EINT0 (DMA0)	0	DMA0 external interrupt 0 enable (’C32 only)
EINT1 (DMA0)	0	DMA0 external interrupt 1 enable (’C32 only)
EINT2 (DMA0)	0	DMA0 external interrupt 2 enable (’C32 only)
EINT3 (DMA0)	0	DMA0 external interrupt 3 enable (’C32 only)
EXINT0 (DMA)	0	DMA serial port 0 transmit interrupt enable (’C30 and ’C31 only)
ERINT0 (DMA)	0	DMA serial port 0 receive interrupt enable (’C30 and ’C31 only)
EXINT1 (DMA)	0	DMA serial port 1 transmit interrupt enable (’C30 only)
ERINT1 (DMA)	0	DMA serial port 1 receive interrupt enable (’C30 only)
EXINT0 (DMA0)	0	DMA0 serial port 1 transmit interrupt enable (’C32 only)
ERINT0 (DMA1)	0	DMA1 serial port 1 receive interrupt enable (’C32 only)

Table 12–7. CPU/DMA Interrupt-Enable Register Bits (Continued)

Abbreviation	Reset Value	Description
ETINT0 (DMA)	0	DMA timer0 interrupt enable (’C30 and ’C31)
ETINT1 (DMA)	0	DMA timer1 interrupt enable (’C30 and ’C31 only)
ETINT0 (DMA0)	0	DMA0 timer1 interrupt enable (’C32 only)
ETINT1 (DMA0)	0	DMA0 timer1 interrupt enable (’C32 only)
ETINT0 (DMA1)	0	DMA1 timer0 interrupt enable (’C32 only)
ETINT1 (DMA1)	0	DMA1 timer1 interrupt enable (’C32 only)
EDINT (DMA)	0	DMA controller interrupt enable (’C30 and ’C31)
EDINT1 (DMA0)	0	DMA0-DMA1 controller interrupt enable (’C32 only)
EDINT0 (DMA1)	0	DMA1-DMA0 controller interrupt enable (’C32 only)
EINT0 (DMA1)	0	DMA1 external interrupt 0 enable (’C32 only)
EINT1 (DMA1)	0	DMA1 external interrupt 1 enable (’C32 only)
EINT2 (DMA1)	0	DMA1 external interrupt 2 enable (’C32 only)
EINT3 (DMA1)	0	DMA1 external interrupt 2 enable (’C32 only)

12.3.5 TMS320C32 DMA Internal Priority Schemes

’C32

Because all accesses made by the two DMA channels take place over one common internal DMA data and address bus, a priority scheme for bus arbitration is required. Within the DMA controller, two priority schemes are used to designate which channel is serviced next:

❑ A fixed priority scheme with channel 0 always having the highest priority and channel 1 the lowest

❑ A rotating priority scheme that places the most recently serviced channel at the bottom of the priority list (default setup after reset)

12.3.5.1 Fixed Priority Scheme

This scheme provides a fixed (unchanging) priority for each channel as follows:

Priority	Channel
Highest	0
Lowest	1

To select fixed priority, set the PRIORITY MODE bit (bit 14) of channel 0’s DMA-channel control register to 1.

12.3.5.2 Rotating Priority Scheme

In a rotating priority scheme, the last channel serviced becomes the lowest priority channel. The other channel sequentially rotates through the priority list with the lowest channel next to the last-serviced channel becoming the highest priority on the following request. The priority rotates every time the channel most recently granted priority completes its access. At system reset, the channels are ordered from highest to lowest priority (0, 1).

To select this scheme, set the PRIORITY MODE bit (bit 14) of channel 0’s DMA control register to 0.

12.3.6 CPU and DMA Controller Arbitration

The DMA controller transfers data on its own internal buses. Arbitration is necessary only when a resource conflict exists between the DMA controller and the CPU. The arbitration causes no delay. When there is no conflict, the CPU and DMA controller accesses proceed in parallel.

All arbitration between the CPU and the DMA controller is on an access basis. DMA controller internal memory access starts during H3 (see Section 8.5, *Clocking Memory Access,* for more information).

When the CPU and DMA controllers request the same resource, priority is determined as follows:

❑ For the ‘C30 and ‘C31, the CPU always has higher priority, thus the DMA must wait until the CPU frees the resource.

❑ For the ‘C32, the DMA channel’s DMA PRI bits (bits 12 and 13 of the channel control register) define the arbitration rules (as shown in Table 12–8). The CPU has higher priority than the DMA when DMA PRI = 00_2; it has lower priority than the DMA when DMA PRI = 11_2. They rotate priority when DMA PRI = 01_2.

Table 12–8.TMS320C32 DMA PRI Bits and CPU/DMA Arbitration Rules

DMA PRI (Bits 13–12)	Description
0 0	DMA access is lower priority than the CPU access. If the DMA channel and the CPU request the same resource, then the CPU has priority. (DMA PRI bits are set to 00_2 at reset.)
0 1	This setting selects *rotating* arbitration, which sets priorities between the CPU and DMA channel by alternating their accesses, but not exactly equally. Priority rotates between CPU and DMA accesses when they conflict during *consecutive instruction cycles***.** The first time the DMA channel and the CPU request the same resource, the CPU has priority. If, in the following instruction cycle, the DMA controller and the CPU again request the same resource, the DMA has priority. Alternate access continues as long as the CPU and DMA requests conflict in consecutive instruction cycles. When there is no conflict in a previous instruction cycle, the CPU has priority.
1 0	Reserved
1 1	DMA access is higher priority than the CPU access. If the DMA channel and the CPU request the same resource, the DMA has priority.

12.3.7 DMA and Interrupts

The DMA controller uses interrupts in the following way:

❑ It can send interrupts to the CPU or other DMA channel when a block transfer finishes. See the TCINT bit field in the DMA global-control register (Figure 12–36, Figure 12–37, or Figure 12–38 on page 12-53). The EDINT bit field (’C30 and ’C31) or the EDINT0 and EDINT1 bit fields (’C32) in the interrupt-enable register must be set to allow the CPU to be interrupted by the DMA.

❑ It can receive interrupts from the external interrupt pins ($\overline{\text{INT}}$3–0), the timers, the serial ports, or other DMA channel.

This section explains how the DMA receives interrupts. This process is called synchronization.

All of the interrupts that the DMA controller receives are detected by the CPU interrupt controller and latched by the CPU in the appropriate interrupt-flag register.

The DMA and the CPU can respond to the same interrupt if the CPU is not involved in any pipeline conflict or in any instruction that halts instruction fetching. Refer to section 7.6.2, *Interrupt Vector Table and Prioritization*, on page 7-29 for more details. It is also possible for different DMA channels to respond to the same interrupt. If the same interrupt is selected for source and destination synchronization, both read and write cycles are enabled with a single incoming interrupt.

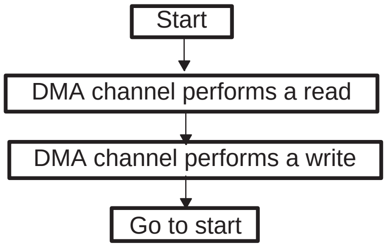
12.3.7.1 Interrupts and Synchronization of DMA Channels

You can use interrupts to synchronize DMA channels. This section describes the following four synchronization mechanisms:

- No synchronization (SYNC = 0 0)

  When SYNC = 0 0, no synchronization is performed. The DMA performs reads and writes whenever there are no conflicts. All interrupts are ignored and are considered to be globally disabled. However, no bits in the DMA interrupt-enable register are changed. Figure 12–43 shows the synchronization mechanism when SYNC = 0 0.

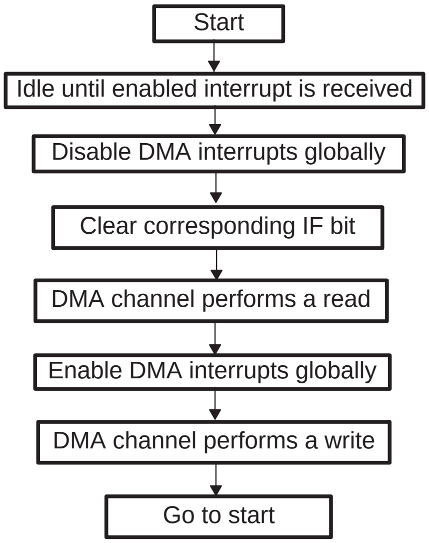
Figure 12–43. Mechanism for No DMA Synchronization



- Source synchronization (SYNC = 0 1)

  When SYNC = 0 1, the DMA is synchronized to the source (see Figure 12–44). A read is not performed until an interrupt is received by the DMA. Then all DMA interrupts are disabled globally. However, no bits in the DMA interrupt-enable register are changed.

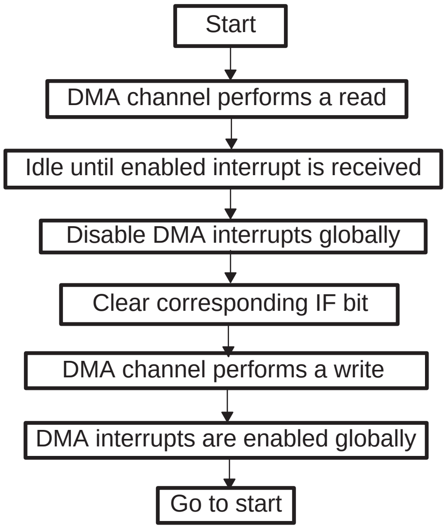
Figure 12–44. Mechanism for DMA Source Synchronization


Start
Idle until enabled interrupt is received
Disable DMA interrupts globally
Clear corresponding IF bit
DMA channel performs a read
Enable DMA interrupts globally
DMA channel performs a write
Go to start

❑ Destination synchronization (SYNC = 1 0)

When SYNC = 1 0, the DMA is synchronized to the destination. First, all interrupts are ignored until the read is complete. Though the DMA interrupts are considered globally disabled, no bits in the DMA interrupt-enable register are changed. A write is not performed until an interrupt is received by the DMA, while the read is performed without waiting for the interrupt. Figure 12–45 shows the synchronization mechanism when SYNC = 1 0.

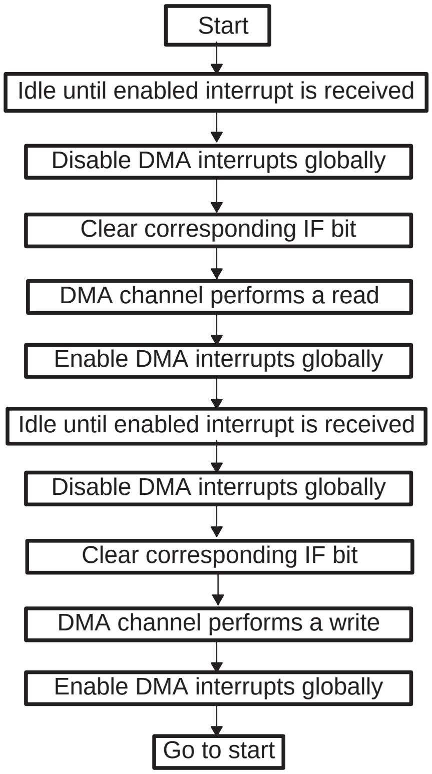
Figure 12–45. Mechanism for DMA Destination Synchronization


Start
DMA channel performs a read
Idle until enabled interrupt is received
Disable DMA interrupts globally
Clear corresponding IF bit
DMA channel performs a write
DMA interrupts are enabled globally
Go to start

- ❑ Source and destination synchronization (SYNC = 1 1)

  When SYNC = 1 1, the DMA is synchronized to both the source and destination. A read is performed when an interrupt is received. Then, a write is performed on the following interrupt. Figure 12–46 shows source and destination synchronization when SYNC = 1 1.

Figure 12–46. Mechanism for DMA Source and Destination Synchronization


Start
Idle until enabled interrupt is received
Disable DMA interrupts globally
Clear corresponding IF bit
DMA channel performs a read
Enable DMA interrupts globally
Idle until enabled interrupt is received
Disable DMA interrupts globally
Clear corresponding IF bit
DMA channel performs a write
Enable DMA interrupts globally
Go to start

12.3.8 DMA Memory Transfer Timing

The ’C30 and ’C31 devices provide one DMA channel, while the ’C32 device provides two DMA channels. The maximum data transfer rate that the ’C3x DMA sustains is one word every two cycles. In the ’C32, the two DMA channels transfer data in a sequential time-slice fashion, rather than simultaneously, because the two channels share one common set of busses.

The data transfer rate for a DMA channel (assuming a single-channel access with no conflicts between CPU or other DMA channels) is as follows:

- On-chip memory and peripheral
  - DMA read: One cycle
  - DMA write: One cycle
- External memory ($\overline{\text{STRB}}$, $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, $\overline{\text{MSTRB}}$)
  - DMA read: Two cycles (one cycle external read followed by one cycle load of internal DMA register)
  - DMA write: Two cycles (identical to CPU write)
- External memory (IO$\overline{\text{STRB}}$)
  - DMA read: Three cycles (two-cycle external read followed by one cycle load of internal DMA register)
  - DMA write: Two cycles (identical to CPU write)

If the DMA started and is transferring data over either external bus, do not modify the bus-control register associated with that bus. If you must modify the bus-control register (see Chapter 9 or 10), stop the DMA, make the modification, and then restart the DMA. Failure to do this may produce an unexpected zero-wait-state bus access.

DMA memory transfer timing can be very complicated, especially if bus resource conflicts occur. However, some rules help you calculate the transfer timing for certain DMA setups. For simplification, the following section focuses on a single-channel DMA memory transfer timing with no conflict with the CPU or other DMA channels. You can obtain the actual DMA transfer timing by combining the calculations for single-channel DMA transfer timing with those for bus resource conflict situations.

12.3.8.1 Single DMA Memory Transfer Timing

When the DMA memory transfer has no conflict with the CPU or any other DMA channels, the number of cycles of a DMA transfer depends on whether the source and destination location are designated as on-chip memory, peripheral, or external ports. When the external port is used, the DMA transfer speed is affected by two factors: the external bus wait state and the read/write conflict (for example, if a write is followed by a read, the read takes one extra half-cycle. See Figure 12–48 footnote on page 12-70). Figure 12–47 through Figure 12–49 show the number of cycles a DMA transfer requires from different sources to different destinations. Entries in the table represent the number of cycles required to do the T transfers, assuming that there are no pipeline conflicts. A timing diagram for the DMA transfers accompanies each figure.

Figure 12–47. DMA Timing When Destination is On Chip

Cycles (H1)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**Rate**
Source on chip	R_1		R_2		R_3		R_4		R_5		R_6		R_7		R_8				(1 + 1) *T*
Destination on chip		W_1		W_2		W_3		W_4		W_5		W_6		W_7					
Source STRB, STRB0, STRB1, MSTRB bus	R_1	R_1	R_1	I		R_2	R_2	R_2	I		R_3	R_3	R_3	I					(2 + C_r +1) *T*
		C_r					C_r					C_r							
Destination on chip					W_1					W_2					W_3				
Source IOSTRB bus	R_1	R_1	R_1	R_1	I		R_2	R_2	R_2	R_2	I		R_3	R_3	R_3	R_3	I		(3 + C_r + 1) *T*
			C_r						C_r						C_r				
Destination on chip						W_1						W_2						W_3	

Legend:

T	=	Number of transfers	W	=	Single-cycle writes
C_r	=	Source-read wait states	R_n	=	Multicycle reads
C_w	=	Destination-write wait states	W_n	=	Multicycle writes
R	=	Single-cycle reads	I	=	Internal register cycle

Figure 12–48. DMA Timing When Destination is an STRB, STRB0, STRB1, MSTRB Bus

Cycles (H1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Rate
Source on chip	R_1		R_2				R_3				R_4				R_5					$(1 + 2 + C_W)\ T$
Destination STRB, STRB0, STRB1, MSTRB bus		W_1	W_1	W_1	W_1	W_2	W_2	W_2	W_2	W_3	W_3	W_3	W_3	W_4	W_4	W_4	W_4	. . .		
				C_W				C_W				C_W				C_W				
Source STRB, STRB0, STRB1 bus	R_1	R_1	R_1	I					R_2	R_2	R_2	I								$(2 + C_r + 2 + C_W)\ T + 0.5\ (T – 1)$[†]
		C_r								C_r										
Destination STRB, STRB0, STRB1 bus					W_1	W_1	W_1	W_1					W_2	W_2	W_2	W_2	. . .			
							C_W								C_W					$(3.5 + Cr + 2 + Cw)\ T + .5\ (T – 1)$[†]
('C30 only) Source IOSTRB	R_1	R_1	R_1	R_1	I	R_2	R_2	R_2	R_2	I	R_3	R_3	R_3	R_3	I	R_4	R_4	R_4	R_4	$(3 + C_r + 2 + C_W) + (2 + C_W + \max[1, C_r – C_W + 1])\ (T–1)$
			C_r					C_r					C_r					C_r		
Destination STRB bus						W_1	W_1	W_1	W_1		W_2	W_2	W_2	W_2		W_3	W_3	W_3	W_3	
								C_W					C_W					C_W		

Legend:

T = Number of transfers		W = Single-cycle writes	
C_r = Source-read wait states		R_n = Multicycle reads	
C_W = Destination-write wait states		W_n = Multicycle writes	
R = Single-cycle reads		I = Internal register cycle	

[†] Write followed by read incurs in one extra half-cycle.

Figure 12–48. DMA Timing When Destination is an STRB, STRB0, STRB1, MSTRB Bus (Continued)

Cycles (H1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Rate
Source IOSTRB bus	R_1	R_1	R_1	R_1	I					R_2	R_2	R_2	R_2	I					$(3 + C_r + 2 + C_W)\ T + 0.5\ (T – 1)$†
			C_W									C_W							
Destination STRB0, STRB1, or MSTRB bus						W_1	W_1	W_1	W_1						W_2	W_2	W_2	W_2	
								C_W									C_W		
('C30 only) Source STRB bus	R_1	R_1	R_1	I		R_2	R_2	R_2	I		R_3	R_3	R_3	I					$(2 + C_r + 2 + C_W) + (2 + C_W + \max[1, C_r – C_W + 1])\ (T–1)$
		C_r					C_r					C_r							
Destination MSTRB bus					W_1	W_1	W_1	W_1		W_2	W_2	W_2	W_2		W_3	W_3	W_3	W_3	
							C_W					C_W					C_W		

Legend:

T	=	Number of transfers	W	=	Single-cycle writes
C_r	=	Source-read wait states	R_n	=	Multicycle reads
C_W	=	Destination-write wait states	W_n	=	Multicycle writes
R	=	Single-cycle reads	I	=	Internal register cycle

† Write followed by read incurs in one extra half-cycle.

Figure 12–49. DMA Timing When Destination is an IOSTRB Bus

Cycles (H1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Rate
Source on chip	R_1		R_2				R_3				R_4				R_5				
Destination IOSTRB		W_1	W_1	W_1	W_1	W_2	W_2	W_2	W_2	W_3	W_3	W_3	W_3	W_4	W_4	W_4	W_4		$1 + (2 + C_W)\ T$
				C_W				C_W				C_W				C_W			
('C30 only) Source STRB bus	R_1	R_1	R_1	I		R_2	R_2	R_2	I		R_3	R_3	R_3	I					$(2 + C_r + 2 + C_W) + (2 + C_W + \max(1, C_r - C_W + 1))\ (T - 1)$
		C_r					C_r					C_r							
Destination IOSTRB bus					W_1	W_1	W_1	W_1		W_2	W_2	W_2	W_2		W_3	W_3	W_3	W_3	
							C_W					C_W					C_W		
Source STRB0, STRB1, MSTRB bus	R_1	R_1	R_1	I					R_2	R_2	R_2	I							$(2 + C_r + 2 + C_W)\ T + (T–1)$†
		C_r								C_r									
Destination IOSTRB					W_1	W_1	W_1	W_1					W_2	W_2	W_2	W_2			
							C_W								C_W				

Legend:

T	=	Number of transfers	W	=	Single-cycle writes
Cr	=	Source-read wait states	R_n	=	Multicycle reads
Cw	=	Destination-write wait states	W_n	=	Multicycle writes
R	=	Single-cycle reads	I	=	Internal register cycle

† Write followed by read incurs in one extra cycle.

12.3.9 DMA Initialization/Reconfiguration

You can control the DMA through memory-mapped registers located on the dedicated peripheral bus. Following is the general procedure for initializing and/or reconfiguring the DMA:

1) Halt the DMA by clearing the START bits of the DMA global-control register. You can do this by writing a 0 to the DMA global-control register. The DMA is halted on $\overline{\text{RESET}}$.

2) Configure the DMA through the DMA global-control register (with START = 00), as well as the DMA source, destination, and transfer-counter registers, if necessary. Refer to Section 12.3.11 on page 12-74 for more information.

3) Start the DMA by setting the START bits of the DMA global-control register as necessary.

12.3.10 Hints for DMA Programming

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

❑ Reset the DMA register before starting it. This clears any previously latched interrupt that may no longer exist.

❑ In the event of a CPU-DMA access conflict, the CPU always prevails. Carefully allocate the different 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.

Note: Expansion and Peripheral Buses

The expansion and peripheral buses cannot be accessed simultaneously because they are multiplexed into a common port (see Figure 2–1 on page 2-3). This might increase CPU-DMA access conflicts.

❑ Ensure that each interrupt is received when you use interrupt synchronization; 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.

The following are indications that the DMA has finished a set of transfers:

❑ The DINT bit in the IF register is set to 1 (interrupt polling). This requires that you set the TCINT bit in the DMA control register first. This interrupt-polling method does not cause any additional CPU-DMA access conflict.

- ❑ The transfer counter has a zero value. However, the transfer counter is decremented after the DMA read operation finishes (not after the write operation). Nevertheless, a transfer counter with a 0 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 CPU/DMA access conflicts.

12.3.11 DMA Programming Examples

Example 12–8, Example 12–9, and Example 12–10 illustrate initialization procedures for the DMA.

When linking the examples, you should allocate section memory addresses carefully to avoid CPU-DMA conflict. In the C3x, the CPU always prevails in cases of conflict. In the event of a CPU program/DMA data conflict, cache enabling helps if the *.text* section is in external memory. For example, when linking the code in Example 12–8, Example 12–9, and Example 12–10, the *.text* section can be allocated into RAM0, *.data* into RAM1, and *.bss* into RAM1, where RAM0 corresponds to on-chip RAM block 0 and RAM1 corresponds to on-chip RAM block 1.

In Example 12–8, 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 program initializes the ST and IE registers for interrupt processing.

Example 12–8. 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 DMA TO CPU
        OR  2000H,ST             ; ENABLE CPU INTERRUPTS GLOBALLY
        LDI @CONTROL,R0          ; INITIALIZE DMA GLOBAL CONTROL REGISTER
        STI R0,*AR0              ; START DMA TRANSFER
        BU $
        .END
```

Example 12–9 sets up the DMA to transfer data (128 words) from the serial port 0 input register to an array buffer with serial port receive interrupt (RINT0). The DMA sends an interrupt to the CPU when the data transfer completes.

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; no CPU serial-port interrupt-vector setting is required.

Example 12–9. 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   00200400H      ; 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)

SPORT      .WORD   808040H        ; SERIAL-PORT GLOBAL-CONTROL REG ADDRESS
SGCCTRL    .WORD   0A300080H      ; SERIAL-PORT GLOBAL-CONTROL REG INITIALIZATION
SRCTRL     .WORD   111H           ; SERIAL-PORT RX PORT CONTROL REG INITIALIZATION
STCTRL     .WORD   3C0H           ; SERIAL-PORT TIMER-CONTROL REG INITIALIZATION
STPERIOD   .WORD   00020000H      ; SERIAL-PORT TIMER PERIOD
SPRESET    .WORD   01300080H          ; SERIAL-PORT RESET
RESET      .WORD   0H             ; SERIAL-PORT TIMER RESET
           .TEXT

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 12–10 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 8(H1) 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; 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 12–10. 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  048C0044H    ; SERIAL-PORT GLOBAL-CONTROL REG INITIALIZATION
SXCTRL    .WORD  111H         ; SERIAL-PORT TX PORT CONTROL REG INITIALIZATION
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 12–10. 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 are as follows:

❑ 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 memory order; the highest addressed member of the block is to become 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 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 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 13

Assembly Language Instructions

The ’C3x assembly language instruction set supports numeric-intensive, signal-processing, and general-purpose applications. (The addressing modes used with the instructions are described in Chapter 5.)

The ’C3x instruction set can also use one of 20 condition codes with any of the 10 conditional instructions, such as LDF*cond*. This chapter defines the condition codes and flags.

The assembler allows optional syntax forms to simplify the assembly language for special-case instructions. These optional forms are listed and explained.

Each of the individual instructions is described and listed in alphabetical order (see subsection 13.6.2, *Optional Assembler Syntax*, on page 13-34). Example instructions demonstrate the special format and explain its content.

This chapter discusses these topics:

Topic **Page**

13.1 Instruction Set

The 'C3x instruction set is well suited to digital signal processing and other numeric-intensive applications. All instructions are a single machine word long, and most instructions require one cycle to execute. In addition to multiply and accumulate instructions, the 'C3x possesses a full complement of general-purpose instructions.

The instruction set contains 113 instructions organized into the following functional groups:

❑ Load and store
❑ 2-operand arithmetic/logical
❑ 3-operand arithmetic/logical
❑ Program control
❑ Interlocked operations
❑ Parallel operations

Each of these groups is discussed in the following subsections.

13.1.1 Load and Store Instructions

The 'C3x supports 13 load and store instructions (see Table 13–1). These instructions can:

❑ Load a word from memory into a register
❑ Store a word from a register into memory
❑ Manipulate data on the system stack

Two of these instructions can load data conditionally. This is useful for locating the maximum or minimum value in a data set. See Section 13.5 on page 13-28 for detailed information on condition codes.

Table 13–1. Load and Store Instructions

Instruction	Description	Instruction	Description
LDE	Load floating-point exponent	POP	Pop integer from stack
LDF	Load floating-point value	POPF	Pop floating-point value from stack
LDF*cond*	Load floating-point value conditionally	PUSH	Push integer on stack
LDI	Load integer	PUSHF	Push floating-point value on stack
LDI*cond*	Load integer conditionally	STF	Store floating-point value
LDM	Load floating-point mantissa	STI	Store integer
LDP	Load data page pointer		

13.1.2 2-Operand Instructions

The 'C3x supports 35 2-operand arithmetic and logical instructions. The two operands are the source and destination. The source operand can be a memory word, a register, or a part of the instruction word. The destination operand is always a register.

As shown in Table 13–2, these instructions provide integer, floating-point or logical operations, and multiprecision arithmetic.

Table 13–2. 2-Operand Instructions

Instruction	Description	Instruction	Description
ABSF	Absolute value of a floating-point number	NORM	Normalize floating-point value
ABSI	Absolute value of an integer	NOT	Bitwise-logical complement
ADDC†	Add integers with carry	OR†	Bitwise-logical OR
ADDF†	Add floating-point values	RND	Round floating-point value
ADDI†	Add integers	ROL	Rotate left
AND†	Bitwise-logical AND	ROLC	Rotate left through carry
ANDN†	Bitwise-logical AND with complement	ROR	Rotate right
ASH†	Arithmetic shift	RORC	Rotate right through carry
CMPF†	Compare floating-point values	SUBB†	Subtract integers with borrow
CMPI†	Compare integers	SUBC	Subtract integers conditionally
FIX	Convert floating-point value to integer	SUBF†	Subtract floating-point values
FLOAT	Convert integer to floating-point value	SUBI†	Subtract integer
LSH†	Logical shift	SUBRB	Subtract reverse integer with borrow
MPYF†	Multiply floating-point values	SUBRF	Subtract reverse floating-point value
MPYI†	Multiply integers	SUBRI	Subtract reverse integer
NEGB	Negate integer with borrow	TSTB†	Test bit fields
NEGF	Negate floating-point value	XOR†	Bitwise-exclusive OR
NEGI	Negate integer		

† 2- and 3-operand versions

13.1.3 3-Operand Instructions

Whereas 2-operand instructions have a single source operand (or shift *count*) and a destination operand, 3-operand instructions can have two source operands (or one source operand and a *count* operand) and a destination operand. A source operand can be a memory word or a register. The destination of a 3-operand instruction is always a register.

Table 13–3 lists the instructions that have 3-operand versions. You can omit the *3* in the mnemonic from 3-operand instructions (see subsection 13.6.2 on page 13-34).

Table 13–3. 3-Operand Instructions

Instruction	Description	Instruction	Description
ADDC3	Add with carry	MPYF3	Multiply floating-point values
ADDF3	Add floating-point values	MPYI3	Multiply integers
ADDI3	Add integers	OR3	Bitwise-logical OR
AND3	Bitwise-logical AND	SUBB3	Subtract integers with borrow
ANDN3	Bitwise-logical AND with complement	SUBF3	Subtract floating-point values
ASH3	Arithmetic shift	SUBI3	Subtract integers
CMPF3	Compare floating-point values	TSTB3	Test bit fields
CMPI3	Compare integers	XOR3	Bitwise-exclusive OR
LSH3	Logical shift		

13.1.4 Program-Control Instructions

The program-control instruction group consists of all of those instructions (17) that affect program flow. The repeat mode allows repetition of a block of code (RPTB) or of a single line of code (RPTS). Both standard and delayed (single-cycle) branching are supported. Several program-control instructions can perform conditional operations. (See Section 13.5 on page 13-28 for detailed information on condition codes.) Table 13–4 lists the program-control instructions.

Table 13–4. Program-Control Instructions

Instruction	Description	Instruction	Description
B*cond*	Branch conditionally (standard)	IDLE	Idle until interrupt
B*cond*D	Branch conditionally (delayed)	NOP	No operation
BR	Branch unconditionally (standard)	RETI*cond*	Return from interrupt conditionally
BRD	Branch unconditionally (delayed)	RETS*cond*	Return from subroutine conditionally
CALL	Call subroutine	RPTB	Repeat block of instructions
CALL*cond*	Call subroutine conditionally	RPTS	Repeat single instruction
DB*cond*	Decrement and branch conditionally (standard)	SWI	Software interrupt
DB*cond*D	Decrement and branch conditionally (delayed)	TRAP*cond*	Trap conditionally
IACK	Interrupt acknowledge		

13.1.5 Low-Power Control Instructions

The low-power control instruction group consists of three instructions that affect the low-power modes. The low-power idle (IDLE2) instruction allows extremely low-power mode. The divide-clock-by-16 (LOPOWER) instruction reduces the rate of the input clock frequency. The restore-clock-to-regular-speed (MAXSPEED) instruction causes the resumption of full-speed operation. Table 13–5 lists the low-power control instructions.

Table 13–5. Low-Power Control Instructions

Instruction	Description	Instruction	Description
IDLE2	Low-power idle	MAXSPEED	Restore clock to regular speed
LOPOWER	Divide clock by 16		

13.1.6 Interlocked-Operations Instructions

The five interlocked-operations instructions (Table 13–6) support multiprocessor communication and the use of external signals to allow for powerful synchronization mechanisms. They also ensure the integrity of the communication and result in a high-speed operation. Refer to Chapter 7 for examples of the use of interlocked instructions.

Table 13–6. Interlocked-Operations Instructions

Instruction	Description	Instruction	Description
LDFI	Load floating-point value, interlocked	STFI	Store floating-point value, interlocked
LDII	Load integer, interlocked	STII	Store integer, interlocked
SIGI	Signal, interlocked		

13.1.7 Parallel-Operations Instructions

The 13 parallel-operations instructions make a high degree of parallelism possible. Some of the ’C3x instructions can occur in pairs that are executed in parallel. These instructions offer the following features:

- ❏ Parallel loading of registers
- ❏ Parallel arithmetic operations
- ❏ Arithmetic/logical instructions used in parallel with a store instruction

Each instruction in a pair is entered as a separate source statement. The second instruction in the pair must be preceded by two vertical bars (||). Table 13–7 lists the valid instruction pairs.

Table 13–7. Parallel Instructions

(a) Parallel arithmetic with store instructions

Mnemonic	Description
ABSF || STF	Absolute value of a floating-point number and store floating-point value
ABSI || STI	Absolute value of an integer and store integer
ADDF3 || STF	Add floating-point values and store floating-point value
ADDI3 || STI	Add integers and store integer
AND3 || STI	Bitwise-logical AND and store integer
ASH3 || STI	Arithmetic shift and store integer
FIX || STI	Convert floating-point to integer and store integer

Table 13–7. Parallel Instructions (Continued)

(a) Parallel arithmetic with store instructions (Continued)

Mnemonic	Description
FLOAT || STF	Convert integer to floating-point value and store floating-point value
LDF || STF	Load floating-point value and store floating-point value
LDI || STI	Load integer and store integer
LSH3 || STI	Logical shift and store integer
MPYF3 || STF	Multiply floating-point values and store floating-point value
MPYI3 || STI	Multiply integer and store integer
NEGF || STF	Negate floating-point value and store floating-point value
NEGI || STI	Negate integer and store integer
NOT || STI	Complement value and store integer
OR3 || STI	Bitwise-logical OR value and store integer
STF || STF	Store floating-point values
STI || STI	Store integers
SUBF3 || STF	Subtract floating-point value and store floating-point value
SUBI3 || STI	Subtract integer and store integer
XOR3 || STI	Bitwise-exclusive OR values and store integer

Table 13–7. Parallel Instructions (Continued)

(b) Parallel load instructions

Mnemonic	Description
LDF || LDF	Load floating-point value
LDI || LDI	Load integer

(c) Parallel multiply and add/subtract instructions

Mnemonic	Description
MPYF3 || ADDF3	Multiply and add floating-point value
MPYF3 || SUBF3	Multiply and subtract floating-point value
MPYI3 || ADDI3	Multiply and add integer
MPYI3 || SUBI3	Multiply and subtract integer

These parallel instructions have been enhanced on the following devices:

- ’C31 silicon revision 6.0 or greater
- ’C32 silicon revision 2.0 or greater

These devices support greater combinations of operands by also allowing the use of any CPU register whenever an indirect operand is required. The particular instruction description details the operand combination.

To support these new modes, you need to invoke the TMS320 floating-point code generation tools (version 5.0 or later) with the following switches:

- C Compiler
  - ‘C31: `CL30 –v31 –gsrev6`
  - ‘C32: `CL30 –v32 –gsrev2`
- Assembler
  - ‘C31: `asm30 –v31 –msrev6`
  - ‘C32: `asm30 –v32 –msrev2`

13.1.8 Illegal Instructions

The ’C3x has no illegal instruction-detection mechanism. Fetching an illegal (undefined) opcode can cause the execution of an undefined operation. Proper use of the TI TMS320 floating-point software tools will not generate an illegal opcode. Only the following conditions can cause the generation of an illegal opcode:

- Misuse of the tools
- An error in the ROM code
- Defective RAM

13.2 Instruction Set Summary

Table 13–8 lists the ’C3x instruction set in alphabetical order. Each table entry provides the instruction mnemonic, description, and operation.

Table 13–8. Instruction Set Summary

Mnemonic	Description	Operation
ABSF	Absolute value of a floating-point number	|*src*| → R*n*
ABSI	Absolute value of an integer	|*src*| → Dreg
ADDC	Add integers with carry	*src* + Dreg + C → Dreg
ADDC3	Add integers with carry (3-operand)	*src*1 + *src*2 + C → Dreg
ADDF	Add floating-point values	*src* + R*n* → R*n*
ADDF3	Add floating-point values (3-operand)	*src*1 + *src*2 → R*n*
ADDI	Add integers	*src* + Dreg → Dreg
ADDI3	Add integers (3 operand)	*src*1 + *src*2 + → Dreg
AND	Bitwise-logical AND	Dreg AND *src* → Dreg
AND3	Bitwise-logical AND (3-operand)	*src*1 AND *src*2 → Dreg
ANDN	Bitwise-logical AND with complement	Dreg AND $\overline{\mathit{src}}$ → Dreg
ANDN3	Bitwise-logical ANDN (3-operand)	*src*1 AND $\overline{\mathit{src}2}$ → Dreg
ASH	Arithmetic shift	If *count* ≥ 0: (Shifted Dreg left by *count*) → Dreg Else: (Shifted Dreg right by |*count*|) → Dreg
ASH3	Arithmetic shift (3-operand)	If *count* ≥ 0: (Shifted *src* left by *count*) → Dreg Else: (Shifted *src* right by |*count*|) → Dreg

Legend:	AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
	C	carry bit	RM	repeat mode bit
	C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
	count	shift value (general addressing modes)	RS	repeat start register
	cond	condition code	SP	stack pointer
	Daddr	destination memory address	Sreg	register address (any register)
	Dreg	register address (any register)	ST	status register
	GIE	global interrupt enable register	*src*	general addressing modes
	N	any trap vector 0–27	*src*1	3-operand addressing modes
	PC	program counter	*src*2	3-operand addressing modes
	RC	repeat counter register	TOS	top of stack

Table 13–8. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
B*cond*	Branch conditionally (standard)	If *cond* = true: If C*src* is a register, C*src* → PC If C*src* is a value, C*src* + PC → PC Else, PC + 1 → PC
B*cond*D	Branch conditionally (delayed)	If *cond* = true: If C*src* is a register, C*src* → PC If C*src* is a value, C*src* + PC + 3 → PC Else, PC + 1 → PC
BR	Branch unconditionally (standard)	Value → PC
BRD	Branch unconditionally (delayed)	Value → PC
CALL	Call subroutine	PC + 1 → TOS Value → PC
CALL*cond*	Call subroutine conditionally	If *cond* = true: PC + 1 → TOS If C*src* is a register, C*src* → PC If C*src* is a value, C*src* + PC → PC Else, PC + 1 → PC
CMPF	Compare floating-point values	Set flags on R*n* – *src*
CMPF3	Compare floating-point values (3-operand)	Set flags on *src*1 – *src*2
CMPI	Compare integers	Set flags on Dreg – *src*
CMPI3	Compare integers (3-operand)	Set flags on *src*1 – *src*2

Legend:	AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
	C	carry bit	RM	repeat mode bit
	C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
	count	shift value (general addressing modes)	RS	repeat start register
	cond	condition code	SP	stack pointer
	Daddr	destination memory address	Sreg	register address (any register)
	Dreg	register address (any register)	ST	status register
	GIE	global interrupt enable register	*src*	general addressing modes
	N	any trap vector 0–27	*src*1	3-operand addressing modes
	PC	program counter	*src*2	3-operand addressing modes
	RC	repeat counter register	TOS	top of stack

Table 13–8. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
DB*cond*	Decrement and branch conditionally (standard)	AR*n* – 1 → AR*n* If *cond* = true and AR*n* ≥ 0: If C*src* is a register, C*src* → PC If C*src* is a value, C*src* + PC + 1 → PC Else, PC + 1 → PC
DB*cond*D	Decrement and branch conditionally (delayed)	AR*n* – 1 → AR*n* If *cond* = true and AR*n* ≥ 0: If C*src* is a register, C*src* → PC If C*src* is a value, C*src* + PC + 3 → PC Else, PC + 1 → PC
FIX	Convert floating-point value to integer	Fix (*src*) → Dreg
FLOAT	Convert integer to floating-point value	Float(*src*) → R*n*
IACK	Interrupt acknowledge	Dummy read of *src* $\overline{\text{IACK}}$ toggled low, then high
IDLE	Idle until interrupt	PC + 1 → PC Idle until next interrupt
IDLE2	Low-power idle	Idle until next interrupt stopping internal clocks
LDE	Load floating-point exponent	*src*(exponent) → R*n*(exponent)
LDF	Load floating-point value	*src* → R*n*
LDF*cond*	Load floating-point value conditionally	If *cond* = true, *src* → R*n* Else, R*n* is not changed
LDFI	Load floating-point value, interlocked	Signal interlocked operation *src* → R*n*
LDI	Load integer	*src* → Dreg

Legend:	AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
	C	carry bit	RM	repeat mode bit
	C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
	count	shift value (general addressing modes)	RS	repeat start register
	cond	condition code	SP	stack pointer
	Daddr	destination memory address	Sreg	register address (any register)
	Dreg	register address (any register)	ST	status register
	GIE	global interrupt enable register	*src*	general addressing modes
	N	any trap vector 0–27	*src*1	3-operand addressing modes
	PC	program counter	*src*2	3-operand addressing modes
	RC	repeat counter register	TOS	top of stack

Table 13–8. Instruction Set Summary (Continued)

Mnemonic	**Description**	**Operation**
LDI*cond*	Load integer conditionally	If *cond* = true, *src* → Dreg Else, Dreg is not changed
LDII	Load integer, interlocked	Signal interlocked operation *src* → Dreg
LDM	Load floating-point mantissa	*src* (mantissa) → R*n* (mantissa)
LDP	Load data page pointer	*src* → data page pointer
LOPOWER	Divide clock by 16	H1/16 → H1
LSH	Logical shift	If *count* ≥ 0: (Dreg left-shifted by *count*) → Dreg Else: (Dreg right-shifted by |*count*|) → Dreg
LSH3	Logical shift (3-operand)	If *count* ≥ 0: (*src* left-shifted by *count*) → Dreg Else: (*src* right-shifted by |*count*|) → Dreg
MAXSPEED	Restore clock to regular speed	H1/16 → H1
MPYF	Multiply floating-point values	*src* × R*n* → R*n*
MPYF3	Multiply floating-point value (3-operand)	*src*1 × *src*2 → R*n*
MPYI	Multiply integers	*src* × Dreg → Dreg
MPYI3	Multiply integers (3-operand)	*src*1 × *src*2 → Dreg
NEGB	Negate integer with borrow	0 – *src* – C → Dreg
NEGF	Negate floating-point value	0 – *src* → R*n*
NEGI	Negate integer	0 – *src* → Dreg

Legend:	AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
	C	carry bit	RM	repeat mode bit
	C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
	count	shift value (general addressing modes)	RS	repeat start register
	cond	condition code	SP	stack pointer
	Daddr	destination memory address	Sreg	register address (any register)
	Dreg	register address (any register)	ST	status register
	GIE	global interrupt enable register	*src*	general addressing modes
	N	any trap vector 0–27	*src*1	3-operand addressing modes
	PC	program counter	*src*2	3-operand addressing modes
	RC	repeat counter register	TOS	top of stack

Table 13–8. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
NOP	No operation	Modify AR*n* if specified
NORM	Normalize floating-point value	Normalize (*src*) → R*n*
NOT	Bitwise-logical complement	$\overline{src}$ → Dreg
OR	Bitwise-logical OR	Dreg OR *src* → Dreg
OR3	Bitwise-logical OR (3-operand)	*src*1 OR *src*2 → Dreg
POP	Pop integer from stack	*SP– – → Dreg
POPF	Pop floating-point value from stack	*SP– – → R*n*
PUSH	Push integer on stack	Sreg → *++ SP
PUSHF	Push floating-point value on stack	R*n* → *++ SP
RETI*cond*	Return from interrupt conditionally	If *cond* = true or missing: *SP– – → PC 1 → ST (GIE) Else, continue
RETS*cond*	Return from subroutine conditionally	If *cond* = true or missing: *SP– – → PC Else, continue
RND	Round floating-point value	Round (*src*) → R*n*
ROL	Rotate left	Dreg rotated left 1 bit → Dreg
ROLC	Rotate left through carry	Dreg rotated left 1 bit through carry → Dreg
ROR	Rotate right	Dreg rotated right 1 bit → Dreg
RORC	Rotate right through carry	Dreg rotated right 1 bit through carry → Dreg

Legend:	AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
	C	carry bit	RM	repeat mode bit
	C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
	count	shift value (general addressing modes)	RS	repeat start register
	cond	condition code	SP	stack pointer
	Daddr	destination memory address	Sreg	register address (any register)
	Dreg	register address (any register)	ST	status register
	GIE	global interrupt enable register	*src*	general addressing modes
	N	any trap vector 0–27	*src*1	3-operand addressing modes
	PC	program counter	*src*2	3-operand addressing modes
	RC	repeat counter register	TOS	top of stack

Table 13–8. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
RPTB	Repeat block of instructions	*src* → RE 1 → ST (RM) Next PC → RS
RPTS	Repeat single instruction	*src* → RC 1 → ST (RM) Next PC → RS Next PC → RE
SIGI	Signal, interlocked	Signal interlocked operation Wait for interlock acknowledge Clear interlock
STF	Store floating-point value	R*n* → Daddr
STFI	Store floating-point value, interlocked	R*n* → Daddr Signal end of interlocked operation
STI	Store integer	Sreg → Daddr
STII	Store integer, interlocked	Sreg → Daddr Signal end of interlocked operation
SUBB	Subtract integers with borrow	Dreg – *src* – C → Dreg
SUBB3	Subtract integers with borrow (3-operand)	*src*1 – *src*2 – C → Dreg
SUBC	Subtract integers conditionally	If Dreg – *src* ≥ 0: [(Dreg – *src*) << 1] OR 1 → Dreg Else, Dreg << 1 → Dreg
SUBF	Subtract floating-point values	R*n* – *src* → R*n*
SUBF3	Subtract floating-point values (3-operand)	*src*1 – *src*2 → R*n*

Legend:	AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
	C	carry bit	RM	repeat mode bit
	C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
	count	shift value (general addressing modes)	RS	repeat start register
	cond	condition code	SP	stack pointer
	Daddr	destination memory address	Sreg	register address (any register)
	Dreg	register address (any register)	ST	status register
	GIE	global interrupt enable register	*src*	general addressing modes
	N	any trap vector 0–27	*src*1	3-operand addressing modes
	PC	program counter	*src*2	3-operand addressing modes
	RC	repeat counter register	TOS	top of stack

Table 13–8. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
SUBI	Subtract integers	Dreg – *src* → Dreg
SUBI3	Subtract integers (3-operand)	*src*1 – *src*2 → Dreg
SUBRB	Subtract reverse integer with borrow	*src* – Dreg – C → Dreg
SUBRF	Subtract reverse floating-point value	*src* – R*n* → R*n*
SUBRI	Subtract reverse integer	*src* – Dreg → Dreg
SWI	Software interrupt	Perform emulator interrupt sequence
TRAP*cond*	Trap conditionally	If *cond* = true or missing: Next PC → * ++ SP Trap vector N → PC 0 → ST (GIE) Else, continue
TSTB	Test bit fields	Dreg AND *src*
TSTB3	Test bit fields (3-operand)	*src*1 AND *src*2
XOR	Bitwise-exclusive OR	Dreg XOR *src* → Dreg
XOR3	Bitwise-exclusive OR (3-operand)	*src*1 XOR *src*2 → Dreg

Legend:

AR*n*	auxiliary register *n* (AR7–AR0)	RE	repeat interrupt register
C	carry bit	RM	repeat mode bit
C*src*	conditional-branch addressing modes	R*n*	register address (R7–R0)
count	shift value (general addressing modes)	RS	repeat start register
cond	condition code	SP	stack pointer
Daddr	destination memory address	Sreg	register address (any register)
Dreg	register address (any register)	ST	status register
GIE	global interrupt enable register	*src*	general addressing modes
N	any trap vector 0–27	*src*1	3-operand addressing modes
PC	program counter	*src*2	3-operand addressing modes
RC	repeat counter register	TOS	top of stack

13.3 Parallel Instruction Set Summary

Table 13–9 lists the ’C3x instruction set in alphabetical order. Each table entry shows the instruction mnemonic, description, and operation. Refer to Section 13.1 for a functional listing of the instructions and individual instruction descriptions.

Table 13–9. Parallel Instruction Set Summary

(a) Parallel arithmetic with store instructions

Mnemonic	Description	Operation
ABSF	Absolute value of a floating point	|*src*2| → *dst*1
|| STF		|| *src*3 → *dst*2
ABSI	Absolute value of an integer	|*src*2| → *dst*1
|| STI		|| *src*3 → *dst*2
ADDF3	Add floating-point value	*src*1 + *src*2 → *dst*1
|| STF		|| *src*3 → *dst*2
ADDI3	Add integer	*src*1 + *src*2 → *dst*1
|| STI		|| *src*3 → *dst*2
AND3	Bitwise-logical AND	*src*1 AND *src*2 → *dst*1
|| STI		|| *src*3 → *dst*2
ASH3	Arithmetic shift	If *count* ≥ 0:
|| STI		(*src*2 << *count*) → *dst*1
		|| *src*3 → *dst*2
		Else:
		(*src*2 >> |*count*|) → *dst*1
		|| *src*3 → *dst*2
FIX	Convert floating-point value to integer	Fix (*src*2) → *dst*1
|| STI		|| *src*3 → *dst*2
FLOAT	Convert integer to floating-point value	Float(*src*2) → *dst*1
|| STF		|| *src*3 → *dst*2

Legend:

count	register addr (R7–R0)
*dst*1	register addr (R7–R0)
*dst*2	indirect addr (*disp* = 0, 1, IR0, IR1)
op1, op2, op4, and op5	Any two of these operands must be specified using register addr; the remaining two must be specified using indirect.
op3	register addr (R0 or R1)
op6	register addr (R2 or R3)
*src*1	register addr (R7–R0)
*src*2	indirect addr (*disp* = 0, 1, IR0, IR1)
*src*3	register addr (R7–R0)

Table 13–9. Parallel Instruction Set Summary (Continued)

(a) Parallel arithmetic with store instructions (Continued)

<table>
<tr><th></th><th>Mnemonic</th><th>Description</th><th></th><th>Operation</th></tr>
<tr><td></td><td>LDF</td><td>Load floating-point value</td><td></td><td><i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STF</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>LDI</td><td>Load integer</td><td></td><td><i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STI</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>LSH3</td><td>Logical shift</td><td colspan="2">If <i>count</i> ≥ 0:</td></tr>
<tr><td>||</td><td>STI</td><td></td><td></td><td><i>src</i>2 &lt;&lt; <i>count</i> → <i>dst</i>1</td></tr>
<tr><td></td><td></td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td></td><td></td><td colspan="2">Else:</td></tr>
<tr><td></td><td></td><td></td><td></td><td><i>src</i>2 &gt;&gt; |<i>count</i>| → <i>dst</i>1</td></tr>
<tr><td></td><td></td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>MPYF3</td><td>Multiply floating-point value</td><td></td><td><i>src</i>1 x <i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STF</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>MPYI3</td><td>Multiply integer</td><td></td><td><i>src</i>1 x <i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STI</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>NEGF</td><td>Negate floating-point value</td><td></td><td>0 – <i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STF</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>NEGI</td><td>Negate integer</td><td></td><td>0 – <i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STI</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>NOT</td><td>Complement</td><td></td><td>$\overline{\mathit{src}1} \rightarrow \mathit{dst}1$</td></tr>
<tr><td>||</td><td>STI</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>OR3</td><td>Bitwise-logical OR</td><td></td><td><i>src</i>1 OR <i>src</i>2 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STI</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>STF</td><td>Store floating-point value</td><td></td><td><i>src</i>1 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STF</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
<tr><td></td><td>STI</td><td>Store integer</td><td></td><td><i>src</i>1 → <i>dst</i>1</td></tr>
<tr><td>||</td><td>STI</td><td></td><td>||</td><td><i>src</i>3 → <i>dst</i>2</td></tr>
</table>

Legend:	*count*	register addr (R7–R0)	op3	register addr (R0 or R1)
	*dst*1	register addr (R7–R0)	op6	register addr (R2 or R3)
	*dst*2	indirect addr (*disp* = 0, 1, IR0, IR1)	*src*1	register addr (R7–R0)
	op1, op2, op4, and op5		*src*2	indirect addr (*disp* = 0, 1, IR0, IR1)
		Any two of these operands must be specified using register addr; the remaining two must be specified using indirect.	*src*3	register addr (R7–R0)

Table 13–9. Parallel Instruction Set Summary (Continued)

(a) Parallel arithmetic with store instructions (Continued)

Mnemonic	Description	Operation
SUBF3	Subtract floating-point value	*src*1 – *src*2 → *dst*1
|| STF		|| *src*3 → *dst*2
SUBI3	Subtract integer	*src*1 – *src*2 → *dst*1
|| STI		|| *src*3 → *dst*2
XOR3	Bitwise-exclusive OR	*src*1 XOR *src*2 → *dst*1
|| STI		|| *src*3 → *dst*2

(b) Parallel load instructions

Mnemonic	Description	Operation
LDF	Load floating-point value	*src*2 → *dst*1
|| LDF		|| *src*4 → *dst*2
LDI	Load integer	*src*2 → *dst*1
|| LDI		|| *src*4 → *dst*2

(c) Parallel multiply and add/subtract instructions

Mnemonic	Description	Operation
MPYF3	Multiply and add floating-point value	op1 x op2 → op3
|| ADDF3		|| op4 + op5 → op6
MPYF3	Multiply and subtract floating-point value	op1 x op2 → op3
|| SUBF3		|| op4 – op5 → op6
MPYI3	Multiply and add integer	op1 x op2 → op3
|| ADDI3		|| op4 + op5 → op6
MPYI3	Multiply and subtract integer	op1 x op2 → op3
|| SUBI3		|| op4 – op5 → op6

Legend:

count	register addr (R7–R0)	op3	register addr (R0 or R1)
*dst*1	register addr (R7–R0)	op6	register addr (R2 or R3)
*dst*2	indirect addr (*disp* = 0, 1, IR0, IR1)	*src*1	register addr (R7–R0)
op1, op2, op4, and op5		*src*2	indirect addr (*disp* = 0, 1, IR0, IR1)
	Any two of these operands must be specified using register addr; the remaining two must be specified using indirect.	*src*3	register addr (R7–R0)

13.4 Group Addressing Mode Instruction Encoding

The six addressing types (covered in Section 6.1, *Addressing Types*, on page 6-2) form these four groups of addressing modes:

- ❑ General addressing modes (G)
- ❑ 3-operand addressing modes (T)
- ❑ Parallel addressing modes (P)
- ❑ Conditional-branch addressing modes (B)

13.4.1 General Addressing Modes

Instructions that use the general addressing modes are general-purpose instructions, such as ADDI, MPYF, and LSH. Such instructions usually have this form:

dst operation *src* → *dst*

In the syntax, the destination operand is signified by *dst* and the source operand by *src*; *operation* defines an operation to be performed on the operands using the general addressing modes. Bits 31–29 are 0, indicating general addressing mode instructions. Bits 22 and 21 specify the general addressing mode (G) field, which defines how bits 15–0 are to be interpreted for addressing the *src* operand.

Options for bits 22 and 21 (G field) are as follows:

G	Mode
0 0	Register (all CPU registers unless specified otherwise)
0 1	Direct
1 0	Indirect
1 1	Immediate

If the *src* and *dst* fields contain register specifications, the value in these fields contains the CPU register addresses as defined by Table 13–10. For the general addressing modes, the following values of AR*n* are valid:

AR*n*, 0 $\le n \le$ 7

Figure 13–1 shows the encoding for the general addressing modes. The notation mod*n* indicates the modification field that goes with the AR*n* field. Refer to Table 13–10 on page 13-22 for further information.

Figure 13–1. Encoding for General Addressing Modes

		G	**Destination**	**Source Operands**			
31 29	**28 23**	**22 21**	**20 16**	**15 11**	**10 8**	**7 5**	**4 0**
0 0 0	operation	0 0	*dst*	0 0 0 0 0 0 0 0 0 0 0			*src*
0 0 0	operation	0 1	*dst*	direct			
0 0 0	operation	1 0	*dst*	mod*n*	AR*n*	*disp*	
0 0 0	operation	1 1	*dst*	immediate			

Table 13–10. Indirect Addressing

(a) Indirect addressing with displacement

Mod Field	Syntax	Operation	Description
00000	*+AR*n*(*disp*)	addr = AR*n* + *disp*	With predisplacement add
00001	*–AR*n*(*disp*)	addr = AR*n* – *disp*	With predisplacement subtract
00010	*++AR*n*(*disp*)	addr = AR*n* + *disp* AR*n* = AR*n* + *disp*	With predisplacement add and modify
00011	*––AR*n*(*disp*)	addr = AR*n* – *disp* AR*n* = AR*n* – *disp*	With predisplacement subtract and modify
00100	*AR*n*++(*disp*)	addr = AR*n* AR*n* = AR*n* + *disp*	With postdisplacement add and modify
00101	*AR*n*––(*disp*)	addr = AR*n* AR*n* = AR*n* – *disp*	With postdisplacement subtract and modify
00110	*AR*n*++(*disp*)%	addr = AR*n* AR*n* = circ(AR*n* + *disp*)	With postdisplacement add and circular modify
00111	*AR*n*––(*disp*)%	addr = AR*n* AR*n* = circ(AR*n* – *disp*)	With postdisplacement subtract and circular modify

(b) Indirect addressing with index register IR0

Mod Field	Syntax	Operation	Description
01000	*+AR*n*(IR0)	addr = AR*n* + IR0	With preindex (IR0) add
01001	*–AR*n*(IR0)	addr = AR*n* – IR0	With preindex (IR0) subtract
01010	*++AR*n*(IR0)	addr = AR*n* + IR0 AR*n* = AR*n* + IR0	With preindex (IR0) add and modify
01011	*––AR*n*(IR0)	addr = AR*n* – IR0 AR*n* = AR*n* – IR0	With preindex (IR0) subtract and modify
01100	*AR*n*++(IR0)	addr = AR*n* AR*n* = AR*n* + IR0	With postindex (IR0) add and modify
01101	*AR*n*––(IR0)	addr= AR*n* AR*n* = AR*n* – IR0	With postindex (IR0) subtract and modify
01110	*AR*n*++(IR0)%	addr = AR*n* AR*n* = circ(AR*n* + IR0)	With postindex (IR0) add and circular modify
01111	*AR*n*––(IR0)%	addr = AR*n* AR*n* = circ(AR*n*– IR0)	With postindex (IR0) subtract and circular modify

Legend:	addr	memory address	++	add and modify
	AR*n*	auxiliary registers AR0–AR7	––	subtract and modify
	circ( )	address in circular addressing	%	where circular addressing is performed
	disp	displacement	IR*n*	index register IR0 or IR1

Table 13–10. Indirect Addressing (Continued)

(c) Indirect addressing with index register IR1

Mod Field	Syntax	Operation	Description
10000	*+ARn(IR1)	addr = ARn + IR1	With preindex (IR1) add
10001	*–ARn(IR1)	addr = ARn – IR1	With preindex (IR1) subtract

10010	*++ARn(IR1)	addr = ARn + IR1
		ARn = ARn + IR1
			With preindex (IR1) add and modify

10011	*––ARn(IR1)	addr = ARn – IR1
		ARn = ARn – IR1
			With preindex (IR1) subtract and modify

10100	*ARn++(IR1)	addr = ARn
		ARn = ARn + IR1
			With postindex (IR1) add and modify

10101	*ARn––(IR1)	addr = ARn
		ARn = ARn – IR1
			With postindex (IR1) subtract and modify

10110	*ARn++(IR1)%	addr = ARn
		ARn = circ(ARn + IR1)
			With postindex (IR1) add and circular modify

10111	*ARn––(IR1)%	addr = ARn
		ARn = circ(ARn – IR1)
			With postindex (IR1) subtract and circular modify

(d) Indirect addressing (special cases)

Mod Field	Syntax	Operation	Description
11000	*ARn	addr = ARn	Indirect

11001	*ARn++(IR0)B	addr = ARn
		ARn = B(ARn + IR0)
			With postindex (IR0) add
			and bit-reversed modify

Legend:	addr	memory address	circ( )	address in circular addressing
	ARn	auxiliary registers AR0–AR7	++	add and modify by one
	B	where bit-reversed addressing is performed	%	where circular addressing is performed
	B()	bit-reversed address	IRn	index register IR0 or IR1
			––	subtract and modify by one

13.4.2 3-Operand Addressing Modes

Instructions that use the 3-operand addressing modes, such as ADDI3, LSH3, CMPF3, or XOR3, usually have this form:

*src*1 operation *src*2 → *dst*

where the destination operand is signified by *dst* and the source operands by *src*1 and *src*2; *operation* defines an operation to be performed.

Note:

The *3* can be omitted from a 3-operand instruction mnemonic.

Bits 31–29 are set to the value of 001, indicating 3-operand addressing mode instructions. Bits 22 and 21 specify the 3-operand addressing mode (T) field, which defines how bits 15–0 are to be interpreted for addressing the SRC operands. Bits 15–8 define the SRC1 address; bits 7–0 define the SRC2 address. Options for bits 22 and 21 (T) are as follows:

T	***src*1 addressing modes**	***src*2 addressing modes**
0 0	Register mode (any CPU register)	Register mode (any CPU register)
0 1	Indirect mode (*disp* = 0, 1, IR0, IR1)	Register mode (any CPU register)
1 0	Register mode (any CPU register)	Indirect mode (*disp* = 0, 1, IR0, IR1)
1 1	Indirect mode (*disp* = 0, 1, IR0, IR1)	Indirect mode (*disp* = 0, 1, IR0, IR1)

Figure 13–2 shows the encoding for 3-operand addressing. If the *src*1 and *src*2 fields both modify the same auxiliary register, both addresses are correctly generated. However, only the value created by the *src*1 field is saved in the auxiliary register specified. The assembler issues a warning if you specify this condition.

The following values of AR*n* and AR*m* are valid:

AR*n*,0 ≤ *n* ≤ 7
AR*m*,0 ≤ *m* ≤ 7

The notation *modm* or *modn* indicates the modification field that goes with the AR*m* or AR*n* field, respectively. Refer to Table 13–10 on page 13-22 for further information.

In indirect addressing of the 3-operand addressing mode, displacements (if used) are allowed to be 0 or 1, and the index registers (IR0 and IR1) can be used. The displacement of 1 is implied and is not explicitly coded in the instruction word.

Figure 13–2. Encoding for 3-Operand Addressing Modes

		T	**Destination**	***src*1**			***src*2**		
31 28	**27 23**	**22 21**	**20 16**	**15 13**	**12 11**	**10 8**	**7 5**	**4 3**	**2 0**
0 0 1	operation	0 0	*dst*	0 0 0	*src*1		0 0 0	*src*2	
0 0 1	operation	0 1	*dst*	mod*n*		AR*n*	0 0 0	*src*2	
0 0 1	operation	1 0	*dst*	0 0 0	*src*1		mod*n*		AR*n*
0 0 1	operation	1 1	*dst*	mod*n*		AR*n*	mod*m*		AR*m*

13.4.3 Parallel Addressing Modes

Instructions that use parallel addressing, indicated by **||** (two vertical bars), allow the most parallelism possible. The destination operands are indicated as d1 and d2, signifying *dst*1 and *dst*2, respectively (see Figure 13–3). The source operands, signified by *src*1 and *src*2, use the extended-precision registers. *Operation* refers to the parallel operation to be performed.

Figure 13–3. Encoding for Parallel Addressing Modes

31 30	**29 26**	**25 24**	**23**	**22**	**21 19**	**18 16**	**15 11**	**10 8**	**7 3**	**2 0**
1 0	operation	P	d1	d2	*src*1	*src*2	mod*n*	AR*n*	mod*m*	AR*m*

The parallel addressing mode (P) field specifies how the operands are to be used, that is, whether they are source or destination. The specific relationship between the P field and the operands is detailed in the description of the individual parallel instructions. However, the operands are always encoded in the same way. Bits 31 and 30 are set to the value of 10, indicating parallel addressing mode instructions. Bits 25 and 24 specify the parallel addressing mode (P) field, which defines how to interpret bits 21–0 for addressing the *src* operands. Bits 21–19 define the *src*1 address, bits 18–16 define the *src*2 address, bits 15–8 the *src*3 address, and bits 7–0 the *src* 4 address. The notations mod*n* and mod*m* indicate which modification field goes with which AR*n* or AR*m* (auxiliary register) field, respectively. The following list describes the parallel addressing operands:

*src*1 = R*n*	$0 \le n \le 7$ (extended-precision registers R0 – R7)
*src*2 = R*n*	$0 \le n \le 7$ (extended-precision registers R0–R7)
d1	If 0, *dst*1 is R0. If 1, *dst*1 is R1.
d2	If 0, *dst*2 is R2. If 1, *dst*2 is R3.
P	$0 \le P \le 3$
*src*3	indirect (*disp* = 0, 1, IR0, IR1)
*src*4	indirect (*disp* = 0, 1, IR0, IR1)

As in the 3-operand addressing mode, indirect addressing in the parallel addressing mode allows for displacements of 0 or 1 and the use of the index registers (IR0 and IR1). The displacement of 1 is implied and is not explicitly coded in the instruction word.

In the encoding shown for this mode in Figure 13–3, if the *src*3 and *src*4 fields use the same auxiliary register, both addresses are correctly generated, but only the value created by the *src*3 field is saved in the auxiliary register specified. The assembler issues a warning if you specify this condition.

The encoding of these parallel addressing modes has been extended in the following devices:

- ’C31 silicon revision 6.0 or greater
- ’C32 silicon revision 2.0 or greater

These addressing mode extensions also allow the use of any CPU register whenever an indirect operand is required in *src3* and/or *src4* operand. Figure 13–4 shows the encoding for extended parallel addressing instructions.

Figure 13–4. Encoding for Extended Parallel Addressing Instructions

31 30 29	**28 26**	**25 24**	**23**	**22**	**21 19**	**18 16**	**15 13**	**12 8**	**9 5**	**4 0**
1 0	operation	P	d1	d2	*src*1	*src*2	1 1 1	*src*3	1 1 1	*src*4

13.4.4 Conditional-Branch Addressing Modes

Instructions using the conditional-branch addressing modes (B*cond*, B*cond*D, CALL*cond*, DB*cond*, and DB*cond*D) can perform a variety of conditional operations. Bits 31–27 are set to the value of 01101, indicating conditional-branch addressing mode instructions. Bit 26 is set to 0 or 1; 0 selects DB*cond*, 1 selects B*cond*. Selection of bit 25 determines the conditional-branch addressing mode (B). If B = 0, register addressing is used; if B = 1, PC-relative addressing is used. Bit 21 sets the type of branch: D = 0 for a standard branch or D = 1 for a delayed branch. The condition field (*cond*) specifies the condition checked to determine what action to take, that is, whether to branch (see Table 13–12 on page 13-30 for a list of condition codes). Figure 13–5 shows the encoding for conditional-branch addressing.

Figure 13–5. Encoding for Conditional-Branch Addressing Modes

*(a) DB*cond *(D)*

31 26	**25**	**24 22**	**21**	**20 16**	**15 5**	**4 0**
0 1 1 0 1 1	B	AR*n*	D	*cond*	0 0 0 0 0 0 0 0 0 0 0	*src* reg
0 1 1 0 1 1	B	AR*n*	D	*cond*	immediate (PC relative)	

*(b) B*cond *(D)*

31 26	**25**	**24 22**	**21**	**20 16**	**15 5**	**4 0**
0 1 1 0 1 0	B	0 0 0	D	*cond*	0 0 0 0 0 0 0 0 0 0 0	*src* reg
0 1 1 0 1 0	B	0 0 0	D	*cond*	immediate (PC relative)	

*(c) CALL*cond

31 26	**25**	**24 22**	**21**	**20 16**	**15 5**	**4 0**
0 1 1 1 0 0	B	0 0 0	0	*cond*	0 0 0 0 0 0 0 0 0 0 0	*src* reg
0 1 1 1 0 0	B	0 0 0	0	*cond*	immediate (PC relative)	

13.5 Condition Codes and Flags

The ’C3x provides 20 condition codes (00000–10100, excluding 01011) that you can place in the *cond* field of any of the conditional instructions, such as RETS*cond* or LDF*cond*. The conditions include signed and unsigned comparisons, comparisons to 0, and comparisons based on the status of individual condition flags. All conditional instructions can accept the suffix U to indicate unconditional operation.

Seven condition flags provide information about properties of the result of arithmetic and logical instructions. The condition flags are stored in the status register (ST) and are affected by an instruction only when either of the following two cases occurs:

❑ The destination register is one of the extended-precision registers (R7–R0). (This allows for modification of the registers used for addressing but does not affect the condition flags during computation.)

❑ The instruction is one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3). (This makes it possible to set the condition flags according to the contents of any of the CPU registers.)

The condition flags are modified by most instructions when either of the preceding conditions is established and either of the following two cases occurs:

❑ A result is generated when the specified operation is performed to infinite precision. This is appropriate for compare and test instructions that do not store results in a register. It is also appropriate for arithmetic instructions that produce underflow or overflow.

❑ The output is written to the destination register, as shown in Table 13–11. This is appropriate for other instructions that modify the condition flags.

Table 13–11. Output Value Formats

Type of Operation	Output Format
Floating point	8-bit exponent, one sign bit, 31-bit fraction
Integer	32-bit integer
Logical	32-bit unsigned integer

Figure 13–6 shows the condition flags in the low-order bits of the status register. Following the figure is a list of status register condition flags with a description of how the flags are set by most instructions. For specific details of the effect of a particular instruction on the condition flags, see the description of that instruction in Section 13.6, *Individual Instruction Descriptions,* on page 13-32.

Figure 13–6. Status Register

13 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	PRGW status (’C32 only)	INT config (’C32 only)	GIE	CC	CE	CF	xx	RM	OVM	LUF	LV	UF	N	Z	V	C
	R	R/W	R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Note: xx = reserved bit, read as 0
R = read, W = write

LUF **Latched floating-point underflow condition flag.** LUF is set whenever UF (floating-point underflow flag) is set. LUF can be cleared only by a processor reset or by modifying it in the status register (ST).

LV **Latched overflow conditionfFlag.** LV is set whenever V (overflow condition flag) is set. Otherwise, it is unchanged. LV can be cleared only by a processor reset or by modifying it in the status register (ST).

UF **Floating-point underflow conditionflag.** A floating-point underflow occurs whenever the exponent of the result is less than or equal to –128. If a floating-point underflow occurs, UF is set, and the output value is set to 0. UF is cleared if a floating-point underflow does not occur.

N **Negative condition flag.** Logical operations assign N the state of the MSB of the output value. For logical operations, V is set to the state of the MSB. For integer and floating-point operations, N is set if the result is negative and cleared otherwise. A 0 is positive.

Z **Zero condition flag.** For logical, integer, and floating-point operations, Z is set if the output is 0 and cleared otherwise.

V **Overflow condition flag.** For integer operations, V is set if the result does not fit into the format specified for the destination (that is, $-2^{32} \le$ result $\le 2^{32} - 1$). Otherwise, V is cleared. For floating-point operations, V is set if the exponent of the result is greater than 127; otherwise,V is cleared. Logical operations always clear V.

C **Carry flag.** When an integer addition is performed, C is set if a carry occurs out of the bit corresponding to the MSB of the output. When an integer subtraction is performed, C is set if a borrow occurs into the bit corresponding to the MSB of the output. Otherwise, for integer operations, C is cleared. The carry flag is unaffected by floating-point and logical operations. For shift instructions, this flag is set to the last bit shifted out; for a 0 shift *count*, this is set to 0.

Table 13–12 lists the condition mnemonic, code, description, and flag for each of the 20 condition codes.

Table 13–12. Condition Codes and Flags

(a) Unconditional compares

Condition	Code	Description	Flag[†]
U	00000	Unconditional	Irrevelant

(b) Unsigned compares

Condition	Code	Description	Flag[†]
LO	00001	Lower than	C
LS	00010	Lower than or same as	C OR Z
HI	00011	Higher than	~C AND ~Z
HS	00100	Higher than or same as	~C
EQ	00101	Equal to	Z
NE	00110	Not equal to	~Z

(c) Signed compares

Condition	Code	Description	Flag[†]
LT	00111	Less than	N
LE	01000	Less than or equal to	N OR Z
GT	01001	Greater than	~N AND ~Z
GE	01010	Greater than or equal to	~N
EQ	00101	Equal to	Z
NE	00110	Not equal to	~Z

[†] ~ = logical complement (not true condition)

Table 13–12. Condition Codes and Flags (Continued)

(d) Compare to zero

Condition	Code	Description	Flag[†]
Z	00101	Zero	Z
NZ	00110	Not zero	~Z
P	01001	Positive	~N AND ~Z
N	00111	Negative	N
NN	01010	Non-negative	~N

(e) Compare to condition flags

Condition	Code	Description	Flag[†]
NN	01010	Non-negative	~N
N	00111	Negative	N
NZ	00110	Nonzero	~Z
Z	00101	Zero	Z
NV	01100	No overflow	~V
V	01101	Overflow	V
NUF	01110	No underflow	~UF
UF	01111	Underflow	UF
NC	00100	No carry	~C
C	00001	Carry	C
NLV	10000	No latched overflow	~LV
LV	10001	Latched overflow	LV
NLUF	10010	No latched floating-point underflow	~LUF
LUF	10011	Latched floating-point underflow	LUF
ZUF	10100	Zero or floating-point underflow	Z OR UF

[†] ~ = logical complement (not true condition)

13.6 Individual Instructions

This section contains the individual assembly language instructions for the ’C3x. The instructions are listed in alphabetical order. Information for each instruction includes assembler syntax, operation, operands, encoding, description, cycles, status bits, mode bit, and examples.

Definitions of the symbols and abbreviations, as well as optional syntax forms allowed by the assembler, precede the individual instruction description section. Also, an example instruction shows the special format used and explains its content.

A functional grouping of the instructions, as well as a complete instruction set summary, can be found in Section 13.1 on page 13-2. Appendix A lists the opcodes for all of the instructions. Refer to Chapter 6 for information on memory addressing. Code examples using many of the instructions are provided in Chapter 1, *Software Applications*, of the *TMS320C3x General-Purpose Applications User’s Guide*.

13.6.1 Symbols and Abbreviations

Table 13–13 lists the symbols and abbreviations used in the individual instruction descriptions.

Table 13–13. Instruction Symbols

Symbol	Meaning
src	Source operand
*src*1	Source operand 1
*src*2	Source operand 2
*src*3	Source operand 3
*src*4	Source operand 4
dst	Destination operand
*dst*1	Destination operand 1
*dst*2	Destination operand 2
disp	Displacement
cond	Condition
count	Shift count
G	General addressing modes
T	3-operand addressing modes
P	Parallel addressing modes
B	Conditional-branch addressing modes
|*x*|	Absolute value of x
x → *y*	Assign the value of x to destination y
x(man)	Mantissa field (sign + fraction) of x
x(exp)	Exponent field of x
op1 || *op2*	Operation 1 performed in parallel with operation 2
x AND *y*	Bitwise-logical AND of x and y
x OR *y*	Bitwise-logical OR of x and y
x XOR *y*	Bitwise-logical XOR of x and y
~*x*	Bitwise-logical complement of x
x << *y*	Shift x to the left y bits
x >> *y*	Shift x to the right y bits
*++SP	Increment SP and use incremented SP as address
*SP– –	Use SP as address and decrement SP
AR*n*	Auxiliary register *n*
IR*n*	Index register *n*
R*n*	Register address *n*
RC	Repeat count register
RE	Repeat end address register
RS	Repeat start address register
ST	Status register
C	Carry bit
GIE	Global interrupt enable bit
N	Trap vector
PC	Program counter
RM	Repeat mode flag
SP	System stack pointer

13.6.2 Optional Assembler Syntax

The assembler allows a relaxed syntax form for some instructions. These optional forms simplify the assembly language so that special-case syntax can be ignored. A list of the optional syntax forms follows.

- You can omit the destination register on unary arithmetic and logical operations when the same register is used as a source. For example,

  `ABSI  R0,R0` *can be written as* `ABSI R0`

  Instructions affected: ABSI, ABSF, FIX, FLOAT, NEGB, NEGF, NEGI, NORM, NOT, RND

- You can write all 3-operand instructions without the 3. For example,

  `ADDI3 R0,R1,R2` *can be written as* `ADDI R0,R1,R2`

  Instructions affected: ADDC3, ADDF3, ADDI3, AND3, ANDN3, ASH3, LSH3, MPYF3, MPYI3, OR3, SUBB3, SUBF3, SUBI3, XOR3

  This also applies to all of the pertinent parallel instructions.

- You can write all 3-operand comparison instructions without the 3. For example,

  `CMPI3 R0,*AR0` *can be written as* `CMPI R0,*AR0`

  Instructions affected: CMPI3, CMPF3, TSTB3

- Indirect operands with an explicit 0 displacement are allowed. In 3-operand or parallel instructions, operands with 0 displacement are automatically converted to no-displacement mode. For example:

  `LDI   *+AR0(0),R1` is legal.

  Also

  `ADDI3 *+AR0(0),R1,R2` *is equivalent to* `ADDI3 *AR0,R1,R2`

- You can write indirect operands with no displacement, in which case a displacement of 1 is assumed. For example,

  `LDI *AR0++(1),R0` *can be written as* `LDI *AR0++,R0`

- All conditional instructions accept the suffix U to indicate unconditional operation. Also, you can omit the U from unconditional short branch instructions. For example:

  `BU label` *can be written as* `B label`

- You can write labels with or without a trailing colon. For example:

  `label0:     NOP`
  `label1      NOP`
  `label2:` *label assembles to next source line*

- Empty expressions are not allowed for the displacement in indirect mode:

  ```
  LDI   *+AR0(),R0         is not legal.
  ```

- You can precede long immediate mode operands (destination of BR and CALL) with an @ sign:

  ```
  BR label             can be written as          BR @label
  ```

- You can use the LDP pseudo-op to load a register (usually DP) with the eight most significant bits (MSBs) of a relocatable address:

  ```
  LDP   addr,REG        or               LDP @addr,REG
  ```

  The @ sign is optional.

  If the destination register is the DP, you can omit the DP in the operand. LDP generates an LDI instruction with an immediate operand and a special relocation type.

- You can write parallel instructions in either order. For example:

  ```
  ADDI                 can be written as                  STI
  || STI                                              || ADDI
  ```

- You can write the parallel bars indicating part 2 of a parallel instruction anywhere on the line from column 0 to the mnemonic. For example:

  ```
  ADDI                 can be written as                 ADDI
  || STI                                                || STI
  ```

- If the second operand of a parallel instruction is the same as the third (destination register) operand, you can omit the third operand. This allows you to write 3-operand parallel instructions that look like normal 2-operand instructions. For example,

  ```
  ADDI  *AR0,R2,R2     can be written as     ADD *AR0,R2
  || MPYI *AR1,R0,R0                    || MPYI *AR1,R0
  ```

  Instructions affected (applies to all parallel instructions that have a register second operand): ADDI, ADDF, AND, MPYI, MPYF, OR, SUBI, SUBF, XOR.

- You can write all commutative operations in parallel instructions in either order. For example, you can write the ADDI part of a parallel instruction in either of two ways:

  ```
  ADDI  *AR0,R1,R2        or        ADDI R1,*AR0,R2
  ```

  Instructions affected include: parallel instructions containing any of the following: ADDI, ADDF, MPYI, MPYF, AND, OR, XOR.

- Use the syntax in Table 13–14 to designate CPU registers in operands. Note the alternate notation R*n*, 0 ≤ *n* ≤ 27, which is used to designate any CPU register.

Table 13–14. CPU Register Syntax

Assemblers Syntax	Alternate Register Syntax	Assigned Function
R0	R0	Extended-precision register
R1	R1	Extended-precision register
R2	R2	Extended-precision register
R3	R3	Extended-precision register
R4	R4	Extended-precision register
R5	R5	Extended-precision register
R6	R6	Extended-precision register
R7	R7	Extended-precision register
AR0	R8	Auxiliary register
AR1	R9	Auxiliary register
AR2	R10	Auxiliary register
AR3	R11	Auxiliary register
AR4	R12	Auxiliary register
AR5	R13	Auxiliary register
AR6	R14	Auxiliary register
AR7	R15	Auxiliary register
DP	R16	Data-page pointer
IR0	R17	Index register 0
IR1	R18	Index register 1
BK	R19	Block-size register
SP	R20	Active stack pointer
ST	R21	Status register
IE	R22	CPU/DMA interrupt enable
IF	R23	CPU interrupt flags
IOF	R24	I/O flags
RS	R25	Repeat start address
RE	R26	Repeat end address
RC	R27	Repeat counter

13.6.3 Individual Instruction Descriptions

Each assembly language instruction for the ’C3x is described in this section in alphabetical order. The description includes the assembler syntax, operation, operands, encoding, description, cycles, status bits, mode bit, and examples.

Syntax

INST *src*, *dst*

or

INST1 *src2*, *dst1*
|| **INST2** *src3*, *dst2*

Each instruction begins with an assembler syntax expression. You can place labels either before the command (instruction mnemonic) on the same line or on the preceding line in the first column. The optional comment field that concludes the syntax is not included in the syntax expression. Space(s) are required between each field (label, command, operand, and comment fields).

The syntax examples illustrate the common one-line syntax and the two-line syntax used in parallel addressing. Note that the two vertical bars **||** that indicate a parallel addressing pair can be placed anywhere before the mnemonic on the second line. The first instruction in the pair can have a label, but the second instruction cannot have a label.

Operation

$|src| \rightarrow dst$

or

$|src2| \rightarrow dst1$
|| $src3 \rightarrow dst2$

The instruction operation sequence describes the processing that occurs when the instruction is executed. For parallel instructions, the operation sequence is performed in parallel. Conditional effects of status-register-specified modes are listed for such conditional instructions as B*cond.*

Operands

src general-addressing modes (G):

0 0	register (Rn, $0 \le n \le 27$)
0 1	direct
1 0	indirect
1 1	immediate

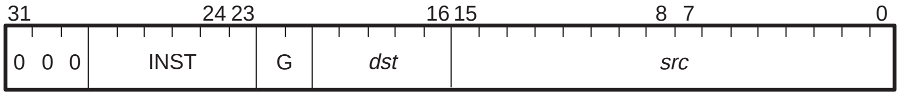
dst	register (Rn, $0 \le n \le 27$)

or

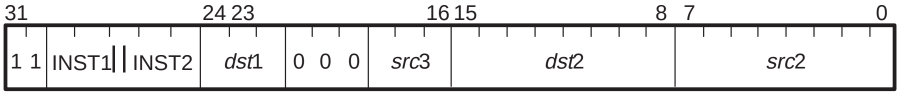
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R$n1$, $0 \le n1 \le 7$)
src3	register (R$n2$, $0 \le n2 \le 7$)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Operands are defined according to the addressing mode and/or the type of addressing used. Note that indirect addressing uses displacements and the index registers. See Chapter 6 for detailed information on addressing.

Opcode



or



Encoding examples are shown using general addressing and parallel addressing. The instruction pair for the parallel addressing example consists of INST1 and INST2.

Description Instruction execution and its effect on the rest of the processor or memory contents is described. Any constraints on the operands imposed by the processor or the assembler are discussed. The description parallels and supplements the information given by the operation block.

Cycles 1

The digit specifies the number of cycles required to execute the instruction.

Status Bits

LUF **Latched floating-point underflow condition flag.** 1 if a floating-point underflow occurs; unchanged otherwise.

LV **Latched overflow condition flag.** 1 if an integer or floating-point overflow occurs; unchanged otherwise.

UF **Floating-point underflow condition flag.** 1 if a floating-point underflow occurs; 0 otherwise.

N **Negative condition flag.** 1 if a negative result is generated; 0 otherwise. In some instructions, this flag is the MSB of the output.

Z **Zero condition flag.** 1 if a 0 result is generated; 0 otherwise. For logical and shift instructions, 1 if a 0 output is generated; 0 otherwise.

V **Overflow condition flag.** 1 if an integer or floating-point overflow occurs; 0 otherwise.

C **Carry flag.** 1 if a carry or borrow occurs; 0 otherwise. For shift instructions, this flag is set to the value of the last bit shifted out; 0 for a shift count of 0.

The seven condition flags stored in the status register (ST) are modified by the majority of instructions only if the destination register is R7–R0. The flags provide information about the properties of the result or the output of arithmetic or logical operations.

Mode Bit **OVM Overflow mode flag.** In general, integer operations are affected by the OVM bit value (described in Table 3–2 on page 3-6).

Example

```
INST @98AEh,R5
```

	Before Instruction		After Instruction
R5	07 6690 0000	R5	00 6690 1000
R5 decimal	2.30562500e+02	R5 decimal	1.80126953e+00
DP	080	DP	080
LUF	0	LUF	0
LV	0	LV	0
UF	0	UV	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
8098AEh	5CDF	8098AEh	5CDF
0200h	1234	0200h	1234

The sample code presented in the above format shows the effect of the code on system pointers (for example, DP or SP), registers (for example, R1 or R5), memory at specific locations, and the seven status bits. The values given for the registers include the leading 0s to show the exponent in floating-point operations. Decimal conversions are provided for all register and memory locations. The seven status bits are listed in the order in which they appear in the assembler and simulator (see Section 13.5 on page 13-28 and Table 13–12 on page 13-30 for further information on these seven status bits).

Syntax **ABSF** *src*, *dst*

Operation |*src*| → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 0 0 0 0 0	G	*dst*	*src*

Description The absolute value of the *src* operand is loaded into the *dst* register. The *src* and *dst* operands are assumed to be floating-point numbers.

An overflow occurs if *src* (*man*) = 80000000h and *src* (*exp*) = 7Fh. The result is *dst* (*man*) = 7FFFFFFFh and *dst* (*exp*) = 7Fh.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	0
N	0
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example ABSF `R4,R7`

	Before Instruction			**After Instruction**	
R4	`5C 8000 F971`	`–9.90337307e+27`	R4	`5C 8000 F971`	`–9.90337307e+27`
R7	`7D 2511 00AE`	`5.48527255e+37`	R7	`5C 7FFF 068F`	`9.90337307e+27`
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`0`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	

Syntax

    ABSF *src2*, *dst1*
|| **STF** *src3*, *dst2*

Operation

    |*src2* | → *dst1*
|| *src3* → *dst2*

Operands

src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16	15 8	7 0
1 1	0 0 1 0 0	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

A floating-point absolute value and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STF) reads from a register and the operation being performed in parallel (ABSF) writes to the same register, STF accepts the contents of the register as input before it is modified by the ABSF.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*. If *src3* and *dst1* point to the same register, *src3* is read before the write to *dst1.*

An overflow occurs if *src* (*man*) = 80000000h and *src* (*exp*) = 7Fh. The result is *dst* (*man*) = 7FFFFFFFh and *dst* (exp) = 7Fh.

Cycles

1

Status Bits

LUF	Unaffected
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	0
N	0
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
          ABSF     *++AR3(IR1) ,R4
||        STF      R4,*–AR7(1)
```

	Before Instruction			After Instruction	
R4	07 33C0 0000	1.79750e+02	R4	05 74C0 0000	6.118750e+01
AR3	80 9800		AR3	8098AF	
AR7	80 98C5		AR7	8098C5	
IR1	0AF		IR1	0AF	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data Memory					
8098AF	58B4000	–6.118750e+01	8098AF	58B4000	–6.118750e+01
8098C4	0		8098C4	733C000	1.79750e+02

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **ABSI** *src*, *dst*

Operation |*src*| → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 0 0 0 0 1	G	*dst*	*src*

Description The absolute value of the *src* operand is loaded into the *dst* register. The *src* and *dst* operands are assumed to be signed integers.

An overflow occurs if *src* = 80000000h. If ST(OVM) = 1, the result is *dst* = 7FFFFFFFh. If ST(OVM) = 0, the result is *dst* = 80000000h.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	0
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example 1

```
ABSI     R0,R0
or
ABSI     R0
```

	Before Instruction			After Instruction	
R0	00 FFFF FFCB	–53	R0	00 0000 0035	53

Example 2

```
ABSI     *AR1,R3
```

	Before Instruction			After Instruction	
R3	00 0000 0000		R3	00 0000 0035	53
AR1	00 0020		AR1	00 0020	
Data memory					
20	0FFFFFFCB	–53	20	0FFFFFFCB	–53

Syntax

ABSI *src2*, *dst1*

|| **STI** *src3, dst2*

Operation

|*src2* | → *dst1*
|| *src3* → *dst2*

Operands

src2	indirect	(*disp* = 0, 1, IR0, IR1)
dst1	register	(R*n*1, 0 ≤ 1 ≤ 7)
src3	register	(R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect	(*disp* = 0, 1, IR0, IR1)

This instruction’s operands have been augmented in the following devices:

- ❑ ’C31 silicon revision 6.0 or greater
- ❑ ’C32 silicon revision 2.0 or greater

src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16	15 8	7 0
1 1	0 0 1 0 1	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

An integer absolute value and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ABSI) writes to the same register, STI accepts the contents of the register as input before it is modified by the ABSI.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

An overflow occurs if *src* = 80000000h. If ST(OVM) = 1, the result is *dst* = 7FFFFFFFh. If ST(OVM) = 0, the result is *dst* = 80000000h.

Cycles

1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF Unaffected
LV 1 if an integer overflow occurs; unchanged otherwise
UF 0
N 0
Z 1 if a 0 result is generated; 0 otherwise
V 1 if an integer overflow occurs; 0 otherwise
C Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example

```
          ABSI      *–AR5(1),R5
||        STI       R1,*AR2––(IR1)
```

	Before Instruction			After Instruction	
R1	00 0000 0042	66	R1	00 0000 0042	66
R5	00 0000 0000		R5	00 0000 0035	53
AR2	80 98FF		AR2	80 98F0	
AR5	80 99E2		AR5	80 99E2	
IR1	0F		IR1	0F	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098FF	2	–53	8098FF	42	–53
8099E1	0FFFFFFCB	2	8099E1	0FFFFFFCB	66

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **ADDC** *src*, *dst*

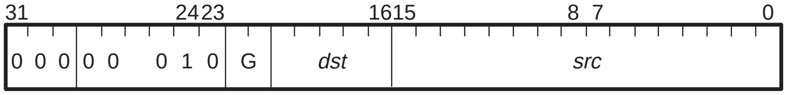
Operation *dst* + *src* + C → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode



Description The sum of the *dst* and *src* operands and the carry (C) flag is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a carry occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `ADDC R1,R5`

	Before Instruction			**After Instruction**	
R1	00 FFFF 5C25	–41,947	R1	00 FFFF 5C25	–41,947
R5	00 FFFF 019E	–65,122	R5	00 FFFE 5DC4	–107,068
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Syntax **ADDC3** *src2*, *src1*, *dst*

Operation *src1* + *src2* + C → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	any CPU register
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	any CPU register
0 1	any CPU register
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst any CPU register

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 0 0 0 0	T	*dst*	*src*1	*src*2

Description The sum of the *src1* and *src2* operands and the carry (C) flag is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
U	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a carry occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example 1

```
ADDC3     *AR5++(IR0),R5,R2
 or
ADDC3     R5,*AR5++(IR0),R2
```

	Before Instruction			After Instruction	
R2	00 0000 0000		R2	00 0000 0032	50
R5	00 0000 0066	102	R5	00 0000 0066	102
AR5	80 9908		AR5	80 9918	
IR0	10		IR0	10	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	1		C	1	
Data memory					
809908	0FFFFFFCB	–53	809908	0FFFFFFCB	–53

Example 2

```
ADDC3     R2, R7, R0
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 0000 123F	4671
R2	00 0000 02BC	700	R2	00 0000 02BC	700
R7	00 0000 0F82	3970	R7	00 0000 0F82	3970
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	1		C	0	

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **ADDF** *src*, *dst*

Operation *dst* + *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31		24 23	16 15	8 7 0
0 0 0	0 0 0 0 1 1	G	*dst*	*src*

Description The sum of the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `ADDF *AR4++(IR1),R5`

	Before Instruction			After Instruction	
R5	05 7980 0000	6.23750e+01	R5	09 052C 0000	5.3268750e+02
AR	4809800		AR4	80992B	
IR	112B		IR1	12B	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809800	86B2800	4.7031250e+02	809800	86B2800	4.7013250e+02

Syntax **ADDF3** *src2*, *src1*, *dst*

Operation *src1* + *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 7)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31		24 23	16 15	8 7	0
0 0 1	0 0 0 0 0 1	T	*dst*	*src*1	*src*2

Description The sum of the *src1* and *src2* operands is loaded into the *dst* register. The *src1, src2,* and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
ADDF3     R6,R5,R1
or
ADDF3     R5,R6,R1
```

	Before Instruction			After Instruction	
R1	00 0000 0000		R1	09 052C 0000	5.3268750e+02
R5	05 7980 0000	6.23750e+01	R5	05 7980 0000	6.23750e+01
R6	08 6B28 0000	4.7031250e+02	R6	08 6B28 0000	4.7031250e+02
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Example 2

```
ADDF3     *+AR1(1),*AR7++(IR0),R4
```

	Before Instruction			After Instruction	
R4	00 0000 0000		R4	07 0DB2 0000	1.41695313e+02
AR1	80 9820		AR1	80 9820	
AR7	80 99FO		AR7	80 99F8	
IR0	8		IR0	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UV	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809821h	700F000	1.28940e+02	809821h	700F000	1.28940e+02
8099F0h	34C2000	1.27590e+01	8099F0h	34C2000	1.27590e+01

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

  ADDF3 *src2, src1, dst1*
|| **STF** *src3, dst2*

Operation

  src1 + *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3 register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

> This instruction's operands have been augmented in the following devices:
>
> ❑ 'C31 silicon revision 6.0 or greater
> ❑ 'C32 silicon revision 2.0 or greater
>
> *src1* register (R*n*1, 0 ≤ *n*1 ≤ 7)
> *src2* indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
> *dst1* register (R*n*2, 0 ≤ *n*2 ≤ 7)
> *src3* register (R*n*3, 0 ≤ *n*3 ≤ 7)
> dst2 indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	0 0 1 1 0	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

A floating-point addition and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STF) reads from a register and the operation being performed in parallel (ADDF3) writes to the same register, STF accepts as input the contents of the register before it is modified by the ADDF3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF 1 if a floating-point underflow occurs; unchanged otherwise
LV 1 if a floating-point overflow occurs; unchanged otherwise
UF 1 if a floating-point underflow occurs; 0 otherwise
N 1 if a negative result is generated; 0 otherwise
Z 1 if a 0 result is generated; 0 otherwise
V 1 if a floating-point overflow occurs; 0 otherwise
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
            ADDF3     *+AR3(IR1),R2,R5
||          STF       R4,*AR2
```

	Before Instruction			After Instruction	
R2	07 0C80 0000	1.4050e+02	R2	07 0C80 0000	1.4050e+02
R4	05 7B40 0000	6.281250e+01	R4	05 7B40 0000	6.281250e+01
R5	00 0000 0000		R5	08 2020 0000	3.20250e+02
AR2	80 98F3		AR2	80 98F3	
AR3	80 9800		AR3	80 9800	
IR1	0A5		IR1	0A5	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UV	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098A5h	733C000	1.79750e+02	8098A5h	733C000	1.79750e+02
8098F3h	0		8098F3h	57B4000	6.28125e+01

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **ADDI** *src, dst*

Operation *dst* + *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 0 0 1 0 0	G	*dst*	*src*

Description The sum of the *dst* and *src* operands is loaded into the the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a carry occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example

```
ADDI      R3,R7
```

	Before Instruction			**After Instruction**	
R3	`00 FFFF FFCB`	–53	R3	`00 FFFF FFCB`	–53
R7	`35`	53	R7	`00 0000 0000`	
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`0`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	

Syntax **ADDI3** *<src2 >,<src1 >,<dst >*

Operation *src1* + *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	any CPU register
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	any CPU register
0 1	any CPU register
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst any CPU register

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 0 0 1 0	T	*dst*	*src*1	*src*2

Description The sum of the *src1* and *src2* operands is loaded into the *dst* register. The *src1, src2,* and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a carry occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example 1 `ADDI3 R4,R7,R5`

	Before Instruction			After Instruction	
R4	00 0000 00DC	220	R4	00 0000 00DC	220
R5	00 0000 0010	16	R5	00 0000 017C	380
R7	00 0000 00A0	160	R7	00 0000 00A0	160
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Example 2 `ADDI3 *–AR3(1),*AR6––(IR0),R2`

	Before Instruction			After Instruction	
R2	00 0000 0010	16	R2	00 0000 6598	26,000
AR3	80 9802		AR3	80 9802	
AR6	80 9930		AR6	80 9918	
IR0	18		IR0	18	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UV	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809801	2AF8	11,000	809801	2AF8	11,000
809930	3A98	15,000	809930	3A98	15,000

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

    ADDI3 *src2, src1, dst1*
|| **STI** *src3, dst2*

Operation

    src1 + *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	0 0 1 1 1	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

An integer addition and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ADDI3) writes to the same register, STI accepts the contents of the register as input before it is modified by the ADDI3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a carry occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example

```
        ADDI3     *AR0––(IR0),R5,R0
||      STI       R3,*AR7
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 0000 0208	520
R3	00 0000 0035	53	R3	00 0000 0035	53
R5	00 0000 00DC	220	R5	00 0000 00DC	220
AR0	80 992C		AR0	80 9920	
AR7	80 983B		AR7	80 983B	
IR0	OC		IR0	OC	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UV	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80992C	12C	300	80992C	12C	300
80983B	0		80983B	35	53

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **AND** *src, dst*

Operands *dst* AND *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate (not sign extended)

dst any CPU register

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 0 0 1 0 1	G	*dst*	*src*

Description The bitwise-logical AND between the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
AND       R1,R2
```

	Before Instruction		**After Instruction**
R1	00 0000 0080	R1	00 0000 0080
R2	00 0000 0AFF	R2	00 0000 0080
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	1	C	1

Syntax **AND3** *src2, src1, dst*

Operation *src1* AND *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	any CPU register
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	any CPU register
0 1	any CPU register
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24 23		16 15	8 7	0
0 0 1	0 0 0 0 1 1	T	*dst*	*src*1	*src*2

Description The bitwise-logical AND between the *src1* and *src2* operands is loaded into the destination register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
AND3      *AR0––(IR0),*+AR1,R4
```

	Before Instruction	After Instruction
R4	00 0000 0000	00 0000 0020
AR0	80 98F4	80 98A4
AR1	80 9951	80 9951
IR0	50	50
LUF	0	0
LV	0	0
UF	0	0
N	0	0
Z	0	0
V	0	0
C	0	0
Data memory		
8098F4h	30	30
809952h	123	123

Example 2

```
AND3      *–AR5,R7,R4
```

	Before Instruction	After Instruction
R4	00 0000 0000	00 0000 0002
R7	00 0000 0002	00 0000 0002
AR5	80 985C	80 985C
LUF	0	0
LV	0	0
UF	0	0
N	0	0
Z	0	0
V	0	0
C	0	0
Data memory		
80985Bh	0AFF	0AFF

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

	AND3	*src2, src1, dst1*
||	**STI**	*src3, dst2*

Operation

src1 AND *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16 15	8 7	0
1 1	0 1 0 0 0	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

A bitwise-logical AND and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (AND3) writes to the same register, STI accepts the contents of the register as input before it is modified by the AND3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
        AND3     *+AR1(IR0),R4,R7
||      STI      R3,*AR2
```

	Before Instruction			After Instruction	
R0	00 0000 0008		R0	00 0000 0008	
R3	00 0000 0035	53	R3	00 0000 0035	53
R4	00 0000 A323		R4	00 0000 A323	
R7	00 0000 0000		R7	00 0000 0003	
AR1	80 99F1		AR1	80 99F1	
AR2	80 983F		AR2	80 983F	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8099F9h	5C53		8099F9h	5C53	
80983Fh	0		80983Fh	35	53

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **ANDN** *src*, *dst*

Operation *dst* AND ~*src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate (not sign extended)

dst any CPU register

Opcode

31	24	23	16	15 8 7 0
0 0 0	0 0 0 1 1 0	G	*dst*	*src*

Description The bitwise-logical AND between the *dst* operand and the bitwise-logical complement (~) of the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
ANDN @980Ch,R2
```

	Before Instruction		After Instruction
R2	00 0000 0C2F	R2	00 0000 042D
DP	080	DP	080
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
80980Ch	0A02	80980Ch	0A02

Syntax **ANDN3** *src2, src1, dst*

Operation *src1* AND ~*src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	any CPU register
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	any CPU register
0 1	any CPU register
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IO0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 0 1 0 0	T	*dst*	*src*1	*src*2

Description The bitwise-logical AND between the *src1* operand and the bitwise-logical complement (~) of the *src2* operand is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
ANDN3 R5,R3,R7
```

	Before Instruction		After Instruction
R3	00 0000 0C2F	R3	00 0000 0C2F
R5	00 0000 0A02	R5	00 0000 0A02
R7	00 0000 0000	R7	00 0000 042D
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0

Example 2

```
ANDN3 R1,*AR5++(IR0),R0
```

	Before Instruction		After Instruction
R0	00 0000 0000	R0	00 0000 0F30
R1	00 0000 00CF	R1	00 0000 00CF
AR5	80 9825	AR5	80 982A
IR0	5	IR0	5
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809825h	0FFF	809825h	0FFF

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **ASH** *count, dst*

Operation If (*count* ≥ 0):

dst << *count* → *dst*

Else:

dst >> |*count* | → *dst*

Operands *count* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24	23	16	15 8 7 0
0 0 0	0 0 0 1 1 1	G	*dst*	*count*

Description The seven LSBs of the *count* operand are used to generate the 2s-complement shift count of up to 32 bits.

If the *count* operand is greater than 0, the *dst* operand is left shifted by the value of the *count* operand. Low-order bits that are shifted in are zero filled, and high-order bits are shifted out through the carry (C) bit.

Arithmetic left shift:

C ← *dst* ← 0

If the *count* operand is less than 0, the *dst* operand is right shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are sign-extended as it is right shifted. Low-order bits are shifted out through the C bit.

Arithmetic right shift:

sign of *dst* → *dst* → C

If the *count* operand is 0, no shift is performed, and the C bit is set to 0. The *count* and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Set to the value of the last bit shifted out; 0 for a shift *count* of 0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1 `ASH R1,R3`

	Before Instruction			After Instruction
R1	00 0000 0010	16	R1	00 0000 0010
R3	00 000A E000		R3	00 E000 0000
LUF	0		LUF	0
LV	0		LV	1
UF	0		UF	0
N	0		N	1
Z	0		Z	0
V	0		V	1
C	0		C	0

Example 2 `ASH @98C3h,R5`

	Before Instruction			After Instruction	
R5	00 AEC0 0001		R5	00 FFFF FFAE	
DP	80		DP	80	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	0	
C	0		C	1	
Data memory					
8098C3h	0FFE8	–24	8098C3h	0FFE8	–24

Syntax **ASH3** *count, src, dst*

Operation If (*count* ≥ 0):

src << *count* → *dst*

Else:

src >> |*count* | → *dst*

Operands *count* 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 27)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 0 1 0 1	T	*dst*	*src*	*count*

Description The seven LSBs of the *count* operand are used to generate the 2s-complement shift count of up to 32 bits.

If the *count* operand is greater than 0, the *src* operand is left shifted by the value of the *count* operand. Low-order bits that are shifted in are zero filled, and high-order bits are shifted out through the status register's C bit.

Arithmetic left shift:

C ← *src* ← 0

If the *count* operand is less than 0, the *src* operand is right shifted by the absolute value of the *count* operand. The high-order bits of the *src* operand are sign extended as they are right shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right shift:

sign of *src* → *src* → C

If the *count* operand is 0, no shift is performed, and the C bit is set to 0. The *count*, *src*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF Unaffected
LV 1 if an integer overflow occurs; unchanged otherwise
UF 0
N MSB of the output
Z 1 if a 0 result is generated; 0 otherwise
V 1 if an integer overflow occurs; 0 otherwise
C Set to the value of the last bit shifted out; 0 for a shift *count* of 0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
ASH3      *AR3––(1),R5,R0
```

	Before Instruction		After Instruction
R0	00 0000 0000	R0	00 02B0 0000
R5	00 0000 02B0	R5	00 0000 02B0
AR3	80 9921	AR3	80 9920
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809921h	10 16	809921h	10 16

Example 2 `ASH3 R1,R3,R5`

	Before Instruction			After Instruction	
R1	`00 FFFF FFF8`	–8	R1	`00 FFFF FFF8`	–8
R3	`00 FFFF CB00`		R3	`00 FFFF CB00`	
R5	`00 0000 0000`		R5	`00 FFFF FFCB`	
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`1`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8.5.2 for the effects of operand ordering on the cycle count.

Syntax

ASH3 *count, src2, dst1*
|| **STI** *src3, dst2*

Operation

If (coun*t* ≥ 0):
src2 << *count* → *dst1*

Else:
src2 >> |*count*| → *dst1*
|| *src3* → *dst2*

Operands

count register (R*n*1, 0 ≤ *n*1 ≤ 7)

src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3 register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

> This instruction's operands have been augmented in the following devices:
>
> ❑ 'C31 silicon revision 6.0 or greater
> ❑ 'C32 silicon revision 2.0 or greater
>
> *src2* indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
> *dst1* register (R*n*1, 0 ≤ *n*1 ≤ 7)
> *src3* register (R*n*2, 0 ≤ *n*2 ≤ 7)
> *dst2* indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16	15 8	7 0
1 1	0 1 0 0 1	*dst*1	*count*	*src*3	*dst*2	*src*2

Description

The seven LSBs of the *count* operand register are used to generate the 2s-complement shift count of up to 32 bits.

If the *count* operand is greater than 0, the *src2* operand is left shifted by the value of the *count* operand. Low-order bits shifted in are zero filled, and high-order bits are shifted out through the C bit.

Arithmetic left shift:

C ← *src2* ← 0

If the *count* operand is less than 0, the *src2* operand is right hifted by the absolute value of the *count* operand. The high-order bits of the *src2* operand are sign extended as they are right shifted. Low-order bits are shifted out through the C bit.

Arithmetic right shift:

sign of *src2* → *src2* → C

If the *count* operand is 0, no shift is performed, and the C bit is set to 0. The *count* and *dst* operands are assumed to be signed integers.

All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ASH3) writes to the same register, STI accepts the contents of the register as input before it is modified by the ASH3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Set to the value of the last bit shifted out; 0 for a shift count of 0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
            ASH3      R1,*AR6++(IR1),R0
||          STI       R5,*AR2
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 FFFF FFAE	
R1	00 0000 FFE8	–24	R1	00 0000 FFE8	–24
R5	00 0000 0035	53	R5	00 0000 0035	53
AR2	80 98A2		AR2	80 98A2	
AR6	80 9900		AR6	80 998C	
IR1	8C		IR1	8C	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809900h	0AE000000		809900h	0AE000000	
8098A2h	0		8098A2h	35	53

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **B*cond*** *src*

Operation

```
If cond is true:
   If src is in register-addressing mode (Rn, 0 ≤ n ≤ 27),
      src → PC.
   If src is in PC-relative mode (label or address),
      displacement + PC + 1 → PC.
Else, continue
```

Operands *src* conditional-branch addressing modes (B):

0	register
1	PC relative

Opcode

31		24 23	16	15 8 7 0
0 1 1 0 1 0	B	0 0 0 0	*cond*	Register or displacement

Description B*cond* signifies a standard branch that executes in four cycles. A branch is performed if the condition is true (since a pipeline flush also occurs on a true condition; see Section 8.2, *Pipeline Conflicts*, on page 8-4). If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label – (PC of branch instruction + 1). This displacement is stored as a 16-bit signed integer in the 16 LSBs of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The 'C3x provides 20 condition codes that you can use with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
BZ R0
```

	Before Instruction		After Instruction
R0	00 0003 FF00	R0	00 0003 FF00
PC	2B00	PC	3 FF00
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	1	Z	1
V	0	V	0
C	0	C	0

Note:

If a BZ instruction is executed immediately following a RND instruction with a 0 operand, the branch is not performed, because the 0 flag is not set. To circumvent this problem, execute a BZUF instead of a BZ instruction.

Syntax **B*cond* D** *src*

Operation

```
If cond is true:
   If src is in register-addressing mode (Rn, 0 ≤ n ≤ 27),
     src → PC.
   If src is in PC-relative mode (label or address),
     displacement + PC + 3  → PC.
Else, continue
```

Operands *src* conditional-branch addressing modes (B):

0	register
1	PC relative

Opcode

31							24	23			16	15 … 8 7 … 0
0	1	1	0	1	0	B	0	0	0	1	*cond*	Register or displacement

Description B*cond*D signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch, and the three instructions following B*cond*D do not affect the condition.

A branch is performed if the condition is true. If the *src* operand is expressed in register-addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label – (PC of branch instruction + 3). This displacement is stored as a 16-bit signed integer in the 16 LSBs of the branch instruction. This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. The ’C3x provides 20 condition codes that you can use with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are *set on the previous instruction* only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
BNZD 36 (36 = 24h)
```

	Before Instruction		**After Instruction**
PC	`0050`	PC	`0077`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`0`

Note:

Delayed branches disable interrupts until the completion of the three instructions that follow the delayed branch, regardless if the branch is or is not performed. The following instructions cannot be used in the next three instructions following a delayed branch: B*cond*, B*cond*D, BR, BRD, CALL, CALL*cond*, DB*cond*, DB*cond*D, IDLE, IDLE2, RETI*cond*, RETS*cond*, RPTB, RPTS, TRAP*cond*.

Syntax **BR** *src*

Operation *src* → PC

Operands *src* long-immediate addressing mode

Opcode

31	24	23 16 15 8 7 0
0 1 1 0 0 0 0	0	*src*

Description BR performs a PC-relative branch that executes in four cycles, since a pipeline flush also occurs upon execution of the branch (see Section 8.2, *Pipeline Conflicts*, on page 8-4). An unconditional branch is performed. The *src* operand is assumed to be a 24-bit unsigned integer. Note that bit 24 = 0 for a standard branch.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `BR 805Ch`

	Before Instruction		After Instruction
PC	0080	PC	805C
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0

Syntax **BRD** *src*

Operation *src* → PC

Operands *src* long-immediate addressing mode

Opcode

31	24	23 16 15 8 7 0
0 1 1 0 0 0 0	1	*src*

Description BRD signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch.

An unconditional branch is performed. The *src* operand is assumed to be a 24-bit unsigned integer. Note that bit 24 = 1 for a delayed branch.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `BRD 2Ch`

	Before Instruction		**After Instruction**
PC	001B	PC	002C
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0

Syntax **CALL** *src*

Operation Next PC → *++SP
src → PC

Operands *src* long-immediate addressing mode

Opcode

31 24	23	16 15 8 7 0
0 1 1 0 0 0 1	0	*src*

Description A call is performed. The next PC value is pushed onto the system stack. The *src* operand is loaded into the PC. The *src* operand is assumed to be a 24-bit unsigned-immediate operand. Since the CALL instruction takes 4 cycles to execute, the pipeline is flushed.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `CALL 123456h`

	Before Instruction		After Instruction
PC	0005	PC	123456
SP	809801	SP	809802
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
		809802h	6

Syntax **CALL*cond*** *src*

Operation If *cond* is true:

Next PC → *++SP
If *src* is in register addressing mode (R*n*, 0 ≤ *n* ≤ 27),
src → PC.
If *src* is in PC-relative mode (label or address),
displacement + PC + 1 → PC.

Else, continue

Operands *src* conditional-branch addressing modes (B):

0	register
1	PC relative

Opcode

31		24 23	16	15 8 7 0
0 1 1 1 0 0	B	0 0 0 0	*cond*	Register or displacement

Description A call is performed if the condition is true. If the condition is true, the next PC value is pushed onto the system stack. If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label – (PC of call instruction + 1). This displacement is stored as a 16-bit signed integer in the 16 LSBs of the call instruction word. This displacement is added to the PC of the call instruction plus 1 to generate the new PC. This instruction flushes the pipeline as shown in Example 8–13 on page 8-18.

The ’C3x provides 20 condition codes that can be used with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 5

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `CALLNZ R5`

	Before Instruction		After Instruction
R5	00 0000 0789	R5	00 0000 0789
PC	0123	PC	0789
SP	809835	SP	809836
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
		809836h	124

Syntax **CMPF** *src, dst*

Operation *dst – src*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31 24 23 16 15 8 7 0

0 0 0	0 0 1 0 0 0	G	*dst*	*src*

Description The *src* operand is subtracted from the *dst* operand. The result is not loaded into any register, which allows for nondestructive compares. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified for all destination registers (R27 – R0).

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `CMPF *+AR4,R6`

	Before Instruction			After Instruction	
R6	07 0C80 0000	1.4050e+02	R6	07 0C80 0000	1.4050e+02
AR4	80 98F2		AR4	80 98F2	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	1	
V	0		V	0	
C	0		C	0	
Data memory					
8098F3h	070C8000	1.4050e+02	8098F3h	070C8000	1.4050e+02

Syntax **CMPF3** *src2, src1*

Operation *src1 – src2*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 7)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 0 1 1 0	T	0 0 0 0 0	*src*1	*src*2

Description The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register, which allows for nondestructive compares. The *src1* and *src2* operands are assumed to be floating-point numbers. Although this instruction has only two operands, it is designated as a 3-operand instruction because operands are specified in the 3-operand format.

Cycles 1

Status Bits These condition flags are modified for all destination registers (R27–R0).

LUF 1 if a floating-point underflow occurs; unchanged otherwise
LV 1 if a floating-point overflow occurs; unchanged otherwise
UF 1 if a floating-point underflow occurs; 0 otherwise
N 1 if a negative result is generated; 0 otherwise
Z 1 if a 0 result is generated; 0 otherwise
V 1 if a floating-point overflow occurs; 0 otherwise
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
CMPF3     *AR2,*AR3––(1)
```

	Before Instruction			After Instruction	
AR2	80 9831		AR2	80 9831	
AR3	80 9852		AR4	80 9851	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809831h	77A7000	2.5044e+02	809831h	77A7000	2.5044e+02
809852h	57A2000	6.253125e+01	809852h	57A2000	6.253125e+01

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **CMPI** *src, dst*

Operation *dst* – *src*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31		24 23	16 15	8 7 0
0 0 0	0 0 1 0 0 1	G	*dst*	*src*

Description The *src* operand is subtracted from the *dst* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified for all destination registers (R27–R0).

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `CMPI R3,R7`

	Before Instruction			**After Instruction**	
R3	00 0000 0898	2200	R3	00 0000 0898	2200
R7	00 0000 03E8	1000	R7	00 0000 03E8	1000
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	0	
C	0		C	1	

Syntax **CMPI3** *src2, src1*

Operation *src1* – *src2*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

*src*2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 27)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24 23		16	15 8	7 0
0 0 1	0 0 0 1 1 1	T	0 0 0 0 0	*src*1	*src*2

Description The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register, which allows for nondestructive compares. The *src1* and *src2* operands are assumed to be signed integers. Although this instruction has only two operands, it is designated as a 3-operand instruction because operands are specified in the 3-operand format.

Cycles 1

Status Bits These condition flags are modified for all destination registers (R27–R0).

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
CMPI3  R7,R4
```

	Before Instruction			After Instruction	
R4	00 0000 0898	2200	R4	00 0000 0898	2200
R7	00 0000 03E8	1000	R7	00 0000 03E8	1000
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

DBcond AR*n, src*

Operation

```
ARn – 1 → ARn
If cond is true and ARn ≥ 0 :
   If src is in register addressing mode (Rn, 0 ≤ n ≤ 27),
      src → PC.
   If src is in PC-relative mode (label or address),
      displacement + PC + 1 → PC.
Else, continue
```

Operands

src conditional-branch addressing modes (B):

0	register
1	PC relative

AR*n* register (0 ≤ *n* ≤ 7)

Opcode

31							24 23		16	15 8 7 0
0	1	1	0	1	1	B	AR*n*	0	*cond*	Register or displacement

Description

DB*cond* signifies a standard branch that executes in four cycles because the pipeline must be flushed if *cond* is true. The specified auxiliary register is decremented and a branch is performed if the condition is true and the specified auxiliary register is greater than or equal to 0. The condition flags are those set by the last previous instruction that affects the status bits.

The auxiliary register is treated as a 24-bit signed integer. The 8 MSBs are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 24 LSBs of the auxiliary register. Note that the branch condition does not depend on the auxiliary register decrement.

If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative addressing mode, the assembler generates a displacement: displacement = label – (PC of branch instruction + 1). This integer is stored as a 16-bit signed integer in the 16 LSBs of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The ’C3x provides 20 condition codes that can be used with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R0–R7) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
CMPI     200,R3
DBLT     AR3,R2
```

	Before Instruction		After Instruction
R2	00 0000 009F	R2	00 0000 009F
R3	00 0000 0080	R3	00 0000 0080
AR3	00 0012	AR3	00 0011
PC	005F	PC	009F
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	1	N	1
Z	0	Z	0
V	0	V	0
C	0	C	0

Syntax **DB*cond*D** AR*n*, *src*

Operation

AR*n* – 1 → AR*n*
If *cond* is true and AR*n* ≥ 0:
  If *src* is in register addressing mode (R*n*, 0 ≤ *n* ≤ 27)
    src → PC
  If *src* is in PC-relative mode (label or address)
    displacement + PC + 3 → PC.

Else, continue

Operands *src* conditional-branch addressing modes (B):

0	register
1	PC relative

AR*n* register (0 ≤ *n* ≤ 7)

Opcode

31		24 23		16	15 8 7 0
0 1 1 0 1 1	B	AR*n*	1	*cond*	Register or displacement

Description

DB*cond* D signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch. The specified auxiliary register is decremented, and a branch is performed if the condition is true and the specified auxiliary register is greater than or equal to 0. The condition flags are those set by the last previous instruction that affects the status bits. The three instructions following the DB*cond*D do not affect the condition.

The auxiliary register is treated as a 24-bit signed integer. The 8 MSBs are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 24 LSBs of the auxiliary register. The branch condition does not depend on the auxiliary register decrement.

If the *src* operand is expressed in register-addressing mode, the contents of the specified register are loaded into the PC. If the *src* is expressed in PC-relative addressing, the assembler generates a displacement: displacement = label – (PC of branch instruction + 3). This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. Note that bit 21 = 1 for a delayed branch.

The ’C3x provides 20 condition codes that you can use with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
CMPI      26h,R2
DBZD      AR5, $+110h
```

	Before Instruction		After Instruction
R2	00 0000 0026	R2	00 0000 0026
AR5	00 0067	AR5	00 0066
PC	0100	PC	0210
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	1
V	0	V	0
C	0	C	0

Syntax **FIX** *src, dst*

Operation fix(*src*) → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 0 1 0 1 0	G	*dst*	*src*

Description The floating-point operand *src* is rounded down to the nearest integral value less than or equal to floating-point value, and the result is loaded into the *dst* register. The *src* operand is assumed to be a floating-point number and the *dst* operand a signed integer.

The exponent field of the *dst* register (bits 39–32) is not modified.

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit 2s-complement integer. In the case of integer overflow, the result is saturated in the direction of overflow.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
FIX  R1,R2
```

	Before Instruction			After Instruction	
R1	0A 2820 0000	1.3454e+3	R1	0A 2820 0000	13454e+3
R2	00 0000 0000		R2	00 0000 0541	1345
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Syntax

```
   FIX  src2, dst1
|| STI  src3, dst2
```

Operation

```
   fix(src2) → dst1
|| src3 → dst2
```

Operands

src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3 register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

> This instruction’s operands have been augmented in the following devices:
>
> - ❑ ’C31 silicon revision 6.0 or greater
> - ❑ ’C32 silicon revision 2.0 or greater
>
> *src2* indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
> *dst1* register (R*n*1, 0 ≤ *n*1 ≤ 7)
> *src3* register (R*n*2, 0 ≤ *n*2 ≤ 7)
> *dst2* indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16	15 8	7 0
1 1	0 1 0 1 0	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

A floating-point-to-integer conversion is performed. All registers are read at the beginning and loaded at the end of the execute cycle. This means that, if one of the parallel operations (STI) reads from a register, and the operation being performed in parallel (FIX) writes to the same register, STI accepts the contents of the register as input before it is modified by FIX.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit 2s-complement integer. In the case of integer overflow, the result is saturated in the direction of overflow.

Cycles

1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF Unaffected
LV 1 if an integer overflow occurs; unchanged otherwise
UF 0
N 1 if a negative result is generated; 0 otherwise
Z 1 if a 0 result is generated; 0 otherwise
V 1 if an integer overflow occurs; 0 otherwise
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
           FIX       *++AR4(1),R1
||         STI       R0,*AR2
```

	Before Instruction			**After Instruction**	
R0	00 0000 00DC	220	R0	00 0000 00DC	220
R1	00 0000 0000		R1	00 0000 00B3	179
AR2	80 983C		AR2	80 983C	
AR4	80 98A2		AR4	80 98A3	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	O	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098A3h	733C000	1.7950e+02	8098A3	733C000	1.79750e+02
80983Ch	0		80983C	0DC	220

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **FLOAT** *src, dst*

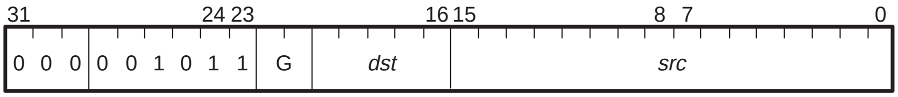
Operation float *(src)* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode



Description The integer operand *src* is converted to the floating-point value equal to it; the result is loaded into the *dst* register. The *src* operand is assumed to be a signed integer; the *dst* operand is assumed to be a floating-point number.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
FLOAT *++AR2(2),R5
```

	Before Instruction			After Instruction	
R5	00 034C 2000	1.27578125e+01	R5	00 72E0 0000	1.74e+02
AR2	80 9800		AR2	80 9802	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809802	0AE	174	809802	0AE	174

Syntax

    FLOAT *src2, dst1*
|| **STF** *src3*, *dst2*

Operation

    float*(src2)* → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3 register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2 register (*disp* = 0, 1, IR0, IR1)

This instruction’s operands have been augmented in the following devices:

- ❏ ’C31 silicon revision 6.0 or greater
- ❏ ’C32 silicon revision 2.0 or greater

src2 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3 register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2 register (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16 15	8	7 0
1 1	0 1 0 1 1	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

An integer-to-floating-point conversion is performed. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STF) reads from a register and the operation being performed in parallel (FLOAT) writes to the same register, then STF accepts the contents of the register as input before it is modified by FLOAT.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit

OVM Operation is affected by OVM bit value.

Example

```
            FLOAT     *+AR2(IR0),R6
    ||      STF       R7,*AR1
```

	Before Instruction			After Instruction	
R6	00 0000 0000		R6	07 2E00 0000	1.740e+02
R7	03 4C20 0000	1.27578125e+01	R7	03 4C20 0000	1.27578125e+01
AR1	80 9933		AR1	80 9933	
AR2	80 98C5		AR2	80 98C5	
IR0	8		IR0	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098CD	0AE	174	8098CD	0AE	174
809933	0		809933	034C2000	1.27578125e+01

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **IACK** *src*

Operation Perform a dummy read operation with $\overline{\text{IACK}}$ = 0.
At end of dummy read, set $\overline{\text{IACK}}$ to 1.

Operands *src* general addressing modes (G):

0 1	direct
1 0	indirect

Opcode

31 24 23 16 15 8 7 0

0 0 0	1 1 0 1 1 0	G	0 0 0 0 0	*src*

Description A dummy read operation is performed if off-chip memory is specified. $\overline{\text{IACK}}$ is set to 0, regardless of *src* location, a half H1 cycle after the beginning of the decode phase of the IACK instruction. At the first half of the H1 cycle of the completion of the dummy read, $\overline{\text{IACK}}$ is set to 1. The $\overline{\text{IACK}}$ signal will not be extended due to multicycle reads with wait states. This instruction can be used to generate an external interrupt acknowledge. The $\overline{\text{IACK}}$ signal and the address can be used to signal interrupt acknowledge to external devices. The data read by the processor is unused.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
IACK *AR5
```

	Before Instruction	After Instruction
$\overline{\text{IACK}}$	1	1
PC	300	301
LUF	0	0
LV	0	0
UF	0	0
N	0	0
Z	0	0
V	0	0
C	0	0

Syntax **IDLE**

Operation 1 → ST(GIE)
Next PC → PC
Idle until interrupt.

Operands None

Opcode

31							24	23							16	15							8	7							0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Description The global-interrupt-enable bit is set, the next PC value is loaded into the PC, and the CPU idles until an unmasked interrupt is received. When the interrupt is received, the contents of the PC are pushed onto the active system stack, the interrupt vector is read, and the interrupt service routine is executed.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
IDLE      ; The processor idles until a reset
          ; or unmasked interrupt occurs.
```

For correct device operation, the three instructions after a delayed branch should not be IDLE or IDLE2 instructions.

CAUTION

Syntax **IDLE2** (supported by: ’LC31, ’C32, ’C30 silicon revision 7.x or greater, ’C31 silicon revision 5.x or greater)

Operation

1 → ST(GIE)
Next PC → PC
Idle until interrupt.

Operands None

Opcode

```
31          24 23                      16 15                 8 7                      0
| 0 0 0 | 0 0 1 1 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 |
```

Description The IDLE2 instruction serves the same function as IDLE, except that it removes the functional clock input from the internal device. This allows for extremely low power mode. The PC is incremented once, and the device remains in an idle state until one of the external interrupts (INT0–3) is asserted.

In IDLE2 mode, the ’C3x devices that support this mode behave as follows:

- The CPU, peripherals, and memory retain their previous states.
- When the device is in the functional (nonemulation) mode, the clocks stop with H1 high and H3 low.
- The device remains in IDLE2 until one of the four external interrupts ($\overline{\text{INT3}}$–$\overline{\text{INT0}}$) is asserted for at least two H1 cycles. When one of the four interrupts is asserted, the clocks start after a delay of one H1 cycle. The clocks can start up in the phase opposite that in which they were stopped (that is, H1 might start high when H3 was high before stopping, and H3 might start high when H1 was high before stopping). However, the H1 and H3 clocks remain 180° out of phase with each other.
- During IDLE2 operation, one of the four external interrupts must be asserted for at least two H2 cycles to be recognized and serviced by the CPU. For the processor to recognize only one interrupt when it restarts operation, the interrupt must be asserted for less than three cycles.
- When the device is in emulation mode, the H1 and H3 clocks continue to run normally, and the CPU operates as if an IDLE instruction had been executed. The clocks continue to run for correct operation of the emulator.

CAUTION

For correct device operation, the three instructions after a delayed branch should not be IDLE or IDLE2 instructions.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
IDLE2     ; The processor idles until a reset
          ; or interrupt occurs.
```

Syntax **LDE** *src, dst*

Operation *src*(*exp*) → *dst*(*exp*)

Operands *src* general addressing modes (G):

0 0	register (R*n*, $0 \le n \le 7$)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, $0 \le n \le 7$)

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 0 1 1 0 1	G	*dst*	*src*

Description The exponent field of the *src* operand is loaded into the exponent field of the *dst* register. No modification of the *dst* register mantissa field is made unless the value of the exponent loaded is the reserved value of the exponent for 0 as determined by the precision of the *src* operand. Then the mantissa field of the *dst* register is set to 0. The *src* and *dst* operands are assumed to be floating-point numbers. Immediate values are evaluated in the short floating-point format.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `LDE R0,R5`

	Before Instruction			After Instruction	
R0	02 0005 6F30	4.00066337e+00	R0	02 0005 6F30	4.00066337e+00
R5	0A 056F E332	1.06749648e+03	R5	02 056F E332	4.16990814e+00
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Syntax **LDF** *src, dst*

Operation *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 0 1 1 1 0	G	*dst*	*src*

Description The *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `LDF  @9800h,R2`

	Before Instruction			After Instruction	
R2	00 0000 0000		R2	01 0C52 A000	2.19254303e+00
DP	080		DP	080	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809800	010C52A0	2.19254303e+00	809800	010C52A0	2.19254303e+00

Syntax

LDF*cond* *src, dst*

Operation

If *cond* is true:
src → *dst.*

Else:
dst is unchanged.

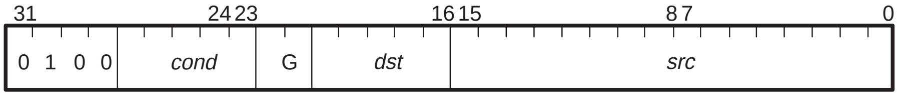
Operands

src general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode



Description

If the condition is true, the *src* operand is loaded into the *dst* register; otherwise, the *dst* register is unchanged. The *dst* and *src* operands are assumed to be floating-point numbers.

The 'C3x provides 20 condition codes that can be used with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Note that an LDFU (load floating-point unconditionally) instruction is useful for loading R7–R0 without affecting condition flags. Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
LDFZ  R3,R5
```

	Before Instruction			After Instruction	
R3	2C FF2C D500	1.77055560e+13	R3	2C FF2C D500	1.77055560e+13
R5	5F 0000 003E	3.96140824e+28	R5	2C FF2C D500	1.77055560e+13
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	1		Z	1	
V	0		V	0	
C	0		C	0	

Syntax **LDFI** *src, dst*

Operation Signal interlocked operation
src → *dst*

Operands *src* general addressing modes (G):

0 1 direct
1 0 indirect (disp = 0–255, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 0 1 1 1 1	G	*dst*	*src*

Description The *src* operand is loaded into the *dst* register. An interlocked operation is signaled over XF0 and XF1. The *src* and *dst* operands are assumed to be floating-point numbers. Only direct and indirect modes are allowed. See Section 7.4, *Interlocked Operations*, on page 7-13 for a detailed description.

Cycles 1 if XF1 = 0 (see Section 7.4 on page 7-13)

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `LDFI *+AR2,R7`

	Before Instruction			**After Instruction**	
R7	00 0000 0000		R7	05 84C0 0000	–6.28125e+01
AR2	80 98F1		AR2	80 98F1	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098F2h	584C000	–6.28125e+01	8098F2h	584C000	–6.28125e+01

Syntax

  LDF *src2, dst2*
|| **LDF** *src1, dst1*

Operation

  src2 → *dst2*
|| *src1* → *dst1*

Operands

src1	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst2	register (R*n*2, 0 ≤ *n*2 ≤ 7)

This instruction's operands have been augmented on the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst2	register (R*n*2, 0 ≤ *n*2 ≤ 7)

Opcode

31	24	23		16	15 8	7 0
1 1	0 0 0 1 0	*dst*2	*dst*1	0 0 0	*src*1	*src*2

Description

Two floating-point loads are performed in parallel. If the LDFs load the same register, the assembler issues a warning. The result is that of LDF *src2, dst2.*

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
        LDF     *––AR1(IR0),R7
||      LDF     *AR7++(1),R3
```

Before Instruction			After Instruction		
R3	00 0000 0000		R0	00 0000 0008	
R7	00 0000 0000		R3	05 7B40 0000	6.281250e+01
AR1	80 985F		R7	07 0C80 0000	1.4050e+02
AR7	80 988A		AR1	80 9857	
IR0	8		AR7	80 988B	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809857h	70C8000	1.4050e+02	809857h	70C8000	1.4050e+02
80988Ah	57B4000	6.281250e+01	80988Ah	57B4000	6.281250e+01

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

LDF *src2, dst1*
|| **STF** *src3, dst2*

Operation

src2 → *dst1*
|| *src3* → *dst2*

Operands

src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented on the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	0 1 1 0 0	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

A floating-point load and a floating-point store are performed in parallel.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
        LDF      *AR2––(1),R1
||      STF      R3,*AR4++(IR1)
```

	Before Instruction			After Instruction	
R1	00 0000 0000		R1	07 0C80 0000	1.4050e+02
R3	05 7B40 0000	6.28125e+01	R3	05 7B40 0000	6.28125e+01
AR2	80 98E7		AR2	80 98E6	
AR4	80 9900		AR4	80 9910	
IR1	10		IR1	10	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098E7h	70C8000	1.4050e+02	8098E7h	70C8000	1.4050e+02
809900h	0		809900h	57B4000	6.28125e+01

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **LDI** *src, dst*

Operation *src* → *dst*

Operands *src* general addressing modes (G):

0 0 any CPU register
0 1 direct
1 0 indirect (disp = 0–255, IR0, IR1)
1 1 immediate

dst any CPU register

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 1 0 0 0 0	G	*dst*	*src*

Description The *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers. An alternate form of LDI, LDP, is used to load the data-page pointer register (DP). See the LDP instruction in Section 13.6.2 *Optional Assembler Syntax* beginning on page 13-34.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `LDI *–AR1(IR0),R5`

	Before Instruction			After Instruction	
R5	00 0000 03C5	965	R5	00 0000 0026	38
AR1	2C		AR1	2C	
IR0	5		IR0	5	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
27h	26	38	27h	26	38

Syntax

LDI*cond* *src, dst*

Operation

If *cond* is true:
src → *dst,*

Else:
dst is unchanged.

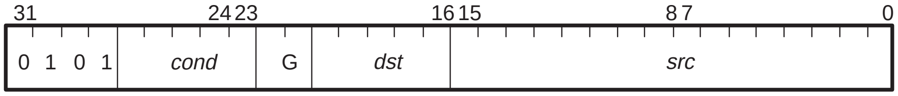
Operands

src general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode



Description

If the condition is true, the *src* operand is loaded into the *dst* register. Otherwise, the *dst* register is unchanged. Regardless of the condition, a read of the *src* takes place. The *dst* and *src* operands are assumed to be signed integers.

The ’C3x provides 20 condition codes that can be used with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Note that an LDIU (load integer unconditionally) instruction is useful for loading R7–R0 without affecting the condition flags. Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
LDIZ *AR0++,R6
```

	Before Instruction			After Instruction	
R6	00 0000 0FE2	4,066	R6	00 0000 0FE2	4,066
AR0	80 98F0		AR0	80 98F1	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098F0h	027C	636	8098F0h	027C	636

Note: Auxiliary Register Arithmetic

The test condition does not affect the auxiliary register arithmetic. (AR modification always occurs.)

Syntax **LDII** *src, dst*

Operation Signal interlocked operation
src → *dst*

Operands *src* general addressing modes (G):

0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)

dst any CPU register

Opcode

31 24 23 16 15 8 7 0

0 0 0	0 1 0 0 0 1	G	*dst*	*src*

Description The *src* operand is loaded into the *dst* register. An interlocked operation is signaled over XF0 and XF1. The *src* and *dst* operands are assumed to be signed integers. Note that only the direct and indirect modes are allowed. See Section 7.4, *Interlocked Operations*, on page 7-13 for a detailed description.

Cycles 1 if XF = 0 (see Section 7.4 on page 7-13)

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF Unaffected
LV Unaffected
UF 0
N 1 if a negative result is generated; 0 otherwise
Z 1 if a 0 result is generated; 0 otherwise
V 0
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `LDII @985Fh,R3`

	Before Instruction		After Instruction
R3	`00 0000 0000`	R3	`00 0000 00DC`
DP	`80`	DP	`80`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`0`
Data memory			
80985Fh	`0DC`	80985Fh	`0DC`

Syntax

  LDI *src*2, *dst2*
|| **LDI** *src1, dst1*

Operation

  src2 → *dst2*
|| *src1* → *dst1*

Operands

src1 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1)
dst2 register (R*n*2, 0 ≤ *n*2 ≤ 7)

> This instruction's operands have been augmented on the following devices:
>
> ❑ 'C31 silicon revision 6.0 or greater
> ❑ 'C32 silicon revision 2.0 or greater
>
> *src1* indirect (*disp* = 0, 1, IR0, IR1)
> *dst1* register (R*n*1, 0 ≤ *n*1 ≤ 7)
> *src2* indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
> *dst2* register (R*n*2, 0 ≤ *n*2 ≤ 7)

Opcode

31	24	23		16	15 8	7 0
1 1	0 0 0 1 1 1	*dst*2	*dst*1	0 0 0	*src*1	*src*2

Description

Two integer loads are performed in parallel. The assembler issues a warning if the LDIs load the same register. The result is that of LDI *src2, dst2.*

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
        LDI *–AR1(1),R7
||      LDI *AR7++(IR0),R1
```

	Before Instruction		After Instruction	
R1	00 0000 0000		00 0000 02EE	750
R7	00 0000 0000		00 0000 00FA	250
AR1	80 9826		80 9826	
AR7	80 98C8		80 98D8	
IR0	10		10	
LUF	0		0	
LV	0		0	
UF	0		0	
N	0		0	
Z	0		0	
V	0		0	
C	0		0	
Data memory				
809825h	0FA	250	0FA	250
8098C8h	2EE	750	2EE	750

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

LDI *src2, dst1*
|| **STI** *src3, dst2*

Operation

src2 → *dst1*
|| *src3* → *dst2*

Operands

src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented on the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16	15 8	7 0
1 1	0 1 1 0 1	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

An integer load and an integer store are performed in parallel. If *src2* and *dst2* point to the same location, *src2* is read before the *dst2* is written.

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
        LDI *–AR1(1),R2
||      STI R7,*AR5++(IR0)
```

	Before Instruction		After Instruction	
R2	00 0000 0000		00 0000 00DC	220
R7	00 0000 0035	53	00 0000 0035	53
AR1	80 98E7		80 98E7	
AR5	80 982C		80 9834	
IR0	8		8	
LUF	0		0	
LV	0		0	
UF	0		0	
N	0		0	
Z	0		0	
V	0		0	
C	0		0	
Data memory				
8098E6h	0DC	220	0DC	220
80982Ch	0		35	53

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **LDM** *src, dst*

Operation *src* (*man*) → *dst* (*man*)

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31–29	28–23	22–21	20–16	15–0
0 0 0	0 1 0 0 1 0	G	*dst*	*src*

Description The mantissa field of the *src* operand is loaded into the mantissa field of the *dst* register. The *dst* exponent field is not modified. The *src* and *dst* operands are assumed to be floating-point numbers. If the *src* operand is from memory, the entire memory contents are loaded as the mantissa. If immediate addressing mode is used, bits 15–12 of the instruction word are forced to 0 by the assembler.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `LDM 156.75,R2 (156.75 = 071CC00000h)`

	Before Instruction		After Instruction	
R2	00 0000 0000	R2	00 1CC0 0000	1.22460938e+00
LUF	0	LUF	0	
LV	0	LV	0	
UF	0	UF	0	
N	0	N	0	
Z	0	Z	0	
V	0	V	0	
C	0	C	0	

Syntax **LDP** *src,* DP

Operation *src* → data-page pointer

Operands *src* is the 8 MSBs of the absolute 24-bit source address (*src*).
The “DP” in the operand is optional.

Opcode

31 24 23 16 15 8 7 0

0 0 0	0 1 0 0 0 0	1 1	1 0 0 0 0	0 0 0 0 0 0 0 0	*src*

Description This pseudo-op is an alternate form of the LDUI instruction, except that LDP is always in the immediate addressing mode. The *src* operand field contains the eight MSBs of the absolute 24-bit *src* address (essentially, only bits 23–16 of *src* are used). These eight bits are loaded into the eight LSBs of the data-page pointer.

The eight LSBs of the pointer are used in direct addressing as a pointer to the page of data being addressed. There is a total of 256 pages, each page 64K words long. Bits 31–8 of the pointer are reserved and should be kept set to 0.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
LDP @809900h, DP
```

or

```
LDP @809900h
```

	Before Instruction		After Instruction
DP	065	DP	080
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0

Syntax **LOPOWER** (supported by: ’LC31 and ’C32, ’C31 silicon revision 5.0 or greater, ’C30 silicon revision 7.0 or greater)

Operation H1 → *H1/16*

Operands None

Opcode

31	23	0
0 0 0	1 0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Description The device continues to execute instructions, but at the reduced rate of the CLKIN frequency divided by 16 (that is, in LOPOWER mode, a ’C3x device that supports this mode with a CLKIN frequency of 32 MHz performs in the same way as a 2-MHz ’C3x device, which has an instruction-cycle time of 1000 ns). This allows for low-power operation.

The ’C3x CPUs slow down during the read phase of the LOPOWER instruction. To exit the LOPOWER power-down mode, invoke the MAXSPEED instruction (opcode = 1080 0000 h). The ’C3x resumes full-speed operation during the read phase of the MAXSPEED instruction.

> **CAUTION**
>
> **Do not run the IDLE2 instruction in the LOPOWER mode.**

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
LOPOWER   ; The processor slows down operation to
          ; 1/16th of the H1 clock.
```

Syntax **LSH** *count, dst*

Operation If *count* ≥ 0:
dst << *count* → *dst*

Else:
dst >> |*count* | → *dst*

Operands *count* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24	23	16	15 8 7 0
0 0 0	0 1 0 0 1 1	G	*dst*	*count*

Description The seven LSBs of the *count* operand are used to generate the 2s-complement shift count. If the *count* operand is greater than 0, the *dst* operand is left shifted by the value of the *count* operand. Low-order bits shifted in are zero filled, and high-order bits are shifted out through the carry (C) bit.

Logical left shift:

C ← *dst* ← 0

If the *count* operand is less than 0, the *dst* is right shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are zero filled as they are shifted to the right. Low-order bits are shifted out through the C bit.

Logical right shift:

0 → *dst* → C

If the *count* operand is 0, no shift is performed, and the C bit is set to 0. The *count* operand is assumed to be a signed integer, and the *dst* operand is assumed to be an unsigned integer.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 output is generated; 0 otherwise
V	0
C	Set to the value of the last bit shifted out; 0 for a shift *count* of 0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1 `LSH  R4,R7`

	Before Instruction			After Instruction	
R4	00 0000 0018	24	R4	00 0000 0018	24
R7	00 0000 02AC		R7	00 AC00 0000	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	1	
C	0		C	0	

Example 2 `LSH *–AR5(IR1),R5`

	Before Instruction			After Instruction	
R5	00 12C0 0000		R5	00 0001 2C00	
AR5	80 9908		AR5	80 9908	
IR0	4		IR0	4	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809904h	0FFFFFFF4	–12	809904h	0FFFFFFF4	–12

Syntax **LSH3** *count, src, dst*

Operation If *count* ≥ 0:
  src << *count* → *dst*

Else:
  src >> |*count* | → *dst*

Operands *src* 3-operand addressing modes (T):

0 0	any CPU register
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (*disp* = 0, 1, IR0, IR1)

count 3-operand addressing modes (T):

0 0	any CPU register
0 1	any CPU register
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 1 0 0 0	T	*dst*	*src*	*count*

Description The seven LSBs of the *count* operand are used to generate the 2s-complement shift count.

If the *count* operand is greater than 0, a copy of the *src* operand is left shifted by the value of the *count* operand, and the result is written to the *dst*. (The *src* is not changed.) Low-order bits shifted in are zero filled, and high-order bits are shifted out through the carry (C) bit.

Logical left shift:

C ← *src* ← 0

If the *count* operand is less than 0, the *src* operand is right shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are 0-filled as they are shifted to the right. Low-order bits are shifted out through the C bit.

Logical right shift:

0 → *src* → C

If the *count* operand is 0, no shift is performed, and the C bit is set to 0. The *count* operand is assumed to be a signed integer. The *src* and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF Unaffected
LV Unaffected
UF 0
N MSB of the output
Z 1 if a 0 output is generated; 0 otherwise
V 0
C Set to the value of the last bit shifted out; 0 for a shift *count* of 0; unaffected if *dst* is not R7–R0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
LSH3 R4,R7,R2
```

	Before Instruction			After Instruction	
R2	00 0000 0000		R2	00 AC00 0000	
R4	00 0000 0018	24	R4	00 0000 0018	24
R7	00 0000 02AC		R7	00 0000 02AC	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	1	
C	0		C	0	

Example 2

```
LSH3 *–AR4(IR1),R5,R3
```

	Before Instruction			After Instruction	
R3	00 0000 0000		R3	00 0001 2C00	
R5	00 12C0 0000		R5	00 12C0 0000	
AR4	80 9908		AR4	80 9908	
IR1	4		IR1	4	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809904h	0FFFFFFF4	–12	809904h	0FFFFFFF4	–12

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

  LSH3 c*ount, src2, dst1*
|| **STI** *src3*, *dst2*

Operation

If *count* ≥ 0:
  src2 << *count* → *dst1*
Else:
  src2 >> |*count* | → *dst1*
|| *src3* → *dst2*

Operands

count register (R*n*1, 0 ≤ *n*1 ≤ 7)
src1 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*3, 0 ≤ *n*3 ≤ 7)
src2 register (R*n*4, 0 ≤ *n*4 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

This instruction’s operands have been augmented in the following devices:

- ’C31 silicon revision 6.0 or greater
- ’C32 silicon revision 2.0 or greater

src2 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3 register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	0 1 1 1 1 0	*dst*1	*count*	*src*3	*dst*2	*src*2

Description

The seven LSBs of the *count* operand are used to generate the 2s-complement shift count.

If the *count* operand is greater than 0, a copy of the *src2* operand is left shifted by the value of the *count* operand, and the result is written to the *dst1*. (The *src2* is not changed.) Low-order bits shifted in are zero filled, and high-order bits are shifted out through the carry (C) bit.

Logical left shift:

  C ← *src2* ← 0

If the *count* operand is less than 0, the *src2* operand is right shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are 0-filled as they are shifted to the right. Low-order bits are shifted out through the C bit.

Logical right shift:

0 → *src2* → C

If the *count* operand is 0, no shift is performed, and the carry bit is set to 0.

The *count* operand is assumed to be a 7-bit signed integer, and the *src2* and *dst1* operands are assumed to be unsigned integers. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (LSH3) writes to the same register, STI accepts as input the contents of the register before it is modified by the LSH3.

If *src2* and *dst2* point to the same location, *src2* is read before *dst2* is written.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 output is generated; 0 otherwise
V	0
C	Set to the value of the last bit shifted out; 0 for a shift *count* of 0

Mode Bit **OVM** Operation is affected by OVM bit value.

Example 1

```
   LSH3    R2,*++AR3(1),R0
|| STI     R4,*–AR5
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 AC00 0000	
R2	00 0000 0018	24	R2	00 0000 0018	24
R4	00 0000 00DC	220	R4	00 0000 00DC	220
AR3	80 98C2		AR3	80 98C3	
AR5	80 98A3		AR5	80 98A3	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098C3h	0AC		8098C3h	0AC	
8098A2h	0		8098A2h	0DC	220

Example 2

```
  LSH3   R7,*AR2––(1),R2
||STI    R0,*+AR0(1)
```

	Before Instruction			After Instruction	
R0	00 0000 012C	300	R0	00 0000 012C	300
R2	00 0000 0000		R2	00 0002 C000	
R7	00 FFFF FFF4	–12	R7	00 FFFF FFF4	–12
AR0	80 98B7		AR0	80 98B7	
AR2	80 9863		AR2	80 9862	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809863h	2C000000		809863h	2C000000	
8098B8h	0		8098B8h	12C	300

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **MAXSPEED** (supported by ’C31, ’C32, ’C31 silicon revision 5.0 or greater, ’C30 silicon revision 7.0 or greater)

Operation *H1/16* → *H1*

Operands None

Opcode

31								23							16	15							8	7							0
0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Description Exits LOPOWER power-down mode (invoked by LOPOWER instruction with opcode 10800001h). The ’LC31 or ’C32 resumes full-speed operation during the read phase of the MAXSPEED instruction.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
MAXSPEED  ; The processor resumes full-speed operation.
```

Syntax **MPYF** *src, dst*

Operation *dst* × *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 1 0 1 0 0	G	*dst*	*src*

Description The product of the *dst* and *src* operands is loaded into the *dst* register. The *src* operand is assumed to be a single-precision floating-point number, and the *dst* operand is an extended-precision floating-point number.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `MPYF R0,R2`

	Before Instruction			**After Instruction**	
R0	07 0C80 0000	1.4050e+02	R0	07 0C80 0000	1.4050e+02
R2	03 4C20 0000	1.27578125e+01	R2	0A 600F 2000	1.79247266e+03
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Syntax **MPYF3** *src2, src1, dst*

Operation *src1* × *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

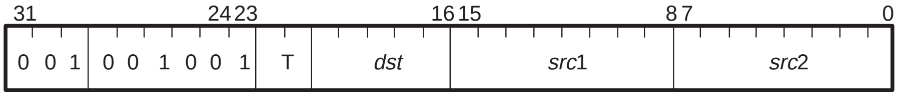
0 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 7)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode



Description The product of the *src1* and *src2* operands is loaded into the *dst* register. The *src1* and *src2* operands are assumed to be single-precision floating-point numbers, and the *dst* operand is an extended-precision floating-point number.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
MPYF3 R0,R7,R1
```

	Before Instruction			After Instruction	
R0	05 7B40 0000	6.281250e+01	R0	05 7B40 0000	6.281250e+01
R1	00 0000 0000		R1	0D 306A 3000	1.12905469e+04
R7	07 33C0 0000	1.79750e+02	R7	07 33C0 0000	1.79750e+02
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Example 2

```
MPYF3 *+AR2(IR0),R7,R2
```

or

```
MPYF3 R7,*+AR2(IR0),R2
```

	Before Instruction			After Instruction	
R2	00 0000 0000		R2	0D 09E4 A000	8.82515625e+03
R7	05 7B40 0000	6.281250e+01	R7	05 7B40 0000	6.281250e+01
AR2	80 9800		AR2	80 9800	
IR0	12A		IR0	12A	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80992Ah	70C8000	1.4050e+02	80992Ah	70C8000	1.4050e+02

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

MPYF3 *srcA, srcB, dst1*
|| **ADDF3** *srcC, srcD, dst2*

Operation

$srcA \times srcB \rightarrow dst1$
|| $srcC + srcD \rightarrow dst2$

Operands

srcA *srcB* *srcC* *srcD*	Any two indirect (*disp* = 0, 1 IR0, IR1) Any two register ($0 \leq \mathrm{R}n \leq 7$)
dst1	register (*d*1): 0 = R0 1 = R1
dst2	register (*d*2): 0 = R2 1 = R3
src1	register (R*n*, $0 \leq n \leq 7$)
src2	register (R*n*, $0 \leq n \leq 7$)
src3	indirect (*disp* = 0, 1, IR0, IR1)
src4	indirect (disp = 0–255, IR0, IR1)
P	parallel addressing modes ($0 \leq \mathrm{P} \leq 3$)

This instruction’s operands have been augmented in the following devices:

❑ ’C31 silicon version 6.0 or greater
❑ ’C32 silicon version 2.0 or greater

srcA, *srcB*, *srcC*, *srcD* can be one of the following combinations:

Register **(0 ≤ R*n* ≤ 7)**	**Indirect** **(*disp* = 0, 1, IR0, IR1)**	**Any CPU Register**
2	2	–
2	1	1
2	–	2

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

src1 register (R*n*, 0 ≤ *n* ≤ 7)
src2 register (R*n*, 0 ≤ *n* ≤ 7)
src3 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
src4 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register

P parallel addressing modes (0 ≤ P ≤ 3)

Version 4.7 or earlier of TMS320 floating-point code-generation tools

P	*srcA*	*srcB*	*srcD*	*srcC*
00	*src4* ×	*src3*,	*src1* +	*src2*
01	*src3* ×	*src1*,	*src4* +	*src2*
10	*src1* ×	*src2*,	*src3* +	*src4*
11	*src3* ×	*src1*,	*src2* +	*src4*

Version 5.0 or later

P	*srcA*	*srcB*	*srcD*	*srcC*
00	*src3* ×	*src4*,	*src1* +	*src2*
01	*src3* ×	*src1*,	*src4* +	*src2*

10	*src1* × *src2*, *src3* + *src4*
11	*src3* × *src1*, *src2* + *src4*

Opcode

31		24	23			16	15 8	7 0
1 0	0 0 0 0	P	d1	d2	*src*1	*src*2	*src*3	*src*4

Description A floating-point multiplication and a floating-point addition are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (MPYF3) reads from a register and the operation being performed in parallel (ADDF3) writes to the same register, then MPYF3 accepts the contents of the register as input before it is modified by the ADDF3.

Any combination of addressing modes can be coded for the four possible source operands as long as two are coded as indirect and two are coded as register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used, and the P field is encoded accordingly.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles 1 (see *Note: Cycle Count*)

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	0
Z	0
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
             MPYF3     *AR5++(1),*––AR1(IR0),R0
||           ADDF3     R5,R7,R3
```

Note: Cycle Count

One cycle if:

- *src3* and *src4* are in internal memory
- *src3* is in internal memory and *src4* is in external memory

Two cycles if:

- *src3* is in external memory and *src4* is in internal memory
- *src3* and *src4* are in external memory

For more information see Section 8.5, *Clocking Memory Accesses,* on page 8-24.

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	04 6718 0000	2.88867188e+01
R3	00 0000 0000	6.281250e+01	R3	08 2020 0000	3.20250e+02
R5	07 33C0 0000	1.79750e+02	R5	07 33C0 0000	1.79750e+02
R7	07 0C80 0000	1.4050e+02	R7	07 0C80 0000	1.4050e+02
AR1	80 98A8		AR1	80 98A4	
AR5	80 98C5		AR5	80 98C6	
IR0	4		IR0	4	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098C5h	34C0000	1.2750e+01	8098C5h	34C0000	1.2750e+01
8098A4h	1110000	2.265625e+00	8098A4h	1110000	2.265625e+00

Syntax

    MPYF3 *src2, src1, dst*
|| **STF** *src3, dst2*

Operation

$src1 \times src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*3, 0 ≤ *n*3 ≤ 7)
src3 register (R*n*4, 0 ≤ *n*4 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

This instruction’s operands have been augmented in the following devices:

- ❑ ’C31 silicon revision 6.0 or greater
- ❑ ’C32 silicon revision 2.0 or greater

src1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1 register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3 register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	0 1 1 1 1	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

A floating-point multiplication and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (MPYF3) writes to a register and the operation being performed in parallel (STF) reads from the same register, then STF accepts the contents of the register as input before it is modified by the MPYF3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF	1 if a floating-point underflow occurs; 0 unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
          MPYF3     *–AR2(1),R7,R0
||        STF       R3,*AR0––(IR0)
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	0D 09E4 A000	8.82515625e+03
R3	08 6B28 0000	4.7031250e+02	R3	08 6B28 0000	4.7031250e+02
R7	05 7B40 0000	6.281250e+01	R7	05 7B40 0000	6.281250e+01
AR0	80 9860		AR0	80 9858	
AR2	80 982B		AR2	80 982B	
IR0	8		IR0	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80982Ah	70C8000	1.4050e+02	80982Ah	70C8000	1.4050e+02
809860h	0		809860h	86B280000	4.7031250e+02

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

```
   MPYF3  srcA, srcB, dst1
|| SUBF3  srcC, srcD, dst2
```

Operation

srcA × *srcB* → *dst1*
|| *srcD* – *srcC* → *dst2*

Operands

srcA
srcB
srcC
srcD

Any two indirect (*disp* = 0, 1, IR0, IR1)
Any two register (0 ≤ R*n* ≤ 7)

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

src1	register (R*n*, 0 ≤ *n* ≤ 7)
src2	register (R*n*, 0 ≤ *n* ≤ 7)
src3	indirect (*disp* = 0, 1, IR0, IR1)
src4	indirect (*disp* = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

This instruction’s operands have been augmented in the following devices:

❑ ’C31 silicon version 6.0 or greater
❑ ’C32 silicon version 2.0 or greater

srcA, *srcB*, *srcC*, *srcD* can be one of the following combinations:

Register **(0 ≤ R*n* ≤ 7)**	**Indirect** **(*disp* = 0, 1, IR0, IR1)**	**Any CPU Register**
2	2	–
2	1	1
2	–	2

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

src1 register (R*n*, 0 ≤ *n* ≤ 7)
src2 register (R*n*, 0 ≤ *n* ≤ 7)
src3 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
src4 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register

P parallel addressing modes (0 ≤ P ≤ 3)

Version 4.7 or earlier of TMS320 floating-point code-generation tools

<u>P</u>	<u>*srcA*</u> <u>*srcB*</u> <u>*srcD*</u> <u>*srcC*</u>
00	*src4* × *src3, src1* – *src2*
01	*src3* × *src1, src4* – *src2*
10	*src1* × *src2, src3* – *src4*
11	*src3* × *src1, src2* – *src4*

Version 5.0 or later

<u>P</u>	<u>*srcA*</u> <u>*srcB*</u> <u>*srcD*</u> <u>*srcC*</u>
00	*src3* × *src4, src1* – *src2*
01	*src3* × *src1, src4* – *src2*
10	*src1* × *src2, src3* – *src4*
11	*src3* × *src1, src2* – *src4*

Opcode

31		24	23			16	15 8	7 0
1 0	0 0 0 1	P	d1	d2	*src*1	*src*2	*src*3	*src*4

Description

A floating-point multiplication and a floating-point subtraction are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (MPYF3) reads from a register and the operation being performed in parallel (SUBF3) writes to the same register, MPYF3 accepts as input the contents of the register before it is modified by the SUBF3.

Any combination of addressing modes can be coded for the four possible source operands as long as two are coded as indirect and two are coded register. The assignment of the source operands *srcA – srcD* to the *src1 – src4* fields varies, depending on the combination of addressing modes used, and the P field is encoded accordingly.

Cycles

1

Note: Cycle Count

One cycle if:

- ❑ *src3* and *src4* are in internal memory
- ❑ *src3* is in internal memory and *src4* is in external memory

Two cycles if:

- ❑ *src3* is in external memory and *src4* is in internal memory
- ❑ *src3* and *src4* are in external memory

For more information see Section 8.5, *Clocking Memory Accesses,* on page 8-24.

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF 1 if a floating-point underflow occurs; unchanged otherwise
LV 1 if a floating-point overflow occurs; unchanged otherwise
UF 1 if a floating-point underflow occurs; 0 otherwise
N 0
Z 0
V 1 if a floating-point overflow occurs; 0 otherwise
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
        MPYF3   R5,*++AR7(IR1),R0
||      SUBF3   R7,*AR3––(1),R2
```

or

```
        MPYF3   *++AR7(IR1), R5,R0
||      SUBF3   R7,*AR3––(1),R2
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	04 6718 0000	2.88867188e+01
R2	00 0000 0000		R2	05 E300 0000	–3.9250e+01
R5	03 4C00 0000	1.2750e+01	R5	03 4C00 0000	1.2750e+01
R7	07 33C0 0000	1.79750e+02	R7	07 33C0 0000	1.79750e+02
AR3	80 98B2		AR3	80 98B1	
AR7	80 9904		AR7	80 990C	
IR1	8		IR1	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80990Ch	1110000	2.250e+00	80990Ch	1110000	2.250e+00
8098B2h	70C8000	1.4050e+02	8098B2h	70C8000	1.4050e+02

Syntax **MPYI** *src, dst*

Operation *dst* × *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24 23		16	15 8 7 0
0 0 0	0 1 0 1 0 1	G	*dst*	*src*

Description The product of the *dst* and *src* operands is loaded into the *dst* register. The *src* and *dst* operands, when read, are assumed to be 24-bit signed integers. The result is assumed to be a 48-bit signed integer. The output to the *dst* register is the 32 LSBs of the result.

Integer overflow occurs when any of the 16 MSBs of the 48-bit result differs from the MSB of the 32-bit output value.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `MPYI R1,R5`

	Before Instruction			After Instruction	
R1	00 0033 C251	3,392,081	R1	00 0033 C251	3,392,081
R5	00 0078 B600	7,910,912	R5	00 E21D 9600	–501,377,536
LUF	0		LUF	0	
LV	0		LV	1	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	1	
C	0		C	0	

Syntax **MPYI3** *src2, src1, dst*

Operation $src1 \times src2 \rightarrow dst$

Operands *src1* 3-operand addressing modes (T):

0 0	any CPU register
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	any CPU register
0 1	any CPU register
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16	15 8	7 0
0 0 1	0 0 1 0 1 0	T	*dst*	*src*1	*src*2

Description The product of the *src1* and *src2* operands is loaded into the *dst* register. The *src1* and *src2* operands are assumed to be 24-bit signed integers. The result is assumed to be a signed 48-bit integer. The output to the *dst* register is the 32 LSBs of the result.

Integer overflow occurs when any of the 16 MSBs of the 48-bit result differs from the MSB of the 32-bit output value.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example 1

```
MPYI3 *AR4,*–AR1(1),R2
```

	Before Instruction			After Instruction	
R2	00 0000 0000		R2	00 0000 94AC	38,060
AR1	80 98F3		AR1	80 98F3	
AR4	80 9850		AR4	80 9850	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809850h	0AD	173	809850h	0AD	173
8098F2h	0DC	220	8098F2h	0DC	220

Example 2

```
MPYI3     *––AR4(IR0),R2,R7
```

	Before Instruction			After Instruction	
R2	00 0000 00C8	200	R2	00 0000 00C8	200
R7	00 0000 0000		R7	00 0000 2710	10,000
AR4	80 99F8		AR4	80 99F0	
IR0	8		IR0	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8099F0h	32	50	8099F0h	32	50

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

```
   MPYI3   srcA, srcB, dst1
|| ADDI3   srcC, srcD, dst2
```

Operation

$srcA \times srcB \rightarrow dst1$

|| $srcD + srcC \rightarrow dst2$

Operands

srcA *srcB* *srcC* *srcD*	Any two indirect (*disp* = 0, 1, IR0, IR1) Any two register (0 ≤ R*n* ≤ 7)

srcA, *srcB*, *srcC*, *srcD* can be one of the following combinations:

dst1	register (*d1*): 0 = R0 1 = R1
dst2	register (*d2*): 0 = R2 1 = R3
src1	register (R*n*, 0 ≤ *n* ≤ 7)
src2	register (R*n*, 0 ≤ *n* ≤ 7)
src3	indirect (*disp* = 0, 1, IR0, IR1)
src4	indirect (*disp* = 0, 1, IR0, IR1)
P	parallel addressing modes (0 ≤ P ≤ 3)

This instruction’s operands have been augmented in the following devices:

- ❑ ’C31 silicon version 6.0 or greater
- ❑ ’C32 silicon version 2.0 or greater

srcA, *srcB*, *srcC*, *srcD* can be one of the following combinations:

Register **(0 ≤ R*n* ≤ 7)**	**Indirect** **(*disp* = 0,1,IR0,IR1)**	**Any CPU Register**
2	2	–
2	1	1
2	–	2

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

src1 register (R*n*, 0 ≤ *n* ≤ 7)
src2 register (R*n*, 0 ≤ *n* ≤ 7)
src3 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
src4 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register

P parallel addressing modes (0 ≤ P ≤ 3)

Version 4.7 or earlier of TMS320 floating-point code-generation tools

P	*srcA*		*srcB*	*srcD*		*srcC*
00	*src4*	×	*src3,*	*src1*	+	*src2*
01	*src3*	×	*src1,*	*src4*	+	*src2*
10	*src1*	×	*src2,*	*src4*	+	*src4*
11	*src3*	×	*src1,*	*src2*	+	*src4*

Version 5.0 or later

P	*srcA*		*srcB*	*srcD*		*srcC*
00	*src3*	×	*src4,*	*src1*	+	*src2*
01	*src3*	×	*src1,*	*src4*	+	*src2*
10	*src1*	×	*src2,*	*src3*	+	*src4*
11	*src3*	×	*src1,*	*src2*	+	*src4*

Opcode

31		24	23			16	15 8	7 0
1 0	0 0 1 0	P	d1	d2	*src*1	*src*2	*src*3	*src*4

Description

An integer multiplication and an integer addition are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYI3) reads from a register and the operation being performed in parallel (ADDI3) writes to the same register, MPYI3 accepts the contents of the register as input before it is modified by the ADDI3.

Any combination of addressing modes can be coded for the four possible source operands as long as two are coded as indirect and two are coded as register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used, and the P field is encoded accordingly. To simplify processing when the order is not significant, the assembler may change the order of operands in commutative operations.

Cycles

1 (see *Note: Cycle Count* on page 13–167)

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	0
Z	0
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit

OVM Operation is affected by OVM bit value.

Example

```
         MPYI3     R7,R4,R0
||       ADDI3     *–AR3,*AR5––(1),R3
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 0000 07D0	2000
R3	00 0000 0000		R3	00 0000 0000	
R4	00 0000 0064	100	R4	00 0000 0064	100
R7	00 0000 0014	20	R7	00 0000 0014	20
AR3	80 981F		AR3	80 981F	
AR5	80 996E		AR5	80 996D	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80981Eh	0FFFFFFCB	–53	80981Eh	0FFFFFFCB	–53
80996Eh	35	53	80996Eh	35	53

Note: Cycle Count

One cycle if:

- ❑ *src3* and *src4* are in internal memory
- ❑ *src3* is in internal memory and *src4* is in external memory

Two cycles if:

- ❑ *src3* is in external memory and *src4* is in internal memory
- ❑ *src3* and *src4* are in external memory

For more information see Section 8.5, *Clocking Memory Accesses,* on page 8-24.

Syntax

    MPYI3 *src2, src1, dst1*
|| **STI** *src3, dst2*

Operation

src1 × *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*3, 0 ≤ *n*3 ≤ 7)
src3 register (R*n*4, 0 ≤ *n*4 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

This instruction’s operands have been augmented in the following devices:

- ❑ ’C31 silicon revision 6.0 or greater
- ❑ ’C32 silicon revision 2.0 or greater

src1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1 register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3 register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	1 0 0 0 0	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

An integer multiplication and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (MPYI3) writes to the same register, STI accepts as input the contents of the register before it is modified by the MPYI3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Integer overflow occurs when any of the 16 MSBs of the 48-bit result differ from the MSB of the 32-bit output value.

Cycles

1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example

```
        MPYI3     *++AR0(1),R5,R7
||      STI       R2,*–AR3(1)
```

	Before Instruction			After Instruction	
R2	00 0000 00DC	220	R2	00 0000 00DC	220
R5	00 0000 0032	50	R5	00 0000 0032	50
R7	00 0000 0000		R7	00 0000 2710	10000
AR0	80 995A		AR0	80 995B	
AR3	80 982F		AR3	80 982F	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80995Bh	0C8	200	80995Bh	0C8	200
80982Eh	0		80982Eh	ODC	220

Note: Cycle Count

See Section 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

MPYI3 *srcA, srcB, dst1*
|| **SUBI3** *srcC, srcD, dst2*

Operation

srcA × *srcB* → *dst1*
|| *srcD* – *srcC* → *dst2*

Operands

srcA
srcB
srcC
srcD

Any two indirect (*disp* = 0, 1, IR0, IR1)
Any two register (0 ≤ R*n* ≤ 7)

srcA, *srcB*, *srcC*, *srcD* can be one of the following combinations:

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

src1 register (R*n*, 0 ≤ *n* ≤ 7)
src2 register (R*n*, 0 ≤ *n* ≤ 7)
src3 indirect (*disp* = 0, 1, IR0, IR1)
src4 indirect (*disp* = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

This instruction’s operands have been augmented in the following devices:

❑ ’C31 silicon version 6.0 or greater
❑ ’C32 silicon version 2.0 or greater

srcA, *srcB*, *srcC*, *srcD* can be one of the following combinations:

Register **(0 ≤ R*n* ≤ 7)**	**Indirect** **(*disp* = 0, 1, IR0, IR1)**	**Any CPU Register**
2	2	–
2	1	1
2	–	2

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

src1 register (R*n*, 0 ≤ *n* ≤ 7)
src2 register (R*n*, 0 ≤ *n* ≤ 7)
src3 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
src4 indirect (*disp* = 0, 1, IR0, IR1) or any CPU register

P parallel addressing modes (0 ≤ P ≤ 3)

Version 4.7 or earlier of TMS320 floating-point code-generation tools

P	*srcA*		*srcB*	*srcD*		*srcC*
00	*src4*	×	*src3,*	*src1*	+	*src2*
01	*src3*	×	*src1,*	*src4*	+	*src2*
10	*src1*	×	*src2,*	*src4*	+	*src4*
11	*src3*	×	*src1,*	*src2*	+	*src4*

Version 5.0 or later

P	srcA		srcB	srcD		srcC
00	*src3*	×	*src4,*	*src1*	+	*src2*
01	*src3*	×	*src1,*	*src4*	+	*src2*
10	*src1*	×	*src2,*	*src3*	+	*src4*
11	*src3*	×	*src1,*	*src2*	+	*src4*

Opcode

31		24	23			16	15 8	7 0
1 0	0 0 1 1	P	d1	d2	*src*1	*src*2	*src*3	*src*4

Description

An integer multiplication and an integer subtraction are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (MPYI3) reads from a register and the operation being performed in parallel (SUBI3) writes to the same register, MPYI3 accepts the contents of the register as input before it is modified by the SUBI3.

Any combination of addressing modes can be coded for the four possible source operands as long as two are coded as indirect and two are coded as register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used, and the P field is encoded accordingly. To simplify processing when the order is not significant, the assembler may change the order of operands in commutative operations.

Integer overflow occurs when any of the 16 MSBs of the 48-bit result differs from the MSB of the 32-bit output value.

Cycles

1 (see *Note: Cycle Count* on page 13–173)

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	1 if an integer underflow occurs; 0 otherwise
N	0
Z	0
V	1 if an integer overflow occurs; 0 otherwise
C	Unaffected

Mode Bit

OVM Operation is affected by OVM bit value.

Example

```
          MPYI3     R2,*++AR0(1),R0
||        SUBI3     *AR5−−(IR1),R4,R2
```

or

```
          MPYI3     *++AR0(1),R2,R0
||        SUBI3     *AR5––(IR1),R4,R2
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 0000 1324	4900
R2	00 0000 0032	50	R2	00 0000 0320	800
R4	00 0000 07D0	2000	R4	00 0000 07D0	2000
AR0	80 98E3		AR0	80 98E4	
AR5	80 99FC		AR5	80 99F0	
IR1	0C		IR1	0C	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098E4h	62	98	8098E4h	62	98
8099FCh	4B0	1200	8099FCh	4B0	1200

Note: Cycle Count

One cycle if:

- ❑ *src3* and *src4* are in internal memory
- ❑ *src3* is in internal memory and *src4* is in external memory

Two cycles if:

- ❑ *src3* is in external memory and *src4* is in internal memory
- ❑ *src3* and *src4* are in external memory

For more information see Section 8.5, *Clocking Memory Accesses,* on page 8-24.

Syntax **NEGB** *src, dst*

Operation 0 – *src* – C → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31 24 23 16 15 8 7 0

0 0 0	0 1 0 1 1 0	G	*dst*	*src*

Description The difference of the 0, *src*, and C operands is loaded into the *dst* register. The *dst* and *src* are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `NEGB R5,R7`

	Before Instruction			After Instruction	
R5	00 FFFF FFCB	–53	R5	00 FFFF FFCB	–53
R7	00 0000 0000		R7	00 0000 0034	52
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	1		C	1	

Syntax **NEGF** *src, dst*

Operation 0 – *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31 24 23 16 15 8 7 0

0 0 0	0 1 0 1 1 1	G	*dst*	*src*

Description The difference of the 0 and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example

```
NEGF *++AR3(2),R1
```

	Before Instruction			After Instruction	
R1	05 7B40 0025	6.28125006e+01	R1	07 F380 0000	–1.4050e+02
AR3	80 9800		AR3	80 9802	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809802h	70C8000	1.4050e+02	809802h	70C8000	1.4050e+02

Syntax

NEGF *src2, dst1*
|| **STF** *src3, dst2*

Operation

0 – *src2* → *dst1*
|| *src3* → *dst2*

Operands

src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❏ 'C31 silicon revision 6.0 or greater
- ❏ 'C32 silicon revision 2.0 or greater

src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	1 0 0 0 1	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

A floating-point negation and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (NEGF) writes to the same register, STF accepts the contents of the register as input before it is modified by the NEGF.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
          NEGF      *AR4––(1),R7
||        STF       R2,*++AR5(1)
```

	Before Instruction			After Instruction	
R2	07 33C0 0000	1.79750e+02	R2	07 33C0 0000	1.79750e+02
R7	00 0000 0000		R7	05 84C0 0000	–6.281250e+01
AR4	80 98E1		AR4	80 98E0	
AR5	80 9803		AR5	80 9804	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098E1h	57B400000	6.281250e+01	8098E1h	57B4000	6.281250e+01
809804h	0		809804h	733C000	1.79750e+02

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **NEGI** *src, dst*

Operation 0 – *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31 …	… 24 23	…	… 16 15	… 8 7 … 0
0 0 0	0 1 1 0 0 0	G	*dst*	*src*

Description The difference of the 0 and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `NEGI 174,R5    (174 = 0AEh)`

	Before Instruction			**After Instruction**	
R5	`00 0000 00DC`	`220`	R5	`00 FFFF FF52`	`–174`
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`1`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`1`	

Syntax

```
    NEGI   src2, dst1
||  STI    src3, dst2
```

Operation

$0 - src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3 register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

> This instruction's operands have been augmented in the following devices:
>
> ❑ 'C31 silicon revision 6.0 or greater
> ❑ 'C32 silicon revision 2.0 or greater
>
> *src2* indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
> *dst1* register (R*n*1, 0 ≤ *n*1 ≤ 7)
> *src3* register (R*n*2, 0 ≤ *n*2 ≤ 7)
> *dst2* indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	1 0 0 1 0	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

An integer negation and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (NEGI) writes to the same register, STI accepts the contents of the register as input before it is modified by the NEGI.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit

OVM Operation is affected by OVM bit value.

Example

```
          NEGI      *–AR3,R2
||        STI       R2,*AR1++
```

	Before Instruction			After Instruction	
R2	00 0000 0019	25	R2	00 FFFF FF24	–220
AR1	80 98A5		AR1	80 98A6	
AR3	80 982F		AR3	80 982F	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	0	
C	0		C	1	
Data memory					
80982Eh	0DC	220	80982Eh	0DC	220
8098A5h	0		8098A5h	19	25

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **NOP** *src*

Operation No ALU or multiplier operations.
AR*n* is modified if *src* is specified in indirect mode.

Operands *src* general addressing modes (G):

0 0	register	(no operation)
1 0	indirect	(modify AR*n*, 0 ≤ *n* ≤ 7) (disp = 0–255, IR0, IR1)

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 1 1 0 0 1	G	0 0 0 0 0	*src*

Description If the *src* operand is specified in the indirect mode, the specified addressing operation is performed, and a dummy memory read occurs. If the *src* operand is omitted, no operation is performed.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
NOP
```

	Before Instruction		**After Instruction**
PC	3A	PC	3B

Example 2

```
NOP  *AR3––(1)
```

	Before Instruction		**After Instruction**
AR3	80 9900	AR3	80 98FF
PC	5	PC	6

Syntax **NORM** *src, dst*

Operation norm (*src*) → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, $0 \le n \le 7$)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

Opcode

31 – 29	28 – 23	22 – 21	20 – 16	15 – 0
0 0 0	0 1 1 0 1 0	G	*dst*	*src*

Description The *src* operand is assumed to be an unnormalized floating-point number; that is, the implied bit is set equal to the sign bit. The *dst* is set equal to the normalized *src* operand with the implied bit removed. The *dst* operand exponent is set to the *src* operand exponent minus the size of the left shift necessary to normalize the *src*. The *dst* operand is assumed to be a normalized floating-point number.

If *src (exp*) = –128 and *src (man*) = 0, then *dst* = 0, Z = 1, and UF = 0.

If *src* (*exp*) = –128 and *src* (*man*) ≠ 0, then *dst* = 0, Z = 0, and UF = 1.

For all other cases of the *src,* if a floating-point underflow occurs, then *dst (man*) is forced to 0 and *dst (exp*) = –128.

If *src (man*) = 0, then *dst (man*) = 0 and *dst (exp*) = –128.

Refer to Section 5.7, *Normalization Using the NORM Instruction*, on page 5-37 for more information.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	Unaffected
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `NORM R1,R2`

	Before Instruction		After Instruction	
R1	`04 0000 3AF5`	R1	`04 0000 3AF5`	
R2	`07 0C80 0000`	R2	`F2 6BD4 0000`	`1.12451613e – 04`
LUF	`0`	LUF	`0`	
LV	`0`	LV	`0`	
UF	`0`	UF	`0`	
N	`0`	N	`0`	
Z	`0`	Z	`0`	
V	`0`	V	`0`	
C	`0`	C	`0`	

Syntax **NOT** *src, dst*

Operation ~*src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst any CPU register

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 1 1 0 1 1	G	*dst*	*src*

Description The bitwise-logical complement of the *src* operand is loaded into the *dst* register. The complement is formed by a logical NOT of each bit of the *src* operand. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `NOT @982Ch,R4`

	Before Instruction		After Instruction
R4	00 0000 0000	R4	00 FFFF A1D0
DP	080	DP	080
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	1
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
80982Ch	5E2F	80982Ch	5E2F

Syntax

	NOT	*src2, dst1*
||	**STI**	*src3, dst2*

Operation

$\sim src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src3	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	1 0 0 1 1	*dst*1	0 0 0	*src*3	*dst*2	*src*2

Description

A bitwise-logical NOT and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (NOT) writes to the same register, STI accepts the contents of the register as input before it is modified by the NOT.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
          NOT      *+AR2,R3
||        STI      R7,*––AR4 (IR1)
```

	Before Instruction			After Instruction	
R3	00 0000 0000		R3	00 FFFF F3D0	
R7	00 0000 00DC	220	R7	00 0000 00DC	220
AR2	80 99CB		AR2	80 99CB	
AR4	80 9850		AR4	80 9840	
IR1	10		IR1	10	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8099CCh	0C2F		8099CCh	0C2F	
809840h	0		809840h	0DC	220

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **OR** *src, dst*

Operation *dst* OR *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate (not sign-extended)

dst any CPU register

Opcode

31	24 23		16	15 8 7 0
0 0 0	1 0 0 0 0 0	G	*dst*	*src*

Description The bitwise-logical OR between the *src* and *dst* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
OR *++AR1(IR1),R2
```

	Before Instruction		After Instruction
R2	00 1256 0000	R2	00 1256 2BCD
AR1	80 9800	AR1	80 9804
IR1	4	IR1	4
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809804h	2BCD	809804h	2BCD

Syntax **OR3** *src2, src1, dst*

Operation *src1* OR *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 27)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 1 0 1 1	T	*dst*	*src*1	*src*2

Description The bitwise-logical OR between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2,* and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `OR3 *++AR1(IR1),R2,R7`

	Before Instruction		After Instruction
R2	00 1256 0000	R2	00 1256 0000
R7	00 0000 0000	R7	0 1256 2BCD
AR1	80 9800	AR1	80 9804
IR1	4	IR1	4
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809804h	2BCD	809804h	2BCD

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

```
   OR3   src2, src1, dst1
|| STI   src3, dst2
```

Operation

src1 OR *src2* → *dst1*
| *src3* → *dst2*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
*dst*2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31		24 23		16	15 8	7 0
1 1	1 0 1 0 0	*dst*1	*src*1	*src*3	*dst*2	*src*2

A bitwise-logical OR and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (OR3) writes to the same register, then STI accepts the contents of the register as input before it is modified by the OR3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7−R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
        OR3      *++AR2,R5,R2
||      STI      R6,*AR1−−
```

	Before Instruction			After Instruction	
R2	00 0000 0000		R2	00 0080 9800	
R5	00 0080 0000		R5	00 0080 0000	
R6	00 0000 00DC	220	R6	00 0000 00DC	220
AR1	80 9883		AR1	80 9882	
AR2	80 9830		AR2	80 9831	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809831h	9800		809831h	9800	
809883h	0		809883h	0DC	220

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **POP** *dst*

Operation *SP– – → *dst*

Operands *dst* register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 1 1 1 0 0	0 1	*dst*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Description The top of the current system stack is popped and loaded into the *dst* register (32 LSBs). The top of the stack is assumed to be a signed integer. The POP is performed with a postdecrement of the stack pointer. The exponent bits of an extended-precision register (R7–R0) are left unmodified.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `POP R3`

	Before Instruction			After Instruction	
R3	`00 0000 12DA`	`4,826`	R3	`00 FFFF 0DA4`	`–62,044`
SP	`809856`		SP	`809855`	
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`1`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	
Data memory					
809856h	`FFFF0DA4`	`–62,044`	809856h	`FFFF0DA4`	`–62,044`

Syntax **POPF** *dst*

Operation *SP– – → *dst1*

Operands *dst* register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31–29	28–23	22–21	20–16	15–0
0 0 0	0 1 1 1 0 1	0 1	*dst*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Description The top of the current system stack (32 MSBs) is popped and loaded into the *dst* register. The top of the stack is assumed to be a floating-point number. The POP is performed with a postdecrement of the stack pointer. The eight LSBs of an extended-precision register (R7–R0) are zero-filled.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
UF	0
LV	Unaffected
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `POPF R4`

	Before Instruction			After Instruction	
R4	02 5D2E 0123	6.91186578e+00	R4	5F 2C13 0200	5.32544007e+28
SP	80984A		SP	809849	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80984Ah	5F2C1302	5.32544007e+28	80984Ah	5F2C1302	5.32544007e+28

Syntax **PUSH** *src*

Operation *src* → *++SP

Operands *src* register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	0 1 1 1 1 0	0 1	*src*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Description The contents of the *src* register (32 LSBs) are pushed on the current system stack. The *src* is assumed to be a signed integer. The PUSH is performed with a preincrement of the stack pointer. The integer or mantissa portion of an extended-precision register (R7–R0) is saved with this instruction.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
PUSH R6
```

	Before Instruction			**After Instruction**	
R6	02 5C12 8081	633,415,688	R6	02 5C12 8081	633,415,688
SP	8098AE		SP	8098AF	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098AFh	0	−62,044	8098AFh	5C128081	1,544,716,417

Syntax **PUSHF** *src*

Operation *src* → *++SP

Operands *src* register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24	23	16	15 8 7 0
0 0 0	0 1 1 1 1 1	0 1	*src*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Description The contents of the *src* register (32 MSBs) are pushed on the current system stack. The *src* is assumed to be a floating-point number. The PUSH is performed with a preincrement of the stack pointer. The eight LSBs of the mantissa are not saved. (Note the difference in R2 and the value on the stack in the example below.)

Cycles 1

Status Bits

LUF Unaffected
LV Unaffected
UF Unaffected
N Unaffected
Z Unaffected
V Unaffected
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `PUSHF R2`

	Before Instruction			**After Instruction**	
R2	`02 5C12 8081`	`6.87725854e+00`	R2	`02 5C12 8081`	`6.87725854e+00`
SP	`809801`		SP	`809802`	
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`0`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	
Data memory					
809802h	`0`		809802h	`025C1280`	`6.87725830e+00`

Syntax **RETI*cond***

Operation If *cond* is true:

*SP – – → PC
1 → ST (GIE).

Else, continue.

Operands None

Opcode

```
31                  24 23               16 15              8  7                0
| 0  1  1  1  1 | 0  0  0  0 | 0  0 |  cond  | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
```

Description A conditional return is performed. If the condition is true, the top of the stack is popped to the PC, and a 1 is written to the global interrupt enable (GIE) bit of the status register. This has the effect of enabling all interrupts for which the corresponding interrupt enable bit is a 1.

The ’C3x provides 20 condition codes that can be used with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `RETINZ`

	Before Instruction		After Instruction
PC	0456	PC	0123
SP	809830	SP	80982F
ST	0	ST	2000
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809830h	123	809830h	123

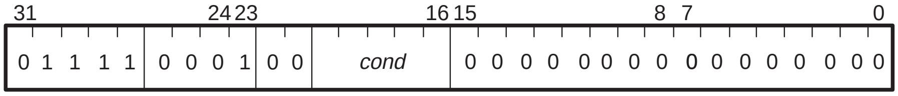
Syntax **RETS*cond***

Operation If *cond* is true:

*SP– – → PC.
Else, continue.

Operands None

Opcode



Description A conditional return is performed. If the condition is true, the top of the stack is popped to the PC.

The ’C3x provides 20 condition codes that you can use with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `RETSGE`

	Before Instruction		After Instruction
PC	`0123`	PC	`0456`
SP	`80983C`	SP	`80983B`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`0`
Data memory			
80983Ch	`456`	80983Ch	`456`

Syntax **RND** *src, dst*

Operation rnd(*src*) → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24 23		16	15 8 7 0
0 0 0	1 0 0 0 1 0	G	*dst*	*src*

Description The result of rounding the *src* operand is loaded into the *dst* register.The *src* operand is rounded to the nearest single-precision floating-point value. If the *src* operand is exactly halfway between two single-precision values, it is rounded to the most positive value.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs or the *src* operand is 0; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	Unaffected
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is affected by OVM bit value.

Example

```
RND R5,R2
```

	Before Instruction		After Instruction	
R2	`00 0000 0000`		`07 33C1 6F00`	`1.79755600e+02`
R5	`07 33C1 6EEF`	`1.79755599e+02`	`07 33C1 6EEF`	`1.79755599e+02`
LUF	0		0	
LV	0		0	
UF	0		0	
N	0		0	
Z	0		0	
V	0		0	
C	0		0	

Note: BZUF Instruction

If a BZ instruction is executed immediately following an RND instruction with a 0 operand, the branch is not performed because the zero flag is not set. To circumvent this problem, execute a BZUF instruction instead of a BZ instruction.

Syntax **ROL** *dst*

Operation *dst* left-rotated 1 bit → *dst*

Operands *dst* register (R*n*, 0 ≤ *n* ≤ 27)

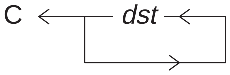
Opcode

31 24 23 16 15 8 7 0

0 0 0	1 0 0 0 1 1	1 1	*dst*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Description The contents of the *dst* operand are left rotated one bit and loaded into the *dst* register. This is a circular rotation, with the MSB simultaneously transferred into the carry (C) bit and the LSB.

Rotate left:



Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF Unaffected
LV Unaffected
UF 0
N MSB of the output
Z 1 if a 0 output is generated; 0 otherwise
V 0
C Set to the value of the bit rotated out of the high-order bit; unaffected if *dst* is not R7 – R0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `ROL R3`

	Before Instruction		After Instruction
R3	`00 8002 5CD4`	R3	`00 0004 B9A9`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`1`

Syntax **ROLC** *dst*

Operation *dst* left-rotated one bit through carry bit → *dst*

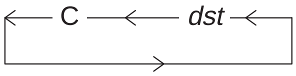
Operands *dst* register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16	15 8 7 0
0 0 0	1 0 0 1 0 0	1 1	*dst*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Description The contents of the *dst* operand are left rotated one bit through the carry (C) bit and loaded into the *dst* register. The MSB is rotated to the carry bit at the same time the carry bit is transferred to the LSB.

Rotate left through carry bit:



Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF Unaffected
LV Unaffected
UF 0
N MSB of the output
Z 1 if a 0 output is generated; 0 otherwise
V 0
C Set to the value of the bit rotated out of the high-order bit; if *dst* is not R7–R0, then C is shifted into the *dst* but not changed

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
ROLC R3
```

	Before Instruction	**After Instruction**
R3	`00 0000 0420`	`00 0000 0841`
LUF	`0`	`0`
LV	`0`	`0`
UF	`0`	`0`
N	`0`	`0`
Z	`0`	`0`
V	`0`	`0`
C	`1`	`0`

Example 2

```
ROLC R3
```

	Before Instruction		**After Instruction**
R3	`00 8000 4281`	R3	`00 0000 8502`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`1`

Syntax **ROR** *dst*

Operation *dst* right-rotated one bit through carry bit → *dst*

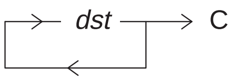
Operands *dst* register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 0 0 1 0 1	1 1	*dst*	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Description The contents of the *dst* operand are right rotated one bit and loaded into the *dst* register. The LSB is rotated into the carry (C) bit and also transferred into the MSB.

Rotate right:



Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF Unaffected
LV Unaffected
UF 0
N MSB of the output
Z 1 if a 0 output is generated; 0 otherwise
V 0
C Set to the value of the bit rotated out of the high-order bit; unaffected if *dst* is not R7–R0

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `ROR R7`

	Before Instruction		After Instruction
R7	00 0000 0421	R7	00 8000 0210
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	1
Z	0	Z	0
V	0	V	0
C	0	C	1

Syntax **RORC** *dst*

Operation *dst* right-rotated one bit through carry bit → *dst*

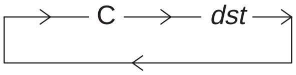
Operands *dst* register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 0 0 1 1 0	1 1	*dst*	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Description The contents of the *dst* operand are right rotated one bit through the status register's carry (C) bit. This could be viewed as a 33-bit shift. The carry bit value is rotated into the MSB of the *dst*, while at the same time the *dst* LSB is rotated into the carry bit.

Rotate right through carry bit:



Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF Unaffected
LV Unaffected
UF 0
N MSB of the output
Z 1 if a 0 output is generated; 0 otherwise
V 0
C Set to the value of the bit rotated out of the high-order bit; if *dst* is not R7 – R0, then C is shifted in but not changed

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `RORC R4`

	Before Instruction		**After Instruction**
R4	`00 8000 0081`	R4	`00 4000 0040`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`1`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`1`

Syntax

RPTB *src*

Operation

src → RE
1 → ST (RM)
Next PC → RS

Operands

src long-immediate addressing mode

Opcode

31 … 24	23 … 0
0 1 1 0 0 1 0 0	*src*

Description

RPTB allows a block of instructions to be repeated RC register + 1 times without any penalty for looping. This instruction activates the block repeat mode of updating the PC. The *src* operand is a 24-bit unsigned immediate value that is loaded into the repeat end-address (RE) register. A 1 is written into the repeat mode bit of status register ST (RM) to indicate that the PC is being updated in the repeat mode. The address of the next instruction is loaded into the repeat start-address (RS) register.

RE should be greater than or equal to RS (RE ≥ RS). Otherwise, the code does not repeat, even though the RM bit remains set to 1.

Cycles

4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
RPTB  127h
```

	Before Instruction		After Instruction
PC	`0123`	PC	`0124`
RE	`0`	RE	`127`
RS	`0`	RS	`124`
ST	`0`	ST	`100`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`0`

CAUTION

Because the block-repeat modes modify the program counter, no other instruction can modify the program counter at the same time. The following two rules apply:

Rule 1: The last instruction in the block (or the only instruction in a block of size 1) cannot be a B*cond*, BR, DB*cond*, CALL, CALL*cond*, TRAP*cond*, RETI*cond*, RETS*cond*, IDLE, IDLE2, RPTB, or RPTS. Example 7–3 on page 7-6 shows an incorrectly placed standard branch.

Rule 2: None of the last four instructions at the bottom of the block (or the only instruction in a block of size 1) can be a B*cond*D, BRD, or DB*cond*D. Example 7–4 on page 7-7 shows an incorrectly placed delayed branch.

If either rule is violated, the PC will be undefined.

Syntax **RPTS** *src*

Operation

src → RC
1 → ST (RM)
1 → S
Next PC → RS
Next PC → RE

Operands *src* general addressing modes (G):

0 0	register
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

Opcode

31	24 23		16	15 8 7 0
0 0 0	1 0 0 1 1 1	G	1 1 0 1 1	*src*

Description The RPTS instruction allows you to repeat a single instruction *src* + 1 times without any penalty for looping. Fetches can also be made from the instruction register (IR), thus avoiding repeated memory access.

The *src* operand is loaded into the repeat counter (RC). A 1 is written into the repeat mode bit of the status register ST (RM). A 1 is also written into the repeat single bit (S). This indicates that the program fetches are to be performed only from the instruction register. The next PC is loaded into the repeat end-address (RE) register and the repeat start-address (RS) register.

For the immediate mode, the *src* operand is assumed to be an unsigned integer and is not sign extended.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `RPTS AR5`

	Before Instruction		After Instruction
AR5	00 00FF	AR5	00 00FF
PC	0123	PC	0124
RC	0	RC	0FF
RE	0	RE	124
RS	0	RS	124
ST	0	ST	100
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0

CAUTION

Because the block-repeat modes modify the program counter, no other instruction can modify the program counter at the same time. Therefore, the repeated instruction cannot be a B*cond*, BR, DB*cond*, CALL, CALL*cond*, TRAP*cond*, RETI*cond*, RETS*cond*, IDLE, IDLE2, RPTB, or RPTS. If this rule is violated, the PC will be undefined.

Note:

The RPTS instruction cannot be interrupted because instruction fetches are halted.

Syntax **SIGI**

Operation Signal interlocked operation.
Wait for interlock acknowledge.
Clear interlock.

Operands None

Opcode

31							24	23							16	15							8	7							0
0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Description An interlocked operation is signaled over XF0 and XF1. After the interlocked operation is acknowledged, the interlocked operation ends. SIGI ignores the external ready signals. Refer to Section 7.4, *Interlocked Operations*, on page 7-13 for detailed information.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
SIGI      ; The processor sets XF0 to 0, idles
          ; until XF1 is set to 0, and then
          ; sets XF0 to 1.
```

Syntax **STF** *src, dst*

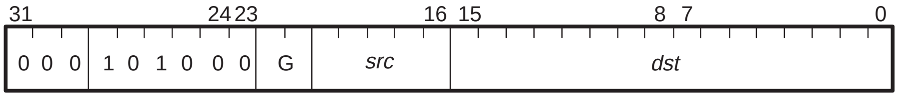
Operation *src* → *dst*

Operands *src* register (R*n*, 0 ≤ *n* ≤ 7)

dst general addressing modes (G):

0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)

Opcode



Description The *src* register is loaded into the *dst* memory location. The *src* and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `STF R2,@98A1h`

	Before Instruction			After Instruction	
R2	05 2C50 1900	4.30782204e+01	R2	05 2C50 1900	4.30782204e+01
DP	080		DP	080	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098A1h	0		8098A1h	52C5019	4.30782204e+01

Syntax **STFI** *src, dst*

Operation *src* → *dst*
Signal end of interlocked operation.

Operands *src* register (R*n*, 0 ≤ *n* ≤ 7)

dst general addressing modes (G):

0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)

Opcode

31								24 23		16 15	8 7	0
0	0	0	1	0	1	0	0	1	G	*src*	*dst*	

Description The *src* register is loaded into the *dst* memory location. An interlocked operation is signaled over pins XF0 and XF1. The *src* and *dst* operands are assumed to be floating-point numbers. Refer to Section 7.4, *Interlocked Operations*, on page 7-13 for detailed information.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `STFI   R3,*–AR4`

	Before Instruction			After Instruction	
R3	`07 33C0 0000`	`1.79750e+02`	R3	`07 33C0 0000`	`1.79750e+02`
AR4	`80 993C`		AR4	`80 993C`	
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`0`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	
Data memory					
80993Bh	`0`		80993Bh	`733C000`	`1.79750e+02`

Note:

The STFI instruction is not interruptible because it completes when ready is signaled. See Section 7.4, *Interlocked Operations,* on page 7-13.

Syntax

```
   STF  src2, dst2
|| STF  src1, dst1
```

Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
dst1	indirect (*disp* = 0, 1, IR0, IR1)
src2	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented on the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
dst1	indirect (*disp* = 0, 1, IR0, IR1)
src2	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register

Opcode

31	24	23		16	15 8	7 0
1 1	0 0 0 0 0	*src*2	0 0 0	*src*1	*dst*1	*dst*2

Description

Two STF instructions are executed in parallel. Both *src1* and *src2* are assumed to be floating-point numbers.

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
         STF       R4,*AR3––
||       STF       R3,*++AR5
```

	Before Instruction			After Instruction	
R3	07 33C0 0000	1.79750e+02	R3	07 33C0 0000	1.79750e+02
R4	07 0C80 0000	1.4050e+02	R4	07 0C80 0000	1.4050e+02
AR3	80 9835		AR3	80 9834	
AR5	80 99D2		AR5	80 99D3	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809835h	0		809835h	070C8000	1.4050e+02
8099D3h	0		8099D3h	0733C000	1.79750e+02

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **STI** *src, dst*

Operation *src* → *dst*

Operands *src* register (R*n*, 0 ≤ *n* ≤ 27)

dst general addressing modes (G):

0 1 direct
1 0 indirect (disp = 0–255, IR0, IR1)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 0 1 0 1 0	G	*src*	*dst*

Description The *src* register is loaded into the *dst* memory location. The *src* and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
STI R4,@982Bh
```

	Before Instruction			After Instruction	
R4	00 0004 2BD7	273,367	R4	00 0004 2BD7	273,367
DP	080		DP	080	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80982Bh	0E5FC	58,876	80982Bh	42BD7	273,367

Syntax **STII** *src, dst*

Operation *src* → *dst*
Signal end of interlocked operation

Operands *src* register (R*n*, 0 ≤ *n* ≤ 27)

dst general addressing modes (G):

0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 0 1 0 1 1	G	*src*	*dst*

Description The *src* register is loaded into the *dst* memory location. An interlocked operation is signaled over pins XF0 and XF1. The *src* and *dst* operands are assumed to be signed integers. Refer to Section 7.4, *Interlocked Operations*, on page 7-13 for detailed information.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `STII R1,@98AEh`

	Before Instruction		After Instruction
R1	00 0000 078D	R1	00 0000 078D
DP	080	DP	080
Data memory			
8098AEh	25C	8098AEh	78D

Note:

The STII instruction is not interruptible because it completes when ready is signaled. See Section 7.4, *Interlocked Operations,* on page 7-13.

Syntax

    STI *src2, dst2*
|| **STI** *src1, dst1*

Operation

    src2 → *dst2*
|| *src1* → *dst1*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
dst1	indirect (*disp* = 0, 1, IR0, IR1)
src2	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented on the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
dst1	indirect (*disp* = 0, 1, IR0, IR1)
src2	register (R*n*2, 0 ≤ *n*2 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register

Opcode

31	24	23		16	15 8	7 0
1 1	0 0 0 0 1	*src*2	0 0 0	*src*1	*dst*1	*dst*2

Description

Two integer stores are performed in parallel. If both stores are executed to the same address, the value written is that of STI *src2, dst2.*

Cycles

1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
        STI R0,*++AR2(IR0)
||      STI R5,*AR0
```

	Before Instruction			After Instruction	
R0	00 0000 00DC	220	R0	00 0000 00DC	220
R5	00 0000 0035	53	R5	00 0000 0035	53
AR0	80 98D3		AR0	80 98D3	
AR2	80 9830		AR2	80 9838	
IR0	8		IR0	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809838h	0		809838h	0DC	220
8098D3h	0		8098D3h	35	53

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **SUBB** *src, dst*

Operation *dst* – *src* – C → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 0 1 1 0 1	G	*dst*	*src*

Description The difference of the *dst, src,* and C operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `SUBB *AR5++(4),R5`

	Before Instruction			After Instruction	
R5	00 0000 00FA	250	R5	00 0000 0032	50
AR5	80 9800		AR5	80 9804	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	1		C	0	
Data memory					
809800h	0C7	199	809800h	0C7	199

Syntax **SUBB3** *src2, src1, dst*

Operation *src1* – *src2* – C → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 27)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7	0
0 0 1	0 0 1 1 0 0	T	*dst*	*src*1	*src*2

Description The difference among the *src1* and *src2* operands and the C flag is loaded into the *dst* register. The *src1, src2,* and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `SUBB3 R5,*AR5++(IR0),R0`

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 0000 0032	50
R5	00 0000 00C7	199	R5	00 0000 00C7	199
AR5	80 9800		AR5	80 9804	
IR0	4		IR0	4	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	1		C	0	
Data memory					
809800h	0FA	250	809800h	0FA	250

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **SUBC** *src, dst*

Operation

```
If ($dst – src \geq 0$):
      ($dst – src$ << 1) OR 1 $\rightarrow dst$
Else:
      $dst$ << 1 $\rightarrow dst$
```

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31–29	28–23 (24 23)	22–21	20–16 (16)	15–0 (15 8 7 0)
0 0 0	1 0 1 1 1 0	G	*dst*	*src*

Description The *src* operand is subtracted from the *dst* operand. The *dst* operand is loaded with a value dependent on the result of the subtraction. If (*dst* – *src*) is greater than or equal to 0, then (*dst* – *src)* is left-shifted one bit, the least significant bit is set to 1, and the result is loaded into the *dst* register. If (*dst* – *src*) is less than 0, *dst* is left-shifted one bit and loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

You can use SUBC to perform a single step of a multi-bit integer division. See the *TMS320C3x General Purpose Applications Guide* for a detailed description.

Cycles 1

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
SUBC  @98C5h,R1
```

	Before Instruction			After Instruction	
R1	00 0000 04F6	1270	R1	00 0000 00C9	201
DP	080		DP	080	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098C5h	492	1170	8098C5h	492	1170

Example 2

```
SUBC 3000,R0  (3000 = 0BB8h)
```

	Before Instruction			After Instruction	
R0	00 0000 07D0	2000	R0	00 0000 0FA0	4000
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Syntax **SUBF** *src*, *dst*

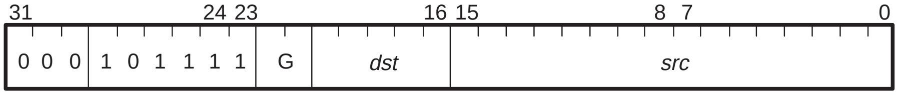
Operation *dst* – *src* → *dst*

Operands *src* general addressing modes (G):

0 0 register (R*n*, 0 ≤ *n* ≤ 7)
0 1 direct
1 0 indirect (disp = 0–255, IR0, IR1)
1 1 immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode



Description The difference between *the dst* operand and the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF 1 if a floating-point underflow occurs; unchanged otherwise
LV 1 if a floating-point overflow occurs; unchanged otherwise
UF 1 if a floating-point underflow occurs; 0 otherwise
N 1 if a negative result is generated; 0 otherwise
Z 1 if a 0 result is generated; 0 otherwise
V 1 if a floating-point overflow occurs; 0 otherwise
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
SUBF      *AR0--(IR0),R5
```

	Before Instruction			After Instruction	
R5	07 33C0 0000	1.79750000e+02	R5	05 1D00 0000	3.9250e+01
AR0	80 9888		AR0	80 9808	
IR0	80		IR0	80	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809888h	70C8000	1.4050e+02	809888h	70C8000	1.4050e+02

Syntax **SUBF3** *src2, src1, dst*

Operation *src1* – *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 7)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 7)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 1 1 0 1	T	*dst*	*src*1	*src*2

Description The difference between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1 `SUBF3 *AR0––(IR0),*AR1,R4`

	Before Instruction			After Instruction	
R4	00 0000 0000		R4	05 1D00 0000	3.9250e+01
AR0	80 9888		AR0	80 9808	
AR1	80 9851		AR1	80 9851	
IR0	80		IR0	80	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809888h	70C8000	1.4050e+02	809888h	70C8000	1.4050e+02
809851h	733C000	1.79750e+02	809851h	733C000	1.79750e+02

Example 2 `SUBF3 R7,R0,R6`

	Before Instruction			After Instruction	
R0	03 4C20 0000	1.27578125e+01	R0	03 4C20 0000	1.27578125e+01
R6	00 0000 0000		R6	05 B7C8 0000	–5.00546875e+01
R7	05 7B40 0000	6.281250e+01	R7	05 7B40 0000	6.281250e+01
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	1	
V	0		V	0	
C	0		C	0	

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

	SUBF3	*src1, src2, dst1*
||	**STF**	*src3, dst2*

Operation

src2 – *src1* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
*dst*2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31			24 23		16	15 8	7 0
1 1	1	0 1 0 1	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

A floating-point subtraction and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STF) reads from a register and the operation being performed in parallel (SUBF3) writes to the same register, STF accepts the contents of the register as input before it is modified by the SUBF3.

If *src3* and *dst1* point to the same location, *src3* is read before the write to *dst1.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7 – R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit

OVM Operation is not affected by OVM bit value.

Example

```
        SUBF3     R1,*–AR4(IR1),R0
||      STF       R7,*+AR5(IR0)
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	06 1B60 0000	7.768750e+01
R1	05 7B40 0000	6.28125e+01	R1	05 7B40 0000	6.28125e+01
R7	07 33C0 0000	1.79750e+02	R7	07 33C0 0000	1.79750e+02
AR4	80 98B8		AR4	80 98B8	
AR5	80 9850		AR5	80 9850	
IR0	10		IR0	10	
IR1	8		IR1	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
8098B0h	70C8000	1.4050e+02	8098B0h	70C8000	1.4050e+02
809860h	0		809860h	733C000	1.79750e+02

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **SUBI** *src, dst*

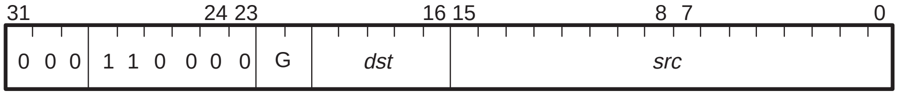
Operation *dst* – *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode



Description The difference between the *dst* operand and the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `SUBI 220,R7`

	Before Instruction			**After Instruction**	
R7	00 0000 0226	550	R7	00 0000 014A	330
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Syntax **SUBI3** *src2, src1, dst*

Operation *src1* – *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 27)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 1 1 1 0	T	*dst*	*src*1	*src*2

Description The difference between the *src1* operand and the *src2* operand is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example 1

```
SUBI3  R7,R2,R0
```

	Before Instruction			After Instruction	
R0	00 0000 0000		R0	00 0000 0032	50
R2	00 0000 0866	2150	R2	00 0000 0866	2150
R7	00 0000 0834	2100	R7	00 0000 0834	2100
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	1	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	

Example 2

```
SUBI3 *–AR2(1),R4,R3
```

	Before Instruction			After Instruction	
R3	00 0000 0000		R3	00 0000 014A	330
R4	00 0000 0226	550	R4	00 0000 0226	550
AR2	80 985E		AR2	80 985E	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80985Dh	0DC	220	80985Dh	0DC	220

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

	SUBI3	*src1, src2, dst1*
||	**STI**	*src3, dst2*

Operation

src2 – *src1* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1)
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

This instruction's operands have been augmented in the following devices:

- ❑ 'C31 silicon revision 6.0 or greater
- ❑ 'C32 silicon revision 2.0 or greater

src1	register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2	indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
dst1	register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3	register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	1 0 1 1 0	*dst*1	*src*1	*src*3	*dst*2	*src*2

Description

An integer subtraction and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (SUBI3) writes to the same register, STI accepts the contents of the register as input before it is modified by the SUBI3.

If *src3* and *dst1* point to the same location, *src3* is read before the write to *dst1.*

Cycles

1

Status Bits

These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit

OVM Operation is affected by OVM bit value.

Example

```
        SUBI3     R7,*+AR2(IR0),R1
||      STI       R3,*++AR7
```

	Before Instruction			After Instruction	
R1	00 0000 0000		R1	00 0000 00C8	200
R3	00 0000 0035	53	R3	00 0000 0035	53
R7	00 0000 0014	20	R7	00 0000 0014	20
AR2	80 982F		AR2	80 982F	
AR7	80 983B		AR7	80 983C	
IR0	10		IR0	10	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80983Fh	0DC	220	80983Fh	0DC	220
80983Ch	0		80983Ch	35	53

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **SUBRB** *src, dst*

Operation *src* – *dst* – C → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 1 0 0 0 1	G	*dst*	*src*

Description The difference of the *src, dst*, and C operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `SUBRB R4,R6`

	Before Instruction			**After Instruction**	
R4	00 0000 03CB	971	R4	00 0000 03CB	971
R6	00 0000 0258	600	R6	00 0000 0172	370
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	1		C	0	

Syntax **SUBRF** *src, dst*

Operation *src* – *dst* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 7)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 7)

Opcode

31–29	28–23	22–21	20–16	15–0
0 0 0	1 1 0 0 1 0	G	*dst*	*src*

Description The difference between the *src* operand and the *dst* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	1 if a floating-point underflow occurs; unchanged otherwise
LV	1 if a floating-point overflow occurs; unchanged otherwise
UF	1 if a floating-point underflow occurs; 0 otherwise
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if a floating-point overflow occurs; 0 otherwise
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `SUBRF @9905h,R5`

	Before Instruction			After Instruction	
R5	05 7B40 0000	6.281250e+01	R5	06 69E0 0000	1.16937500e+02
DP	080		DP	080	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
809905h	733C000	1.79750e+02	809905h	733C000	1.79750e+02

Syntax **SUBRI** *src, dst*

Operation *src* – *dst* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24	23	16	15 … 8 7 … 0
0 0 0	1 1 0 0 1 1	G	*dst*	*src*

Description The difference between the *src* operand and the *dst* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	1 if an integer overflow occurs; unchanged otherwise
UF	0
N	1 if a negative result is generated; 0 otherwise
Z	1 if a 0 result is generated; 0 otherwise
V	1 if an integer overflow occurs; 0 otherwise
C	1 if a borrow occurs; 0 otherwise

Mode Bit **OVM** Operation is affected by OVM bit value.

Example `SUBRI *AR5++(IR0),R3`

	Before Instruction			**After Instruction**	
R3	`00 0000 00DC`	`220`	R3	`00 0000 014A`	`330`
AR5	`80 9900`		AR5	`80 9908`	
IR0	`8`		IR0	`8`	
LUF	`0`		LUF	`0`	
LV	`0`		LV	`0`	
UF	`0`		UF	`0`	
N	`0`		N	`0`	
Z	`0`		Z	`0`	
V	`0`		V	`0`	
C	`0`		C	`0`	
Data memory					
809900h	`226`	`550`	809900h	`226`	`550`

Syntax **SWI**

Operation Performs an emulation interrupt

Operands None

Opcode

31							24	23							16	15							8	7							0
0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Description The SWI instruction performs an emulator interrupt. This is a reserved instruction and should not be used in normal programming.

Cycles 4

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Syntax **TRAP*cond*** N

Operation 0 → ST(GIE)
If *cond* is true:
  Next PC → *++SP,
  Trap vector N → PC.

Else:

  Set ST(GIE) to original state.
  Continue.

Operands N (0 ≤ N ≤ 31)

Opcode

31	24 23	16	15 8 7	0
0 1 1 1 0 1 0	0 0 0 0	*cond*	0 0 0 0 0 0 0 0 0 0 1	N + 20h

Description Interrupts are disabled globally when 0 is written to ST(GIE). If the condition is true, the contents of the PC are pushed onto the system stack, and the PC is loaded with the contents of the specified trap vector (N). If the condition is not true, ST(GIE) is set to its value before the TRAP*cond* instruction changes it.

The ’C3x provides 20 condition codes that can be used with this instruction (see Table 13–12 on page 13-30 for a list of condition mnemonics, condition codes, and flags). Condition flags are set on a previous instruction only when the destination register is one of the extended-precision registers (R7–R0) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.

Cycles 5

Status Bits

LUF	Unaffected
LV	Unaffected
UF	Unaffected
N	Unaffected
Z	Unaffected
V	Unaffected
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `TRAPZ 16`

	Before Instruction		After Instruction
PC	0123	PC	0010
SP	809870	SP	809871
ST	0	ST	0
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
Trap V.16	10	809871h	124

Syntax **TSTB** *src, dst*

Operation *dst* AND *src*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16	15 8 7 0
0 0 0	1 1 0 1 0 0	G	*dst*	*src*

Description The bitwise-logical AND of the *dst* and *src* operands is formed, but the result is not loaded in any register. This allows for nondestructive compares. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified for all destination registers (R27–R0).

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 output is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
TSTB *–AR4(1),R5
```

	Before Instruction			After Instruction	
R5	00 0000 0898	2200	R5	00 0000 0898	2200
AR4	80 99C5		AR4	80 99C5	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	1	
V	0		V	0	
C	0		C	0	
Data memory					
8099C4h	767	1895	8099C4h	767	1895

Syntax **TSTB3** *src2, src1*

Operation *src1* AND *src2*

Operands *src1* 3-operand addressing modes (T):

0 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1	indirect (*disp* = 0, 1, IR0, IR1)
1 0	register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0	register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1	register (R*n*2, 0 ≤ *n*2 ≤ 127)
1 0	indirect (*disp* = 0, 1, IR0, IR1)
1 1	indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23	16	15 8	7 0
0 0 1	0 0 1 1 1 1	T	0 0 0 0 0	*src*1	*src*2

Description The bitwise-logical AND between the *src1* and *src2* operands is formed but is not loaded into any register. This allows for nondestructive compares. The *src1* and *src2* operands are assumed to be unsigned integers. Although this instruction has only two operands, it is designated as a 3-operand instruction because operands are specified in the 3-operand format.

Cycles 1

Status Bits These condition flags are modified for all destination registers (R27 – R0).

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 output is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1 `TSTB3 *AR5––(IR0),*+AR0(1)`

	Before Instruction		After Instruction	
AR0	80 992C		80 992C	
AR5	80 9885		80 9805	
IR0	80		80	
LUF	0		0	
LV	0		0	
UF	0		0	
N	0		0	
Z	0		1	
V	0		0	
C	0		0	
Data memory				
809885h	898	2200	898	2200
80992Dh	767	1895	767	1895

Example 2 `TSTB3 R4,*AR6––(IR0)`

	Before Instruction	After Instruction
R4	00 0000 FBC4	00 0000 FBC4
AR6	80 99F8	80 99F0
IR0	8	8
LUF	0	0
LV	0	0
UF	0	0
N	0	0
Z	0	0
V	0	0
C	0	0
Data memory		
8099F8h	1568	1568

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax **XOR** *src, dst*

Operation *dst* XOR *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (R*n*, 0 ≤ *n* ≤ 27)
0 1	direct
1 0	indirect (disp = 0–255, IR0, IR1)
1 1	immediate

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode

31	24 23		16 15	8 7 0
0 0 0	1 1 0 1 0 1	G	*dst*	*src*

Description The bitwise-exclusive OR of the *src* and *dst* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 output is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example `XOR R1,R2`

	Before Instruction		After Instruction
R1	`00 000F FA32`	R1	`00 000F F412`
R2	`00 000F F5C1`	R2	`00 0000 0FF3`
LUF	`0`	LUF	`0`
LV	`0`	LV	`0`
UF	`0`	UF	`0`
N	`0`	N	`0`
Z	`0`	Z	`0`
V	`0`	V	`0`
C	`0`	C	`0`

Syntax **XOR3** *src2, src1, dst*

Operation *src1* XOR *src2* → *dst*

Operands *src1* 3-operand addressing modes (T):

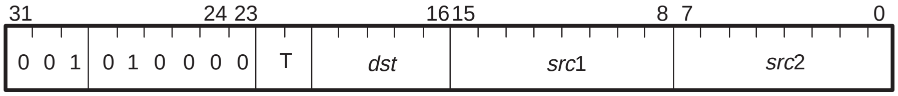
0 0 register (R*n*1, 0 ≤ *n*1 ≤ 27)
0 1 indirect (*disp* = 0, 1, IR0, IR1)
1 0 register (R*n*1, 0 ≤ *n*1 ≤ 27)
1 1 indirect (*disp* = 0, 1, IR0, IR1)

src2 3-operand addressing modes (T):

0 0 register (R*n*2, 0 ≤ *n*2 ≤ 27)
0 1 register (R*n*2, 0 ≤ *n*2 ≤ 27)
1 0 indirect (*disp* = 0, 1, IR0, IR1)
1 1 indirect (*disp* = 0, 1, IR0, IR1)

dst register (R*n*, 0 ≤ *n* ≤ 27)

Opcode



Description The bitwise-exclusive OR between the *src1* and *src2* operands is loaded into the *dst* register. The *src1, src2,* and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits These condition flags are modified only if the destination register is R7 – R0.

LUF Unaffected
LV Unaffected
UF 0
N MSB of the output
Z 1 if a 0 output is generated; 0 otherwise
V 0
C Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example 1

```
XOR3 *AR3++(IR0),R7,R4
```

	Before Instruction		After Instruction
R4	00 0000 0000	R4	00 0000 A53C
R7	00 0000 FFFF	R7	00 0000 FFFF
AR3	80 9800	AR3	80 9810
IR0	10	IR0	10
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809800h	5AC3	809800h	5AC3

Example 2

```
XOR3  R5,*–AR1(1),R1
```

	Before Instruction		After Instruction
R1	00 0000 0000	R1	00 0000 0F33
R5	00 000F FA32	R5	00 000F FA32
AR1	80 9826	AR1	80 9826
LUF	0	LUF	0
LV	0	LV	0
UF	0	UF	0
N	0	N	0
Z	0	Z	0
V	0	V	0
C	0	C	0
Data memory			
809825h	0FF5C1	809825h	0FF5C1

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Syntax

   XOR3 *src2, src1, dst1*
|| **STI** *src3, dst2*

Operation

   src1 XOR *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (R*n*1, 0 ≤ *n*1 ≤ 7)
src2 indirect (*disp* = 0, 1, IR0, IR1)
dst1 register (R*n*2, 0 ≤ *n*2 ≤ 7)
src3 register (R*n*3, 0 ≤ *n*3 ≤ 7)
dst2 indirect (*disp* = 0, 1, IR0, IR1)

> This instruction's operands have been augmented in the following devices:
>
> ❑ 'C31 silicon revision 6.0 or greater
> ❑ 'C32 silicon revision 2.0 or greater
>
> *src1* register (R*n*1, 0 ≤ *n*1 ≤ 7)
> *src2* indirect (*disp* = 0, 1, IR0, IR1) or any CPU register
> *dst1* register (R*n*2, 0 ≤ *n*2 ≤ 7)
> *src3* register (R*n*3, 0 ≤ *n*3 ≤ 7)
> *dst2* indirect (*disp* = 0, 1, IR0, IR1)

Opcode

31	24	23		16	15 8	7 0
1 1	1 0 1 1 1	*dst*	*src*1	*src*3	*dst*2	*src*2

Description

A bitwise-exclusive XOR and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations (STI) reads from a register and the operation being performed in parallel (XOR3) writes to the same register, STI accepts as input the contents of the register before it is modified by the XOR3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2.*

Cycles

1

Status Bits These condition flags are modified only if the destination register is R7–R0.

LUF	Unaffected
LV	Unaffected
UF	0
N	MSB of the output
Z	1 if a 0 output is generated; 0 otherwise
V	0
C	Unaffected

Mode Bit **OVM** Operation is not affected by OVM bit value.

Example

```
        XOR3     *AR1++,R3,R3
||      STI      R6,*–AR2(IR0)
```

	Before Instruction			After Instruction	
R3	00 0000 0085		R3	00 0000 0000	
R6	00 0000 00DC	220	R6	00 0000 00DC	220
AR1	80 987E		AR1	80 987F	
AR2	80 98B4		AR2	80 98B4	
IR0	8		IR0	8	
LUF	0		LUF	0	
LV	0		LV	0	
UF	0		UF	0	
N	0		N	0	
Z	0		Z	0	
V	0		V	0	
C	0		C	0	
Data memory					
80987Eh	85		80987Eh	85	
8098ACh	0		8098ACh	0DC	220

Note: Cycle Count

See subsection 8.5.2, *Data Loads and Stores*, on page 8-24 for the effects of operand ordering on the cycle count.

Appendix A

Instruction Opcodes

The opcode fields for all TMS320C3x instructions are shown in Table A–1. Bits in the table marked with a hyphen are defined in the individual instruction descriptions (see Chapter 13, *Assembly Language Instructions*). Table A–1, along with the instruction descriptions, fully defines the instruction words. The opcodes are listed in numerical order. Note that an undefined operation may occur if an illegal opcode is executed. (An Illegal opcode can only be generated by the misuse of the TMS320 floating-point software tools, by an error in the ROM code, or by a defective RAM.)

Table A–1. TMS320C3x Instruction Opcodes

Instruction	31	30	29	28	27	26	25	24	23
ABSF	0	0	0	0	0	0	0	0	0
ABSI	0	0	0	0	0	0	0	0	1
ADDC	0	0	0	0	0	0	0	1	0
ADDF	0	0	0	0	0	0	0	1	1
ADDI	0	0	0	0	0	0	1	0	0
AND	0	0	0	0	0	0	1	0	1
ANDN	0	0	0	0	0	0	1	1	0
ASH	0	0	0	0	0	0	1	1	1
CMPF	0	0	0	0	0	1	0	0	0
CMPI	0	0	0	0	0	1	0	0	1
FIX	0	0	0	0	0	1	0	1	0
FLOAT	0	0	0	0	0	1	0	1	1
IDLE	0	0	0	0	0	1	1	0	0
IDLE2	0	0	0	0	0	1	1	0	0
LDE	0	0	0	0	0	1	1	0	1
LDF	0	0	0	0	0	1	1	1	0
LDFI	0	0	0	0	0	1	1	1	1
LDI	0	0	0	0	1	0	0	0	0
LDII	0	0	0	0	1	0	0	0	1
LDM	0	0	0	0	1	0	0	1	0
LDP	0	0	0	0	1	0	0	0	0
LSH	0	0	0	0	1	0	0	1	1
LOPOWER	0	0	0	1	0	0	0	0	1
MAXSPEED	0	0	0	1	0	0	0	0	1
MPYF	0	0	0	0	1	0	1	0	0

[†] The opcode is the same for standard and delayed instructions.

Table A–1. TMS320C3x Instruction Opcodes (Continued)

Instruction	31	30	29	28	27	26	25	24	23
MPYI	0	0	0	0	1	0	1	0	1
NEGB	0	0	0	0	1	0	1	1	0
NEGF	0	0	0	0	1	0	1	1	1
NEGI	0	0	0	0	1	1	0	0	0
NOP	0	0	0	0	1	1	0	0	1
NORM	0	0	0	0	1	1	0	1	0
NOT	0	0	0	0	1	1	0	1	1
POP	0	0	0	0	1	1	1	0	0
POPF	0	0	0	0	1	1	1	0	1
PUSH	0	0	0	0	1	1	1	1	0
PUSHF	0	0	0	0	1	1	1	1	1
OR	0	0	0	1	0	0	0	0	0
RND	0	0	0	1	0	0	0	1	0
ROL	0	0	0	1	0	0	0	1	1
ROLC	0	0	0	1	0	0	1	0	0
ROR	0	0	0	1	0	0	1	0	1
RORC	0	0	0	1	0	0	1	1	0
RPTS	0	0	0	1	0	0	1	1	1
STF	0	0	0	1	0	1	0	0	0
STFI	0	0	0	1	0	1	0	0	1
STI	0	0	0	1	0	1	0	1	0
STII	0	0	0	1	0	1	0	1	1
SIGI	0	0	0	1	0	1	1	0	0
SUBB	0	0	0	1	0	1	1	0	1
SUBC	0	0	0	1	0	1	1	1	0
SUBF	0	0	0	1	0	1	1	1	1
SUBI	0	0	0	1	1	0	0	0	0

[†] The opcode is the same for standard and delayed instructions.

Table A–1. TMS320C3x Instruction Opcodes (Continued)

Instruction	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**
SUBRB	0	0	0	1	1	0	0	0	1
SUBRF	0	0	0	1	1	0	0	1	0
SUBRI	0	0	0	1	1	0	0	1	1
TSTB	0	0	0	1	1	0	1	0	0
XOR	0	0	0	1	1	0	1	0	1
IACK	0	0	0	1	1	0	1	1	0
ADDC3	0	0	1	0	0	0	0	0	0
ADDF3	0	0	1	0	0	0	0	0	1
ADDI3	0	0	1	0	0	0	0	1	0
AND3	0	0	1	0	0	0	0	1	1
ANDN3	0	0	1	0	0	0	1	0	0
ASH3	0	0	1	0	0	0	1	0	1
CMPF3	0	0	1	0	0	0	1	1	0
CMPI3	0	0	1	0	0	0	1	1	1
LSH3	0	0	1	0	0	1	0	0	0
MPYF3	0	0	1	0	0	1	0	0	1
MPYI3	0	0	1	0	0	1	0	1	0
OR3	0	0	1	0	0	1	0	1	1
SUBB3	0	0	1	0	0	1	1	0	0
SUBF3	0	0	1	0	0	1	1	0	1
SUB13	0	0	1	0	0	1	1	1	0
TSTB3	0	0	1	0	0	1	1	1	1
XOR3	0	0	1	0	1	0	0	0	0
LDF*cond*	0	1	0	0	–	–	–	–	–
LDI*cond*	0	1	0	1	–	–	–	–	–
BR(D)†	0	1	1	0	0	0	0	–	–
CALL	0	1	1	0	0	0	1	–	–

† The opcode is the same for standard and delayed instructions.

Table A–1. TMS320C3x Instruction Opcodes (Continued)

Instruction	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**
RPTB	0	1	1	0	0	1	0	–	–
SWI	0	1	1	0	0	1	1	–	–
B*cond*(D)†	0	1	1	0	1	0	–	–	–
DBcond(D)†	0	1	1	0	1	1	–	–	–
CALL*cond*	0	1	1	1	0	0	–	–	–
TRAP*cond*	0	1	1	1	0	1	0	–	–
RETI*cond*	0	1	1	1	1	0	0	0	0
RETS*cond*	0	1	1	1	1	0	0	0	1
MPYF3||ADDF3	1	0	0	0	0	0	0	0	–
	1	0	0	0	0	0	0	1	–
	1	0	0	0	0	0	1	0	–
	1	0	0	0	0	0	1	1	–
MPYF3||SUBF3	1	0	0	0	0	1	0	0	–
	1	0	0	0	0	1	0	1	–
	1	0	0	0	0	1	1	0	–
	1	0	0	0	0	1	1	1	–
MPYI3||ADDI3	1	0	0	0	1	0	0	0	–
	1	0	0	0	1	0	0	1	–
	1	0	0	0	1	0	1	0	–
	1	0	0	0	1	0	1	1	–
MPYI3||SUBI3	1	0	0	0	1	1	0	0	–
	1	0	0	0	1	1	0	1	–
	1	0	0	0	1	1	1	0	–
	1	0	0	0	1	1	1	1	–
STF||STF	1	1	0	0	0	0	0	–	–
STI||STI	1	1	0	0	0	0	1	–	–
LDF||LDF	1	1	0	0	0	1	0	–	–
LDI||LDI	1	1	0	0	0	1	1	–	–
ABSF||STF	1	1	0	0	1	0	0	–	–

† The opcode is the same for standard and delayed instructions.

Table A–1. TMS320C3x Instruction Opcodes (Continued)

Instruction	31	30	29	28	27	26	25	24	23
ABSI||STI	1	1	0	0	1	0	1	–	–
ADDF3||STF	1	1	0	0	1	1	0	–	–
ADDI3||STI	1	1	0	0	1	1	1	–	–
AND3||STI	1	1	0	1	0	0	0	–	–
ASH3||STI	1	1	0	1	0	0	1	–	–
FIX||STI	1	1	0	1	0	1	0	–	–
FLOAT||STF	1	1	0	1	0	1	1	–	–
LDF||STF	1	1	0	1	1	0	0	–	–
LDI||STI	1	1	0	1	1	0	1	–	–
LSH3||STI	1	1	0	1	1	1	0	–	–
MPYF3||STF	1	1	0	1	1	1	1	–	–
MPYI3||STI	1	1	1	0	0	0	0	–	–
NEGF||STF	1	1	1	0	0	0	1	–	–
NEGI||STI	1	1	1	0	0	1	0	–	–
NOT||STI	1	1	1	0	0	1	1	–	–
OR3||STI	1	1	1	0	1	0	0	–	–
SUBF3||STF	1	1	1	0	1	0	1	–	–
SUBI3||STI	1	1	1	0	1	1	0	–	–
XOR3||STI	1	1	1	0	1	1	1	–	–
Reserved for reset, traps, and interrupts	0	1	1	1	1	1	1	1	1

† The opcode is the same for standard and delayed instructions.

Appendix B

TMS320C31 Boot Loader Source Code

This appendix contains the source code for the ’C31 boot loader.

```
************************************************************************
*    C31BOOT – TMS320C31 BOOT LOADER PROGRAM
*              (C) COPYRIGHT TEXAS INSTRUMENTS INC., 1990
*
*    NOTE:    1. AFTER DEVICE RESET, THE PROGRAM IS SET TO WAIT FOR
*                THE EXTERNAL INTERRUPTS. THE FUNCTION SELECTION OF
*                THE EXTERNAL INTERRUPTS IS AS FOLLOWS:
*                ---------------------------------------------------
*                 INTERRUPT PIN |             FUNCTION
*                ---------------|-----------------------------------
*                       0       | EPROM boot loader from 1000H
*                ---------------|-----------------------------------
*                       1       | EPROM boot loader from 400000H
*                ---------------|-----------------------------------
*                       2       | EPROM boot loader from FFF000H
*                ---------------|-----------------------------------
*                       3       | Serial port 0 boot loader
*                ---------------------------------------------------
*
*             2. THE EPROM BOOT LOADER LOADS WORD, HALFWORD, OR BYTE-
*                WIDE PROGRAMS TO SPECIFIED LOCATIONS. THE
*                8 LSBs OF FIRST MEMORY SPECIFY THE MEMORY WIDTH OF
*                THE EPROM. IF THE HALFWORD OR BYTE-WIDE PROGRAM IS
*                SELECTED, THE LSBs ARE LOADED FIRST, FOLLOWED BY THE MSBs.
*                THE FOLLOWING WORD CONTAINS THE CONTROL WORD FOR
*                THE LOCAL MEMORY REGISTER. THE PROGRAM BLOCKS FOLLOW.
*                THE FIRST TWO WORDS OF EACH PROGRAM BLOCK CONTAIN
*                THE BLOCK SIZE AND MEMORY ADDRESS TO BE LOADED INTO.
*                WHEN THE ZERO BLOCK SIZE IS READ, THE PROGRAM BLOCK
*                LOADING IS TERMINATED. THE PC WILL BRANCH TO THE
*                STARTING ADDRESS OF THE FIRST PROGRAM BLOCK.
*
*             3. IF SERIAL PORT 0 IS SELECTED FOR BOOT LOADING, THE
*                PROCESSOR WILL WAIT FOR THE INTERRUPT FROM THE
*                RECEIVE SERIAL PORT 0 AND PERFORM THE DOWNLOAD.
*                AS WITH THE EPROM LOADER, PROGRAMS CAN BE LOADED
*                INTO DIFFERENT MEMORY BLOCKS. THE FIRST TWO WORDS OF EACH
*                PROGRAM BLOCK CONTAIN THE BLOCK SIZE AND MEMORY ADDRESS
*                TO BE LOADED INTO. WHEN THE ZERO BLOCK SIZE IS READ,
*                PROGRAM BLOCK LOADING IS TERMINATED. IN OTHER WORDS,
*                IN ORDER TO TERMINATE THE PROGRAM BLOCK LOADING,
*                A ZERO HAS TO BE ADDED AT THE END OF THE PROGRAM BLOCK.
*                AFTER THE BOOT LOADING IS COMPLETED, THE PC WILL BRANCH
*                TO THE STARTING ADDRESS OF THE FIRST PROGRAM BLOCK.
*
************************************************************************
```

```
                    .global check
                    .sect  ”vectors”
  reset             .word  check
  int0              .word  809FC1h
  int1              .word  809FC2h
  int2              .word  809FC3h
  int3              .word  809FC4h
  xint0             .word  809FC5h
  rint0             .word  809FC6h
                    .word  809FC7h
                    .word  809FC8h
  tint0             .word  809FC9h
  tint1             .word  809FCAh
  dint              .word  809FCBh
                    .word  809FCCh
                    .word  809FCDh
                    .word  809FCEh
                    .word  809FCFh
                    .word  809FD0h
                    .word  809FD1h
                    .word  809FD2h
                    .word  809FD3h
                    .word  809FD4h
                    .word  809FD5h
                    .word  809FD6h
                    .word  809FD7h
                    .word  809FD8h
                    .word  809FD9h
                    .word  809FDAh
                    .word  809FDBh
                    .word  809FDCh
                    .word  809FDDh
                    .word  809FDEh
                    .word  809FDFh

********************************************************************************

trap0 .word  809FE0h
trap1 .word  809FE1h
trap2 .word  809FE2h
trap3 .word  809FE3h
trap4 .word  809FE4h
trap5 .word  809FE5h
trap6 .word  809FE6h
trap7 .word  809FE7h
trap8 .word  809FE8h
trap9 .word  809FE9h
trap10.word  809FEAh
```

```
trap11 .word  809FEBh
trap12 .word  809FECh
trap13 .word  809FEDh
trap14 .word  809FEEh
trap15 .word  809FEFh
trap16 .word  809FF0h
trap17 .word  809FF1h
trap18 .word  809FF2h
trap19 .word  809FF3h
trap20 .word  809FF4h
trap21 .word  809FF5h
trap22 .word  809FF6h
trap23 .word  809FF7h
trap24 .word  809FF8h
trap25 .word  809FF9h
trap26 .word  809FFAh
trap27 .word  809FFBh
       .word  809FFCh
       .word  809FFDh
       .word  809FFEh
       .word  809FFFh

*****************************************************************************

                    .space 5

  check:            LDI      4040h,AR0       ; load peripheral mem. map
                    LSH      9,AR0           ; start addr. 808000h
                    LDI      404Ch,SP        ; initialize stack pointer to
                    LSH      9,SP            ; ram0 addr. 809800h
                    LDI      0,R0            ; set start address flag off

  intloop           TSTB     8,IF            ; test for ext int3
                    BNZ      serial          ; on int3 go to serial

                    LDI      8,AR1           ; load 001000h / 2^9 –> AR1
                    TSTB     1,IF            ; test for int0
                    BNZ      eprom_load      ; branch to eprom_load if int0 = 1

                    LDI      2000h,AR1       ; load 400000h / 2^9 –> AR1
                    TSTB     2,IF            ; test for int1
                    BNZ      eprom_load      ; branch to eprom_load if int1 = 1

                    LDI      7FF8h,AR1       ; load FFF000h / 2^9 –> AR1
                    TSTB     4,IF            ; test for int2
                    BZ       intloop         ; if no intX go to intloop

  eprom_load        LSH      9,AR1           ; eprom address = AR1 * 2^9
                    LDI      *AR1++(1),R1    ; load eprom mem. width

                    LDI      sub_w,AR3       ; full–word size subroutine
                                             ; address –> AR3
                    LSH      26,R1           ; test bit 5 of mem. width word
                    BN       load0           ; if ’1’ start PGM loading
                                             ; (32 bits width)
```

```
                  NOP      *AR1++(1)       ; jump last half word from mem. word
                  LDI      sub_h,AR3       ; half word size subroutine
                                           ; address –> AR3
                  LSH      1,R1            ; test bit 4 of mem. width word
                  BN       load0           ; if ’1’ start PGM loading
                                           ; (16 bits width)

                  LDI      sub_b,AR3       ; byte size subroutine address –> AR3
                  ADDI     2,AR1           ; jump last 2 bytes from mem. word

load0             CALLU    AR3             ; load new word
                                           ; according to mem. width
                  STI      R1,*+AR0(64h)   ; set primary bus control

load2             CALLU    AR3             ; load new word according to
                                           ; mem. width
                  LDI      R1,RC           ; set block size for repeat loop
                  CMPI     0,RC            ; if 0 block size start PGM
                  BZ       AR2
                  SUBI     1,RC            ; block size –1

                  CALLU    AR3             ; load new word according to
                                           ; mem. width
                  LDI      R1,AR4          ; set destination address
                  LDI      R0,R0           ; test start address loaded flag
                  LDIZ     R1,AR2          ; load start address if flag off
                  LDI      –1,R0           ; set start & dest. address flag on
                  SUBI     1,AR3           ; sub address with loop

                           CALLUAR3        ; load new word according to
                                           ; mem. width
                  LDI      1,R0            ; set dest. address flag off
                  ADDI     1,AR3           ; sub address without loop
                  BR       load2           ; jump to load a new block
                                           ; when loop completed

                  .space 1

serial            LDI      sub_s,AR3       ; serial words subroutine
                                           ; address –> AR3
                  LDI      111h,R1         ; R1 = 0000111h
                  STI      R1,*+AR0(43h)   ; set CLKR,DR,FSR as serial port pins
                  LDI      0A30h,R2
                  LSH      16,R2           ; R2 = A300000h
                  STI      R2,*+AR0(40h)   ; set serial port global
                                           ; ctrl. register
                  BR       load2           ; jump to load 1st block

                  .space 29

loop_s            RPTB     load_s          ; PGM load loop
sub_s             TSTB     20h,IF
                  BZ       sub_s           ; wait for receive buffer full
                  AND      0FDFh,IF        ; reset interrupt flag
                  LDI      *+AR0(4Ch),R1
                  LDI      R0,R0           ; test load address flag
                  BNN      end_s
load_s            STI      R1,*AR4++(1)    ; store new word to dest. address
end_s             RETSU                    ; return from subroutine

                  .space 22

loop_h            RPTB     load_h          ; PGM load loop
sub_h             LDI      *AR1++(1),R1    ; load LSB half word
                  AND      0FFFFh,R1
                  LDI      *AR1++(1),R2    ; load MSB half word
                  LSH      16,R2
                  OR       R2,R1           ; R1 = a new 32-bit word
                  LDI      R0,R0           ; test load address flag
                  BNN      end_h
load_h            STI      R1,*AR4++(1)    ; store new word to dest. address
end_h             RETSU                    ; return from subroutine

                  .space 26

loop_w            RPTB     load_w          ; PGM load loop
sub_w             LDI      *AR1++(1),R1    ; read a new 32-bit word
                  LDI      R0,R0           ; test load address flag
                  BNN      end_w
load_w            STI      R1,*AR4++(1)    ; store new word to dest. address
end_w             RETSU                    ; return from subroutine

                  .space 14

loop_b            RPTB     load_b          ; PGM load loop
sub_b             LDI      *AR1++(1),R1
                  AND      0FFh,R1         ; load 1st byte ( LSB )
                  LDI      *AR1++(1),R2
                  AND      0FFh,R2
                  LSH      8,R2
                  OR       R2,R1           ; load 2nd byte
                  LDI      *AR1++(1),R2
                  AND      0FFh,R2
                  LSH      16,R2
                  OR       R2,R1           ; load 3rd byte
                  LDI      *AR1++(1),R2    ; load 4th byte ( MSB )
                           LSH 24,R2
                  OR       R2,R1           ; R1 = a new 32-bit word
                  LDI      R0,R0           ; test load address flag
                  BNN      end_b
load_b            STI      R1,*AR4++(1)    ; store new word to dest. address
end_b             RETSU                    ; return from subroutine

                  .space 1

                  .end
```

Appendix C

TMS320C32 Boot Loader Source Code

This appendix includes a description of the ’C32 boot loader sequence of events and a listing of its source code.

Topic **Page**

C.1 Boot-Loader Source Code Description

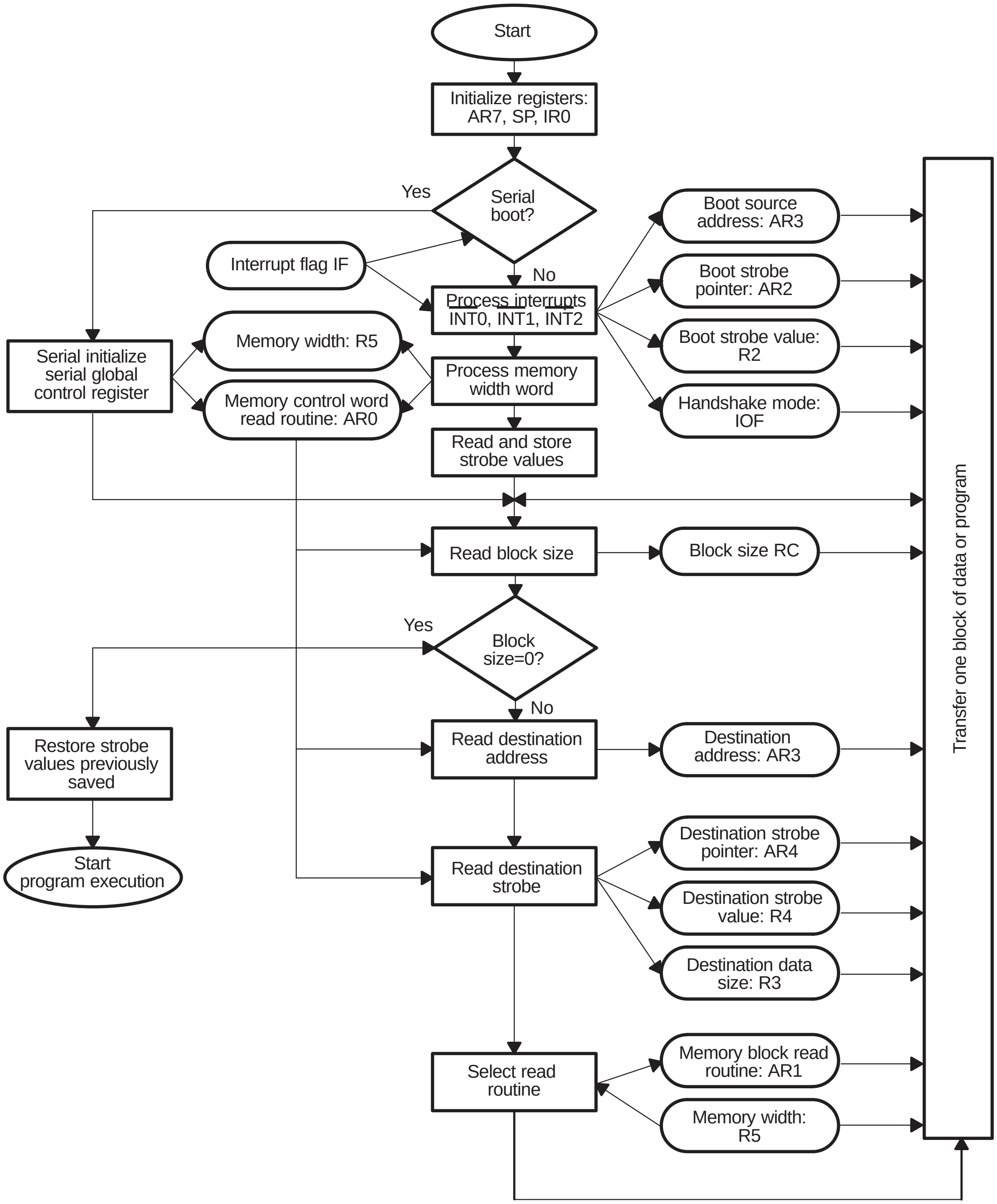
Figure C–1 shows the boot loader program flow chart. The boot loader program starts by initializing three registers: *AR7*, *SP*, and *IR0*. These registers hold the peripheral bus memory map register base address, the timer counter register (used as a stack), and a flag that indicates the first block, respectively. Then, the program checks for serial port boot load or memory boot load mode by processing the bit fields set in the interrupt flag register (*IF*). For a serial port boot load, the program initializes the serial port for 32-bit fixed-burst-mode reads with an externally generated serial port clock and FSR.

For a memory boot load, *AR3* is set to the boot source address, *AR2* points to the boot source strobe control register, and *R2* contains the value that is stored in this strobe control register. The boot loader also sets the bit field $\overline{\text{I}}$/OXF0 of the I/O flag register (IOF) if the handshake mode was selected. Then the boot loader reads the first word of the boot source program. This 32-bit word indicates the boot memory width and the boot load program stores this value in *R5*. *AR0* points to the *read_mc* routine that performs this read.

After reading the memory width word, the boot loader reads IOSTRB, STRB0, and STRB1 control register values of the source program. These values are temporarily saved in the DMA source address register, DMA destination address register, and DMA transfer counter registers, respectively. Then, the program reads the block size with the *read_mc* routine. If the block size is 0, the boot loader restores the values of IOSTRB, STRB0, and STRB1 previously saved and branches to the destination address of the first block loaded and begins program execution. If the block size is not 0, the boot loader stores the block size in the *BK* register. This is used as a counter in a repeat block (*RPTB*) to transfer all the data or program in that block.

For each block, the boot loader reads the destination address and the destination strobe control word. The program stores the destination address in the *AR5* register. The destination strobe control word includes the destination strobe identification, the contents of the destination strobe control register (includes memory width and data size). The boot loader extracts this information from the destination control word and stores the destination strobe-control register memory-mapped address in the *AR4* register, the contents of the destination strobe control register in the *R4* register, and the source data size in the *R3* register. The boot loader sets the *AR1* register to the appropriate read routine *read_s0* for serial port boot load and *read_mb* for memory boot load. The read routine uses these registers to control the transfer of a block of data or program.

Figure C–1. Boot-Loader Flow Chart


Start
Initialize registers: AR7, SP, IR0
Serial boot?
Yes
No
Interrupt flag IF
Process interrupts INT0, INT1, INT2
Boot source address: AR3
Boot strobe pointer: AR2
Boot strobe value: R2
Handshake mode: IOF
Serial initialize serial global control register
Memory width: R5
Memory control word read routine: AR0
Process memory width word
Read and store strobe values
Read block size
Block size RC
Block size=0?
Yes
No
Restore strobe values previously saved
Start program execution
Read destination address
Destination address: AR3
Read destination strobe
Destination strobe pointer: AR4
Destination strobe value: R4
Destination data size: R3
Select read routine
Memory block read routine: AR1
Memory width: R5
Transfer one block of data or program

C.2 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 will start loading the boot table to the destination memory:
*
```

INTERRUPT PIN	BOOT TABLE START ADDRESS	BOOT SOURCE
INT0	1000h (STRB0)	EPROM
INT1	810000h (IOSTRB)	EPROM
INT2	900000h (STRB1)	EPROM
INT3	80804Ch (sport0 Rx)	SERIAL
INT0 and INT3	1000h (STRB0) ASYNC	EPROM,XF0/XF1
INT1 and INT3	810000h (IOSTRB) ASYNC	EPROM,XF0/XF1
INT2 and INT3	900000h (STRB1) ASYNC	EPROM,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 will continuously toggle 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 de
*     scribing the block contents to the boot loader program.
*
*  3. When loading from 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 EPROM whose physical width is less than 32 bits, the
*     physical interface of the EPROM device(s) to the processor should be the
*     same as that of the 32-bit interface. (This involves a specific connection
*     to C32’s strobe and address signals). The reason for such 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 should reside in
*     physical memory of the same width. It is also not permitted to mix prg and
*     data in the same source block.
*
*  6. The boot loader stops boot operation when it finds 0 in the block size
*     control word. Therefore, each boot table should always 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 mem locations
*================================================================================*
*
* AR7 – peripheral memory map               IOF – XF0 (handshake O)
* AR0 – read cntrl data subr pointer        IOF – XF1 (handshake I)
* 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

*================================================================================*


* Init 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 ,
* appropriate boot strobe pointer  --> AR2 and force the boot strobe value to
* reflect 32bit 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 should be 0 or 60h. The first case will
* result in 0 --> DMA cntrl reg, in second case 0 --> IOSTRB reg).
* 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 mem 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 D

Glossary

A

A0–A23: External address pin*s* for data/program memory or I/O devices. These pins are on the primary bus.

address: The location of program code or data stored in memory.

addressing mode: The method by which an instruction interprets its operands to acquire the data it needs.

ALU: *Arithmetic logic unit.* The part of the CPU that performs arithmetic and logic operations.

analog-to-digital (A/D) converter: A converter with internal sample-and-hold circuitry used to translate an analog signal to a digital signal.

ARAU: *Auxiliary-register arithmetic unit.* A 32-bit ALU used to calculate indirect addresses using the auxiliary registers as inputs and outputs.

arithmetic logic unit (ALU): The part of the CPU that performs arithmetic and logic operations.

auxiliary registers (ARn): A set of registers used primarily in address generation.

auxiliary-register arithmetic unit (ARAU): *Auxiliary-register arithmetic unit.* A 32-bit ALU used to calculate indirect addresses using the auxiliary registers as inputs and outputs.

B

bit-reversed addressing: Accessing data from memory, registers, and the instruction word by reversing several bits of an address in order to speed processing of algorithms, such as Fourier transforms.

BK: *Block-size register.* A 32-bit register used by the ARAU in circular addressing to specify the data block size.

boot loader: An on-chip code that loads and executes programs received from a host processor through standard memory devices (including EPROM), with and without handshake, or through the serial port to RAM at power up.

C

carry bit: A bit in the status register (ST) used by the ALU for extended arithmetic operations and accumulator shifts and rotates. The carry bit can be tested by conditional instructions.

circular addressing: Accessing data from memory, registers, and the instruction word by using an auxiliary register to cycle through a range of addresses to create a circular buffer in memory.

context save/restore: A save/restore of system status (status registers, accumulator, product register, temporary register, hardware stack, and auxiliary registers, etc.) when the device enters/exits a subroutine such as an interrupt service routine.

CPU: *Central processing unit*. The unit that coordinates the functions of a processor.

CPU cycle: The time it takes the CPU to go through one logic phase (during which internal values are changed) and one latch phase (during which the values are held constant).

ICPU interrupt flag register (IF): A register that contains CPU, serial ports, timer, and DMA interrupt flags.

cycle: See CPU cycle.

D

D0–D31: External data-bus pins that transfer data between the processor and external data/program memory or I/O devices. *See also LD0–LD31*.

data-address generation logic: Circuitry that generates the addresses for data-memory reads and writes. This circuitry can generate one address per machine cycle. See also *program address generation logic.*

data-page pointer: A 32-bit register used as the 8 most significant bits (MSBs) in addresses generated using direct addressing.

data size: The number of bits (8, 16, or 32) used to represent a particular number.

decode phase: The phase of the pipeline in which the instruction is decoded (identified).

DMA coprocessor: A peripheral that transfers the contents of memory locations independently of the processor (except for initialization).

DMA controller: See DMA coprocessor.

DP: See data-page pointer.

dual-access RAM: Memory that can be accessed twice in a single clock cycle. For example, code that can read from and write to a RAM in one clock cycle.

E

external interrupt: A hardware interrupt triggered by a pin.

extended-precision floating-point format: A 40-bit representation of a floating-point number with a 32-bit mantissa and an 8-bit exponent.

extended-precision register: A 40-bit register used primarily for extended-precision floating-point calculations. Floating-point operations use bits 39–0 of an extended-precision register. Integer operations, however, use only bits 31–0.

F

FIFO buffer: *First-in, first-out buffer.* A portion of memory in which data is stored and then retrieved in the same order in which it was stored. Thus, the first word stored in this buffer is retrieved first.

H

hardware interrupt: An interrupt triggered through physical connections with on-chip peripherals or external devices.

hit: A condition in which, when the processor fetches an instruction, the instruction is available in the cache.

I

$\overline{\text{IACK}}$: *Interrupt acknowledge signal*. An output signal indicating that an interrupt has been received and that the program counter is fetching the interrupt vector that will force the processor into an interrupt service routine.

IE: See internal interrupt enable register.

I/O flag (IOF) register: Controls the function (general-purpose I/O or interrupt) of the external pins. It also contains timer/DMA interrupt flags.

index registers: Two 32-bit registers (IR0 and IR1) that are used by the ARAU for indexing an address.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

internal interrupt enable register: A register (in the CPU register file) that determines whether the CPU or DMA responds to interrupts from external interrupt pins, the serial ports, the timers, and the DMA coprocessor.

interrupt: A signal sent to the CPU that (when not masked) forces the CPU into an ISR. This signal can be triggered by an external device, an on-chip peripheral, or an instruction (TRAP, for example).

interrupt acknowledge ($\overline{\text{IACK}}$): A signal indicating that an interrupt has been received and that the program counter is fetching the interrupt vector location.

interrupt-trap table pointer (ITTP): A bit field in the status register that indicates the starting location (base address) of the interrupt-trap vector table. The base address is formed by left-shifting the value of the ITTP bit field by 8 bits.

ISR: *Interrupt service routine*. A module of code that is executed in response to a hardware or software interrupt.

ITTP: See *interrupt-trap table pointer*.

L

LSB: *Least significant bit.* The lowest-order bit in a word.

M

machine cycle: See *CPU cycle*.

mantissa: A component of a floating-point number consisting of a fraction and a sign bit. The mantissa represents a normalized fraction whose binary point is shifted by the exponent.

maskable interrupt: A hardware interrupt that can be enabled or disabled through software.

memory-mapped register: One of the on-chip registers that point to addresses in memory. Some memory-mapped registers point to data memory, and somepoint to input/output memory.

memory width: The number of bits that can be stored in a single external memory address.

MFLOPS: *Millions of floating point operations per second*. A measure of floating-point processor speed that counts of the number of floating-point operations made per second. Also called megaflops.

microcomputer mode: A mode in which the on-chip ROM (boot loader) is enabled. This mode is selected via the MP/$\overline{\text{MCBL}}$ pin.

microprocessor mode: A mode in which the on-chip ROM is disabled. This mode is selected via the MP/$\overline{\text{MCBL}}$ pin. See also *MP/$\overline{\text{MC}}$ pin.*

MIPS: Million instructions per second.

miss: A condition in which, when the processor fetches an instruction, it is not available in the cache.

MSB: *Most significant bit.* The highest-order bit in a word.

multiplier: A device that generates the product of two numbers.

N

NMI: *Nonmaskable interrupt*. A hardware interrupt that uses the same logic as the maskable interrupts but cannot be masked.

O

overflow flag (OV) bit: A status bit that indicates whether or not an arithmetic operation has exceeded the capacity of the corresponding register.

P

PC: *Program counte*r. A register that contains the address of the next instruction to be fetched.

peripheral bus: A bus that is used by the CPU to communicate to the DMA coprocessor, communication ports, and timers.

pipeline: A method of executing instructions in an assembly-line fashion.

R

RC: See repeat counter register.

read/write (R/$\overline{W}$) pin: A memory-control signal that indicates the direction of transfer when communicating to an external device.

register file: A bank of registers.

repeat-counter (RC) register: A 32-bit register in the CPU register file that specifies the number of times to repeat a block of code when performing a block repeat.

repeat mode: A zero-overhead method for repeating the execution of a block of code. Using repeat modes allows time-critical sections of code to be executed in the shortest possible time.

reset: A means to bring the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to fetch the reset vector.

reset pin: A signal that causes the device to reset.

R/$\overline{W}$: *See read/write pin*.

S

short floating-point format: A 16-bit representation of a floating point number with a 12-bit mantissa and a 4-bit exponent.

short floating-point format for external 16-bit data: A 16-bit representation of a floating point number with an 8-bit mantissa and an 8-bit exponent.

short integer format: A 2s-complement,16-bit format for integer data.

short unsigned-integer format: A 16-bit unsigned format for integer data.

sign-extend: The process of filling the high-order bits of a number with the sign bit.

single-precision floating-point format: A 32-bit representation of a floating-point number with a 24-bit mantissa and an 8-bit exponent.

single-precision integer format: A 2s-complement 32-bit format for integer data.

single-precision unsigned-integer format: A 32-bit unsigned format for integer data.

software interrupt: An interrupt caused by the execution of a TRAP instruction.

ST: See status register.

stack: A block of memory reserved for storing and retrieving data on a first-in last-out basis. It is usually used for storing return addresses and for preserving register values.

status register: A register in the CPU register file that contains global information relating to the current state of the CPU.

T

timer: A programmable peripheral that generates pulses for timed events.

timer-period register: A 32-bit memory-mapped register that specifies the period for the on-chip timer.

W

wait state: A period of time that the CPU must wait for external program, data, or I/O memory to respond when it reads from or writes to that external memory. The CPU waits one extra cycle for every wait state.

wait-state generator: A program that can be modified to generate a limited number of wait states for a given off-chip memory space (lower program, upper program, data, or I/O).

X

XA0–XA13: External address pin*s* for data/program memory or I/O devices. These pins are on the expansion bus of the 'C30. *See also A0–A23*.

XD0–XD31: External data bus pins that transfer data between the processor and external data/program memory or I/O devices of the 'C30. *See also D0–D31*.

Z

zero fill: The process of filling the low- or high-order bits with 0s when loading a number into a larger field.

Index