TMS320C3x User's Guide

2558539-9721 revision J
October 1994

TEXAS INSTRUMENTS

TEXAS INSTRUMENTS

TMS320C3x

User's Guide

1994 Digital Signal Processing Products

Printed in U.S.A., October 1994
2558539-9761 revision J

SPRU031D

User's Guide

TMS320C3x

1994

IMPORTANT NOTICE

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

This user's guide serves as a reference book for the TMS320C3x generation of digital signal processors, which includes the TMS320C30, TMS320C30-27, TMS320C30-40, TMS320C31, TMS320C31-27, TMS320C31-40, TMS320C31-50, TMS320LC31, and TMS320C31PQA. Throughout the book, all references to 'C3x refer collectively to 'C30 and 'C31, and the TMS320C30 and TMS320C31 refer to all speed variations unless an exception is noted. This document provides information to assist managers and hardware/software engineers in application development.

How to Use This Book

This revision of the TMS320C3x User's Guide incorporates the following changes:

- Updated reference list of publications
- Improved description of repeat modes and interrupts in Chapter 6
- Description of power management modes in Chapter 6
- Improved description of serial ports and DMA coprocessor in Chapter 8
- Description of power management instructions in Chapter 10
- Description of low-power-mode interrupt interface in Chapter 12
- More detailed information on MPSD emulator interface, signal timings, and connections between emulator and target system
- Current timing specification in Chapter 13
- TMS320C30PPM pinout, mechanical drawing, and timings in Chapter 13
- Development support description and device/tool part numbers in Appendix B
- Data sheet for current military versions of the 'C3x in Appendix E

Notational Conventions

This document uses the following conventions:

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

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

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

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

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

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's 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's 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 must have at least one value parameter, but you have the option of supplying additional value parameters separated by commas.

Information About Cautions

This book may contain cautions and warnings.

- A **caution** describes a situation that could potentially cause your system to behave unexpectedly.

This is what a caution looks like.

CAUTION

The information in a caution is provided for your information. 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 at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320 Floating-Point DSP Assembly Language Tools User's Guide (literature number SPRU035) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C3x and 'C4x generations of devices.

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

TMS320C3x C Source Debugger (literature number SPRU053) describes the 'C3x debugger for the emulator, evaluation module, and simulator. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

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

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

References

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

❑ **General Digital Signal Processing:**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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. . .

If you want to. . .	Do this. . .
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** FAX: **(713) 274–2324** Electronic Mail: **4389750@mcimail.com.** European fax line: **+33–1–3070–1032**
Report mistakes in this document or any other TI documentation	Fill out and return the reader response card at the end of this book, or send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Trademarks

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 digital signal processors. Since 1982, when the TMS32010 was introduced, the TMS320 family, with its powerful instruction sets, high-speed number-crunching capabilities, and innovative architectures, has established itself as the industry standard. It is ideal for DSP applications.

The 40-ns cycle time of the TMS320C31-50 allows it to execute operations at a performance rate of up to 60 million floating-point instructions per second (MFLOPS) and 30 million instructions per second (MIPS). This performance was previously available only on a supercomputer. The generation's performance is further enhanced through its large on-chip memories, concurrent direct memory access (DMA) controller, and two external interface ports.

This chapter presents the following major topics:

Topic **Page**

1.1 General Description

The TMS320 family consists of five generations: TMS320C1x, TMS320C2x, TMS320C3x, TMS320C4x, and TMS320C5x (see Figure 1–1). The expansion includes enhancements of earlier generations and more powerful new generations of DSPs.

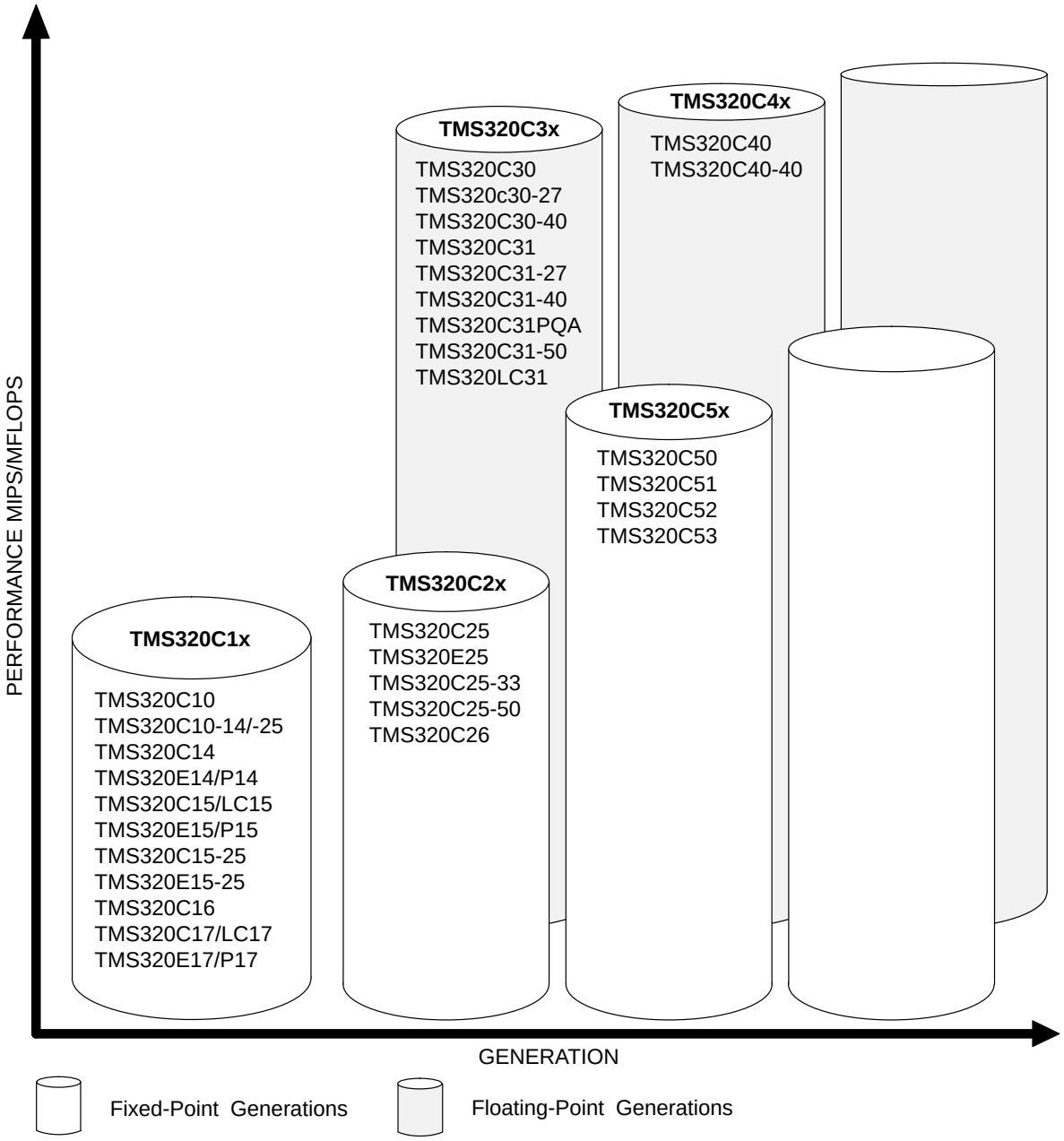
The TMS320's internal busing and special DSP instruction set have the speed and flexibility to execute at up to 50 MFLOPS. The TMS320 family optimizes speed by implementing functions in hardware that other processors implement through software or microcode. This hardware-intensive approach provides power previously unavailable on a single chip.

The emphasis on total system cost has resulted in a less expensive processor that can be designed into systems currently using costly bit-slice processors. Also, cost/performance selection is provided by the different processors in the TMS320C3x generation:

- ❑ TMS320C30: 60-ns, single-cycle execution-time
- ❑ TMS320C30-27: Lower cost; 74-ns, single-cycle execution time
- ❑ TMS320C30-40: Higher speed; 50-ns, single-cycle execution time
- ❑ TMS320C30-50: Highest speed; 40-ns, single-cycle execution time
- ❑ TMS320C31: Low cost; 60-ns, single-cycle execution time
- ❑ TMS320C31-27: Lower cost; 74-ns, single-cycle execution time
- ❑ TMS320C31-40: Low cost; 50-ns, single-cycle execution time
- ❑ TMS320C31PQA: Low cost; extended temperature; 60-ns, single-cycle execution time
- ❑ TMS320C31-50: Highest speed; 40-ns, single-cycle execution time
- ❑ TMS320LC31: Low power; 60-ns, single-cycle execution time, 3.3-volt operation

All of these processors are described in this user's guide. Essentially, their functionality is the same. However, electrical and timing characteristics vary (as described in Chapter 13); part numbering information is found in Section B.2 on page B-7. Throughout this book, TMS320C3x is used to refer to the TMS320C30 and TMS320C31 and all speed variations. TMS320C30 and TMS320C31 are used to refer to all speed variants of those processors where appropriate. Special references, such as TMS320C30-40, are used to note specific exceptions.

Figure 1–1. TMS320 Device Evolution



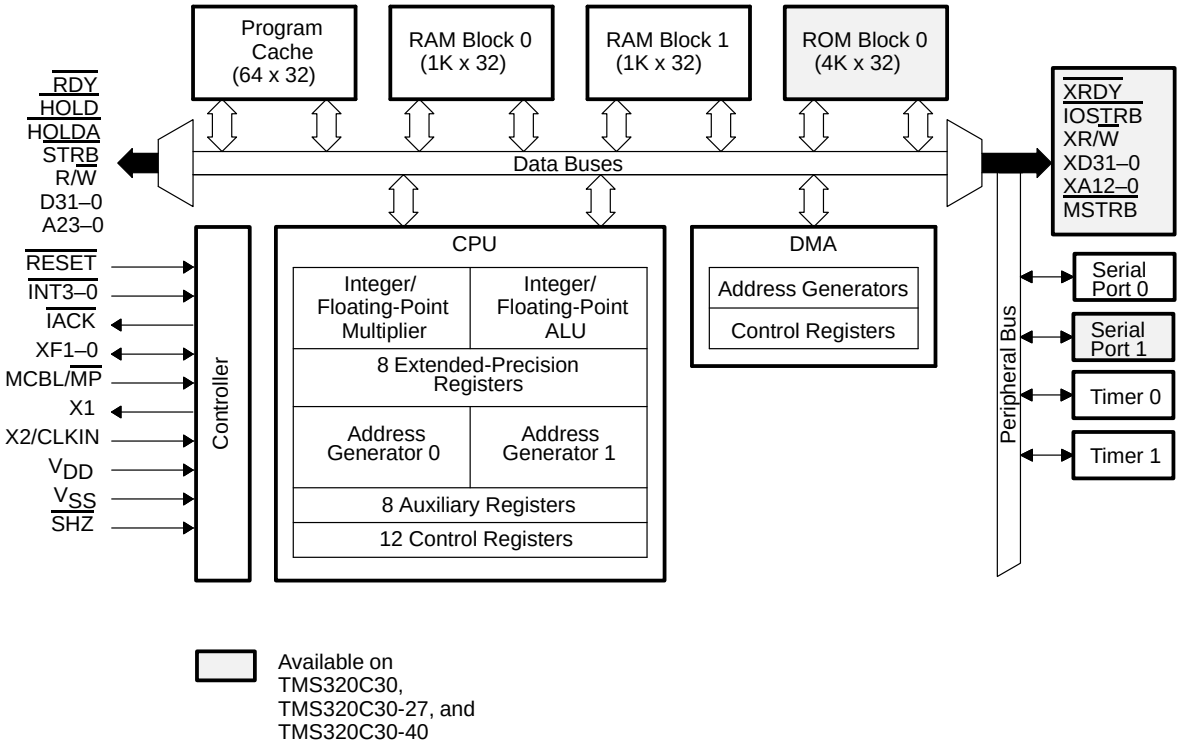
The TMS320C30 and TMS320C31 can perform parallel multiply and arithmetic logic unit (ALU) operations on integer or floating-point data in a single cycle. The processor also possesses a general-purpose register file, a program cache, dedicated auxiliary register arithmetic units (ARAU), internal dual-access memories, one DMA channel supporting concurrent I/O, and a short machine-cycle time. High performance and ease of use are products of those features.

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 TMS320C31), two timers, two serial ports (one on the TMS320C31), and multiple interrupt structure. The TMS320C3x supports a wide variety of system applications from host processor to dedicated coprocessor.

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

Figure 1–2 is a functional block diagram that shows the interrelationships between the various TMS320C3x key components.

Figure 1–2. TMS320C3x Block Diagram



1.2 TMS320C30 Key Features

Some key features of the TMS320C30 are listed below.

- Performance
  - TMS320C30 (33 MHz)
    - 60-ns, single-cycle instruction execution time
    - 33.3 MFLOPS
    - 16.7 MIPS
  - TMS320C30-27
    - 74-ns, single-cycle instruction execution time
    - 27 MFLOPS
    - 13.5 MIPS
  - TMS320C30-40
    - 50-ns, single-cycle instruction execution time
    - 40 MFLOPS
    - 20 MIPS
- One 4K x 32-bit, single-cycle, dual-access, on-chip, read-only memory (ROM) block
- Two 1K x 32-bit, single-cycle, dual-access, on-chip, random access memory (RAM) blocks
- 64- x 32-bit instruction cache
- 32-bit instruction and data words
- 24-bit addresses
- 40-/32-bit floating-point/integer multiplier and ALU
- 32-bit barrel shifter
- Eight extended-precision registers (accumulators)
- Two address generators with eight auxiliary registers and two auxiliary register arithmetic units
- On-chip DMA controller for concurrent I/O and CPU operation
- Integer, floating-point, and logical operations
- Two- and three-operand instructions
- Parallel ALU and multiplier instructions in a single cycle

- Block repeat capability
- Zero-overhead loops with single-cycle branches
- Conditional calls and returns
- Interlocked instructions for multiprocessing support
- Two 32-bit data buses (24- and 13-bit address)
- Two serial ports to support 8/16/24/32-bit transfers
- Two 32-bit timers
- Two general-purpose external flags; four external interrupts
- 181-pin grid array (PGA) package; 1-μm CMOS

1.3 TMS320C31 Key Features

The TMS320C31 is a low-cost 32-bit DSP that offers the advantages of a floating-point processor and ease of use. The TMS320C31 devices are object-code compatible with the TMS320C30. Aside from lacking a ROM block and having a single serial port, the TMS320C31 is functionally equivalent to the TMS320C30 but differs in its respective electrical and timing characteristics. Chapter 13 describes these differences in detail.

- The TMS320C31 (33 MHz) features are identical to those of the TMS320C30 device, except that the TMS320C31 uses a subset of the TMS320C30's standard peripheral and memory interfaces. This maintains the 33-MFLOPS performance of the TMS320C30's core CPU while providing the cost advantages associated with 132-pin plastic quad flat pack (PQFP) packaging.
- The TMS320C31-27 is the slower speed version of the TMS320C31. The TMS320C31-27 delivers 27 MFLOPS and runs at 27 MHz. The reduced speed allows you to realize an immediate system cost reduction by using slower off-chip memories and a lower-cost processor.
- The TMS320C31-40 is a high-speed version of the TMS320C31. The 40-MHz TMS320C31-40 runs with 50-ns cycle time and offers up to 40 MFLOPS in performance.
- The TMS320C31-50 is the highest-speed version of the TMS320C31. The 50-MHz TMS320C31-50 runs with 40-ns cycle time and offers up to 50 MFLOPS in performance.
- The TMS320C31PQA (33 MHz) offers extended-temperature capabilities to TMS320C31 performance. The TMS320C31PQA will operate at case temperatures ranging from –40° C to +85° C, making it a lower-cost floating-point solution for industrial and extended-temperature commercial applications.
- The TMS320LC31 is the low-power version of the TMS320C31. The TMS320LC31 runs with 60-ns cycle time and offers up to 33 MFLOPS in performance at 3.3-volt operation.

Some key features of the TMS320C31, including those which differentiate it from the TMS320C30, are summarized as follows:

- Performance
  - TMS320C31 (PQL/PQA)
    - 60-ns, single-cycle instruction execution time
    - 33.3 MFLOPS
    - 16.7 MIPS (million instructions per second)

- TMS320C31-27
  - 74-ns, single-cycle instruction execution time
  - 27 MFLOPS
  - 13.5 MIPS
- TMS320C31-40
  - 50-ns, single-cycle instruction execution time
  - 40 MFLOPS
  - 20 MIPS
- TMS320C31-50
  - 40-ns, single-cycle instruction execution time
  - 50 MFLOPS
  - 25 MIPS
- TMS320LC31
  - 60-ns, single-cycle instruction execution time
  - 33.3 MFLOPS
  - 16.7 MIPS
  - Low-power, 3.3 volt operation
  - Two power-down nodes; 2-MHz operation and idle

- Flexible boot program loader
- One serial port to support 8-/16-/24-/32-bit transfers
- 132-pin PQFP package, .8 μm CMOS

1.4 Typical Applications

The TMS320 family's versatility, real-time performance, and multiple functions offer flexible design approaches in a variety of applications, which are shown in Table 1–1.

Table 1–1. 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	
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 Interpolation (DSI)	Adaptive Ride Control
Channel Multiplexing	X.25 Packet Switching	Global Positioning
1200- to 19200-bps Modems	Video Conferencing	Navigation
Adaptive Equalizers	Spread Spectrum	Voice Commands
DTMF Encoding/Decoding	Communications	Digital Radio
Data Encryption		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 Machines	Lathe Control	Fetal Monitors
	CAM	MR Imaging

Chapter 2

TMS320C3x Architecture

This chapter gives an architectural overview of the TMS320C3x processor.

Major areas of discussion are listed below.

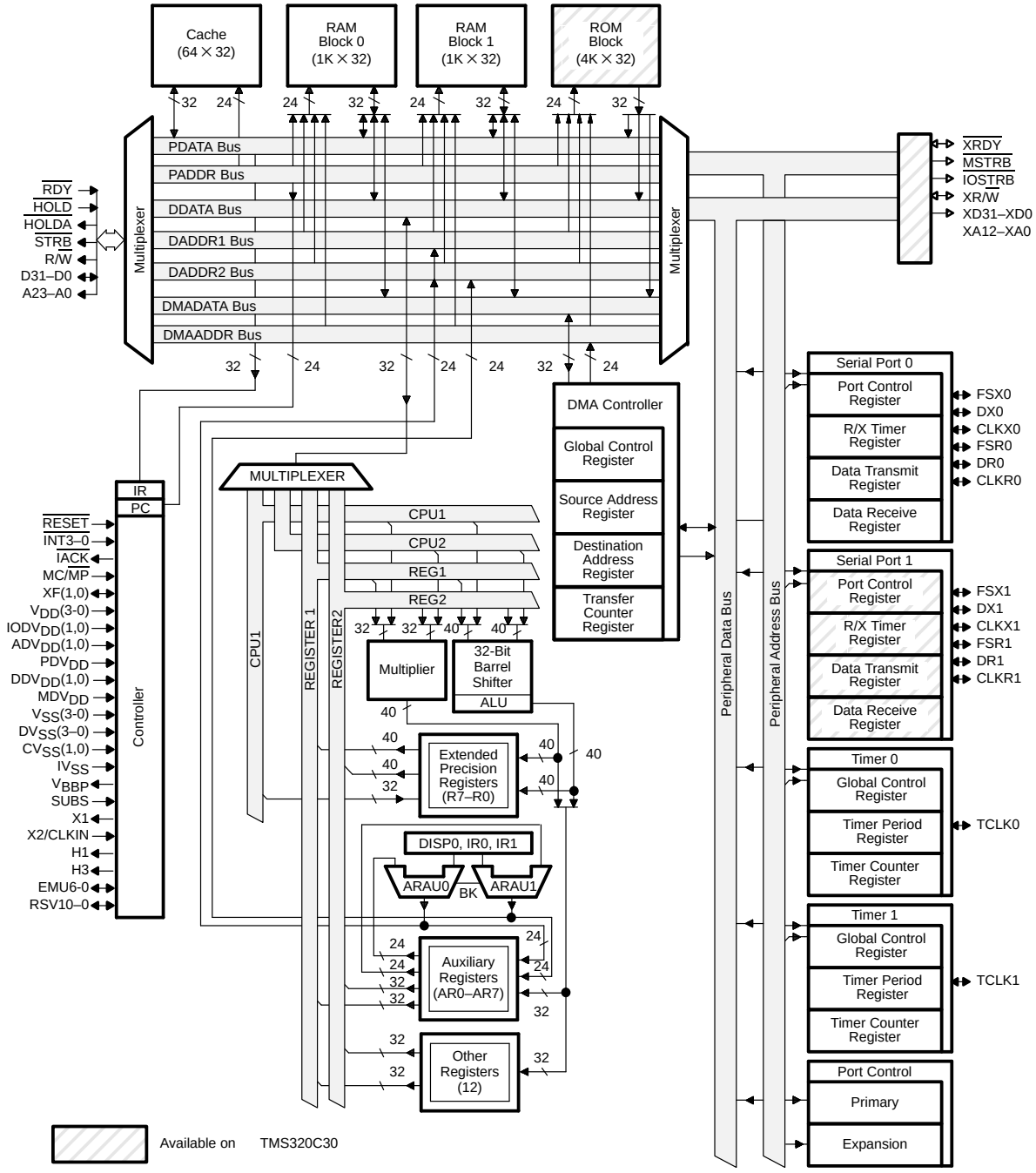
Topic **Page**

2.1 Architectural Overview

The TMS320C3x architecture responds to system demands that are based on sophisticated arithmetic algorithms and 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 direct memory access (DMA) controller.

Figure 2–1 is a block diagram of the TMS320C3x architecture.

Figure 2–1. TMS320C3x Block Diagram



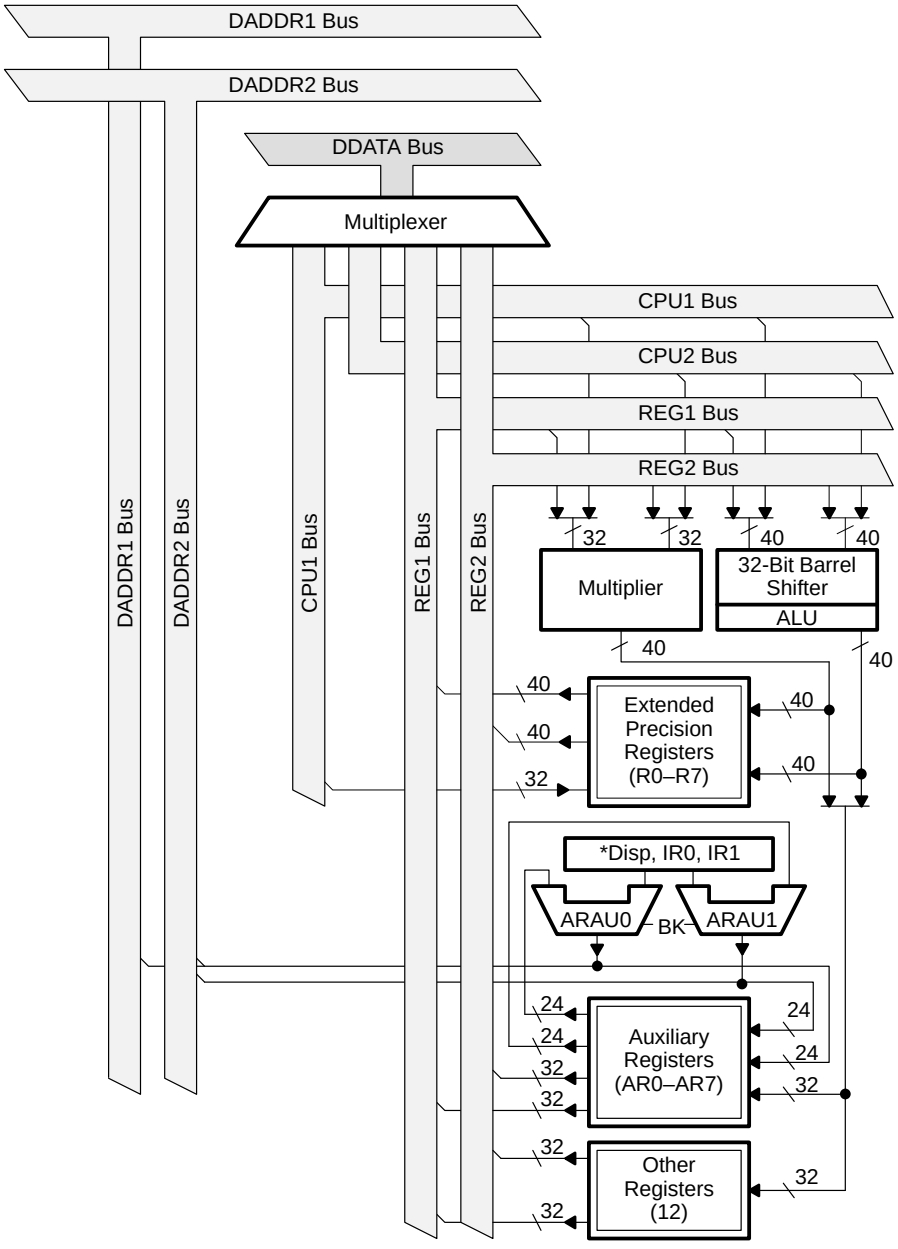
2.2 Central Processing Unit (CPU)

The TMS320C3x has a register-based central processing unit (CPU) architecture. The CPU consists of the following components:

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

Figure 2–2 shows the various CPU components that are discussed in the succeeding subsections.

Figure 2–2. Central Processing Unit (CPU)



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

2.2.1 Multiplier

The multiplier performs single-cycle multiplications on 24-bit integer and 32-bit floating-point values. The TMS320C3x implementation of floating-point arithmetic allows for floating-point operations at fixed-point speeds via a 50-ns instruction cycle and a high degree of parallelism. To gain even higher throughput, you can use parallel instructions to perform a multiply and 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. Refer to Chapter 4 for detailed information on data formats and floating-point operation.

2.2.2 Arithmetic Logic Unit (ALU)

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. Refer to Chapter 4 for detailed information on data formats and floating-point operation.

Internal buses, CPU1/CPU2 and REG1/REG2, carry two operands from memory and two operands from the register file, thus 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. Refer to Chapter 5 for a description of addressing modes.

2.2.4 CPU Register File

The TMS320C3x provides 28 registers in a multiport register file that is tightly coupled to the CPU. All of these registers can be operated upon by the multiplier and ALU and can be used as general-purpose registers. However, 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. Refer to Chapter 6 for detailed information and examples of stack management and register usage.

The register names and assigned functions are listed in Table 2–1. Following the table, the function of each register or group of registers is briefly described. Refer to Chapter 3 for detailed information on each of the CPU registers.

Table 2–1. CPU Registers

Register Name	Assigned Function
R0	Extended-precision register 0
R1	Extended-precision register 1
R2	Extended-precision register 2
R3	Extended-precision register 3
R4	Extended-precision register 4
R5	Extended-precision register 5
R6	Extended-precision register 6
R7	Extended-precision register 7
AR0	Auxiliary register 0
AR1	Auxiliary register 1
AR2	Auxiliary register 2
AR3	Auxiliary register 3
AR4	Auxiliary register 4
AR5	Auxiliary register 5
AR6	Auxiliary register 6
AR7	Auxiliary register 7
DP	Data-page pointer
IR0	Index register 0
IR1	Index register 1
BK	Block size
SP	System stack pointer
ST	Status register
IE	CPU/DMA interrupt enable
IF	CPU interrupt flags
IOF	I/O flags
RS	Repeat start address
RE	Repeat end address
RC	Repeat counter

The **extended-precision registers (R7–R0)** are capable of storing and supporting operations on 32-bit integer 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. Refer to Chapter 4 for extended-precision register formats for floating-point and integer numbers.

The 32-bit **auxiliary registers (AR7–AR0)** can be 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 can also be used as loop counters or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. Refer to Chapter 5 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 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. Refer to Chapter 5 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 of the system stack pointer; a pop performs a postdecrement. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH and POP instructions. Refer to Section 5.5 for information about system stack management.

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. This includes 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. Therefore, 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 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. Refer to subsection 3.1.8 for bit definitions.

The **CPU interrupt flag register (IF)** is also a 32-bit register (see subsection 3.1.9). 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.

The **I/O flags 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 subsection 3.1.10 for detailed information.

The **repeat counter (RC)** is a 32-bit register used to specify the number of times a block of code is to be repeated 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 be repeated, and the 32-bit **repeat end address register (RE)** contains the ending address of the block to be repeated.

The **program counter (PC)** is a 32-bit register containing the address of the next instruction to be fetched. 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.

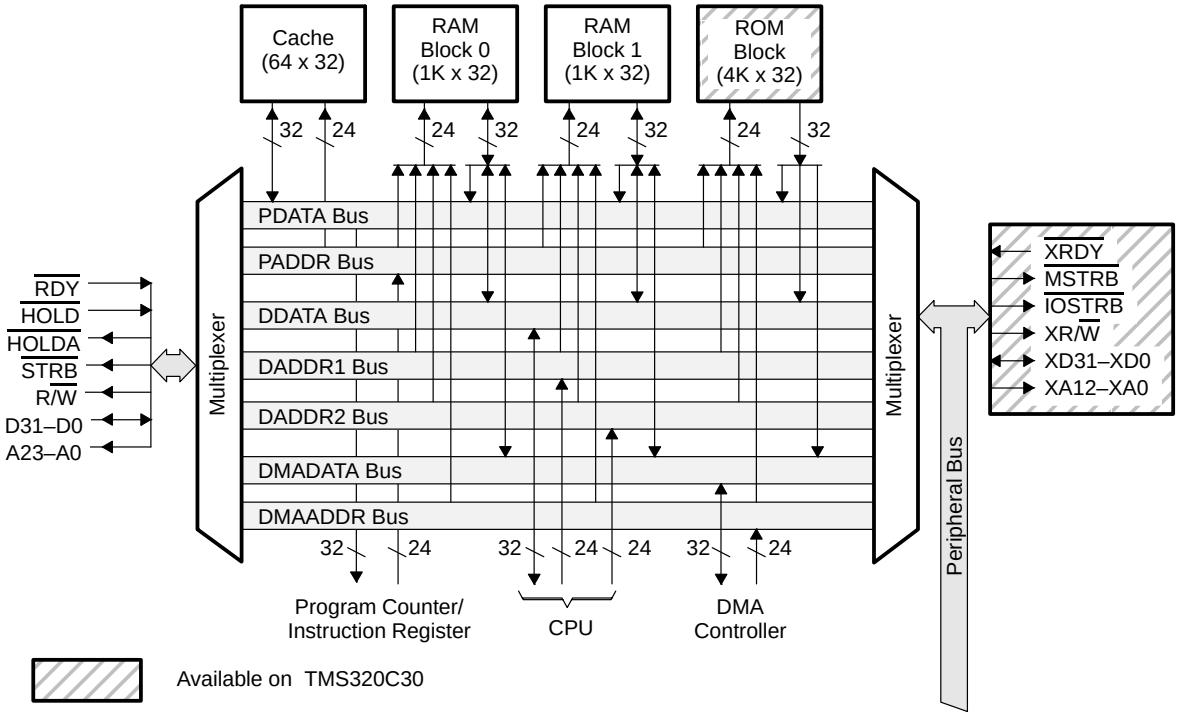
2.3 Memory Organization

The total memory space of the TMS320C3x is 16M (million) 32-bit words. Program, data, and I/O space are contained within this 16M-word address space, thus allowing tables, coefficients, program code, or data to be stored in either RAM or ROM. In this way, memory usage is maximized and memory space allocated as desired.

2.3.1 RAM, ROM, and Cache

Figure 2–3 shows how the memory is organized on the TMS320C3x. RAM blocks 0 and 1 are each 1K x 32 bits. The ROM block, available only on the TMS320C30, is 4K x 32 bits. Each RAM and ROM block is capable of supporting two CPU accesses in a single cycle. The separate program buses, data buses, 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 loading another RAM block, all within a single cycle.

Figure 2–3. Memory Organization



A 64 x 32-bit instruction cache is provided to store often-repeated sections of code, thus greatly reducing the number of off-chip accesses necessary. 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.

Refer to Chapter 3 for detailed information about the memory and instruction cache.

2.3.2 Memory Maps

The memory map depends on whether the processor is running in microprocessor mode (MC/$\overline{MP}$ or MCBL/$\overline{MP}$ = 0) or microcomputer mode (MC/$\overline{MP}$ or MCBL/$\overline{MP}$ = 1). The memory maps for these modes are similar (see Figure 2–4 and Figure 2–5). Locations 800000h–801FFFh are mapped to the expansion bus. When this region, available only on the TMS320C30, is accessed, $\overline{MSTRB}$ is active. Locations 802000h–803FFFh are reserved. Locations 804000h–805FFFh are mapped to the expansion bus. When this region, available only on the TMS320C30, is accessed, $\overline{IOSTRB}$ 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 ($\overline{STRB}$ active).

In microprocessor mode, the 4K on-chip ROM (TMS320C30) or boot loader (TMS320C31) is not mapped into the TMS320C3x memory map. Locations 0h–0BFh consist of interrupt vector, trap vector, and reserved locations, all of which are accessed over the external memory port ($\overline{STRB}$ active). Locations 0C0h–7FFFFFh are also accessed over the external memory port.

In microcomputer mode, the 4K on-chip ROM (TMS320C30) or boot loader (TMS320C31) is mapped into locations 0h–0FFFh. There are 192 locations (0h–0BFh) within this block for interrupt vectors, trap vectors, and a reserved space (TMS320C30). Locations 1000h–7FFFFFh are accessed over the external memory port ($\overline{STRB}$ active).

Section 3.2 on page 3-13 describes the memory maps in greater detail and provides the peripheral bus map and vector locations for reset, interrupts, and traps.

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

CAUTION

Figure 2–4. TMS320C30 Memory Maps

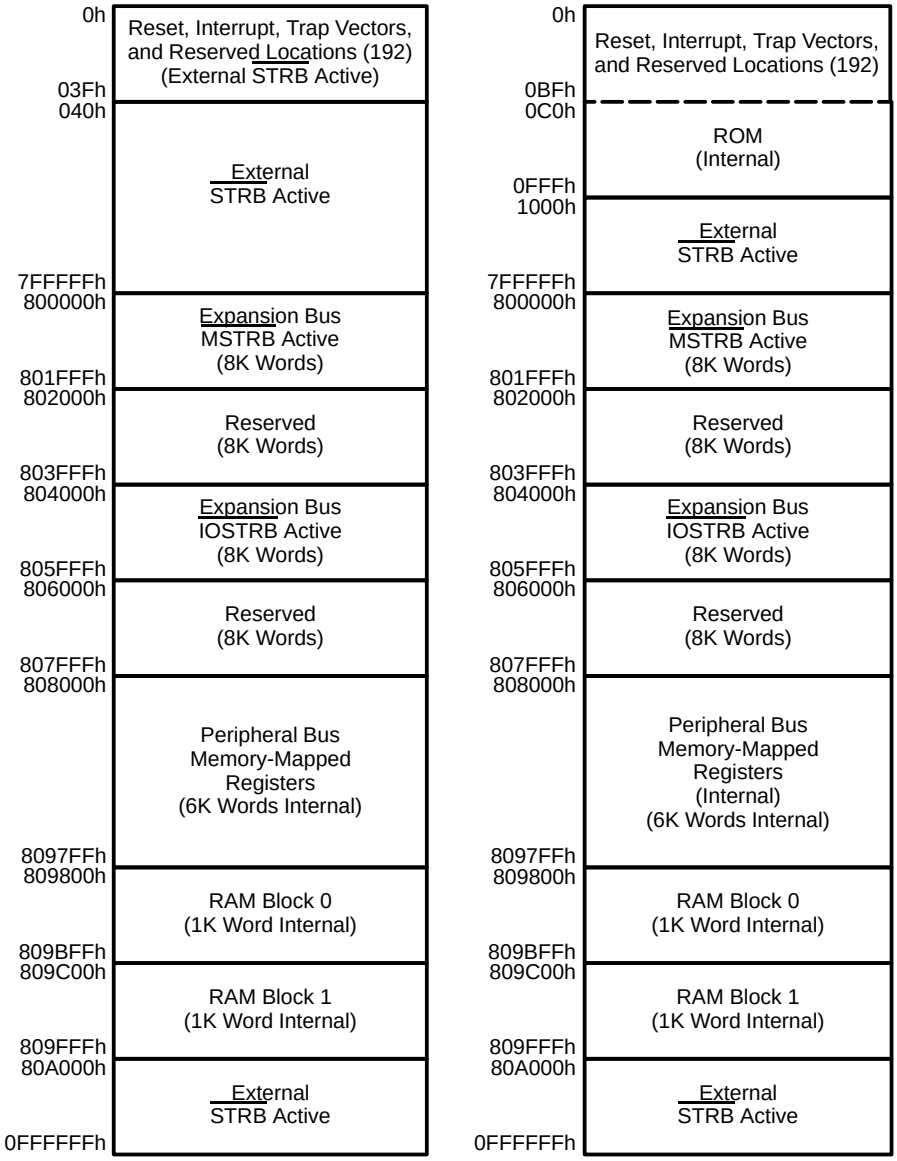
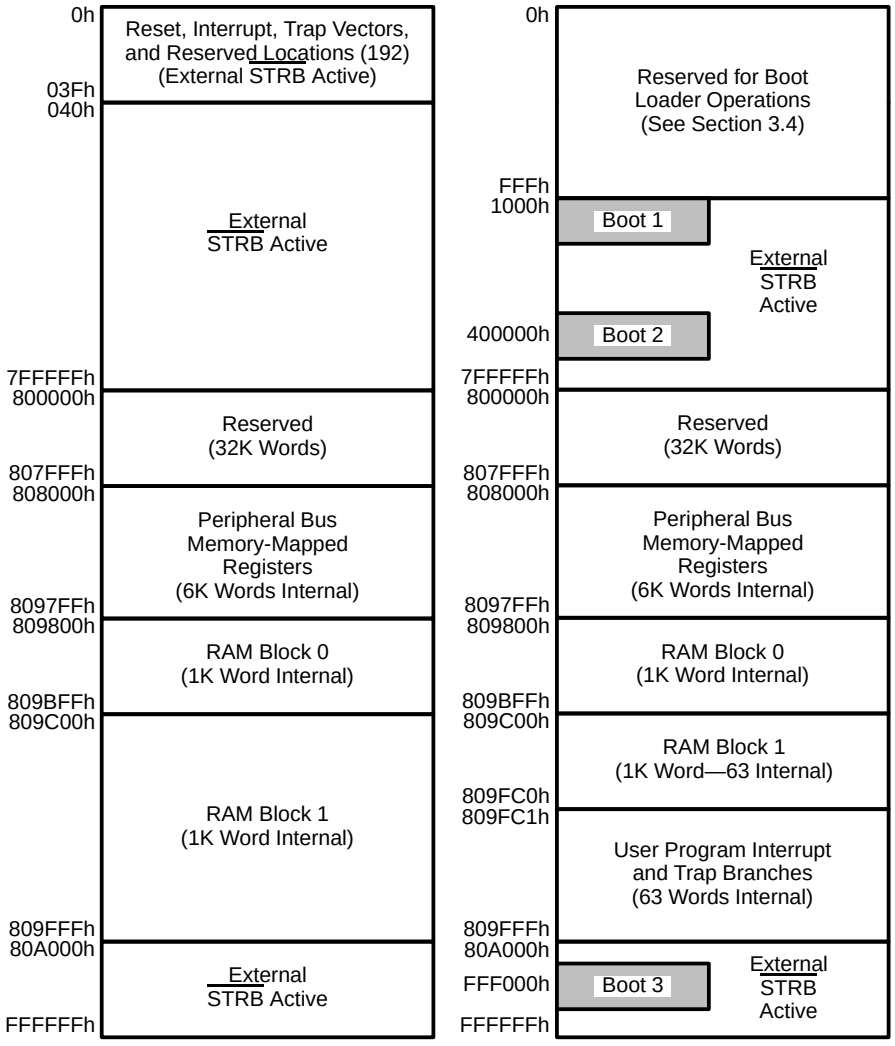


Figure 2–5. TMS320C31 Memory Maps



(a) Microprocessor Mode (b) Microcomputer/Boot Loader Mode

2.3.3 Memory Addressing Modes

The TMS320C3x 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. Refer to Chapter 5 for detailed information on addressing.

Five groups of addressing modes are provided on the TMS320C3x. Six types of addressing can be used within the groups, as shown in the following list:

- General addressing modes:
  - Register. The operand is a CPU register.
  - Short immediate. The operand is a 16-bit immediate value.
  - Direct. The operand is the contents of a 24-bit address.
  - Indirect. An auxiliary register indicates the address of the operand.
- Three-operand addressing modes:
  - Register. Same as for general addressing mode.
  - Indirect. Same as for general addressing mode.
- Parallel addressing modes:
  - Register. The operand is an extended-precision register.
  - Indirect. Same as for general addressing mode.
- Long-immediate addressing mode.

  The Long-immediate operand is a 24-bit immediate value.
- Conditional branch addressing modes:
  - Register. Same as for general addressing mode.
  - PC-relative. A signed 16-bit displacement is added to the PC.

2.4 Instruction Set Summary

Table 2–2 lists the TMS320C3x instruction set in alphabetical order. Each table entry shows the instruction mnemonic, description, and operation. Refer to Chapter 10 for a functional listing of the instructions and individual instruction descriptions.

Table 2–2. Instruction Set Summary

Mnemonic	Description	Operation
ABSF	Absolute value of a floating-point number	|*src*| → Rn
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* + Rn → Rn
ADDF3	Add floating-point values (3 operand)	*src*1 + *src*2 → Rn
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 $\overline{src}$ → Dreg
ANDN3	Bitwise logical ANDN (3 operand)	*src*1 AND $\overline{src2}$ → 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
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*condD*	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

Legend:

C	carry bit	C*src*	conditional-branch addressing modes
cond	condition code	count	shift value (general addressing modes)
Dreg	register address (any register)	PC	program counter
Rn	register address (R7–R0)	*src*	general addressing modes
*src*1	three-operand addressing modes	*src*2	three-operand addressing modes

Table 2–2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
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 Rn – *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
DB*cond*	Decrement and branch conditionally (standard)	ARn – 1 → ARn If *cond* = true and ARn ≥ 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*condD*	Decrement and branch conditionally (delayed)	ARn – 1 → ARn If *cond* = true and ARn ≥ 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*) → Rn
IACK	Interrupt acknowledge	Dummy read of *src* IACK toggled low, then high
IDLE	Idle until interrupt	PC + 1 → PC Idle until next interrupt
LDE	Load floating-point exponent	*src*(exponent) → Rn(exponent)
LDF	Load floating-point value	*src* → Rn
LDF*cond*	Load floating-point value conditionally	If *cond* = true, *src* → Rn Else, Rn is not changed
LDFI	Load floating-point value, interlocked	Signal interlocked operation *src* → Rn
LDI	Load integer	*src* → Dreg
LDI*cond*	Load integer conditionally	If *cond* = true, *src* → Dreg Else, Dreg is not changed

Legend:

ARn	auxiliary register n (AR7–AR0	Rn	register address (R7 — R0)
C*src*	conditional-branch addressing modes	*src*	general addressing modes
cond	condition code	*src*1	three-operand addressing modes
Dreg	register address (any register)	*src*2	three-operand addressing modes
PC	program counter	TOS	top of stack

Table 2–2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
LDII	Load integer, interlocked	Signal interlocked operation *src* → Dreg
LDM	Load floating-point mantissa	*src* (mantissa) → Rn (mantissa)
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
MPYF	Multiply floating-point values	*src* × Rn → Rn
MPYF3	Multiply floating-point value (3 operand)	*src*1 × *src*2 → Rn
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* → Rn
NEGI	Negate integer	0 – *src* → Dreg
NOP	No operation	Modify ARn if specified
NORM	Normalize floating-point value	Normalize (*src*) → Rn
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–– → Rn
PUSH	Push integer on stack	Sreg → *++ SP
PUSHF	Push floating-point value on stack	Rn → *++ SP

Legend:

ARn	auxiliary register n (AR7–AR0)	SP	stack pointer
C	carry bit	Sreg	register address (any register)
Dreg	register address (any register)	*src*	general addressing modes
PC	program counter	*src*1	3-operand addressing modes
Rn	register address (R7–R0)	*src*2	3-operand addressing modes

Table 2–2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
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*) → Rn
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
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	Rn → Daddr
STFI	Store floating-point value, interlocked	Rn → 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

Legend:

C	carry bit	RM	repeat mode bit
cond	condition code	RS	repeat start register
Daddr	destination memory address	Rn	register address (R7–R0)
Dreg	register address (any register)	SP	stack pointer
GIE	global interrupt enable register	ST	status register
PC	program counter	Sreg	register address (any register)
RC	repeat counter register	*src*	general addressing modes
RE	repeat interrupt register		

Table 2–2. Instruction Set Summary (Concluded)

Mnemonic	Description	Operation
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	Rn – *src* → Rn
SUBF3	Subtract floating-point values (3 operand)	*src*1 – *src*2 → Rn
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* – Rn → Rn
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:

C	carry bit	Rn	register address (R7–R0)
cond	condition code	SP	stack pointer
Dreg	register address (any register)	*src*	general addressing modes
GIE	global interrupt enable register	*src*1	3-operand addressing modes
N	any trap vector 0–27	*src*2	3-operand addressing modes
PC	program counter	ST	status register

2.5 Internal Bus Operation

Much of the TMS320C3x's high performance is due to internal busing and parallelism. The separate program buses (PADDR and PDATA), data buses (DADDR1, DADDR2, and DDATA), and DMA buses (DMAADDR and DMADATA) allow for parallel program fetches, data accesses, and DMA accesses. These buses connect all of the physical spaces (on-chip memory, off-chip memory, and on-chip peripherals) supported by the TMS320C30. Figure 2–3 shows these internal buses and their connection to on-chip and off-chip memory blocks.

The 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 2–2 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.

2.6 Parallel Instruction Set Summary

Table 2–3 lists the ’C3x instruction set in alphabetical order. Each table entry shows the instruction mnemonic, description, and operation. Refer to Section 10.3 on page -14 for a functional listing of the instructions and individual instruction descriptions.

Table 2–3. Parallel Instruction Set Summary

Mnemonic	Description	Operation
	Parallel Arithmetic With Store Instructions	
ABSF || STF	Absolute value of a floating point	|*src*2| → *dst*1 || *src*3 → *dst*2
ABSI || STI	Absolute value of an integer	|*src*2| → *dst*1 || *src*3 → *dst*2
ADDF3 || STF	Add floating point	*src*1 + *src*2 → *dst*1 || *src*3 → *dst*2
ADDI3 || STI	Add integer	*src*1 + *src*2 → *dst*1 || *src*3 → *dst*2
AND3 || STI	Bitwise logical AND	*src*1 AND *src*2 → *dst*1 || *src*3 → *dst*2
ASH3 || STI	Arithmetic shift	If count ≥ 0: *src*2 << count → *dst*1 || *src*3 → *dst*2 Else: *src*2 >> |count| → *dst*1 || *src*3 → *dst*2
FIX || STI	Convert floating point to integer	Fix(*src*2) → *dst*1 || *src*3 → *dst*2
FLOAT || STF	Convert integer to floating point	Float(*src*2) → *dst*1 || *src*3 → *dst*2
LDF || STF	Load floating point	*src*2 → *dst*1 || *src*3 → *dst*2
LDI || STI	Load integer	*src*2 → *dst*1 || *src*3 → *dst*2
LSH3 || STI	Logical shift	If count ≥ 0: *src*2 << count → *dst*1 || *src*3 → *dst*2 Else: *src*2 >> |count| → *dst*1 || *src*3 → *dst*2
MPYF3 || STF	Multiply floating point	*src*1 x *src*2 → *dst*1 || *src*3 → *dst*2
MPYI3 || STI	Multiply integer	*src*1 x *src*2 → *dst*1 || *src*3 → *dst*2

Legend:

count	register addr (R7–R0)	*src*1	register addr (R7–R0)
*dst*1	register addr (R7–R0)	*src*2	indirect addr (disp = 0, 1, IR0, IR1)
*dst*2	indirect addr (disp = 0, 1, IR0, IR1)	*src*3	register addr (R7–R0)

Table 2–3. Parallel Instruction Set Summary (Continued)

Mnemonic	Description	Operation
	Parallel Arithmetic With Store Instructions (Concluded)	
NEGF || STF	Negate floating point	0– *src*2 → *dst*1 || *src*3 → *dst*2
NEGI || STI	Negate integer	0 – *src*2 → *dst*1 || *src*3 → *dst*2
NOT || STI	Complement	$\overline{src1}$ → *dst*1 || *src*3 → *dst*2
OR3 || STI	Bitwise logical OR	*src*1 OR *src*2 → *dst*1 || *src*3 → *dst*2
STF || STF	Store floating point	*src*1 → *dst*1 || *src*3 → *dst*2
STI || STI	Store integer	*src*1 → *dst*1 || *src*3 → *dst*2
SUBF3 || STF	Subtract floating point	*src*1 – *src*2 → *dst*1 || *src*3 → *dst*2
SUBI3 || STI	Subtract integer	*src*1 – *src*2 → *dst*1 || *src*3 → *dst*2
XOR3 || STI	Bitwise exclusive OR	*src*1 XOR *src*2 → *dst*1 || *src*3 → *dst*2
	Parallel Load Instructions	
LDF || LDF	Load floating point	*src*2 → *dst*1 || *src*4 → *dst*2
LDI || LDI	Load integer	*src*2 → *dst*1 || *src*4 → *dst*2
	Parallel Multiply And Add/Subtract Instructions	
MPYF3 || ADDF3	Multiply and add floating point	op1 x op2 → op3 || op4 + op5 → op6
MPYF3 || SUBF3	Multiply and subtract floating point	op1 x op2 → op3 || op4 – op5 → op6
MPYI3 || ADDI3	Multiply and add integer	op1 x op2 → op3 || op4 + op5 → op6
MPYI3 || SUBI3	Multiply and subtract integer	op1 x op2 → op3 || op4 – op5 → op6

Legend:

*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)

2.7 External Bus Operation

The TMS320C30 provides two external interfaces: the primary bus and the expansion bus. The TMS320C31 provides one external interface: the primary bus. Both primary and expansion buses consist of a 32-bit data bus and a set of control signals. The primary bus has a 24-bit address bus, whereas the expansion bus has a 13-bit address bus. Both buses can be used to address external program/data memory or I/O space. The buses also have an external $\overline{\text{RDY}}$ signal for wait-state generation. You can insert additional wait states under software control. Refer to Chapter 7 for detailed information on external bus operation.

2.7.1 External Interrupts

The TMS320C3x 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 6.5 (beginning on page 6-18) covers $\overline{\text{RESET}}$ and interrupt processing.

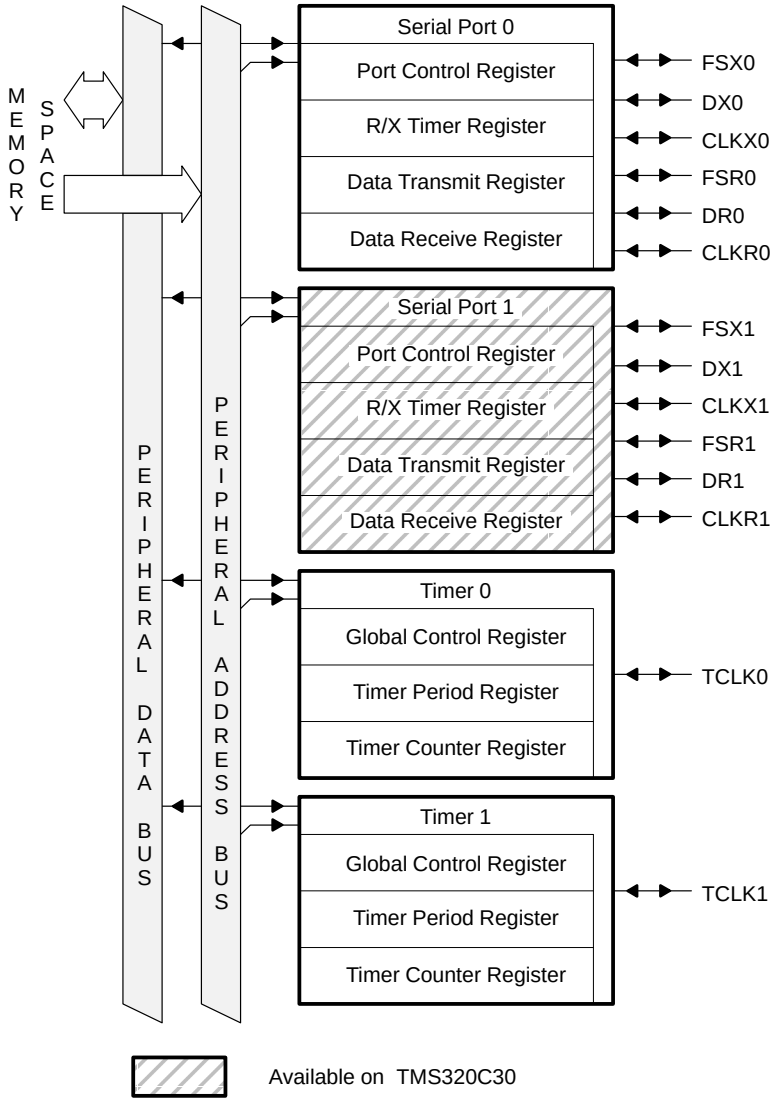
2.7.2 Interlocked-Instruction Signaling

Two external I/O flags, XF0 and XF1, can be configured as input or output pins under software control. These pins are also used by the interlocked operations of the TMS320C3x. The interlocked-operations instruction group supports multiprocessor communication (see Section 6.4 on page 6-12 for examples of the use of interlocked instructions).

2.8 Peripherals

All TMS320C3x 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 TMS320C3x peripherals include two timers and two serial ports (only one serial port is available on the TMS320C31). Figure 2–6 shows the peripherals with associated buses and signals. Refer to Chapter 8 for detailed information on the peripherals.

Figure 2–6. Peripheral Modules



2.8.1 Timers

The two timer modules are general-purpose 32-bit timer/event counters with two signaling modes and internal or external clocking. Each timer has an I/O pin that can be used as an input clock to the timer or as an output signal driven by the timer. The pin can also be configured as a general-purpose I/O pin.

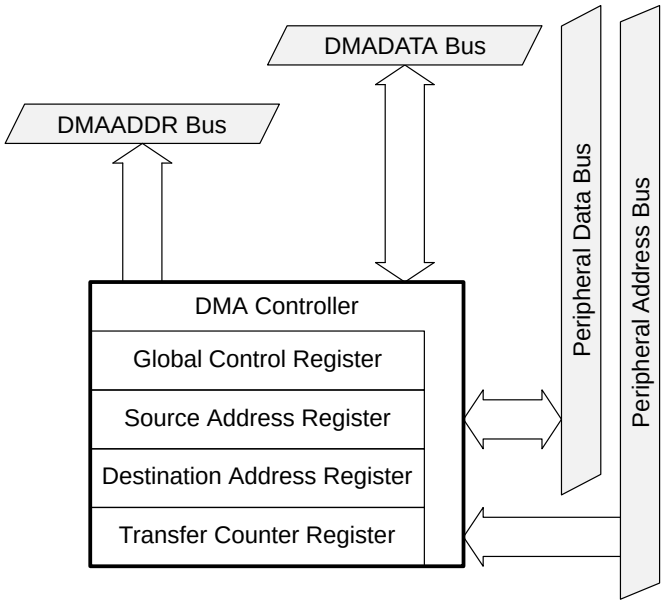
2.8.2 Serial Ports

The two bidirectional serial ports 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 serial port pins are configurable as general-purpose I/O pins. The serial ports can also be configured as timers. A special handshake mode allows TMS320C3xs to communicate over their serial ports with guaranteed synchronization.

2.9 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 operation of the CPU. Therefore, the TMS320C3x 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. Refer to Section 8.3 on page 8-43 for detailed information on the DMA controller. Figure 2–7 shows the DMA controller with associated buses.

Figure 2–7. DMA Controller



2.10 TMS320C30 and TMS320C31 Differences

This section addresses the major memory access differences between the TMS320C31 and the TMS320C30 devices. Observance of these considerations is critical for achieving design goal success.

Table 2–4 shows these differences, which are detailed in the following subsections.

Table 2–4. Feature Set Comparison

Feature	TMS320C31	TMS320C30
Data/program bus	Primary bus: one bus composed of a 32-bit data and a 24-bit address bus	Two buses: • Primary bus: a 32-bit data and a 24-bit address • Expansion bus: a 32-bit data and a 13-bit address
Serial I/O ports	1 serial port (SP0)	2 serial ports (SP0, SP1)
User program/data ROM	Not available	4K words/16K bytes
Program boot loader	User selectable	Not available

2.10.1 Data/Program Bus Differences

The TMS320C31 uses *only the primary bus* and reserves the memory space that was previously used for expansion bus operations.

Be careful! Program access to a reserved area produces unpredictable results.

CAUTION

2.10.2 Serial-Port Differences

Serial port 1 references in Section 8.2 are not applicable to the TMS320C31. The memory locations identified for the associated control registers and buffers are reserved.

2.10.3 Reserved Memory Locations

Table 2–5 identifies TMS320C31 reserved memory locations in addition to those shown in Figure 3–8 on page 3-16.

Table 2–5. TMS320C31 Reserved Memory Locations

Feature	TMS320C31	TMS320C30
0x000000–0x000FFF	Reserved†	Microcomputer program/data ROM mode†
0x800000–0x801FFF	Reserved	Expansion bus $\overline{\text{MSTRB}}$ space
0x804000–0x805FFF	Reserved	Expansion bus $\overline{\text{IOSTRB}}$ space
0x808050	Reserved	SP1 global-control register
0x808052–0x808056	Reserved	SP1 local-control registers
0x808058	Reserved	SP1 data-transmit buffer
0x80805C	Reserved	SP1 receive-transmit buffer
0x808060	Reserved	Expansion bus control register

† Applies to the MCBL and MC modes only.

2.10.4 Effects on the IF and IE Interrupt Registers

The bits associated with serial port 1 in the IE (interrupt enable) register and the IF (interrupt flag) register for the TMS320C30 are not applicable to the TMS320C31. Write only logic 0 data to IE register bits 6, 7, 22, and 23 and to IF register bits 6 and 7. Writing logic 1s to these bits produces unpredictable results.

2.10.5 User Program/Data ROM

The user program/data ROM that is available for the TMS320C30 device does not exist for the TMS320C31. Rather, the memory locations that were allocated to support user program/data ROM operations have been reserved on the TMS320C31 to support microcomputer/boot loader accessing. See Chapter 3 for more information on using the microcomputer/boot loader function.

2.10.6 Development Considerations

If you are developing application code using a TMS320C3x simulator, XDS, or ASM/LNK, TI recommends that you modify the .cfm and .cmd files by removing these memory spaces from the tool's configured memory. This ensures that your developed application performs as expected when the TMS320C31 device is used.

2.11 System Integration

In summary, the TMS320C3x is a powerful DSP system that integrates an innovative, high-performance CPU, two external interface ports, large memories, and efficient buses to support its speed. A single chip contains this system, along with peripherals such as a DMA controller, two serial ports, and two timers. The TMS320C3x system is truly an affordable single-chip solution.

Chapter 3

CPU Registers, Memory, and Cache

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. The registers in the CPU register file support addressing, floating-point/integer operations, stack management, processor status, block repeats, and interrupts.

The TMS320C3x provides a total memory space of 16M (million) 32-bit words containing program, data, and I/O space. Two RAM blocks of 1K x 32 bits each and a ROM block of 4K x 32 bits (available only on the TMS320C30) permit two CPU accesses in a single cycle. The memory maps for the microcomputer and microprocessor modes are similar, except that the on-chip ROM is not used in the microprocessor mode.

A 64- x 32-bit instruction cache stores often-repeated sections of code. This greatly reduces the number of off-chip accesses and allows code to be stored off-chip in slower, lower-cost memories. Three bits in the CPU status register control the clear, enable, or freeze of the cache.

This chapter describes in detail each of the CPU registers, the memory maps, and the instruction cache. Major topics are as follows:

Topic **Page**

3.1 CPU Register File

The TMS320C3x 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. However, 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. Refer to Chapter 5 for detailed information and examples of the use of CPU registers in addressing.

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

Table 3–1. CPU Registers

Register	Assigned Function Name
R0	Extended-precision register 0
R1	Extended-precision register 1
R2	Extended-precision register 2
R3	Extended-precision register 3
R4	Extended-precision register 4
R5	Extended-precision register 5
R6	Extended-precision register 6
R7	Extended-precision register 7
AR0	Auxiliary register 0
AR1	Auxiliary register 1
AR2	Auxiliary register 2
AR3	Auxiliary register 3
AR4	Auxiliary register 4
AR5	Auxiliary register 5
AR6	Auxiliary register 6
AR7	Auxiliary register 7
DP	Data-page pointer
IR0	Index register 0
IR1	Index register 1
BK	Block-size register
SP	System stack pointer
ST	Status register
IE	CPU/DMA interrupt enable
IF	CPU interrupt flags
IOF	I/O flags
RS	Repeat start address
RE	Repeat end address
RC	Repeat counter

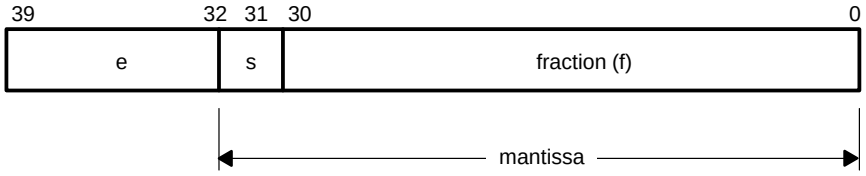
3.1.1 Extended-Precision Registers (R7–R0)

The eight extended-precision registers (R7–R0) are capable of storing and supporting 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)

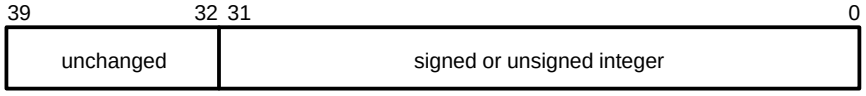
Any instruction that assumes the operands are floating-point numbers uses bits 39–0. Figure 3–1 illustrates the storage of 40-bit floating-point numbers in the extended-precision registers.

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 eight 32-bit auxiliary registers (AR7–AR0) can be accessed by the CPU and modified by the two Auxiliary Register Arithmetic Units (ARAUs). The primary function of the auxiliary registers is the generation of 24-bit addresses. However, they can also be used as loop counters in indirect addressing or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. Refer to Chapter 5 for detailed information and examples of the use of auxiliary registers in addressing.

3.1.3 Data-Page Pointer (DP)

The data-page pointer (DP) is a 32-bit register that is loaded using the LDP instruction. 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. Data pages are 64K words long, with a total of 256 pages. Bits 31–8 are reserved; you should 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. Refer to Chapter 5 for detailed information and examples of the use of index registers in addressing.

3.1.5 Block Size Register (BK)

The 32-bit block size register (BK) is used by the ARAU in circular addressing to specify the data block size (see Section 5.3 on page 5-24).

3.1.6 System Stack Pointer (SP)

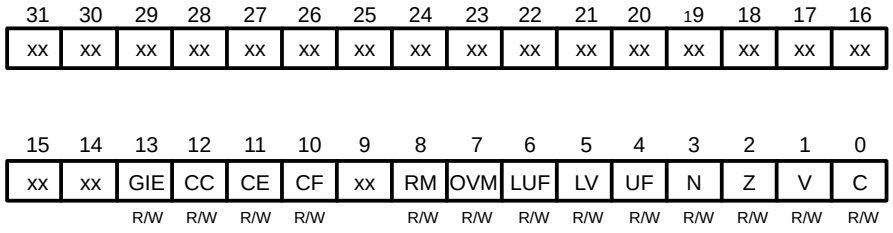
The system stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH, PUSHF, POP, and POPF instructions. Pushes and pops of the stack perform preincrement and postdecrement, respectively, on all 32 bits of the stack pointer. However, only the 24 LSBs are used as an address. Refer to Section 5.5 on page 5-31 for information about system stack management.

3.1.7 Status Register (ST)

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. This includes register load and store operations as well as arithmetic and logical functions. When the status register is loaded, however, the contents of the source operand replace the current contents bit-for-bit, regardless of the state of any bits in the source operand. Therefore, following a load, the contents of the status register are identically equal to the contents of the source operand. This allows the status register to be saved easily and restored. At system reset, 0 is written to this register.

Figure 3–3 shows the format of the status register. Table 3–2 defines the status register bits, their names, and their functions.

Figure 3–3. Status Register



Notes: 1) xx = reserved bit, read as 0
2) R = read, W = write

Table 3–2. Status Register Bits Summary

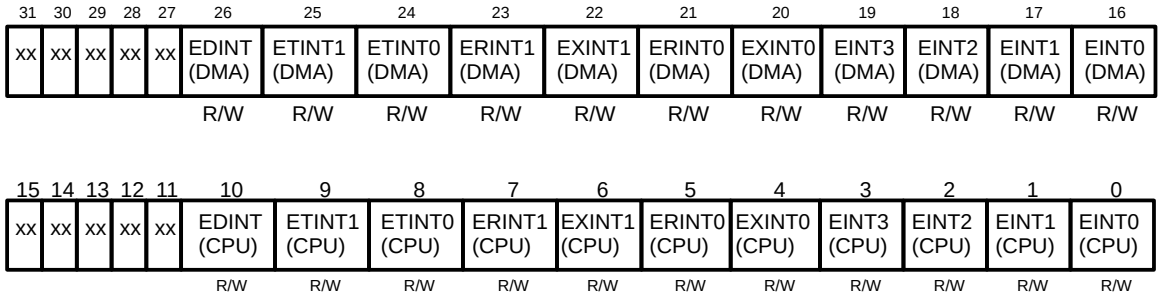
Bit	Name	Reset Value	Function
0†	C	0	Carry flag
1†	V	0	Overflow flag
2†	Z	0	Zero flag
3†	N	0	Negative flag
4†	UF	0	Floating-point underflow flag
5†	LV	0	Latched overflow flag
6†	LUF	0	Latched floating-point underflow flag
7	OVM	0	Overflow mode flag. This flag affects only the integer operations. If OVM = 0, the overflow mode is turned off; integer results that overflow are treated in no special way. If OVM = 1, a) integer results overflowing in the positive direction are set to the most positive 32-bit twos-complement number (7FFFFFFFh), and b) integer results overflowing in the negative direction are set to the most negative 32-bit twos-complement number (80000000h). Note that the function of V and LV is independent of the setting of OVM.
8	RM	0	Repeat mode flag. If RM = 1, the PC is being modified in either the repeat-block or repeat-single mode.
9	Reserved	0	Read as 0
10	CF	0	Cache freeze. When CF = 1, the cache is frozen. If the cache is enabled (CE = 1), fetches from the cache are allowed, but no modification of the state of the cache is performed. This function can be used to save frequently used code resident in the cache. At reset, 0 is written to this bit. Cache clearing (CC = 1) is allowed when CF = 0.
11	CE	0	Cache enable. CE = 1 enables the cache, allowing the cache to be used according to the least recently used (LRU) cache algorithm. CE = 0 disables the cache; no update or modification of the cache can be performed. No fetches are made from the cache. This function is useful for system debugging. At system reset, 0 is written to this bit. Cache clearing (CC = 1) is allowed when CE = 0.
12	CC	0	Cache clear. CC = 1 invalidates all entries in the cache. This bit is always cleared after it is written to and thus always read as 0. At reset, 0 is written to this bit.
13	GIE	0	Global interrupt enable. If GIE = 1, the CPU responds to an enabled interrupt. If GIE = 0, the CPU does not respond to an enabled interrupt.
15–14	Reserved	0	Read as 0
31–16	Reserved	0–0	Value undefined

† The seven condition flags (ST bits 6–0) are defined in Section 10.2 on page -10.

3.1.8 CPU/DMA Interrupt Enable Register (IE)

The CPU/DMA interrupt enable register (IE) is a 32-bit register (see Figure 3–4). The CPU interrupt enable bits are in locations 10–0. The direct memory access (DMA) interrupt enable bits are in locations 26–16. A 1 in a CPU/DMA IE register bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. At reset, 0 is written to this register. Table 3–3 defines the register bits, the bit names, and the bit functions.

Figure 3–4. CPU/DMA Interrupt Enable Register (IE)



Notes: 1) xx = reserved bit, read as 0
2) R = read, W = write

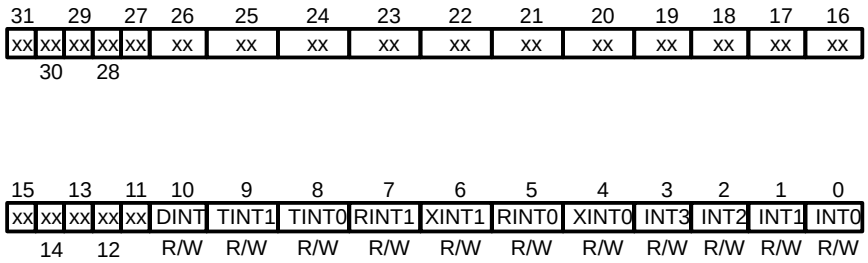
Table 3–3. IE Register Bits Summary

Bit	Name	Reset Value	Function
0	EINT0	0	Enable external interrupt 0 (CPU)
1	EINT1	0	Enable external interrupt 1 (CPU)
2	EINT2	0	Enable external interrupt 2 (CPU)
3	EINT3	0	Enable external interrupt 3 (CPU)
4	EXINT0	0	Enable serial-port 0 transmit interrupt (CPU)
5	ERINT0	0	Enable serial-port 0 receive interrupt (CPU)
6	EXINT1	0	Enable serial-port 1 transmit interrupt (CPU)
7	ERINT1	0	Enable serial-port 1 receive interrupt (CPU)
8	ETINT0	0	Enable timer 0 interrupt (CPU)
9	ETINT1	0	Enable timer 1 interrupt (CPU)
10	EDINT	0	Enable DMA controller interrupt (CPU)
15–11	Reserved	0	Value undefined
16	EINT0	0	Enable external interrupt 0 (DMA)
17	EINT1	0	Enable external interrupt 1 (DMA)
18	EINT2	0	Enable external interrupt 2 (DMA)
19	EINT3	0	Enable external interrupt 3 (DMA)
20	EXINT0	0	Enable serial-port 0 transmit interrupt (DMA)
21	ERINT0	0	Enable serial-port 0 receive interrupt (DMA)
22	EXINT1	0	Enable serial-port 1 transmit interrupt (DMA)
23	ERINT1	0	Enable serial-port 1 receive interrupt (DMA)
24	ETINT0	0	Enable timer 0 interrupt (DMA)
25	ETINT1	0	Enable timer 1 interrupt (DMA)
26	EDINT	0	Enable DMA controller interrupt (DMA)
31–27	Reserved	0–0	Value undefined

3.1.9 CPU Interrupt Flag Register (IF)

Figure 3–5 shows the 32-bit CPU interrupt flag register (IF). 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 lists the bit fields, bit-field names, and bit-field functions of the CPU IF register.

Figure 3–5. CPU Interrupt-Flag Register (IF)



Notes: 1) xx = reserved bit, read as 0
2) R = read, W = write

Table 3–4. IF Register Bits Summary

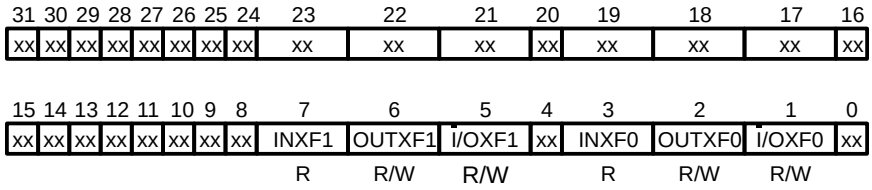
Bit	Name	Reset Value	Function
0	INT0	0	External interrupt 0 flag
1	INT1	0	External interrupt 1 flag
2	INT2	0	External interrupt 2 flag
3	INT3	0	External interrupt 3 flag
4	XINT0	0	Serial-port 0 transmit interrupt flag
5	RINT0	0	Serial-port 0 receive interrupt flag
6	XINT1†	0	Serial-port 1 transmit interrupt flag
7	RINT1†	0	Serial-port 1 receive interrupt flag
8	TINT0	0	Timer 0 interrupt flag
9	TINT1	0	Timer 1 interrupt flag
10	DINT	0	DMA channel interrupt flag
31–11	Reserved	0–0	Value undefined

† Reserved on TMS320C31

3.1.10 I/O Flags Register (IOF)

The I/O flags register (IOF) is shown in Figure 3–6 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 shows the bit fields, bit-field names, and bit-field functions.

Figure 3–6. I/O-Flag Register (IOF)



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

2) R = read, W = write

Table 3–5. IOF Register Bits Summary

Bit	Name	Reset Value	Function
0	Reserved	0	Read as 0
1	$\overline{I}$/OXF0	0	If $\overline{I}$/OXF0 = 0, XF0 is configured as a general-purpose input pin. If $\overline{I}$/OXF0 = 1, XF0 is configured as a general-purpose output pin.
2	OUTXF0	0	Data output on XF0
3	INXF0	0	Data input on XF0. A write has no effect.
4	Reserved	0	Read as 0
5	$\overline{I}$/OXF1	0	If $\overline{I}$/OXF1 = 0, XF1 is configured as a general-purpose input pin. If $\overline{I}$/OXF1 = 1, XF1 is configured as a general-purpose output pin.
6	OUTXF1	0	Data output on XF1
7	INXF1	0	Data input on XF1. A write has no effect.
31–8	Reserved	0–0	Read as 0

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

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

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

Note: RE < RS

If RE < RS, the block of program memory will not be repeated, and the code will not loop backwards. However, the ST(RM) bit remains set to 1.

The repeat-count register (RC) is a 32-bit register used to specify 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.

3.1.12 Program Counter (PC)

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

3.1.13 Reserved Bits and Compatibility

To retain compatibility with future members of the TMS320C3x family of microprocessors, reserved bits that are read as 0 must be written as 0. A reserved bit that has an undefined value must not have its current value modified. In other cases, you should maintain the reserved bits as specified.

3.2 Memory

The TMS320C3x's total memory space of 16M (million) 32-bit words contains program, data, and I/O space, allowing 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 x 32 bits. The ROM block is 4K x 32 bits. Each on-chip RAM and ROM block is capable of supporting 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. Chapter 9 covers this in detail.

3.2.1 TMS320C3x Memory Maps

The memory map depends on whether the processor is running in microprocessor mode (MC/$\overline{\text{MP}}$ or MCBL/$\overline{\text{MP}}$ = 0) or microcomputer mode (MC/$\overline{\text{MP}}$ or MCBL/$\overline{\text{MP}}$ = 1). The memory maps for these modes are similar (see Figure 3–7). Locations 800000h through 801FFFh are mapped to the expansion bus. When this region, available only on the TMS320C30, is accessed, $\overline{\text{MSTRB}}$ is active. Locations 802000h through 803FFFh are reserved. Locations 804000h through 805FFFh are mapped to the expansion bus. When this region, available only on the TMS320C30, is accessed, $\overline{\text{IOSTRB}}$ is active. Locations 806000h through 807FFFh are reserved. All of the memory-mapped peripheral registers are in locations 808000h through 8097FFh. In both modes, RAM block 0 is located at addresses 809800h through 809BFFh, and RAM block 1 is located at addresses 809C00h through 809FFFh. Memory locations 80A000h through 0FFFFFFh are accessed over the primary external memory port ($\overline{\text{STRB}}$ active).

In **microprocessor mode**, the 4K on-chip ROM (TMS320C30) or boot loader (TMS320C31) is not mapped into the TMS320C3x memory map. As shown in Figure 3–7, locations 0h through 03Fh consist of interrupt vector, trap vector, and reserved locations, all of which are accessed over the primary external memory port ($\overline{\text{STRB}}$ active). Interrupt and trap vector locations are shown in Figure 3–9. Locations 040h–7FFFFFh and 80A000L–FFFFFFh are also accessed over the primary external memory port.

In **microcomputer mode**, the 4K on-chip ROM (TMS320C30) or boot loader (TMS320C31) is mapped into locations 0h through 0FFFh. There are 192 locations (0h through BFh) within this block for interrupt vectors, trap vectors, and a reserved space. Locations 1000h–7FFFFFh are accessed over the primary external memory port ($\overline{\text{STRB}}$ active).

Reserved Spaces

Do not read and write to reserved portions of the TMS320C3x memory space and reserved peripheral bus addresses. Doing so might cause the TMS320C3x to halt operation and require a system reset to restart.

Figure 3–7. TMS320C30 Memory Maps

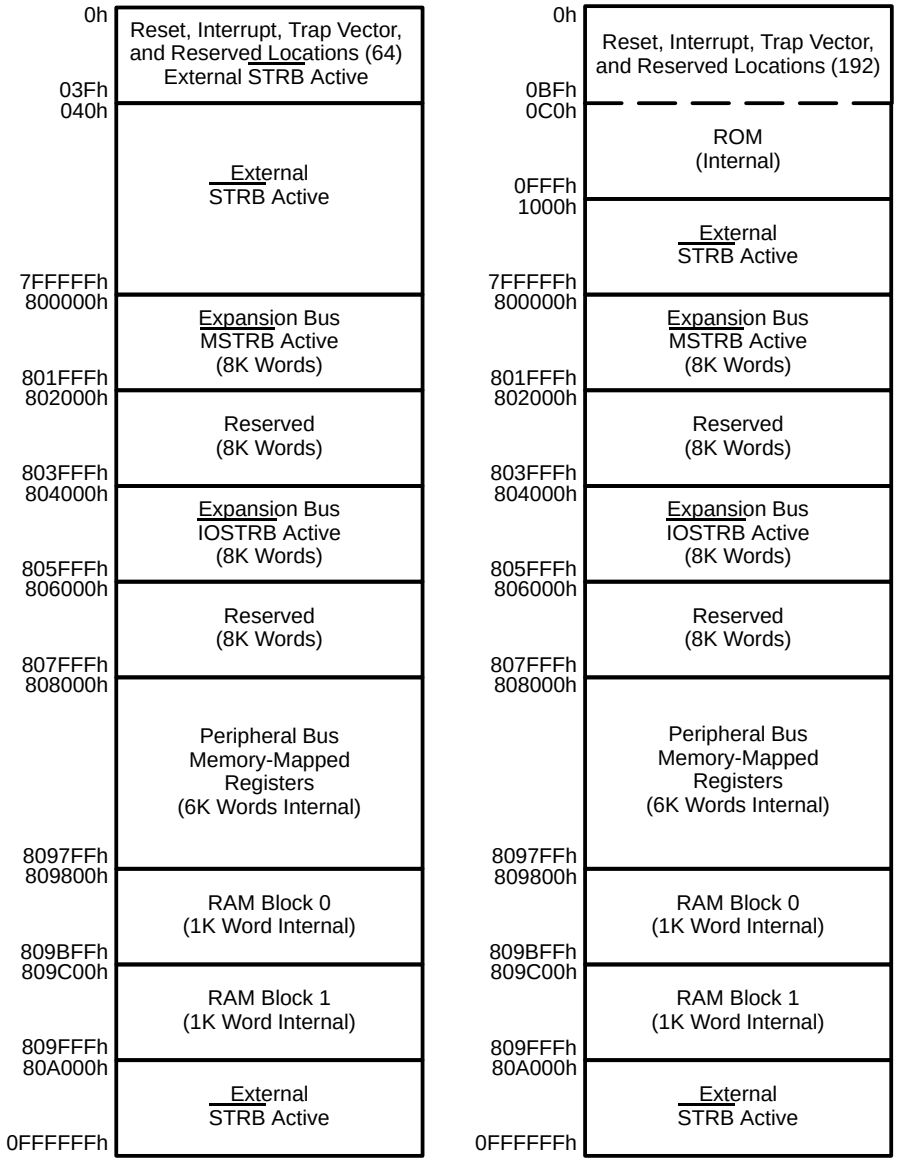
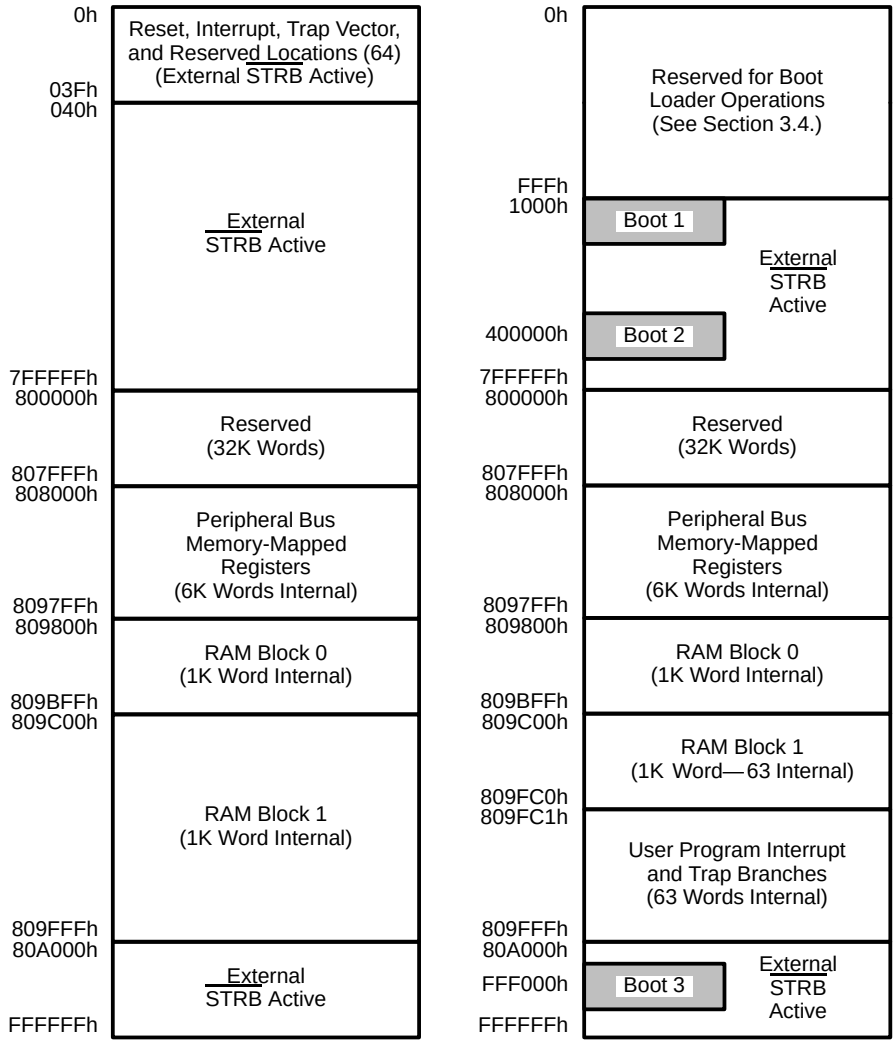


Figure 3–8. TMS320C31 Memory Maps



Boot 1–3 locations are used by the boot-loader function. See Section 3.4 for a complete description. All reserved memory locations are described in Table 2–5 on page 2-31.

3.2.2 TMS320C31 Memory Maps

Setting the TMS320C31 MCBL/$\overline{MP}$ pin determines the mode in which the TMS320C31 can function:

- Microprocessor mode (MCBL/$\overline{MP}$ = 0), or
- Microcomputer/boot loader mode (MCBL/$\overline{MP}$ = 1)

The major difference between these two modes is their memory maps (see Figure 3–8). The program boot load feature is enabled when the MCBL/$\overline{MP}$ pin is driven high during reset.

Figure 3–8 shows the memory locations (internal and external) used by the boot loader to load the source program.

3.2.3 Reset/Interrupt/Trap Vector Map

The addresses for the reset, interrupt, and trap vectors are 00h–3Fh, as shown in Figure 3–9. The reset vector contains the address of the reset routine.

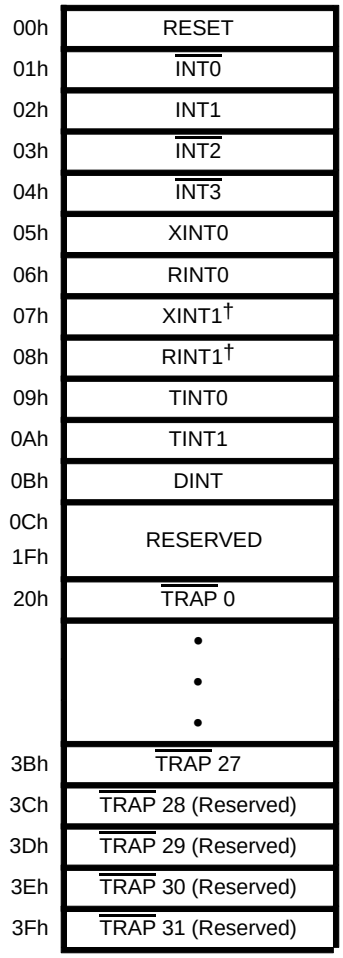
Microprocessor and Microcomputer Modes

In the microprocessor mode of the TMS320C30 and TMS320C31 and the microcomputer mode of the TMS320C30, the interrupt and trap vectors stored in locations 0h–3Fh are the addresses of the starts of the respective 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 3–9.

Microcomputer/Boot Loader Mode

In the microcomputer/boot loader mode of the TMS320C31, 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 3–10.

Figure 3–9. Reset, Interrupt, and Trap-Vector Locations for the TMS320C30/TMS320C31 Microprocessor Mode

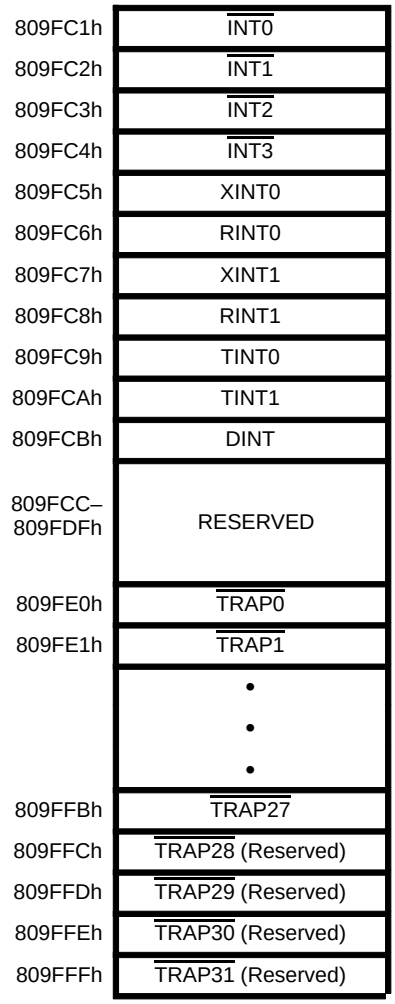


† Reserved on TMS320C31

Note: Traps 28–31

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

Figure 3–10. Interrupt and Trap Branch Instructions for the TMS320C31 Microcomputer Mode



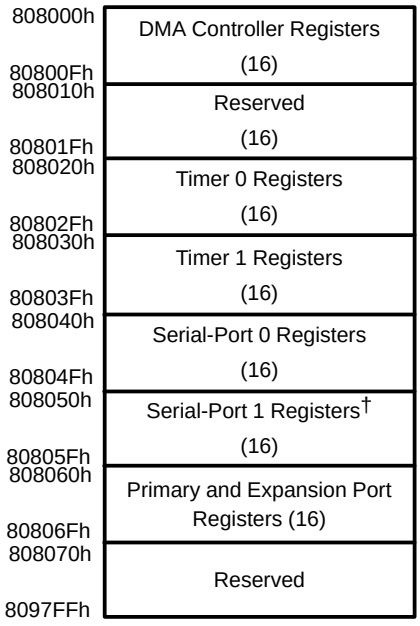
Note: Traps 28–31

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

3.2.4 Peripheral Bus Map

The memory-mapped peripheral registers are located starting at address 808000h. The peripheral bus memory map is shown in Figure 3–11. Each peripheral occupies a 16-word region of the memory map. Locations 808010h through 80801Fh and locations 808070h through 8097FFh are reserved.

Figure 3–11. Peripheral Bus Memory Map



† Reserved on TMS320C31

3.3 Instruction Cache

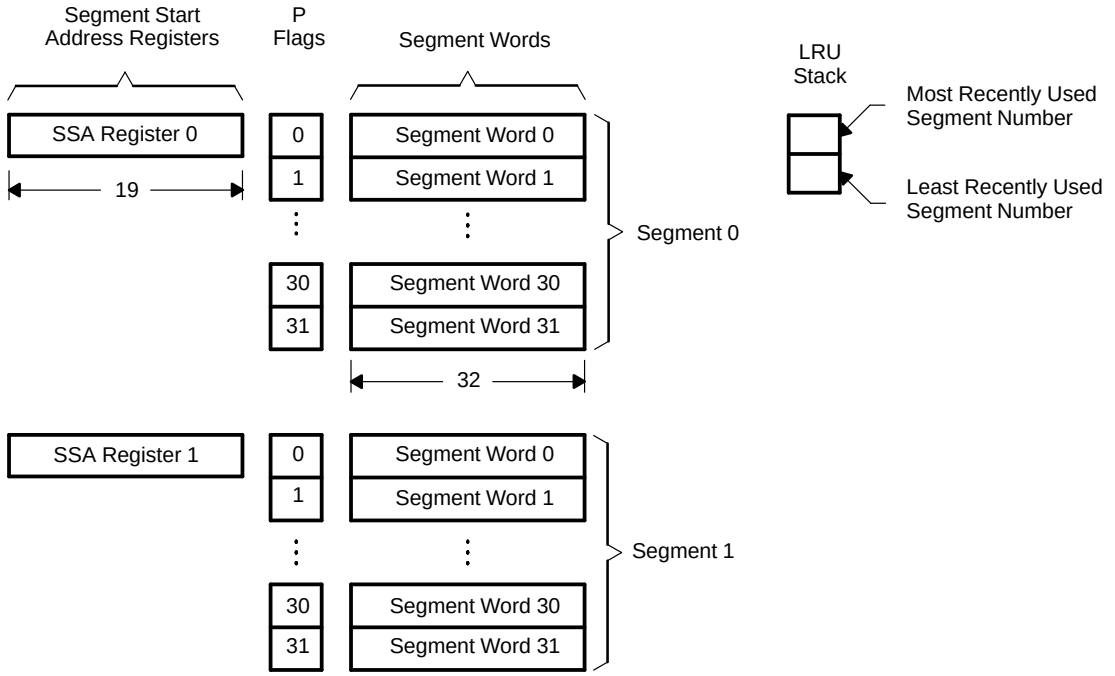
A 64 $\times$ 32-bit instruction cache facilitates maximum system performance by storing sections of code that can be fetched when the device repeatedly accesses time-critical code. This reduces the number of off-chip accesses necessary and allows code to be stored off-chip in slower, lower-cost memories. 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 3.3.2 describes a form of the least recently used (LRU) cache update algorithm.

3.3.1 Cache Architecture

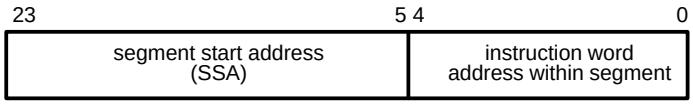
The instruction cache (see Figure 3–12) contains 64 32-bit words of RAM; it is divided into two 32-word segments. Associated with each segment is a 19-bit segment start address (SSA) register. For each word in the cache, there is a corresponding single bit: present (P) flag.

Figure 3–12. Instruction Cache Architecture



When the CPU requests an instruction word from external memory, the cache algorithm checks to determine whether the word is already contained in the instruction cache. Figure 3–13 shows the partitioning of an instruction address as used by the cache control algorithm. 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 whether a word within a particular segment is already present in cache memory.

Figure 3–13. Address Partitioning for Cache Control Algorithm



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 3–12) is maintained for this purpose.

The LRU stack determines which of the two segments qualifies as the least recently used after each access to the cache; therefore, the stack contains either 0,1 or 1,0. 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 segment number, and the number at the bottom of the stack is the least recently used segment number.

At system 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 least recently used 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.

3.3.2 Cache Algorithm

When the TMS320C3x 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:
  1) The instruction word is read from the cache.
  2) The number of the segment containing the 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.
- **Cache Miss.** The cache does not contain the instruction. Following are the types of cache miss:
  - Word 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, 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 least recently used 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 least recently used 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 (i.e., following a branch) are treated by the cache as valid program fetches and can generate cache misses and cache updates.

Take care when using self-modifying code. If an instruction resides in cache and the corresponding location in primary memory is modified, the copy of the instruction in 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.

3.3.3 Cache Control Bits

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

- **Cache Clear Bit (CC)**. Writing a 1 to the cache clear bit (CC) invalidates all entries in the cache. All P flags in the cache are cleared. The CC bit is always cleared after the cache is cleared. It is therefore always read as a 0. At reset, the cache is cleared and 0 is written to this bit.
- **Cache Enable Bit (CE)**. Writing a 1 to this bit enables the cache. When enabled, the cache is used according to the previously described cache algorithm. Writing a 0 to the cache enable bit disables the cache; no updates or modification of the cache can be performed. Specifically, no SSA register updates are performed, no P flags are modified (unless CC = 1), and the LRU stack is not modified. Writing a 1 to CC when the cache is disabled clears the cache, and, thus, the P flags. No fetches are made from the cache when the cache is disabled. At reset, 0 is written to this bit.

- **Cache Freeze Bit (CF)**. When CF = 1, the cache is frozen. If, in addition, the cache is enabled, fetches from the cache are allowed, but no modification of the state of the cache is performed. Specifically, no SSA register updates are performed, no P flags are modified (unless CC = 1), and the LRU stack is not modified. You can use this function to keep frequently used code resident in the cache. Writing a 1 to CC when the cache is frozen clears the cache, and, thus, the P flags. At reset, 0 is written to this bit.

Table 3–6 defines the effect of the CE and CF bits used in combination.

Table 3–6. 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

3.4 Using the TMS320C31 Boot Loader

This section describes how to use the TMS320C31 microcomputer/boot loader (MCBL/$\overline{MP}$)function. This feature is unique to the TMS320C31 and is not available on the TMS320C30 devices. The source code for the boot loader is supplied in Appendix G.

3.4.1 Boot-Loader Operations

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 either reside in one of three memory mapped areas identified as Boot 1, Boot 2, and Boot 3 (see the shaded areas of Figure 3–8), or they are received by means of the serial port.

User-definable byte, half-word, and word-data formats, as well as 32-bit fixed burst loads from the TMS320C31 serial port, are supported. See Section 8.2 on page 8-13 for a detailed description of the serial-port operation.

3.4.2 Invoking the Boot Loader

The boot-loader function is selected by resetting the processor while driving the MCBL/$\overline{MP}$ pin high. Use interrupt pins $\overline{INT3}$ – $\overline{INT0}$ to set the mode of the boot load operation. Figure 3–14 shows the flow of this operation, which depends on the mode selected (external memory or serial boot). Figure 3–15 shows **memory load** operations; Figure 3–16 shows **serial port load** operations.

Figure 3–14. Boot-Loader-Mode Selection Flowchart

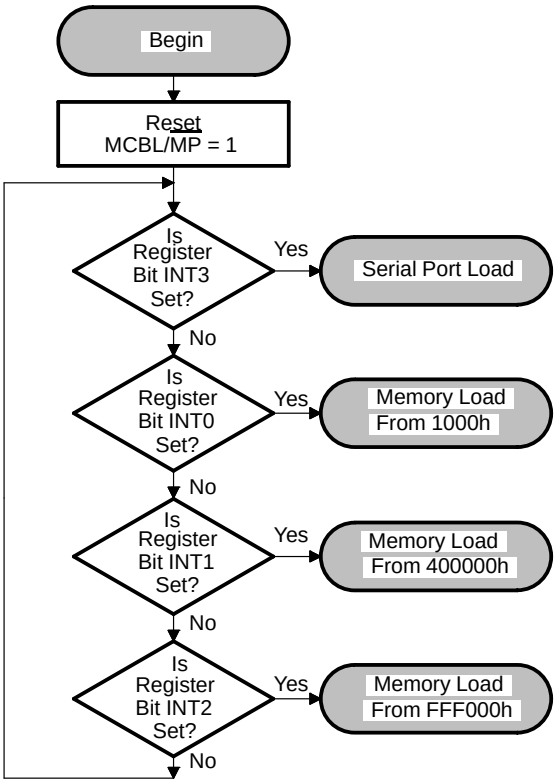


Figure 3–15. Boot-Loader Memory-Load Flowchart

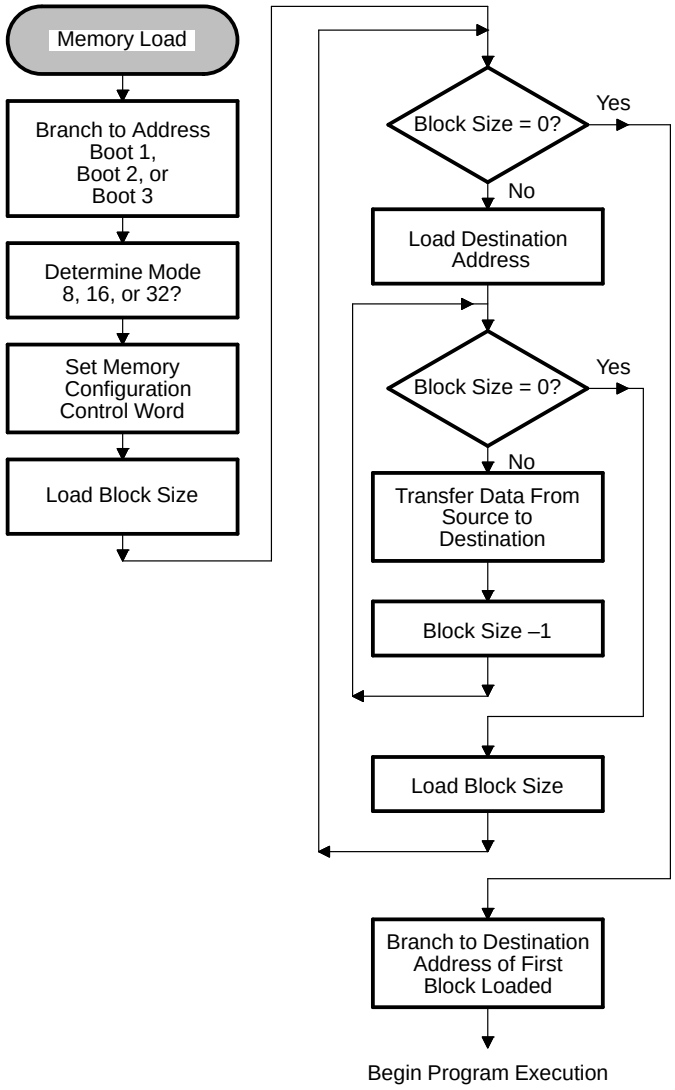
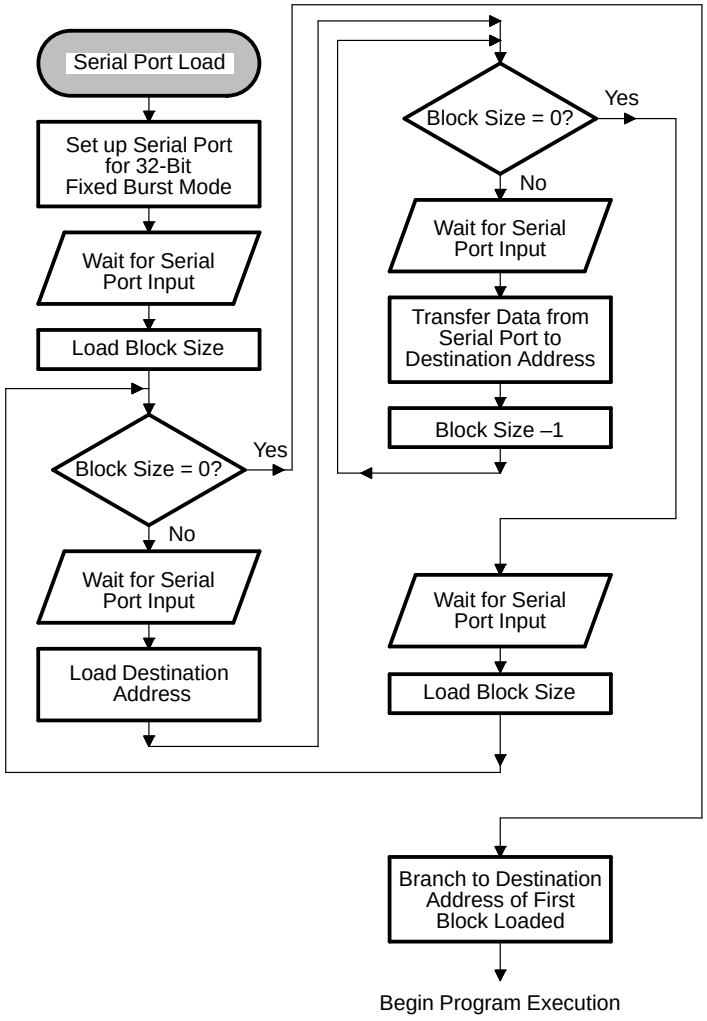


Figure 3–16. Boot-Loader Serial-Port Load-Mode Flowchart



3.4.3 Mode Selection

After reset, the loader mode is determined by polling the status of the INT3–INT0 bits of the IF register. The bits are polled in the order described in the flowchart in Figure 3–14 on page 3-27. Table 3–7 lists the mode options and the interrupt that you can use to set the particular mode. The interrupt can be driven any time after the $\overline{\text{RESET}}$ pin has been deasserted. Unless only one interrupt flag bit is set (INT0, INT1, INT2, or INT3), the boot mode cannot be guaranteed.

Table 3–7. Loader Mode Selection

Active Interrupt	Loader Mode	Memory Addresses
$\overline{\text{INT0}}$	External memory	Boot 1 address 0x001000
$\overline{\text{INT1}}$	External memory	Boot 2 address 0x400000
$\overline{\text{INT2}}$	External memory	Boot 3 address 0xFFF000
$\overline{\text{INT3}}$	32-bit serial	Serial port 0

3.4.4 External Memory Loading

Table 3–8 shows and describes the information that you must specify to define boot memory organization (8, 16, or 32 bits), the code block size, the load destination address, and memory access timing control for the boot memory. You must specify this information before a source program can be externally loaded.

This information must be specified in the first four locations of the Boot 1, Boot 2, or Boot 3 areas. The header is followed by the data or program code that is the block size in length.

Table 3–8. External Memory Loader Header

Location	Description	Valid Data Entries
0	Boot memory type (8, 16, or 32)	0x8, 0x10, or 0x20 specified as a 32-bit number
1	Boot memory configuration (defined # of wait states, etc.)	See Chapter 7 for valid bus-control register entries.
2	Program block size (blk)	Any value 0 < blk < 2^{24}
3	Destination address	Any valid TMS320C31 24-bit address
4	Program code starts here	Any 32-bit data value or valid TMS320C3x instruction

The loader fetches 32 bits of data for each specified location, regardless of what memory configuration width is specified. The data values must reside within or be written to memory, beginning with the value of least significance for each 32 bits of information.

3.4.5 Examples of External Memory Loads

Example 3–1, Example 3–2, and Example 3–3 show memory images for byte-wide, 16-bit-wide, and 32-bit-wide configured memory.

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 loader 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 TMS320C31's internal RAM Block 1, and
  - A single block of memory that is 0x1FF in length.

Example 3–1. 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 code size = 0x1FF
0x1009	0x01	
0x100A	0x00	
0x100B	0x00	
0x100C	0x00	Program load starting address = 0x809C00
0x100D	0x9C	
0x100E	0x80	
0x100F	0x00	

Example 3–2.16-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 code size = 0x1FF
0x1005	0x0000	
0x1006	0x9C00	Program load starting address = 0x809C00
0x1007	0x0080	

Example 3–3.32-Bit-Wide Configured Memory

Address	Value	Comments
0x1000	0x00000020	Memory width = 32
0x1001	0x00001058	Memory type = SWW = 11, WCNT = 2
0x1002	0x000001FF	Program code size = 0x1FF
0x1003	0x00809C00	Program load starting address = 0x809C00

After reading the header, the loader transfers blk, 32-bit words beginning at a specified destination address. 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 block loaded should be a start-up routine to access the other loaded programs.

Each code block has the following header:

```
BLK size              1st location
Destination address      2nd location
```

End the loader function and begin execution of the first code block by appending the value of 0x00000000 to the last block.

It is assumed that at least one block of code will be loaded when the loader is invoked. Initial loader invocation with a block size of 0x00000000 produces unpredictable results.

3.4.6 Serial-Port Loading

Boot loads, by way of the TMS320C31 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 externally generate the serial-port clock and FSR.

As in parallel loading, a header must precede the actual program to be loaded. However, you need only apply the block size and destination address because the loader and your hardware have predefined serial-port speed and data format (i.e., skip data words 0 and 1 from Table 3–8).

The transferred data-bit order must begin with the MSB and end with the LSB.

3.4.7 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 TMS320C3x 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 last 63 locations in RAM Block 1 of the TMS320C31 are assumed to contain branch instructions to the interrupt source routines.

Take care to ensure that these locations are not inadvertently overwritten by loaded program or data values.

Table 3–9 shows the MCBL/$\overline{MP}$ mode interrupt and trap instruction memory maps.

Table 3–9. TMS320C31 Interrupt and Trap Memory Maps

Address	Description
809FC1	$\overline{INT0}$
809FC2	$\overline{INT1}$
809FC3	$\overline{INT2}$
809FC4	$\overline{INT3}$
809FC5	$\overline{XINT0}$
809FC6	$\overline{RINT0}$
809FC7	Reserved
809FC8	Reserved
809FC9	$\overline{TINT0}$
809FCA	$\overline{TINT1}$
809FCB	$\overline{DINT0}$
809FCC–809FDF	Reserved
809FE0	$\overline{TRAP0}$
809FE1	$\overline{TRAP1}$
•	•
•	•
•	•
809FFB	$\overline{TRAP27}$
809FFC–809FFF	Reserved

3.4.8 Precautions

The boot loader builds a one-word-deep stack, starting at location 809801h.

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/MP pin should remain high during the entire boot-loader execution, but it can be changed subsequently at any time. The TMS320C31 does not need to be reset after the MCBL/$\overline{\text{MP}}$ pin is changed. During the change, the TMS320C31 should not access addresses 0h–FFFh.

Chapter 4

Data Formats and Floating-Point Operation

In the TMS320C3x architecture, data is organized into three fundamental types: integer, unsigned-integer, and floating-point. The terms integer and signed-integer are considered to be equivalent. The TMS320C3x 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 TMS320C3x implementation of floating-point arithmetic facilitates floating-point operations at integer speeds while preventing problems with overflow, operand alignment, and other burdensome tasks common in integer operations.

This chapter discusses in detail the data formats and floating-point operations supported in the TMS320C3x. Major topics in this section are as follows:

Topic **Page**

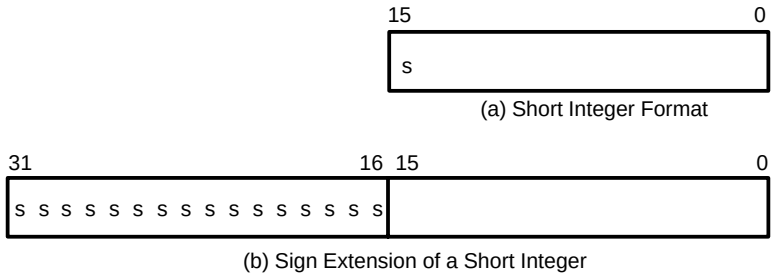
4.1 Integer Formats

The TMS320C3x supports two integer formats: a 16-bit short integer format and a 32-bit single-precision integer format. When extended-precision registers are used as integer operands, only bits 31–0 are used; bits 39–32 remain unchanged and unused.

4.1.1 Short-Integer Format

The short integer format is a 16-bit two's complement integer format for immediate integer operands. For those instructions that assume integer operands, this format is sign-extended to 32 bits (see Figure 4–1). The range of an integer si, represented in the short integer format, is $-2^{15} \leq si \leq 2^{15} - 1$. In Figure 4–1, s = signed bit.

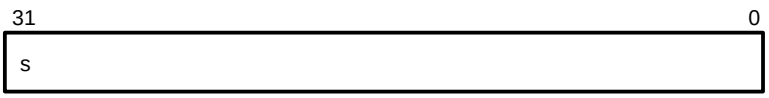
Figure 4–1. Short Integer Format and Sign Extension of Short Integers



4.1.2 Single-Precision Integer Format

In the single-precision integer format, the integer is represented in two's complement notation. The range of an integer *sp*, represented in the single-precision integer format, is $-2^{31} \leq sp \leq 2^{31} - 1$. Figure 4–2 shows the single-precision integer format.

Figure 4–2. Single-Precision Integer Format



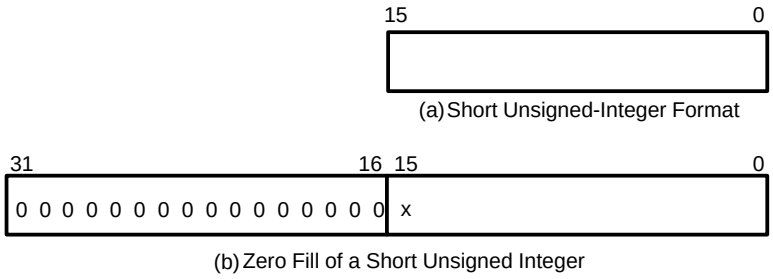
4.2 Unsigned-Integer Formats

The TMS320C3x supports two unsigned-integer formats: a 16-bit short format and a 32-bit single-precision format. In extended-precision registers, the unsigned-integer operands use only bits 31–0; bits 39–32 remain unchanged.

4.2.1 Short Unsigned-Integer Format

Figure 4–3 shows the16-bit, short, unsigned-integer format for immediate unsigned-integer operands. For those instructions that assume unsigned-integer operands, this format is zero-filled to 32 bits. In Figure 4–3, x = most significant bit (MSB) (1 or 0).

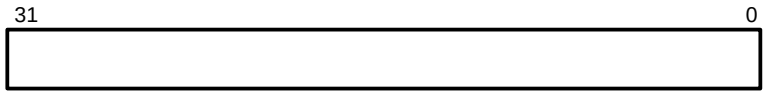
Figure 4–3. Short Unsigned-Integer Format and Zero Fill



4.2.2 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 4–4.

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

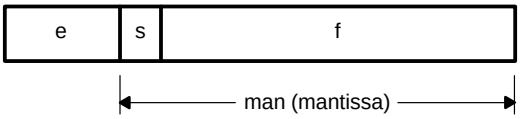


4.3 Floating-Point Formats

All TMS320C3x floating-point formats consist of three fields: an exponent field (e), a single-bit sign field (s), and a fraction field (f). These are stored as shown in Figure 4–5. The exponent field is a two's complement number. The sign field and fraction field may be considered one unit and referred to as the mantissa field (man). The two's complement fraction is combined with the sign bit and the implied most significant bit to create the mantissa. The mantissa represents a normalized two's complement number. A normalized representation implies a most significant nonsign bit, thus providing additional precision. The value of a floating-point number x as a function of the fields e, s, and f is given as

$x = 01.f \times 2^e$	if s = 0, or if the leading 0 is the sign bit and the 1 is the implied most significant nonsign bit
$10.f \times 2^e$	if s = 1, or if the leading 1 is the sign bit and the 0 is the implied most significant nonsign bit
0	if e = most negative two's complement value of the specified exponent field width

Figure 4–5. Generic Floating-Point Format



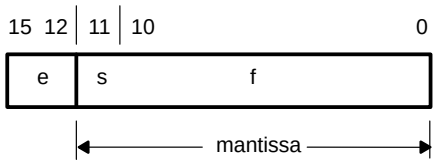
Note: e = exponent field
s = single-bit sign field
f = fraction field

Three floating-point formats are supported on the TMS320C3x. The first is a short floating-point format for immediate floating-point operands, consisting of a 4-bit exponent, a sign bit, and an 11-bit fraction. The second is a single-precision format consisting of an 8-bit exponent, a sign bit, and a 23-bit fraction. The third is an extended-precision format consisting of an 8-bit exponent, a sign bit, and a 31-bit fraction.

4.3.1 Short Floating-Point Format

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

Figure 4–6. 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 two's complement number x in the short floating-point format is given by the following:

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

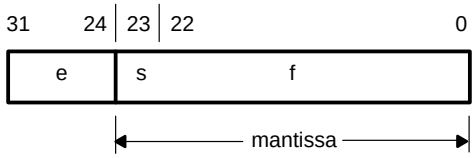
The following examples illustrate the range and precision of the short floating-point format:

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: $x = -2 \times 2^7 = -2.5600 \times 10^2$

4.3.2 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 two's complement 24-bit mantissa field (man) with an implied most significant nonsign bit. See Figure 4–7.

Figure 4–7. 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:

$x = 01.f \times 2^e$ if s = 0
$10.f \times 2^e$ if s = 1
0 if e = –8

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: $x = 1 \times 2^{-127} = 5.8774717 \times 10^{-39}$

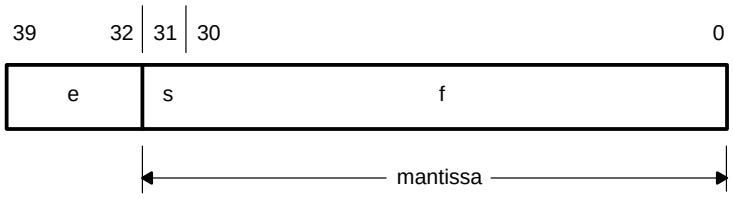
Least Negative: $x = (-1-2^{-23}) \times 2^{-127} = -5.8774724 \times 10^{-39}$

Most Negative: $x = -2 \times 2^{127} = -3.4028236 \times 10^{38}$

4.3.3 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 4–8.

Figure 4–8. 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

$10.f \times 2^e$ if s = 1

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: $x = 1 \times 2^{-127} = 5.8774717541 \times 10^{38}$

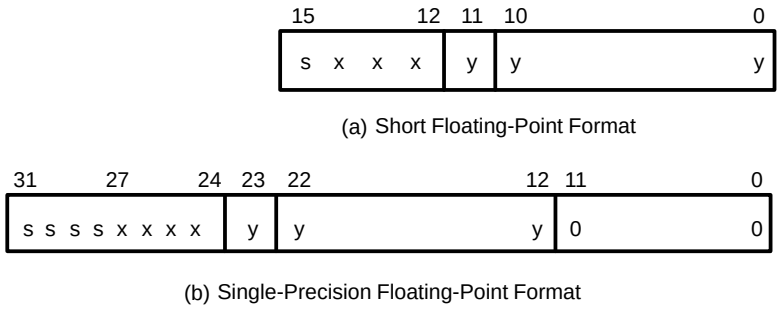
Least Negative: $x = (-1-2^{-31}) \times 2^{-127} = -5.8774717569 \times 10^{-39}$

Most Negative: $x = -2 \times 2^{127} = -3.4028236691 \times 10^{38}$

4.3.4 Conversion Between Floating-Point Formats

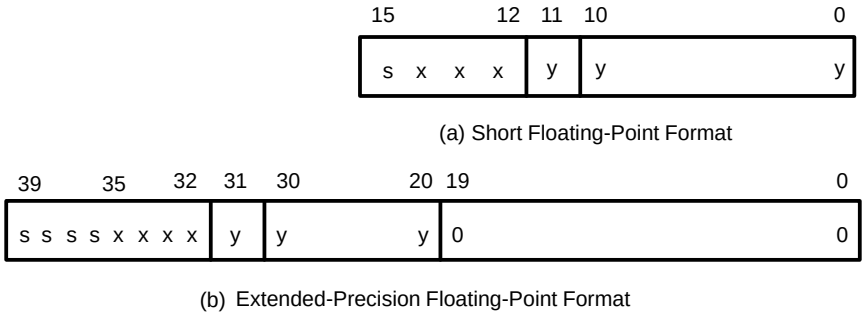
Floating-point operations assume several different formats for inputs and outputs. These formats often require conversion from one floating-point format to another (e.g., short floating-point format to extended-precision floating-point format). Format conversions occur automatically in hardware, with no overhead, as a part of the floating-point operations. Examples of the four conversions are shown in Figure 4–9, Figure 4–10, Figure 4–11, and Figure 4–12. 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 4–9, Figure 4–10, Figure 4–11, and Figure 4–12, s = sign bit of the exponent.

Figure 4–9. Converting From Short Floating-Point Format to Single-Precision Floating-Point Format



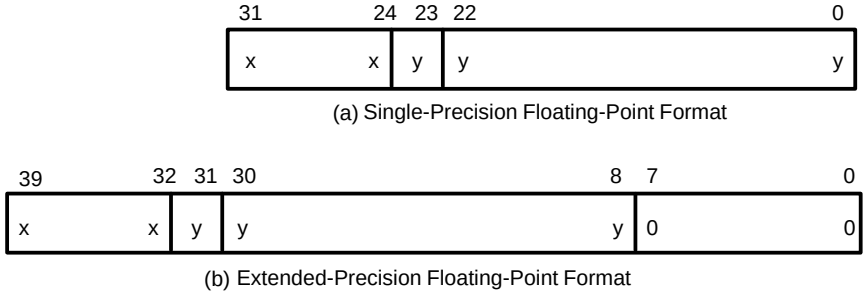
In this format, the exponent field is sign-extended, and the fraction field is filled with 0s.

Figure 4–10. Converting From Short Floating-Point Format to Extended-Precision Floating-Point Format



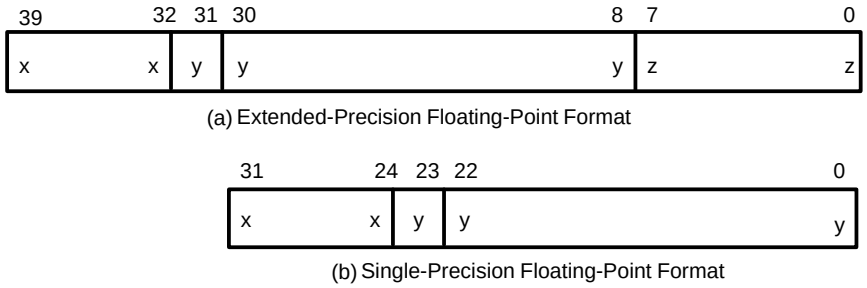
The exponent field in this format is sign-extended, and the fraction field is filled with 0s.

Figure 4–11. Converting From Single-Precision Floating-Point Format to Extended-Precision Floating-Point Format



The fraction field is filled with 0s.

Figure 4–12. Converting From Extended-Precision Floating-Point Format to Single-Precision Floating-Point Format



The fraction field is truncated.

4.4 Floating-Point Multiplication

A floating-point number α can be written in floating-point format as in the following formula:

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

where:

α(man) is the mantissa and α(exp) is the exponent.

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

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

where:

c(man) = α(man) $\times$ b(man), and
c(exp) = α(exp) + b(exp)

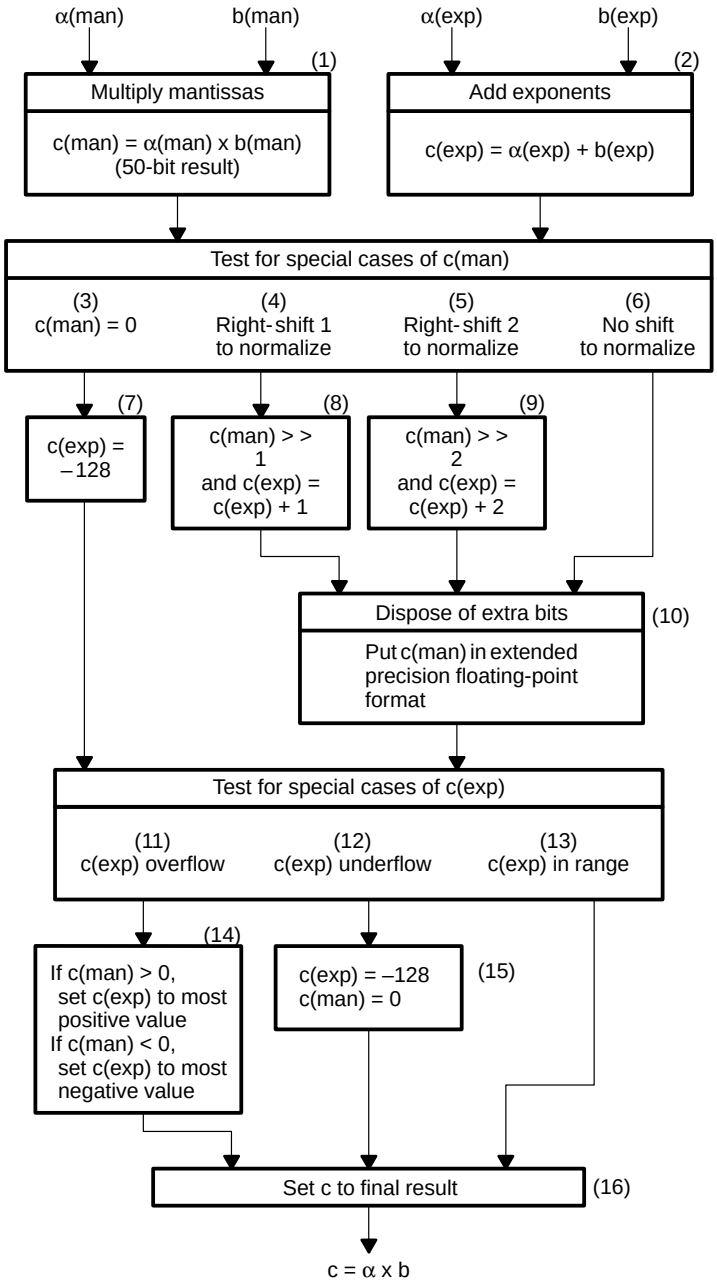
During floating-point multiplication, source operands are always assumed to be in the single-precision floating-point format. If the source of the operands is in short floating-point format, it is extended to the single-precision floating-point format. If the source of the operands is in extended-precision floating-point format, it is 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.

A flowchart for floating-point multiplication is shown in Figure 4–13. In step 1, the 24-bit source operand mantissas are multiplied, producing a 50-bit result c(man). (Note that input and output data are always represented as normalized numbers.) In step 2, the exponents are added, yielding c(exp). Steps 3 through 6 check for special cases. 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 1 bit, thus adding 1 to c(exp). If a right shift of 2 is necessary, then in step 9, c(man) is right-shifted 2 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 4–13. Flowchart for Floating-Point Multiplication



Example 4–1, Example 4–2, Example 4–3, Example 4–4, and Example 4–5 illustrate how floating-point multiplication is performed on the TMS320C3x. For these examples, the implied most significant nonsign bit is made explicit.

Example 4–1. Floating-Point Multiply (Both Mantissas = –2.0)

Let:

$\alpha = -2.0 \times 2^{\alpha(exp)} = 10.00000000000000000000000 \times 2^{\alpha(exp)}$

$b = -2.0 \times 2^{b(exp)} = 10.00000000000000000000000 \times 2^{b(exp)}$

where:

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

Then:

$$\begin{array}{r} 10.00000000000000000000000 \times 2^{\alpha(exp)} \\ \times\ 10.00000000000000000000000 \times 2^{b(exp)} \\ \hline 0100.00000000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp))} \end{array}$$

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:

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

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 4–2. Floating-Point Multiply (Both Mantissas = 1.5)

Let:

$a = 1.5 \times 2^{\alpha(exp)} = 01.10000000000000000000000 \times 2^{\alpha(exp)}$

$b = 1.5 \times 2^{b(exp)} = 01.10000000000000000000000 \times 2^{b(exp)}$

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

$$\begin{array}{r} 01.10000000000000000000000 \times 2^{\alpha(exp)} \\ \times\ 01.10000000000000000000000 \times 2^{b(exp)} \\ \hline 0010.01000000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp))} \end{array}$$

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:

$$\begin{array}{l} 01.10000000000000000000 \times 2^{\alpha(exp)} \\ \times\, 01.10000000000000000000 \times 2^{b(exp)} \\ \hline 01.00100000000000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp) + 1)} \end{array}$$

Example 4–3. Floating-Point Multiply (Both Mantissas = 1.0)

Let:

$\alpha = 1.0 \times 2^{\alpha(exp)} = 01.00000000000000000000 \times 2^{\alpha(exp)}$

$b = 1.0 \times 2^{b(exp)} = 01.00000000000000000000 \times 2^{b(exp)}$

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

$$\begin{array}{l} 01.00000000000000000000 \times 2^{\alpha(exp)} \\ \times\, 01.00000000000000000000 \times 2^{b(exp)} \\ \hline 0001.00000000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp))} \end{array}$$

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

These examples have shown cases where the product of two normalized numbers can be normalized with a shift of 0, 1, or 2. For all normalized inputs with the floating-point format used by the TMS320C3x, a normalized result can be produced by a shift of 0, 1, or 2.

Example 4–4. Floating-Point Multiply Between Positive and Negative Numbers

Let:

$\alpha = 1.0 \times 2^{\alpha(exp)} = 01.00000000000000000000 \times 2^{\alpha(exp)}$

$b = -2.0 \times 2^{b(exp)} = 10.00000000000000000000 \times 2^{b(exp)}$

Then:

$$\begin{array}{l} 01.00000000000000000000 \times 2^{\alpha(exp)} \\ \times\, 10.00000000000000000000 \times 2^{b(exp)} \\ \hline 1110.00000000000000000000000000000000000000000000 \times 2^{(\alpha(exp) + b(exp))} \end{array}$$

The result is $c = -2.0 \times 2^{(\alpha(exp) + b(exp))}$

Example 4–5. Floating-Point Multiply by 0

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

4.5 Floating-Point Addition and Subtraction

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

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

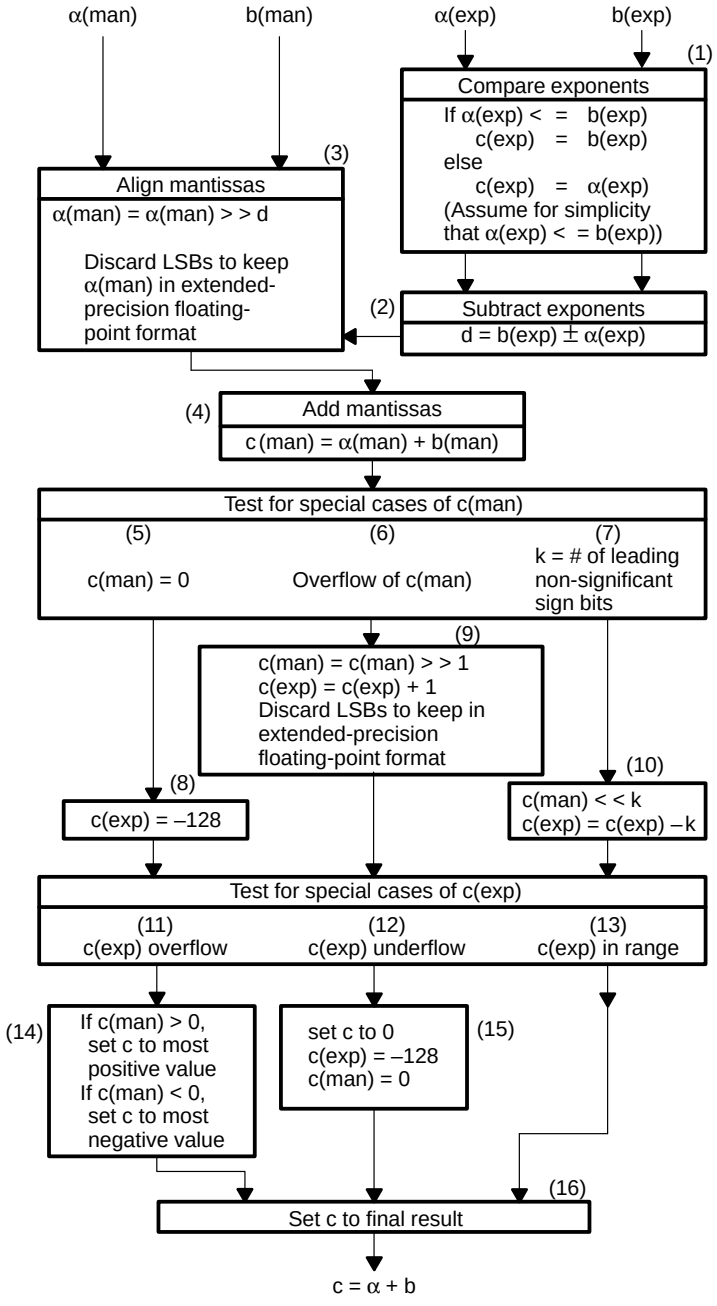
The sum (or difference) of α and b can be defined as:

$$c = \alpha \pm b$$
$$= (\alpha(man) \pm (b(man) \times 2^{-(\alpha(exp)-b(exp))})) \times 2^{\alpha(exp)}, \quad \text{if } \alpha(exp) \geq b(exp)$$
$$= ((\alpha(man) \times 2^{-(b(exp)-\alpha(exp))}) \pm b(man)) \times 2^{b(exp)}, \quad \text{if } \alpha(exp) < b(exp)$$

The flowchart for floating-point addition is shown in Figure 4–14. Since this flowchart assumes signed data, it is also appropriate for floating-point subtraction. In this figure, it is assumed that $\alpha(exp) \leq b(exp)$. In step 1, the source exponents 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. After the mantissas have been aligned, they are added (step 4).

Steps 5 through 7 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 c(man) is right-shifted one bit, and 1 is added to c(exp). Otherwise, step 10 normalizes c by left-shifting c(man) and subtracting c(exp) by the number of leading non-significant sign bits (step 7). 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.

Figure 4–14. Flowchart for Floating-Point Addition



Example 4–6, Example 4–7, Example 4–8, and Example 4–9 describe the floating-point addition and subtraction operations. It is assumed that the data is in the extended-precision floating-point format.

Example 4–6. Floating-Point Addition

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

$\alpha = 1.5 = 01.10000000000000000000000000000000 \times 2^{0}$
$b = 0.5 = 01.00000000000000000000000000000000 \times 2^{-1}$

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

$b = 0.5 = 00.10000000000000000000000000000000 \times 2^{0}$

Then:

$$\begin{array}{l} 01.10000000000000000000000000000000 \times 2^{0} \\ +\ 00.10000000000000000000000000000000 \times 2^{0} \\ \hline 010.00000000000000000000000000000000 \times 2^{0} \end{array}$$

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:

$$\begin{array}{l} 01.10000000000000000000000000000000 \times 2^{0} \\ \pm\ 00.10000000000000000000000000000000 \times 2^{0} \\ \hline 01.00000000000000000000000000000000 \times 2^{1} \end{array}$$

Example 4–7. Floating-Point Subtraction

A subtraction is performed in this example. Let

$\alpha = 01.00000000000000000000000000000001 \times 2^{0}$
$b = 01.00000000000000000000000000000000 \times 2^{0}$

The operation to be 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.

$$\begin{array}{l} 01.00000000000000000000000000000001 \times 2^{0} \\ -\ 01.00000000000000000000000000000000 \times 2^{0} \\ \hline 00.00000000000000000000000000000001 \times 2^{0} \end{array}$$

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:

$$\begin{array}{l} 01.00000000000000000000000000000001 \times 2^{0} \\ -\,01.00000000000000000000000000000000 \times 2^{0} \\ \hline 01.00000000000000000000000000000000 \times 2^{-31} \end{array}$$

Example 4–8. 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

$$\alpha = 01.11111111111111111111111111111111 \times 2^{127}$$
$$b = 10.00000000000000000000000000000000 \times 2^{127}$$

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

$$\begin{array}{l} 01.11111111111111111111111111111111 \times 2^{127} \\ +\,10.00000000000000000000000000000000 \times 2^{127} \\ \hline 11.11111111111111111111111111111111 \times 2^{127} \end{array}$$

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

$$\begin{array}{l} 01.11111111111111111111111111111111 \times 2^{127} \\ +\,10.00000000000000000000000000000000 \times 2^{127} \\ \hline 10.00000000000000000000000000000000 \times 2^{95} \end{array}$$

Example 4–9. 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:

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

$0 \pm 0 = 0$

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

4.6 Normalization Using the NORM Instruction

The NORM instruction normalizes an extended-precision floating-point number that is assumed to be unnormalized. See Example 4–10. 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, and
3) Adjusts the exponent.

Example 4–10. 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

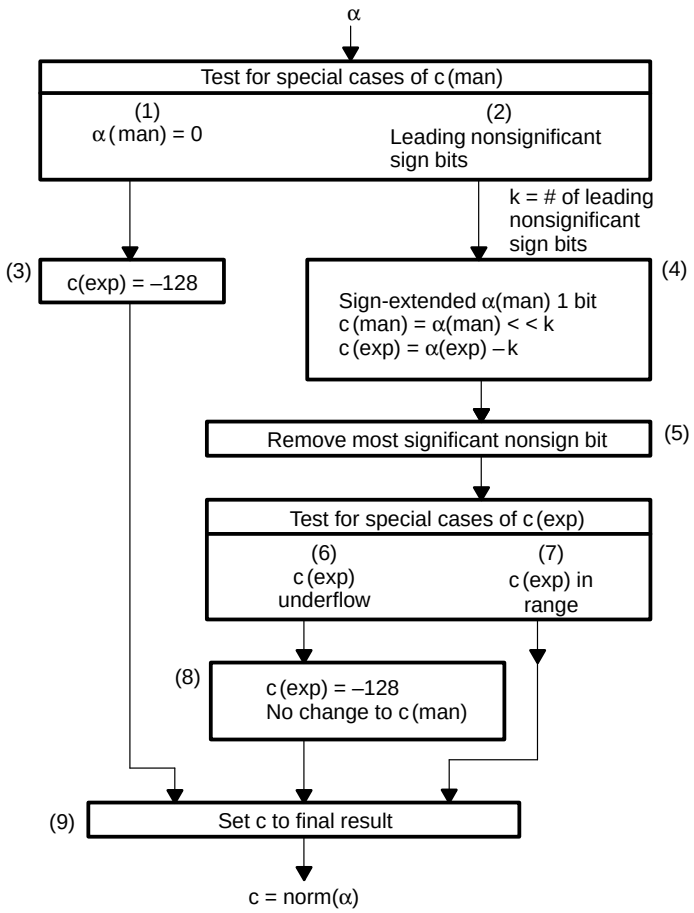
The final 32-bit value output after removing the redundant bit is:

man = 00000000000100000000000000000000, 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 4–15.

Figure 4–15. Flowchart for NORM Instruction Operation



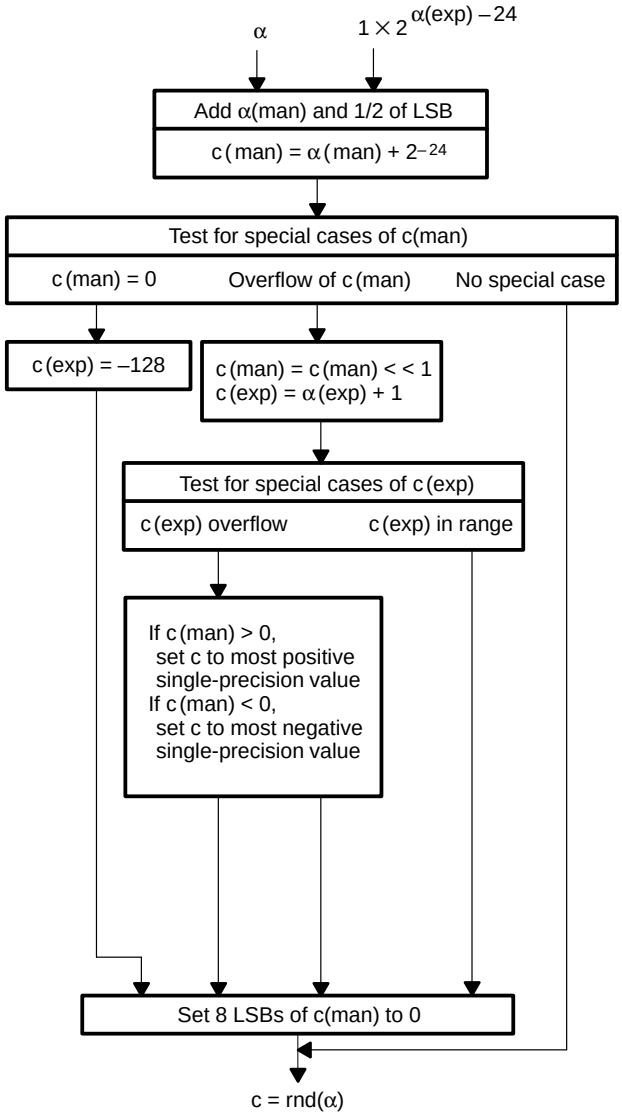
4.7 Rounding: The RND Instruction

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 4–16.

Figure 4–16. Flowchart for Floating-Point Rounding by the RND Instruction



4.8 Floating-Point-to-Integer Conversion

Floating-point to integer conversion, using the FIX instructions, allows extended-precision floating-point numbers to be converted to single-precision integers 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

$$\alpha(\text{exp}) \leq 30$$

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 α(exp) is within the valid range, then α(man), with implied bit included, is sign-extended and right-shifted (rs) by the amount

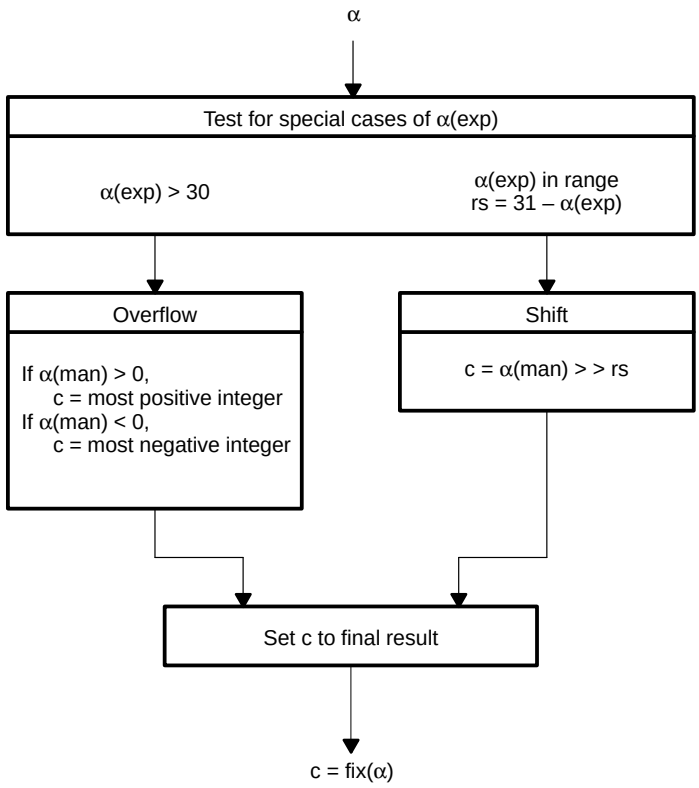
$$\text{rs} = 31 - \alpha(\text{exp})$$

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.

The flowchart for the floating-point-to-integer conversion is shown in Figure 4–17.

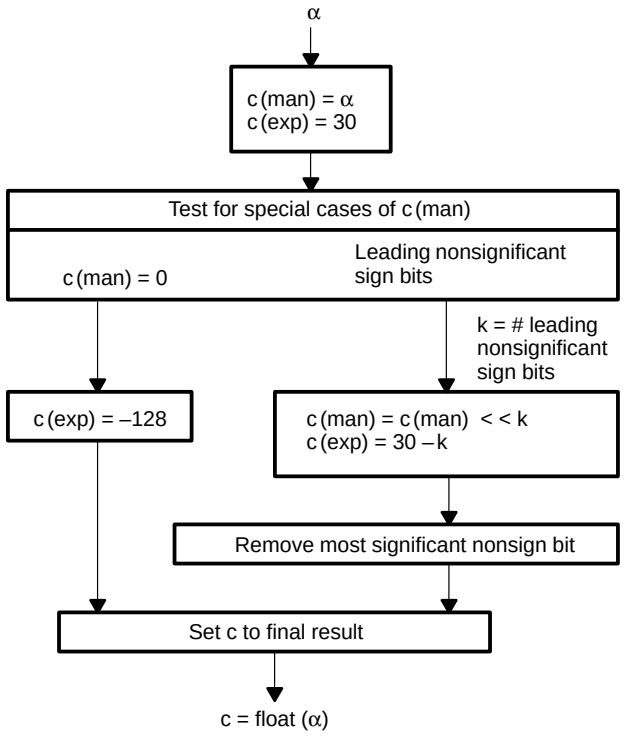
Figure 4–17. Flowchart for Floating-Point-to-Integer Conversion by FIX Instructions



4.9 Integer-to-Floating-Point Conversion

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 4–18.

Figure 4–18. Flowchart for Integer-to-Floating-Point Conversion by FLOAT Instructions



Chapter 5

Addressing

The TMS320C3x supports five groups of powerful addressing modes. Six types of addressing may be used within the groups, which allow access of data from memory, registers, and the instruction word. This chapter details the operation, encoding, and implementation of the addressing modes. It also discusses the management of system stacks, queues, and dequeues in memory.

These are the major topics in this chapter:

Topic **Page**

5.1 Types of Addressing

Six types of addressing allow access of data from memory, registers, and the instruction word:

- Register
- Direct
- Indirect
- Short-immediate
- Long-immediate
- PC-relative

Some types of addressing are appropriate for some instructions but not others. For this reason, the types of addressing are used in the five groups of addressing modes as follows:

- General addressing modes (G):
  - Register
  - Direct
  - Indirect
  - Short-immediate
- Three-operand addressing modes (T):
  - Register
  - Indirect
- Parallel addressing modes (P):
  - Register
  - Indirect
- Conditional-branch addressing modes (B):
  - Register
  - PC-relative

The six types of addressing are discussed first, followed by the five groups of addressing modes.

5.1.1 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 5–1.

Table 5–1. CPU Register Address/Assembler Syntax and Function

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

5.1.2 Direct Addressing

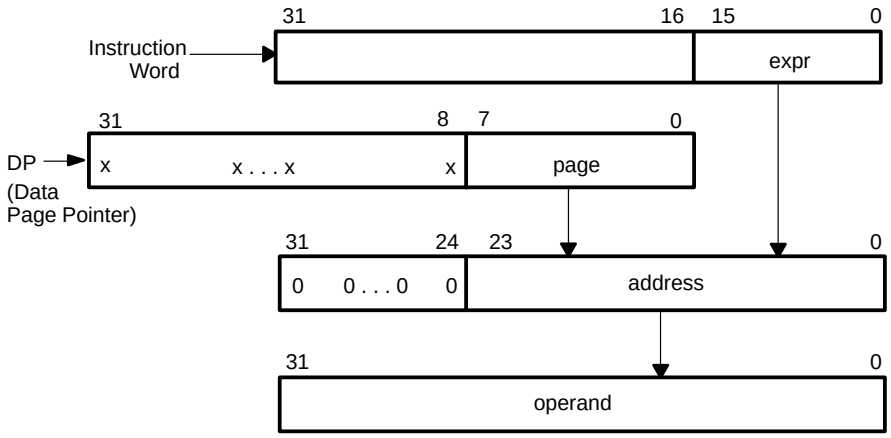
In direct addressing, the data address is formed by the concatenation of the eight least significant bits of the data page pointer (DP) with the 16 least significant bits of the instruction word (expr). This results in 256 pages (64K words per page), giving the programmer a large address space without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

Syntax: @expr

Operation: address = DP concatenated with expr

Figure 5–1 shows the formation of the data address. Example 5–1 is an instruction example with data before and after instruction execution.

Figure 5–1. Direct Addressing



Example 5–1.Direct Addressing

```
ADDI    @0BCDEh,R7
```

Before Instruction:	**After Instruction:**
DP = 8Ah	DP = 8Ah
R7 = 0h	R7 = 12345678h
Data at 8ABCDEh = 12345678h	Data at 8ABCDEh = 12345678h

5.1.3 Indirect Addressing

Indirect addressing is used to specify the address of an operand in memory through the contents of an auxiliary register, optional displacements, and index registers. Only the 24 least significant bits of the auxiliary registers and index registers are used in indirect addressing. This arithmetic is performed by the auxiliary register arithmetic units (ARAUs) on these lower 24 bits and is unsigned. The upper eight bits are unmodified.

The flexibility of indirect addressing is possible because the ARAUs on the TMS320C3x modify auxiliary registers in parallel with operations within the main CPU. Indirect addressing is specified by a five-bit field in the instruction word, referred to as the mod field. A displacement is either an explicit unsigned eight-bit integer contained in the instruction word or an implicit displacement of one. Two index registers, IR0 and IR1, can also be used in indirect addressing. In some cases, an optional addressing scheme using circular or bit-reversed addressing can be used. The mechanism for generating addresses in circular addressing is discussed in Section 5.3 on page 5-24; bit-reversed is discussed in Section 5.4 on page 5-29.

Note: Auxiliary Register

The auxiliary register (ARn) to be used 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 5–1).

Example 5–2.Auxiliary Register Indirect

An auxiliary register (ARn) contains the address of the operand to be fetched.

Operation: operand address = ARn
Assembler Syntax: *ARn
Modification Field: 11000

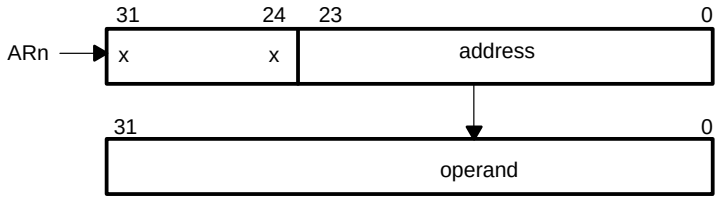


Table 5–2 lists the various kinds of indirect addressing, along with the value of the modification (mod) field, assembler syntax, operation, and function for each. The succeeding 17 examples show the operation for each kind of indirect addressing. Figure 5–2 shows the format in the instruction encoding.

Table 5–2. Indirect Addressing

Mod Field	Syntax	Operation	Description
		Indirect Addressing with Displacement	
00000	*+ARn(disp)	addr = ARn + disp	With predisplacement add
00001	*–ARn(disp)	addr = ARn – disp	With predisplacement subtract
00010	*++ARn(disp)	addr = ARn + disp ARn = ARn + disp	With predisplacement add and modify
00011	*––ARn(disp)	addr = ARn – disp ARn = ARn – disp	With predisplacement subtract and modify
00100	*ARn++(disp)	addr = ARn ARn = ARn + disp	With postdisplacement add and modify
00101	*ARn––(disp)	addr = ARn ARn = ARn – disp	With postdisplacement subtract and modify
00110	*ARn++(disp)%	addr = ARn ARn = circ(ARn + disp)	With postdisplacement add and circular modify
00111	*ARn––(disp)%	addr = ARn ARn = circ(ARn – disp)	With postdisplacement subtract and circular modify
		Indirect Addressing with Index Register IR0	
01000	*+ARn(IR0)	addr = ARn + IR0	With preindex (IR0) add
01001	*–ARn(IR0)	addr = ARn – IR0	With preindex (IR0) subtract
01010	*++ARn(IR0)	addr = ARn + IR0 ARn = ARn + IR0	With preindex (IR0) add and modify
01011	*––ARn(IR0)	addr = ARn – IR0 ARn = ARn – IR0	With preindex (IR0) subtract and modify
01100	*ARn++(IR0)	addr = ARn ARn = ARn + IR0	With postindex (IR0) add and modify
01101	*ARn––(IR0)	addr= ARn ARn = ARn – IR0	With postindex (IR0) subtract and modify
01110	*ARn++(IR0)%	addr = ARn ARn = circ(ARn + IR0)	With postindex (IR0) add and circular modify
01111	*ARn––(IR0)%	addr = ARn ARn = circ(ARn)– IR0	With postindex (IR0) subtract and circular modify

Legend:

addr	memory address	++	add and modify
ARn	auxiliary register AR0–AR7	––	subtract and modify
circ( )	address in circular addressing	%	where circular addressing is performed
disp	displacement		

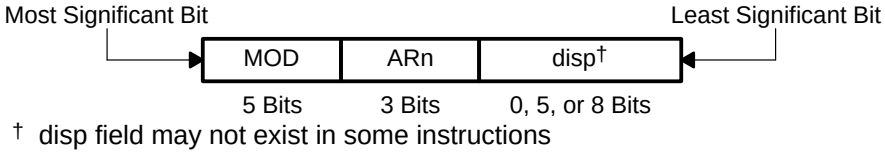
Table 5–2. Indirect Addressing (Continued)

Mod Field	Syntax	Operation	Description
		Indirect Addressing with Index Register IR1	
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
		Indirect Addressing (Special Cases)	
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 register AR0–AR7	++	add and modify
	B	where bit-reversed addressing is performed	%	where circular addressing is performed

Example 5–3, Example 5–4, Example 5–5, Example 5–6, Example 5–7, Example 5–8, Example 5–9, Example 5–10, Example 5–11, Example 5–12, Example 5–13, Example 5–14, Example 5–15, Example 5–16, Example 5–17, Example 5–18, and Example 5–19 exemplify indirect addressing in Table 5–2.

Figure 5–2. Instruction Encoding Format

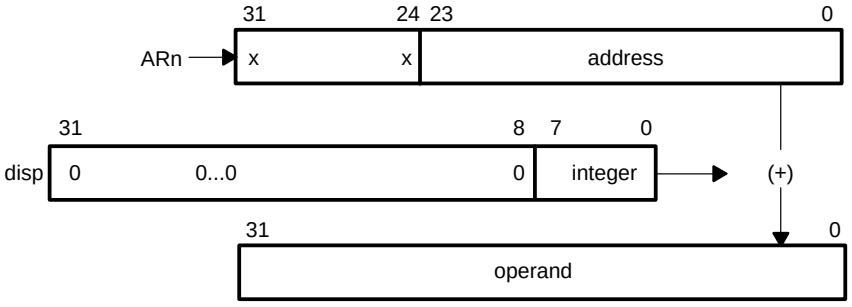


† disp field may not exist in some instructions

Example 5–3. Indirect With Predisplacement Add

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and the displacement (disp). The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

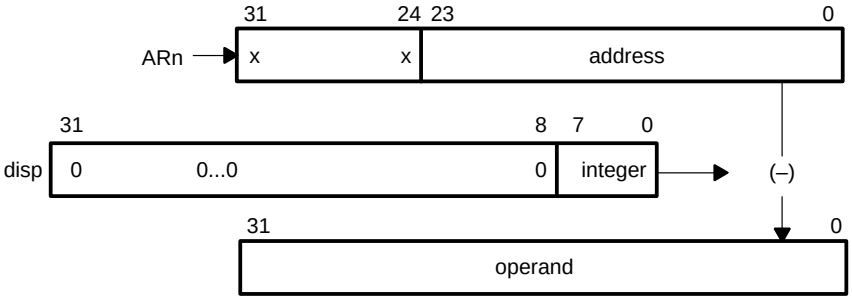
Operation: operand address = ARn + disp
Assembler Syntax: *+ ARn(disp)
Modification Field: 00000



Example 5–4. Indirect With Predisplacement Subtract

The address of the operand to be fetched is the contents of an auxiliary register (ARn) minus the displacement (disp). The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = ARn – disp
Assembler Syntax: *– ARn(disp)
Modification Field: 00001



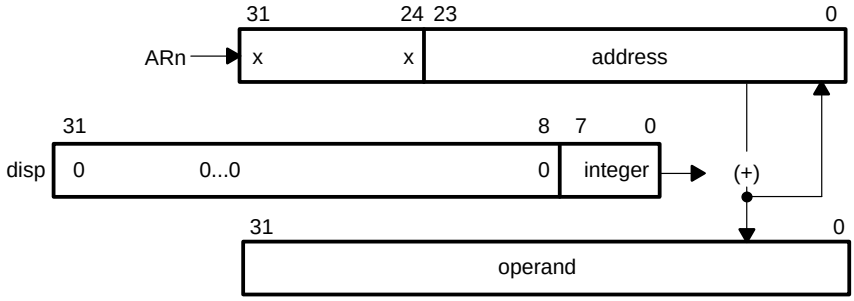
Example 5–5. Indirect With Predisplacement Add and Modify

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and the displacement (disp). The displacement is either an eight-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 = ARn + disp
ARn = ARn + disp

Assembler Syntax: *++ ARn (disp)

Modification Field: 00010



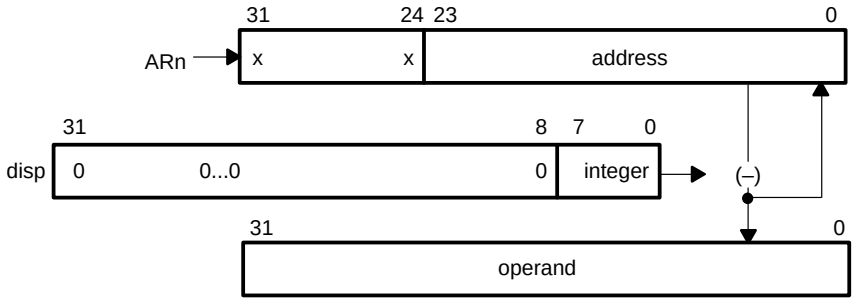
Example 5–6. Indirect With Predisplacement Subtract and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn) minus the displacement (disp). The displacement is either an eight-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 = ARn – disp
ARn = ARn – disp

Assembler Syntax: *–– ARn(disp)

Modification Field: 00011



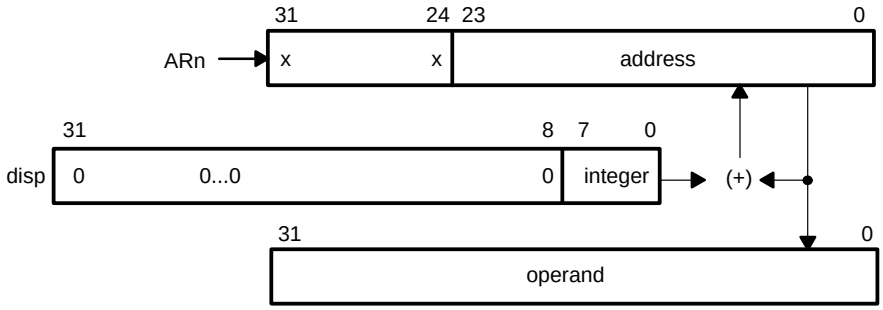
Example 5–7. Indirect With Postdisplacement Add and Modify

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

Operation: operand address = ARn
ARn = ARn + disp

Assembler Syntax: *ARn ++ (disp)

Modification Field: 00100



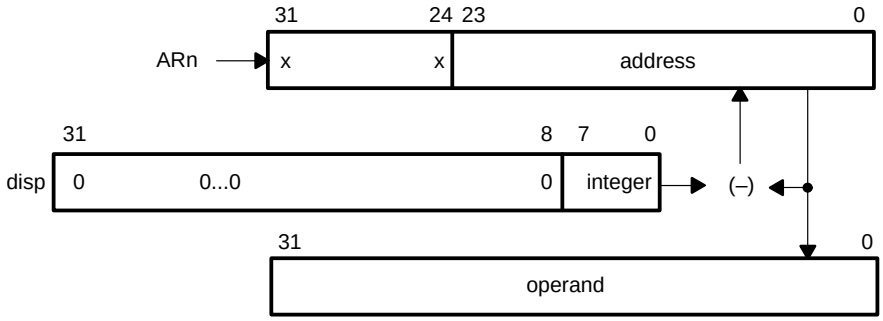
Example 5–8. Indirect With Postdisplacement Subtract and Modify

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

Operation: operand address = ARn
ARn = ARn – disp

Assembler Syntax: *ARn –– (disp)

Modification Field: 00101



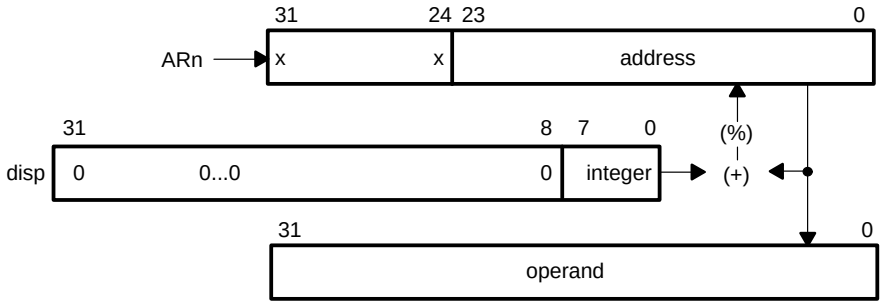
Example 5–9. Indirect With Postdisplacement Add and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). 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 eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = ARn
ARn = circ(ARn+disp)

Assembler Syntax: *ARn ++ (disp)%

Modification Field: 00110



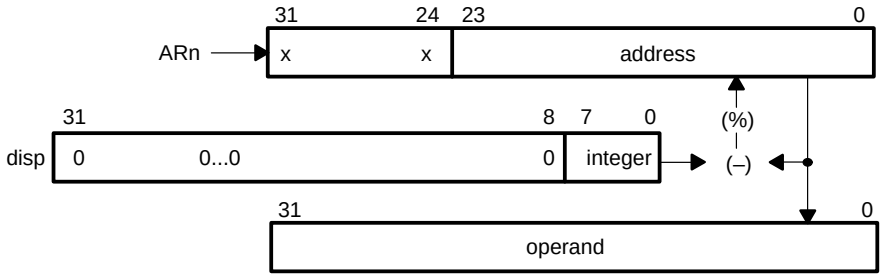
Example 5–10. Indirect With Postdisplacement Subtract and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). 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 eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = ARn
ARn = circ(ARn – disp)

Assembler Syntax: *ARn –– (disp)%

Modification Field: 00111

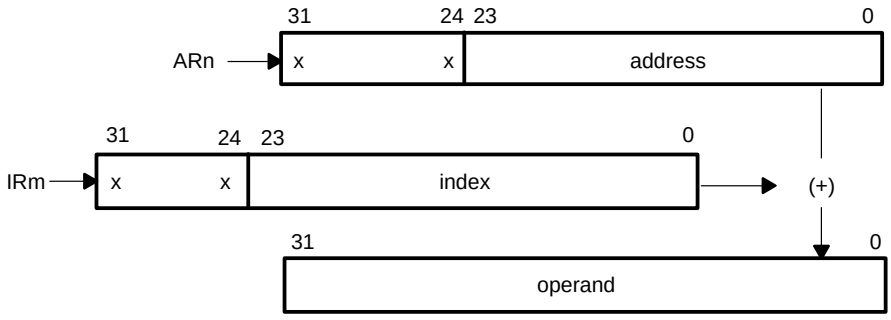


Example 5–11. Indirect With Preindex Add

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and an index register (IR0 or IR1).

Operation: operand address = ARn + IRm
Assembler Syntax: *+ ARn(IRm)

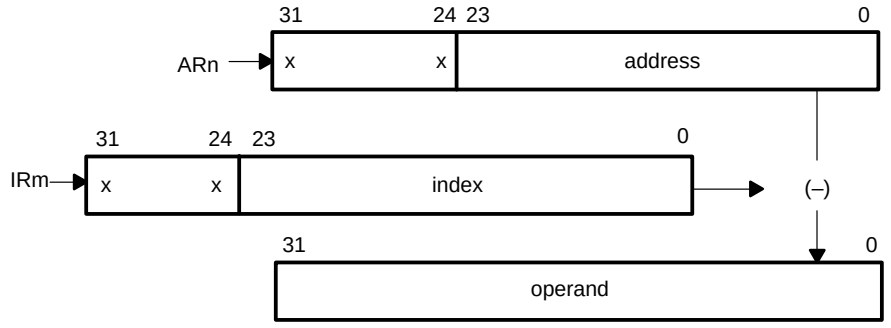
Modification Field: 01000 if m = 0
10000 if m = 1



Example 5–12. Indirect With Preindex Subtract

The address of the operand to be fetched is the difference of an auxiliary register (ARn) and an index register (IR0 or IR1).

Operation: operand address = ARn – IRm
Assembler Syntax: *– ARn(IRm)
Modification Field: 01001 if m = 0
10001 if m = 1



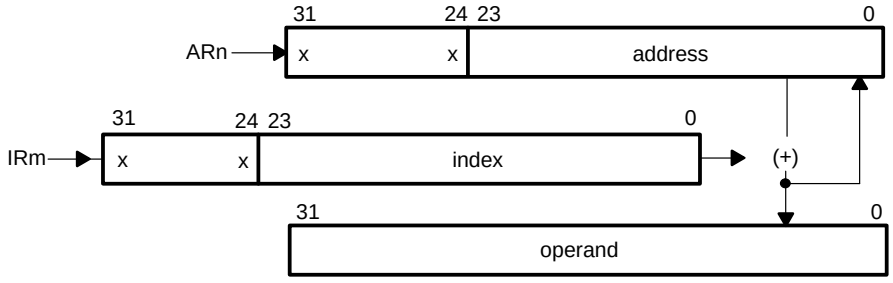
Example 5–13. Indirect With Preindex Add and Modify

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

Operation: operand address = ARn + IRm
ARn = ARn + IRm

Assembler Syntax: *++ ARn(IRm)

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



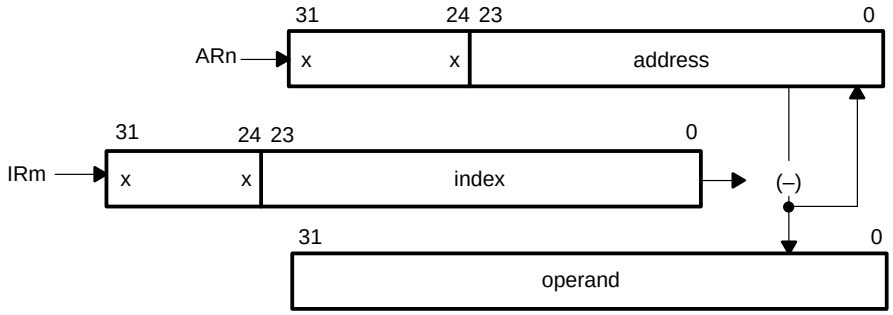
Example 5–14. Indirect With Preindex Subtract and Modify

The address of the operand to be fetched is the difference between an auxiliary register (ARn) and an index register (IR0 or IR1). The resulting address becomes the new contents of the auxiliary register.

Operation: operand address = ARn – IRm
ARn = ARn – IRm

Assembler Syntax: *––ARn(IRm)

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



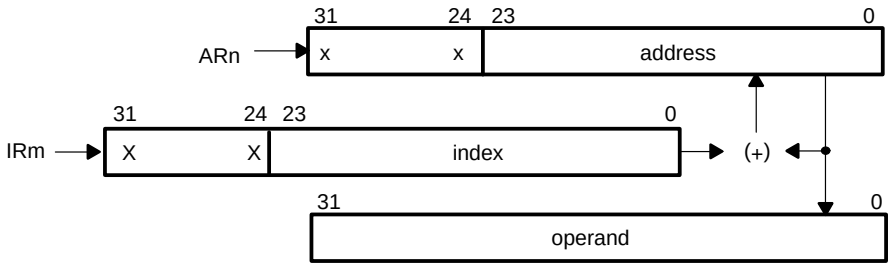
Example 5–15. Indirect With Postindex Add and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register.

Operation: operand address = ARn
ARn = ARn + IRm

Assembler Syntax: *ARn ++ (IRm)

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



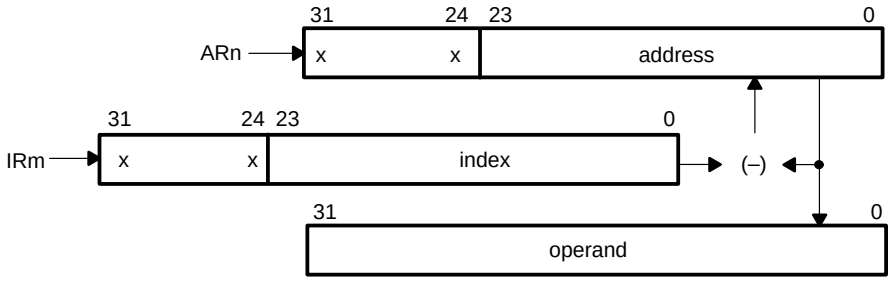
Example 5–16. Indirect With Postindex Subtract and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is subtracted from the auxiliary register.

Operation: operand address = ARn
ARn = ARn – IRm

Assembler Syntax: *ARn –– (IRm)

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



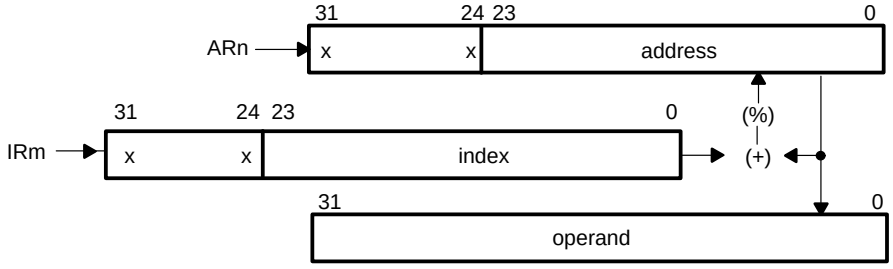
Example 5–17. Indirect With Postindex Add and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). 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 = ARn
ARn = circ(ARn + IRm)

Assembler Syntax: *ARn ++ (IRm)%

Modification Field: 01110 if m = 0
10110 if m = 1



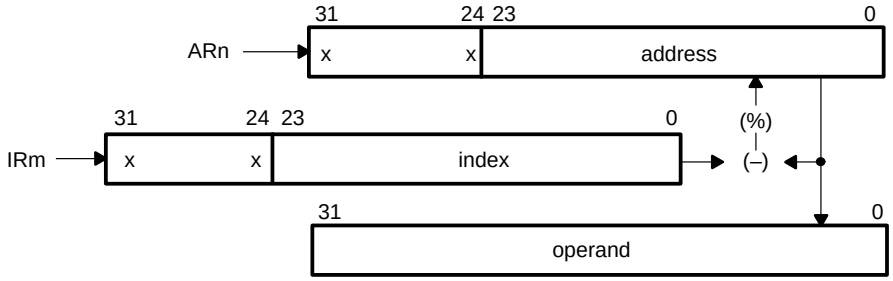
Example 5–18. Indirect With Postindex Subtract and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). 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 = ARn
ARn = circ(ARn – IRm)

Assembler Syntax: *ARn –– (IRm)%

Modification Field: 01111 if m = 0
10111 if m = 1



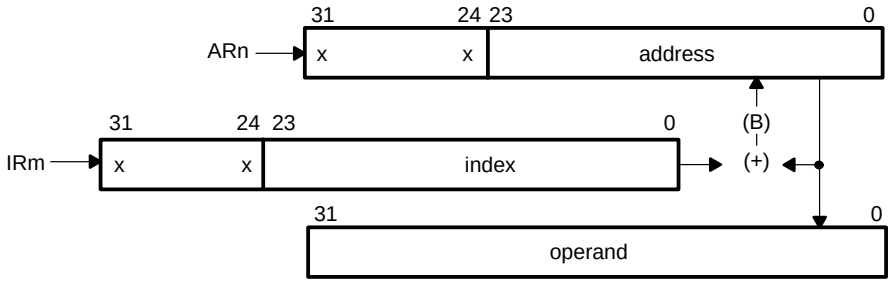
Example 5–19. Indirect With Postindex Add and Bit-Reversed Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). 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 = ARn
ARn = B(ARn + IR0)

Assembler Syntax: *ARn ++ (IR0)B

Modification Field: 11001



5.1.4 Short-Immediate Addressing

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

Syntax: **expr**

Example 5–20 illustrates before- and after-instruction data.

Example 5–20. Short-Immediate Addressing

```
SUBI  1,R0
```

Before Instruction:	**After Instruction**:
R0 = 0h	R0 = 0FFFFFFFFh

5.1.5 Long-Immediate Addressing

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

Syntax: **expr**

Example 5–21 illustrates before- and after-instruction data.

Example 5–21. Long-Immediate Addressing

```
BR      8000h
```

Before Instruction:	**After Instruction:**
PC = 0h	PC = 8000h

5.1.6 PC-Relative Addressing

Program counter (PC)-relative addressing is used for branching. It adds the contents of the 16 or 24 least significant bits 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 least significant bits 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 5–22 illustrates before- and after-instruction data.

Example 5–22. PC-Relative Addressing

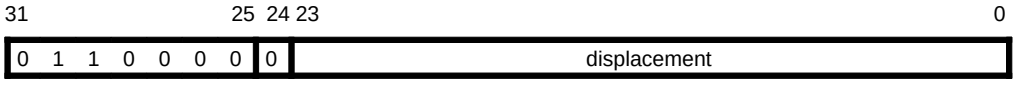
```
BU    NEWPC ;  pc=1001h, NEWPC label = 1005h, displacement = 3
```

Before Instruction decode phase:	**After Instruction execution phase:**
PC = 1002h	PC = 1005h

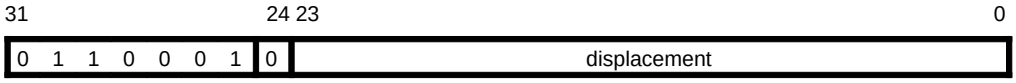
The 24-bit addressing mode encodes the program control instructions (for example, BR, BRD, CALL, RPTB, and RPTBD). Depending on the instruction, the new PC value is derived by adding a 24-bit signed value in the instruction word with the present PC value. Bit 24 determines the type of branch (D = 0 for a standard branch or D = 1 for a delayed branch). Some of the instructions are encoded in Figure 5–3.

Figure 5–3. Encoding for 24-Bit PC-Relative Addressing Mode

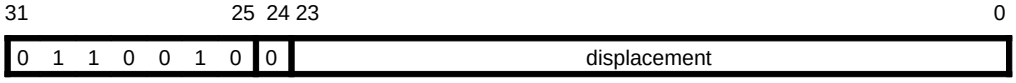
(a) BR, BRD: unconditional branches (standard and delayed)



(b) CALL: unconditional subroutine call



(c) RPTB: repeat block



5.2 Groups of Addressing Modes

Six types of addressing (covered in Section 5.1, beginning on page 5-2) form these four groups of addressing modes:

- General addressing modes (G)
- Three-operand addressing modes (T)
- Parallel addressing modes (P)
- Conditional-branch addressing modes (B)

5.2.1 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*

where 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:

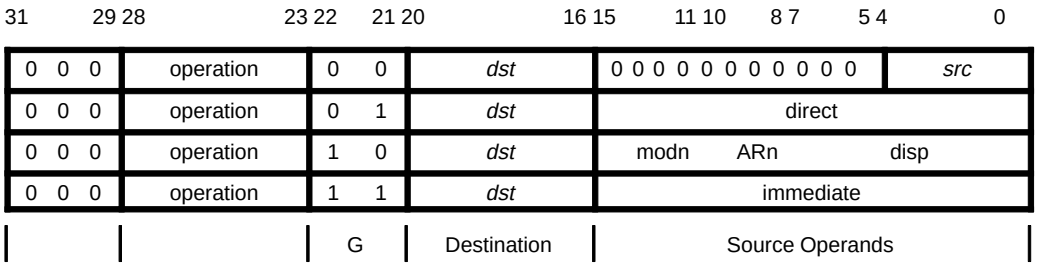
0 0	register (all CPU registers unless specified otherwise)
0 1	direct
1 0	indirect
1 1	immediate

If the *src* and *dst* fields contain register specifications, the value in these fields contains the CPU register addresses as defined by Table 5–1 on page 5-3. For the general addressing modes, the following values of ARn are valid:

ARn, $0 \leq n \leq 7$

Figure 5–4 shows the encoding for the general addressing modes. The notation mod indicates the modification field that goes with the ARn field. Refer to Table 5–2 on page 5-6 for further information.

Figure 5–4. Encoding for General Addressing Modes



5.2.2 Three-Operand Addressing Modes

Instructions that use the three-operand addressing modes, such as ADDI3, LSH3, CMPF3. or XOR3, usually have this form:

SRC1 operation SRC2 → *dst*

where the destination operand is signified by *dst* and the source operands by SRC1 and SRC2; operation defines an operation to be performed. Note that the 3 can be omitted from three-operand instructions.

Bits 31–29 are set to the value of 001, indicating three-operand addressing mode instructions. Bits 22 and 21 specify the three-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	SRC1	SRC2
0 0	register	register
0 1	indirect	register
1 0	register	indirect
1 1	indirect	indirect

Figure 5–5 shows the encoding for three-operand addressing. If the SRC1 and SRC2 fields use the same auxiliary register, both addresses are correctly generated. However, only the value created by the SRC1 field is saved in the auxiliary register specified. The assembler issues a warning if you specify this condition.

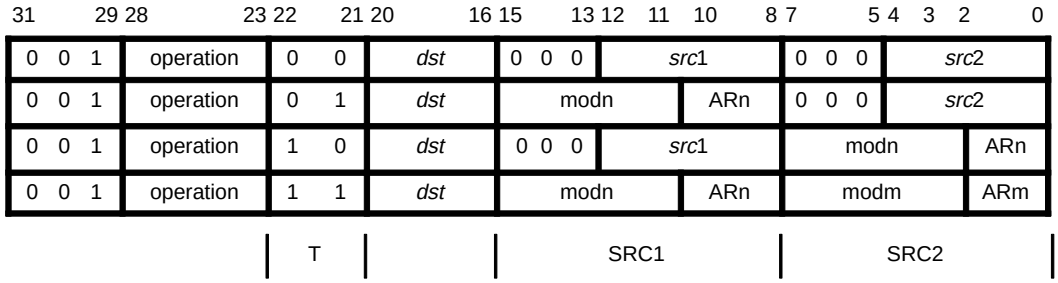
The following values of ARn and ARm are valid:

ARn,$0 \leq n \leq 7$
ARm,$0 \leq m \leq 7$

The notation modm or modn indicates that the modification field goes with the ARm or ARn field, respectively. Refer to Table 5–2 on page 5-6 for further information.

In indirect addressing of the three-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.

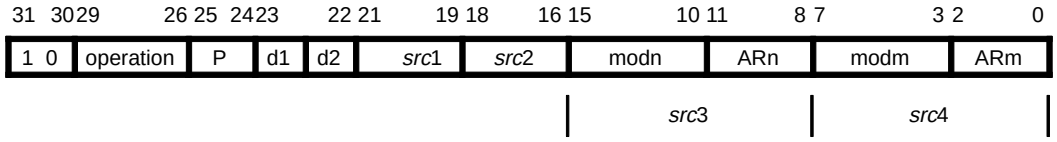
Figure 5–5. Encoding for Three-Operand Addressing Modes



5.2.3 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 5–6). 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 5–6. Encoding for Parallel Addressing Modes



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 (see Chapter 10). 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 bits 21–0 are to be interpreted 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 modn and modm indicate which modification field goes with which ARn or ARm (auxiliary register) field, respectively. Following is a list of the parallel addressing operands:

- *src*1 $0 \leq src1 \leq 7$ (extended-precision registers R0–R7)
- *src*2 $0 \leq src2 \leq 7$ (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 $0 \leq P \leq 3$
- *src*3 indirect (disp = 0, 1, IR0, IR1)
- *src*4 indirect (disp = 0, 1, IR0, IR1)

As in the three-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 5–6 on page 5-21, 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.

5.2.4 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. Selection of 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 Chapter 10 for a list of condition codes). Figure 5–7 shows the encoding for conditional-branch addressing.

Figure 5–7. Encoding for Conditional-Branch Addressing Modes

DB*cond* (D):

31 27	26	25	24 22	21	20 16	15 5	4 0
0 1 1 0 1 1	B	ARn	D	*cond*	0 0 0 0 0 0 0 0 0 0 0		*src* reg
0 1 1 0 1 1	B	ARn	D	*cond*	immediate (PC relative)		

B*cond* (D):

31 27	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)		

CALL*cond*:

31 27	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)		

5.3 Circular Addressing

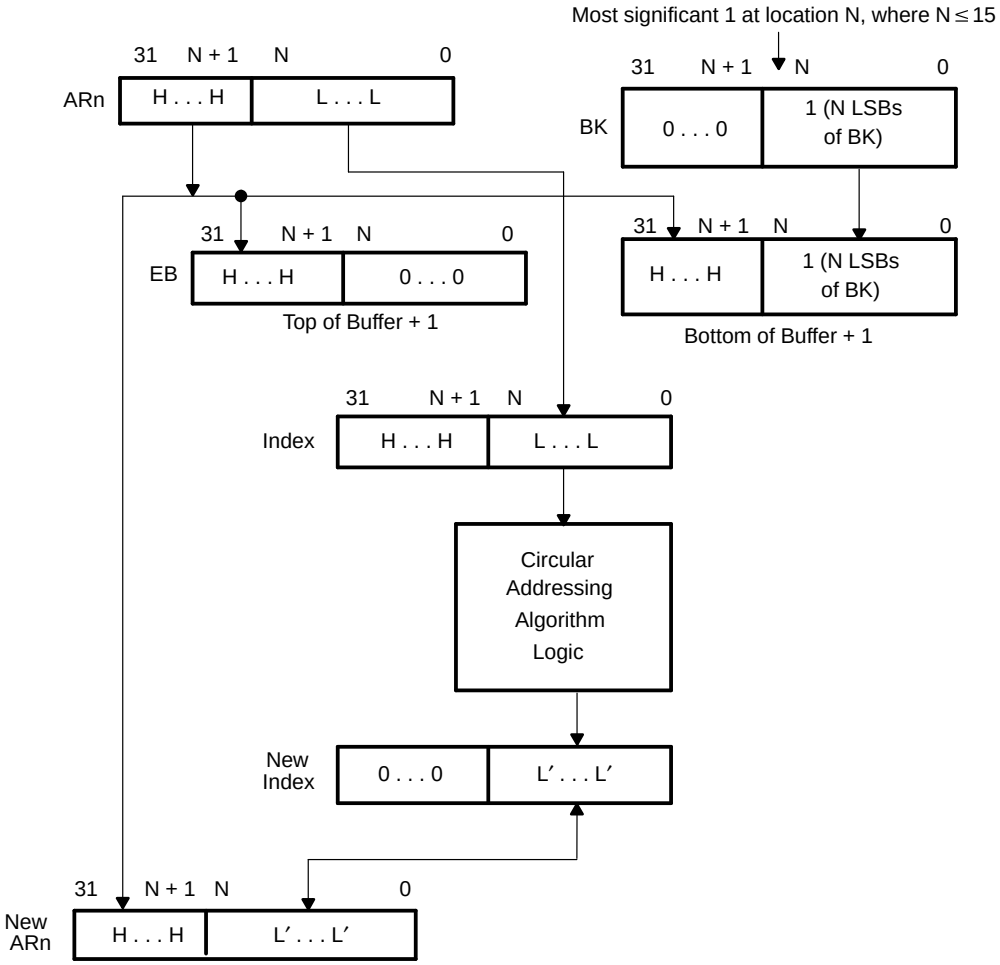
Many algorithms, such as convolution and correlation, require the implementation of a circular buffer in memory. In convolution and correlation, the circular buffer is used to implement a sliding window that contains the most recent data to be processed. As new data is brought in, the new data overwrites the oldest data. Key to the implementation of a circular buffer is the implementation of a circular addressing mode. This section describes the circular addressing mode of the TMS320C3x.

The block size register (BK) specifies the size of the circular buffer. By labeling the most significant 1 of the BK register as bit N, with $N \leq 15$, you can find the address immediately following the bottom of the circular buffer by concatenating bits 31 through N + 1 of a user-selected register (ARn) with bits N through 0 of the BK register. The address of the top of the buffer is referred to as the effective base (EB) and can be found by concatenating bits 31 through N + 1 of ARn, with bits N through 0 of EB being 0.

Figure 5–8 illustrates the relationships between the block size register (BK), the auxiliary registers (ARn), the bottom of the circular buffer, the top of the circular buffer, and the index into the circular buffer.

A circular buffer of size R must start on a K-bit boundary (that is, the K LSBs of the starting address of the circular buffer must be 0), where K is an integer that satisfies $2^K > R$. Since the value R must be loaded into the BK register, $K \geq N + 1$. For example, a 31-word circular buffer must start at an address whose five LSBs are 0 (that is, $XXXXXXXXXXXXXXXXXXXXXXXXXXX00000_2$), and the value 31 must be loaded into the BK register.

Figure 5–8. Flowchart for Circular Addressing



Legend:

ARn	auxiliary register n	BK	blocksize register
EB	effective base	H	high-order bits
L	low-order bits	L′	new low-order bits
LSB	least significant bit	N	bit value

In circular addressing, index refers to the N LSBs 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.
- 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:

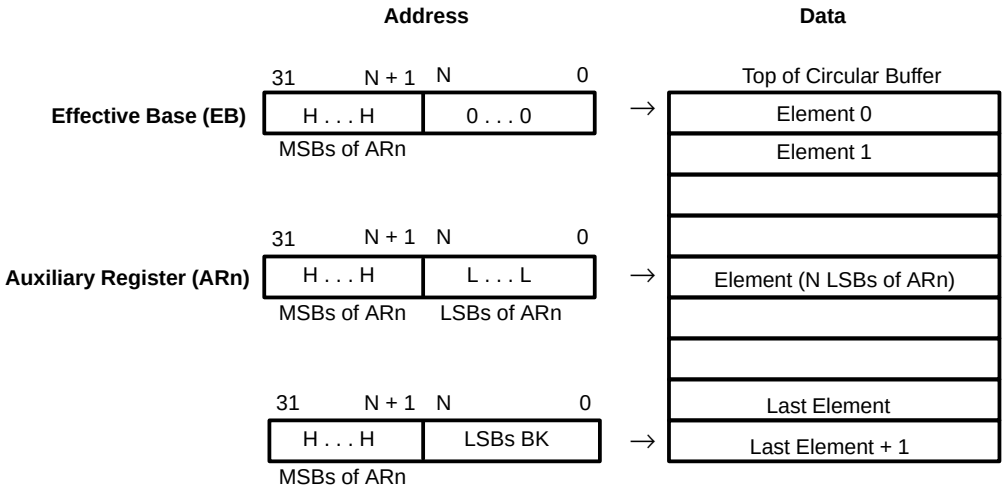
If $0 \leq$ index + step $<$ BK:
index = index + step.

Else if index + step $\geq$ BK:
index = index + step – BK.

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

Figure 5–9 shows how the circular buffer is implemented and illustrates the relationship of the quantities generated and the elements in the circular buffer.

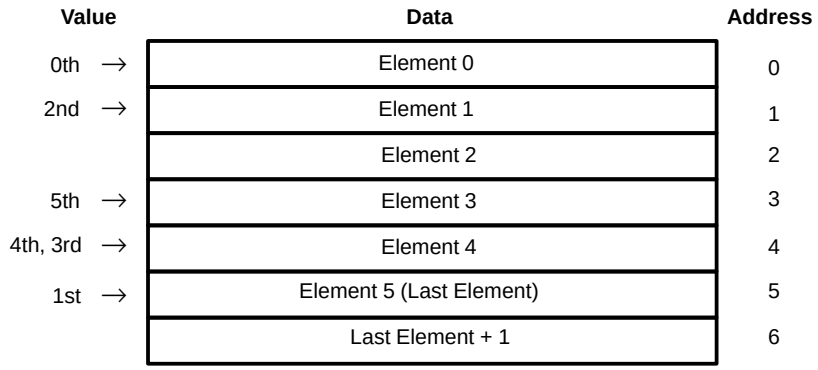
Figure 5–9. Circular Buffer Implementation



Example 5–23 shows circular addressing operation. Assuming that all ARs are four bits, let AR0 = 0000, and BK = 0110 (block size of 6). Example 5–23 shows a sequence of modifications and the resulting value of AR0. Example 5–23 also shows how the pointer steps through the circular queue with a variety of step sizes (both incrementing and decrementing).

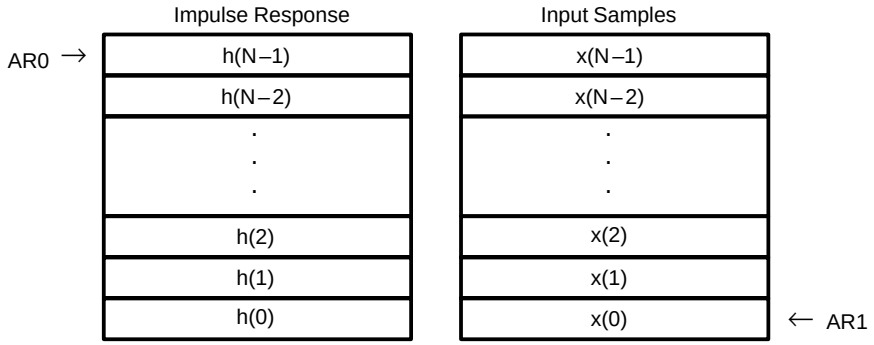
Example 5–23. Circular Addressing

```
*AR0++(5)%      ;  AR0 = 0   (0th 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 5–10 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 TMS320C3x code for the FIR filter shown in Example 5–24.

Figure 5–10. Data Structure for FIR Filters



Example 5–24. FIR Filter Code Using Circular Addressing

```
* Initialization
*
     LDI    N,BK              ; Load block size.
     LDI    H,AR0             ; Load pointer to impulse response.
     LDI    X,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.
```

5.4 Bit-Reversed Addressing

Bit-reversed addressing on the TMS320C3x enhances execution speed and program memory for FFT algorithms that use a variety of radices. The base address of bit-reversed addressing must be located on a boundary of the size of the table. For example, if IR0 = 2^{n-1}, the n LSBs of the base address must be 0. The base address of the data in memory must be on a 2^n boundary. One auxiliary register points to the physical location of a data value. IR0 specifies one-half the size of the FFT; that is, the value contained in IR0 must be equal to 2^{n-1}, where n is an integer and the FFT size is 2^n. When you add IR0 to the auxiliary register by using bit-reversed addressing, addresses are generated in a bit-reversed fashion.

To illustrate this kind of addressing, assume eight-bit auxiliary registers. Let AR2 contain the value 0110 0000 (96). This is the base address of the data in memory. Let IR0 contain the value 0000 1000 (8). Example 5–25 shows a sequence of modifications of AR2 and the resulting values of AR2.

Example 5–25. 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 5–3 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 5–3. 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

5.5 System and User Stack Management

The TMS320C3x 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**

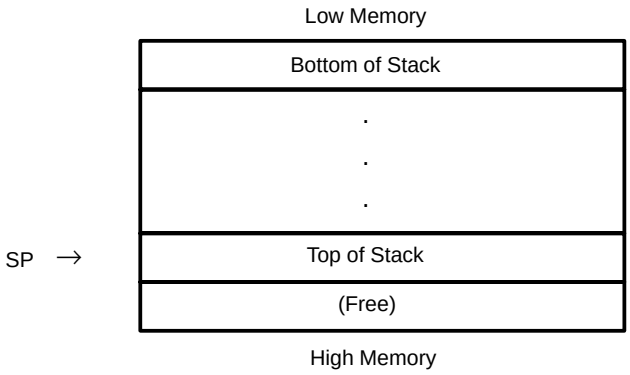
  The dequeue is a double-ended queue linear list for which insertions and deletions are made at either end of the list.

5.5.1 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 5–11). 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 5–11. System Stack Configuration



5.5.2 Stacks

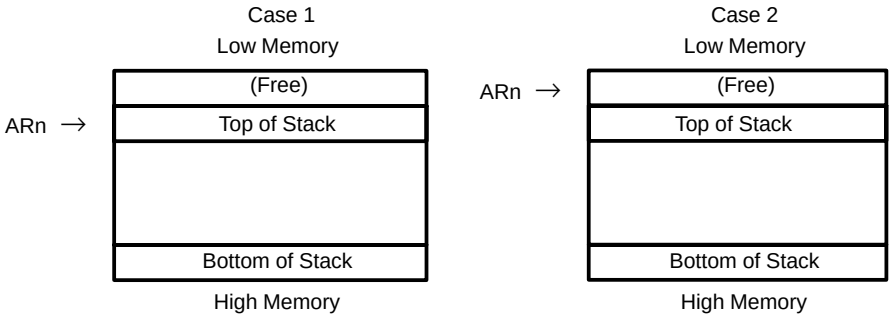
Stacks can be built from low to high memory or high to low memory. Two cases for each type of stack are shown. Stacks can be built using the preincrement/decrement and postincrement/decrement modes of modifying the auxiliary registers (AR). Stack growth from high-to-low memory can be implemented in two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

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

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

Figure 5–12 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 5–12. Implementations of High-to-Low Memory Stacks



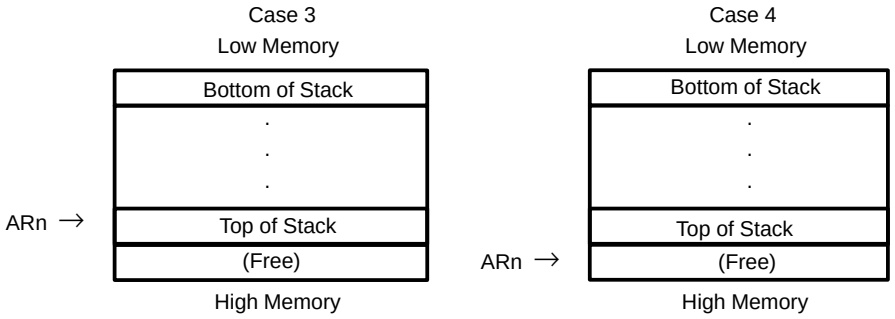
Stack growth from low-to-high memory can be implemented in two ways:

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

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

Figure 5–13 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 5–13. Implementations of Low-to-High Memory Stacks



5.5.3 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 6

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 operations, reset, and interrupts. Because programming includes a variety of constructs, you can select the one suited for your particular application.

Several interlocked operations instructions provide flexible multiprocessor support and, through the use of external signals, a powerful means of synchronization. They also guarantee the integrity of the communication and result in a high-speed operation.

The TMS320C3x supports a nonmaskable external reset signal and a number of internal and external interrupts. These functions can be programmed for a particular application.

This chapter discusses the following major topics:

Topic **Page**

6.1 Repeat Modes

The repeat modes of the TMS320C3x 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 TMS320C3x 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 four-cycle instructions. These four cycles of overhead occur during the initial execution of the loop. All subsequent executions of the loop have no overhead (zero cycle).

Three registers (RS, RE, and RC) are associated with the updating of the program counter (PC) when it is updated in a repeat mode. Table 6–1 describes these registers.

Table 6–1. 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.
RC	Repeat Count Register. Contains one 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.

For correct operation of the repeat modes, you must correctly initialize all of the above-mentioned registers.

6.1.1 Repeat-Mode Control Bits

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

- **RM bit.** The repeat-mode flag (RM) bit in the status register specifies whether the processor is running in the repeat mode.
  - RM = 0 indicates standard instruction fetching mode.
  - RM = 1 indicates repeat-mode instruction fetches.

- **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.
  - S = 0 indicates standard instruction fetches.
  - S = 1 and RM = 1 indicates repeat-single instruction fetches.

6.1.2 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 one less than the number of times to execute the block; for example, an RC value of 4 would execute the block five times. The detailed algorithm for the update of the PC is shown in Example 6–1.

Note: Maximum Number of Repeats

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.

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

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 6–1. 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
```

6.1.3 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 one. 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 6–2.

Example 6–2.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
```

Using the repeat-block mode of modifying the PC facilitates analysis of what would happen in the case of branches within the block. Assume that the next value of the PC will be either PC + 1 or the contents of the RS register. It is thus apparent that this method of block repeat allows much branching within the repeated block. Execution can go anywhere within the user's code via interrupts, subroutine calls, etc. For proper modification of the loop counter, the last instruction of the loop must be fetched. You can stop the repeating of the loop prior to completion by writing a 0 to the repeat counter or writing a 0 to the RM bit of the status register.

6.1.4 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 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 $\rightarrow$ RS
2) PC + 1 $\rightarrow$ RE
3) 1 $\rightarrow$ RM status register bit
4) 1 $\rightarrow$ S bit
5) *src* $\rightarrow$ 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.

6.1.5 Repeat-Mode Restrictions

Since the block repeat modes modify the program counter, other instructions cannot modify the program counter at the same time. There are two restrictions:

- 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 6–3 shows an incorrectly placed standard branch.
- 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 6–4 shows an incorrectly placed delayed branch.

Note: Rule Violation

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

Example 6–3. 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 6–4. 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
```

6.1.6 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 that 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 6–5 illustrates a pipeline conflict. Refer to Chapter 9 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 6–5. Pipeline Conflict in an RPTB Instruction

```
EDC     .word40000000h; 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
ENDLOOPLDI  *AR0,R0    ; The *AR0 read conflicts with
                       ; the instruction fetching
                       ; Then RC decrements to 0
                       ; If cache is enabled, RC decrements
                       ; to FFFF FFFFh
```

6.1.7 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 repeated execution of a block. 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 should 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 should restore RC, RE, RS, and ST, in that order. If the RM bit is not set, you don't need to save and restore these registers.

The order in which the registers are saved/restored is important to guarantee correct operation. The ST register should be restored last, after the RC, RE, and RS registers. ST should 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 should 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.

6.2 Delayed Branches

The TMS320C3x offers three main types of branching: standard, delayed, and conditional delayed.

Standard branches empty the pipeline before performing the branch; this guarantees correct management of the program counter and results in a TMS320C3x branch taking four cycles. Included in this class are repeats, calls, returns, and traps.

Delayed branches on the TMS320C3x do not empty the pipeline, but rather guarantee that the next three instructions will execute before the program counter is modified by the branch. The result is a branch that requires only a single cycle, thus making the speed of the delayed branch very close to that of the optimal block repeat modes of the TMS320C3x. 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 TMS320C3x 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. None of the three instructions that follow a delayed branch can be any of the following (see Example 6–6):

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

Delayed branches disable interrupts until the three instructions following the delayed branch are completed. This is independent of whether the branch is taken.

Note: Incorrect Use of Delayed Branches

If delayed branches are used incorrectly, the PC will be undefined.

Example 6–6. Incorrectly Placed Delayed Branches

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

6.3 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 stack thus provides a return using either the RETS*cond* or RETI*cond* 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 6–1 shows CALL response timing.
- The **CALL*cond*** instruction is similar to the CALL instruction (above) except for the following:
  - It executes only if a specific condition is true (the 20 conditions—including unconditional—are listed in Table 10–9 on page -13).
  - The *src* is either a PC-relative displacement or is in register-addressing mode.

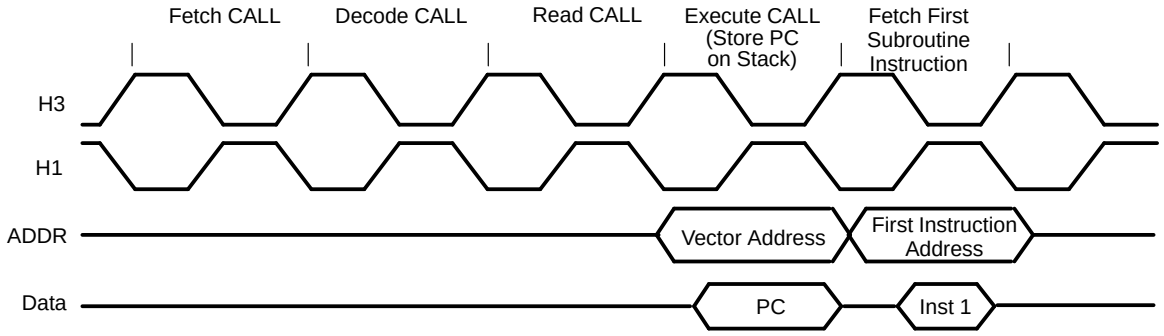
  The condition flags are set by a previous instruction only when the destination register is one of the extended-precision registers (R0–R7) or when one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3) is executed.
- The **TRAP*cond*** instruction also executes only if a specific 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) A vector is retrieved from one of the addresses 20h to 3Fh and is loaded into the PC.

  The particular address is identified by a trap number in the instruction. Using the RETI*cond* to return re-enables interrupts.
- **RETS*cond*** returns execution from any of the above three instructions by popping the top of the stack to the PC. To execute, the specified condition must be true. Conditions are the same as for the CALL*cond* instruction.
- **RETI*cond*** returns from traps or calls like the RETS*cond* (above) 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. Conditions are the same as for the CALL*cond* instruction.

Calls and traps accomplish the same functional task (that is, a subfunction is called and executed, and control is then returned to the calling function). Traps offer several advantages. Among them are the following:

- Interrupts are automatically disabled when a trap is executed. This allows critical code to execute without risk of being interrupted. Thus, traps are generally terminated with a RETI*cond* instruction to re-enable 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, the functions in the kernel can be modified and relocated without the need to recompile each application.

Figure 6–1. CALL Response Timing



6.4 Interlocked Operations

Among the most common multiprocessing configurations is the sharing of global memory by multiple processors. In order 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 TMS320C3x interlocked operations.

The TMS320C3x 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 6–2.

Table 6–2. 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	*src* → *dst* Clear interlock
STII	Store integer to memory, interlocked	*src* → *dst* Clear interlock

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, and 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 ($\overline{RDY}_{int}$ or $\overline{XRDY}_{int}$). ($\overline{RDY}_{int}$ and $\overline{XRDY}_{int}$ are a combination of external ready input and software wait states. Refer to Chapter 7, *External Bus Operation*, 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 ($\overline{RDY}_{int}$ or $\overline{XRDY}_{int}$) is signaled.
3) Leave XF0 set to 0 and end the read cycle.

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, $\overline{STRB}$, $\overline{MSTRB}$, or $\overline{IOSTRB}$ 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 is 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 ($\overline{RDY}_{int}$ or $\overline{XRDY}_{int}$) is signaled.

As in the case for LDFI and LDII, the *dst* of STFI and STII affects XF0. If *dst* is located off-chip ($\overline{STRB}$, $\overline{MSTRB}$, or $\overline{IOSTRB}$ 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 operations are as an STF or STI to internal memory.

The SIGI instruction functions as follows:

1) Sets XF0 to 0.
2) Idles until XF1 is set to 0.
3) Sets XF0 to 1 and ends the operation.

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 ($\overline{RDY}_{int}$ or‘ $\overline{XRDY}_{int}$) in order to be interrupted. Because interrupts are taken on bus cycle boundaries (see Section 6.6), an interrupt may be taken any time after a valid ready. This 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.

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 TMS320C3xs. The following examples illustrate the usefulness of the interlocked operations instructions.

Example 6–7 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 6–7. 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
```

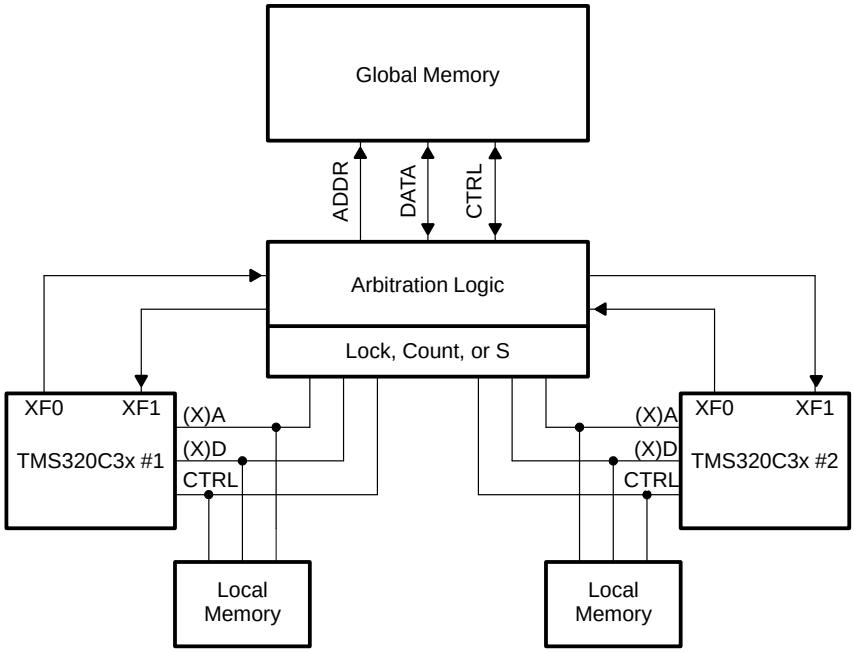
Example 6–8 shows how a location COUNT may contain a count of the number of times a particular operation needs to be performed. This operation may be performed by any processor in the system. If the count is 0, the processor waits until it is nonzero before beginning processing. The example also shows the algorithm for modifying COUNT correctly.

Example 6–8. 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 6–2 illustrates multiple TMS320C3xs sharing global memory and using the interlocked instructions as in Example 6–9, Example 6–10, and Example 6–11.

Figure 6–2. Multiple TMS320C3xs Sharing Global Memory



It might sometimes 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 non-negative 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 accessing and modifying S.

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 TMS320C3x code for V(S) is shown in Example 6–9; code for P(S) is shown in Example 6–10. Compare the code in Example 6–10 to the code in Example 6–8.

Example 6–9. Implementation of V(S)

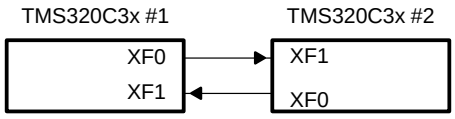
```
V:  LDII  @S,R0     ; Interlocked read of S begins (XFO = 0)
                    ; Contents of S → R0
    ADDI  1,R0      ; Increment R0 (= S)
    STII  R0,@S     ; Update S, end interlock (XF0 = 0)
```

Example 6–10. 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)
```

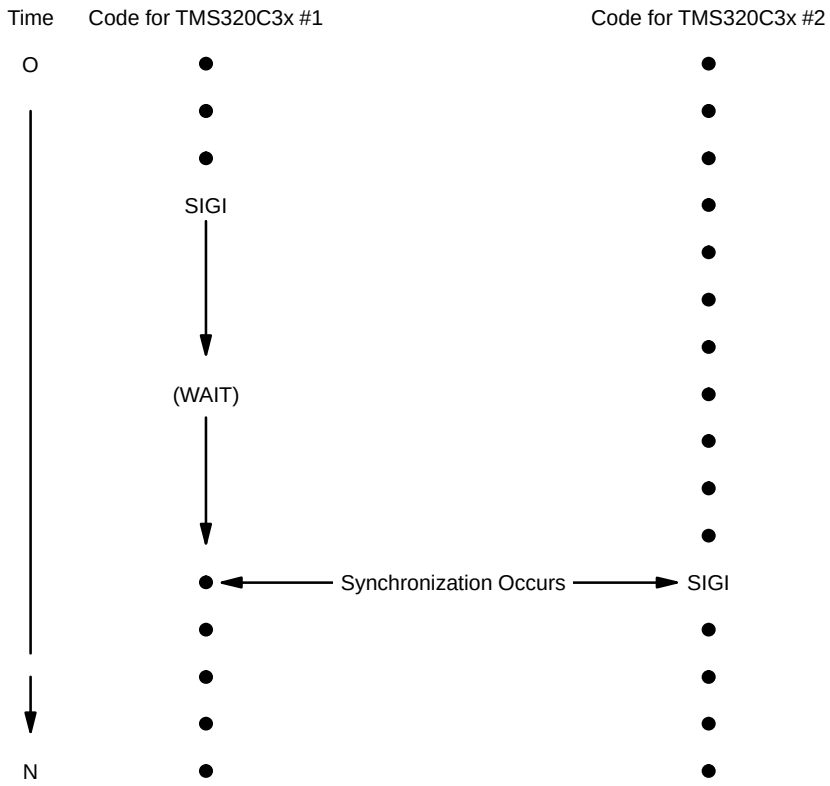
The SIGI operation can synchronize, at an instruction level, multiple TMS320C3xs. Consider two processors connected as shown in Figure 6–3. The code for the two processors is shown in Example 6–11.

Figure 6–3. Zero-Logic Interconnect of TMS320C3xs



Processor #1 runs until it executes the SIGI. It then waits until processor #2 executes a SIGI. At this point, the two processors have synchronized and continue execution.

Example 6–11. Code to Synchronize Two TMS320C3xs at the Software Level



6.5 Reset Operation

The TMS320C3x supports a nonmaskable external reset signal ($\overline{\text{RESET}}$), which is used to perform system reset. This section discusses the reset operation.

At powerup, the state of the TMS320C3x 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 TMS320C3x (see Chapter 13 for more information).

Reset affects the other pins on the device in either a synchronous or asynchronous manner. The synchronous reset is gated by the TMS320C3x's internal clocks. The asynchronous reset directly affects the pins and is faster than the synchronous reset. Table 6–3 shows the state of the TMS320C3x's pins after $\overline{\text{RESET}}$ = 0. Each pin is described according to whether the pin is reset synchronously or asynchronously.

Table 6–3. Pin Operation at Reset

Signal	# Pins	Operation at Reset
		Primary Interface (61 Pins)
D31–D0	32	Synchronous reset; placed in high-impedance state
A23–A0	24	Synchronous reset; placed in high-impedance state
R/$\overline{W}$	1	Synchronous reset; deasserted by going to a high level
STRB	1	Synchronous reset; deasserted by going to a high level
RDY	1	Reset has no effect.
HOLD	1	Reset has no effect.
HOLDA	1	Reset has no effect.
		Expansion Interface (49 Pins)†
XD31–XD0	32	Synchronous reset; placed in high-impedance state
XA12–XA0	13	Synchronous reset; placed in high-impedance state
XR/$\overline{W}$	1	Synchronous reset; placed in high-impedance state
MSTRB	1	Synchronous reset; deasserted by going to a high level
IOSTRB	1	Synchronous reset; deasserted by going to a high level
XRDY	1	Reset has no effect.
		Control Signals (9 Pins)
RESET	1	Reset input pin
INT3–INT0	4	Reset has no effect.
IACK	1	Synchronous reset; deasserted by going to a high level
MC/$\overline{MP}$ or MCBL/$\overline{MP}$	1	Reset has no effect.
XF1–XF0	2	Asynchronous reset; placed in high-impedance state

† Present only on TMS320C30

Table 6–3. Pin Operation at Reset (Continued)

Signal	# Pins	Operation at Reset
		Serial Port 0 Signals (6 Pins)
CLKX0	1	Asynchronous reset; placed in high-impedance state
DX0	1	Asynchronous reset; placed in high-impedance state
FSX0	1	Asynchronous reset; placed in high-impedance state
CLKR0	1	Asynchronous reset; placed in high-impedance state
DR0	1	Asynchronous reset; placed in high-impedance state
FSR0	1	Asynchronous reset; placed in high-impedance state
		Serial Port 1 Signals (6 Pins) †
CLKX1	1	Asynchronous reset; placed in high-impedance state
DX1	1	Asynchronous reset; placed in high-impedance state
FSX1	1	Asynchronous reset; placed in high-impedance state
CLKR1	1	Asynchronous reset; placed in high-impedance state
DR1	1	Asynchronous reset; placed in high-impedance state
FSR1	1	Asynchronous reset; placed in high-impedance state
		Timer 0 Signal (1 Pin)
TCLK0	1	Asynchronous reset; placed in high-impedance state
		Timer 1 Signal (1 Pin)
TCLK1	1	Asynchronous reset; placed in high-impedance state
		Supply and Oscillator Signals (29 Pins)
V_{DD} (3–0)	4	Reset has no effect.
$IODV_{DD}$ (1,0)	2	Reset has no effect.
ADV_{DD} (1,0)	2	Reset has no effect.
PDV_{DD}	1	Reset has no effect.
DDV_{DD} (1,0)	2	Reset has no effect.
MDV_{DD}	1	Reset has no effect.
V_{SS} (3–0)	4	Reset has no effect.

† Present only on TMS320C30

Table 6–3. Pin Operation at Reset (Continued)

Signal	# Pins	Operation at Reset
DV_{SS} (3–0)	2	Reset has no effect.
CV_{SS} (1,0)	2	Reset has no effect.
IV_{SS}	1	Reset has no effect.
V_{BBP}	1	Reset has no effect.
SUBS	1	Reset has no effect.
X1	1	Reset has no effect.
X2/CLKIN	1	Reset has no effect.
H1	1	Synchronous reset. Will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition. See Chapter 13.
H3	1	Synchronous reset. Will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition. See Chapter 13.
		Emulation, Test, and Reserved (18 Pins)
EMU0	1	Undefined
EMU1	1	Undefined
EMU2	1	Undefined
EMU3	1	Undefined
EMU4/$\overline{\text{SHZ}}$	1	Undefined
EMU5†	1	Undefined
EMU6†	1	Undefined
RSV0†	1	Undefined
RSV1†	1	Undefined
RSV2†	1	Undefined
RSV3†	1	Undefined
RSV4†	1	Undefined
RSV5†	1	Undefined
RSV6†	1	Undefined
RSV7†	1	Undefined
RSV8†	1	Undefined
RSV9†	1	Undefined
RSV10†	1	Undefined

† Present only on TMS320C30

At system reset, the following additional operations are performed:

- The peripherals are reset. This is a synchronous operation. The peripheral reset is described in Chapter 8.
- The external bus control registers are reset. The reset values of the control registers are described in Chapter 7.
- The following CPU registers are loaded with 0:
  - ST (CPU status register)
  - IE (CPU/DMA interrupt enable flags)
  - IF (CPU interrupt flags)
  - IOF (I/O flags)
- The reset vector is read from memory location 0h and loaded into the PC. This vector contains the start address of the system reset routine.
- Execution begins. Refer to Example 11–1 on page 11-3 for an illustration of a processor initialization routine.

Multiple TMS320C3xs 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 TMS320C3xs will come up on the same internal phase.

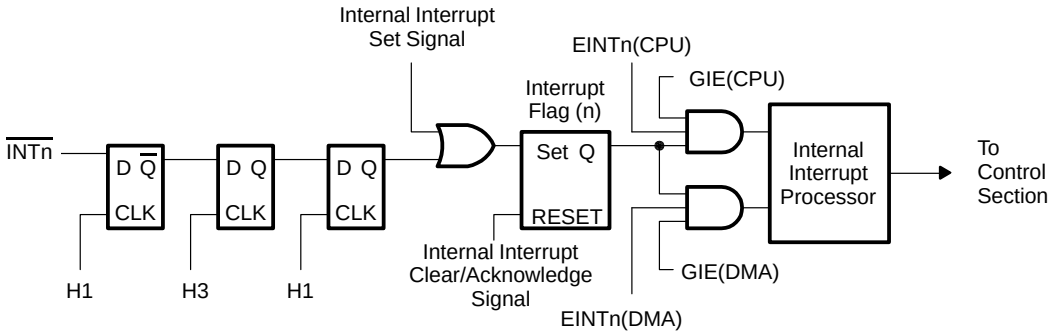
Unless otherwise specified, all registers are undefined after reset.

6.6 Interrupts

The TMS320C3x supports multiple internal and external interrupts, which can be used for a variety of applications. This section discusses the operation of these interrupts.

A functional diagram of the logic used to implement the external interrupt inputs is shown in Figure 6–4; the logic for internal interrupts is similar. Additional information regarding internal interrupts can be found in Chapter 8.

Figure 6–4. Interrupt Logic Functional Diagram



External interrupts are synchronized internally, as illustrated by the three flip-flops clocked by H1 and H3. Once synchronized, the interrupt input will set the corresponding interrupt flag register (IF) bit if the interrupt is active.

External interrupts are latched internally on the falling edge of H1 (see Chapter 13 for timing information). An external interrupt must be held low for at least one H1/H3 cycle to be recognized by the TMS320C3x. Interrupts should be held low for only one or two H1 falling edges. If the interrupt is held low for three or more H1 falling edges, multiple interrupts may be recognized.

6.6.1 Interrupt Vector Table

Table 6–4 and Table 6–5 contain the interrupt vectors. In the microprocessor mode of the TMS320C30 and the TMS320C31 (Table 6–4) and the microcomputer mode of the TMS320C31 (Table 6–5), 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 TMS320C31, the interrupt vector contains a branch instruction to the start of the interrupt service routine.

Table 6–4. Reset, Interrupt, and Trap-Vector Locations for the TMS320C30/TMS320C31 Microprocessor Mode

Address	Routine
00h	RESET
01h	INT0
02h	INT1
03h	INT2
04h	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)

† Reserved on TMS320C31

Table 6–5. Reset, Interrupt, and Trap-Vector Locations for the TMS320C31 Microcomputer Boot Mode

Address	Description
809FC1	$\overline{\text{INT0}}$
809FC2	$\overline{\text{INT1}}$
809FC3	$\overline{\text{INT2}}$
809FC4	$\overline{\text{INT3}}$
809FC5	$\overline{\text{XINT0}}$
809FC6	$\overline{\text{RINT0}}$
809FC7	Reserved
809FC8	Reserved
809FC9	$\overline{\text{TINT0}}$
809FCA	$\overline{\text{TINT1}}$
809FCB	$\overline{\text{DINT0}}$
809FCC–809FDF	Reserved
809FE0	$\overline{\text{TRAP0}}$
809FE1	$\overline{\text{TRAP1}}$
•	•
•	•
•	•
809FFB	$\overline{\text{TRAP27}}$
809FFC–809FFF	Reserved

6.6.2 Interrupt Prioritization

When two interrupts occur in the same clock cycle or when two previously received interrupts are waiting to be serviced, one interrupt will be serviced before the other. The CPU handles this prioritization by servicing the interrupt with the least priority. Table 6–6 shows the priorities assigned to the reset and interrupt vectors.

The CPU controls all prioritization of interrupts (see Table 6–6 for reset and interrupt vector locations and priorities).

Table 6–6. 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 timer 0
TINT1	0Ah	10	Internal interrupt generated by timer 1
DINT	0Bh	11	Internal interrupt generated by DMA controller 0

† Reserved on TMS320C31

6.6.3 Interrupt Control Bits

Four CPU registers contain bits used to control interrupt operation:

- Status Register (ST)

  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. Refer to subsection 3.1.7 on page 3-4 for more information.

- CPU/DMA Interrupt Enable Register (IE)

  This register individually enables/disables CPU and DMA (external, serial port, and timer) interrupts. Refer to subsection 3.1.8 on page 3-7 for more information.

- CPU Interrupt Flag Register (IF)

  This register contains interrupt flag bits that indicate the corresponding interrupt is set. Refer to subsection 3.1.9 on page 3-9 for more information.

- DMA Global Control Register

  Interrupts to the DMA are controlled by the synchronization bits of the DMA global control register. DMA interrupts are independent of the ST (GIE) bit.

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. It should be noted, however, that if $\overline{\text{INTn}}$ is still low when the interrupt acknowledge signal occurs, the interrupt flag bit will be cleared for only one cycle and then set again because $\overline{\text{INTn}}$ is still low. Accordingly, it is theoretically possible that, depending on when the IF register is read, this bit may be 0 even though $\overline{\text{INTn}}$ is 0. When the TMS320C3x is reset, 0 is written to the interrupt flag register, thereby clearing all pending interrupts.

The interrupt flag register bits may be read and written 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.

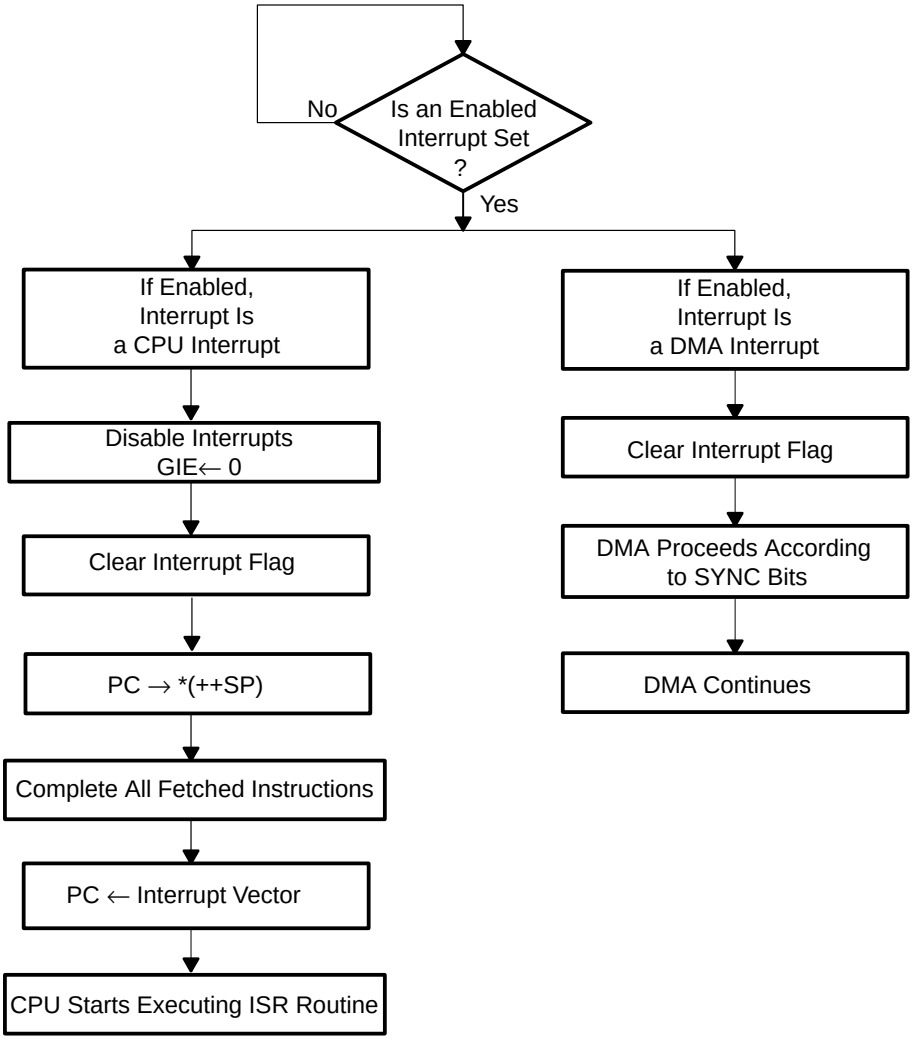
Internal interrupts operate in a similar manner. In the IF register, the bit corresponding to an internal interrupt may be read and written through software. Writing a 1 sets the interrupt latch; writing a 0 clears it. All internal interrupts are one H1/H3 cycle in length.

The CPU global interrupt enable bit (GIE), located in the CPU status register (ST), controls all CPU interrupts. All DMA interrupts are controlled by the DMA global interrupt enable bit, which is not dependent on ST(GIE) and is local to the DMA. The DMA global interrupt enable bit is dependent, in part, on the state of the DMA SYNC bits. It is not directly accessible through software (see Chapter 8). The AND of the interrupt flag bit and the interrupt enables is then connected to the interrupt processor.

6.6.4 Interrupt Processing

The ’C3x allows the CPU and DMA coprocessor to respond to and process interrupts in parallel. Figure 6–5 on page 6-28 shows interrupt processing flow; for exact sequence, refer to Table 6–7 on page 6-29.

Figure 6–5. Interrupt Processing



Note: CPU and DMA Interrupts

CPU and DMA 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 and DMA interrupts are not acknowledged until instruction fetching continues.

Table 6–7. 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	Temporarily disable interrupt until GIE is cleared.	—	interrupt	prog a	prog a–1
3	Read the interrupt vector table.	—	—	interrupt	prog a
4	Clear Interrupt flag; clear GIE bit; 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

In the CPU interrupt processing cycle (left side of Figure 6–5), 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 fetched and loaded into the PC, and the CPU starts executing the first instruction in the interrupt service routine (ISR).

If you wish to make the interrupt service routine interruptible, you can set the GIE bit to 1 after entering the ISR.

The DMA interrupt processing cycle (right side of Figure 6–5) 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.

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 interrupt service routine. However, it may be better suited at the end of the interrupt service routine or be totally unnecessary.

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. This is not the case for 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.

6.6.5 CPU Interrupt Latency

CPU interrupt latency, defined as the time from the acknowledgement of the interrupt to the execution of the first interrupt service routine (ISR) instruction, is at least eight cycles. This is explained in Table 6–7 on page 6-29, where the interrupt is treated as an instruction. It assumed that all of the instructions are single-cycle instructions.

6.6.6 CPU/DMA Interaction

If the DMA is not using interrupts for synchronization of transfers, it will not be 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 will not be 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, 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.

6.6.7 TMS320C3x Interrupt Considerations

Give careful consideration to TMS320C3x 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 is set to 0 by an interrupt. This 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 TMS320C3x 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.

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 traps. For example, you can use TRAP 0 to reset GIE and use TRAP 1 to set GIE. This is accomplished by making TRAP 0 and TRAP 1 be the instructions RETS and RETI, respectively.

Another alternative incorporates the following code fragment, which protects against modifying or saving of the status register by disabling interrupts through the interrupt enable register:

```
PUSH  IE          ;  Save IE register
LDI   0, IE       ;  Clear IE register
NOP               ;
NOP               ;
AND   0DFFFh, ST  ;  Set GIE = 0
POP   IE          ;
                  ;
                  ;
                  ;
```

- Added instructions to avoid pipeline problems
- 2 NOPs or useful instructions
- Instruction that reads or writes to ST register. Added instruction to avoid pipeline problems.

6.6.8 TMS320C30 Interrupt Considerations

The TMS320C30 has two unique exceptions to the interrupt operation.

- 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, and
  - A pipeline conflict has occurred, resulting in a delay in the decode or read phases of the instruction.

  During the decode phase of a conditional trap, interrupts are temporarily disabled to ensure that the trap will execute 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. In the case that the trap condition is *false*, interrupts will be permanently disabled until the GIE bit is intentionally set. The condition does not present itself when the trap condition is *true***,** because normal operation of the instruction causes the GIE to be reset, and standard coding practice will set the GIE to 1 before the trap routine is exited. Several instruction sequences that can 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. Therefore, the following solution is recommended to avoid or rectify the problem.

Insert two NOP instructions immediately prior to 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 TMS320C3x 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 currently gives the CPU write priority; consequently, the asserted interrupt might be lost. This is particularly true if the asserted interrupt has been generated internally (for example, a direct memory access (DMA) interrupt). This situation can arise 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 not to use the interrupt polling technique but to design the external interrupt inputs to have pulse widths of between 1 and 2 instruction cycles. The alternative to strict polling is to periodically enable and disable the interrupts that would be polled, thereby allowing the normal interrupt vectoring to take place; that automatically clears the interrupt flag without affecting other interrupts. If you need to clear a pending interrupt, it is recommended that you use a memory location to indicate that the interrupt is invalid. Then 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 a dummy interrupt due to a long interrupt pulse might be handled:

```
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
            BZ    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                   ;
```

6.6.9 Prioritization and Control

The CPU controls all prioritization of interrupts (see Table 6–8 for reset and interrupt vector locations and priorities). If the DMA is not using interrupts for synchronization of transfers, it will not be affected by the processing of the CPU interrupts. Detected interrupts are responded to by the CPU and DMA on instruction fetch boundaries only. If instruction fetches are halted due to pipeline conflicts or when executing instructions in an RPTS loop, interrupts will not be 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, make 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, thereby returning control to the CPU following the DMA transfer.

Since the DMA and CPU share the same set of interrupt flags, the DMA can 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.

Table 6–8. Reset and Interrupt Vector Locations

Reset or Interrupt	Vector Location	Priority	Function
$\overline{\text{RESET}}$	0h	0	External reset signal input on the $\overline{\text{RESET}}$ pin
$\overline{\text{INT0}}$	1h	1	External interrupt input on the $\overline{\text{INT0}}$ pin
$\overline{\text{INT1}}$	2h	2	External interrupt input on the $\overline{\text{INT1}}$ pin
$\overline{\text{INT2}}$	3h	3	External interrupt input on the $\overline{\text{INT2}}$ pin
$\overline{\text{INT3}}$	4h	4	External interrupt input on the $\overline{\text{INT3}}$ 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 timer 0
TINT1	0Ah	10	Internal interrupt generated by timer 1
DINT	0Bh	11	Internal interrupt generated by DMA controller 0

† Reserved on TMS320C31

6.7 TMS320LC31 Power Management Modes

The TMS320LC31 CPU has been enhanced by the addition of two power management modes:

- IDLE2, and
- LOPOWER.

6.7.1 IDLE2

The H1 instruction clock is held high until one of the four external interrupts is asserted. In IDLE2 mode, the TMS320C31 behaves as follows:

- No instructions are executed.
- The CPU, peripherals, and internal memory retain their previous states.
- The primary bus output pins are idle:
  - The address lines remain in their previous states,
  - The data lines are in the high-impedance state, and
  - The output control signals are inactive.
- When the device is in the functional (non-emulation) mode, the clocks stop with H1 high and H3 low (see Figure 6–6).
- The ’C31 will 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 high if H1 was previously high). The H1 and H3 clocks will remain 180° out of phase with each other (see Figure 6–7).
- For one of the four external interrupts to be recognized and serviced by the CPU during the IDLE2 operation, the interrupt must be asserted for less than three cycles but more than two cycles.
- The instruction following the IDLE2 instruction will not be executed until after the return from interrupt instruction (RETI) is executed.
- When the device is in emulation mode, the H1 and H3 clocks will continue to run normally and the CPU will operate as if an IDLE instruction had been executed. The clocks continue to run for correct operation of the emulator.

Delayed Branch

For correct device operation, the three instructions after a delayed branch should not be IDLE or IDLE2 instructions.

Figure 6–6. IDLE2 Timing

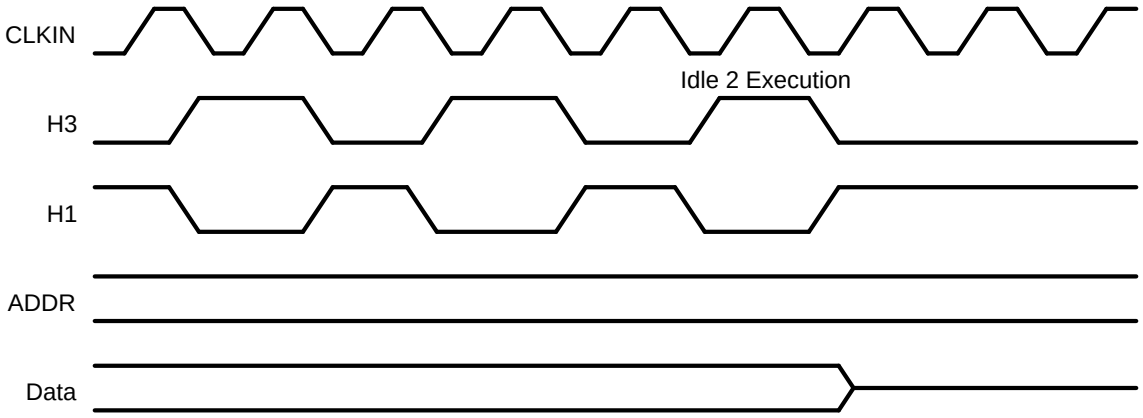
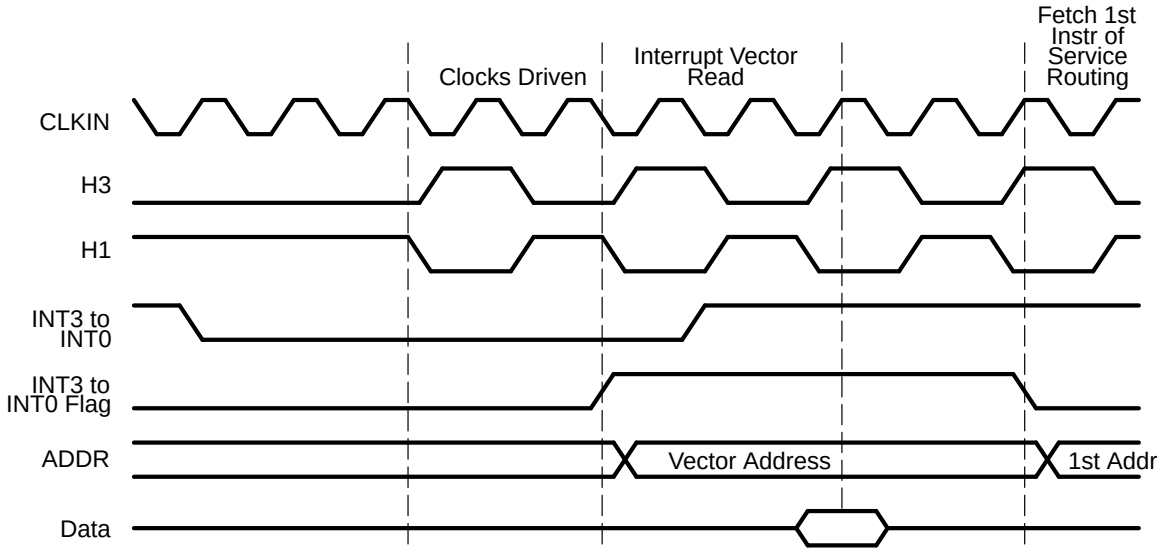


Figure 6–7. Interrupt Response Timing After IDLE2 Operation



6.7.2 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 $\frac{CLKIN\ frequency}{16}$.

A TMS320C31 with a CLKIN frequency of 32 MHz will perform identically to a 2 MHz TMS320C31 with an instruction cycle time of 1,000 ns.

During the read phase of the . . .	The TMS320C31 . . .
LOPOWER instruction (Figure 6–8)	slows to 1/16 of full-speed operation.
MAXSPEED instruction (Figure 6–9)	resumes full-speed operation.

Figure 6–8. LOPOWER Timing

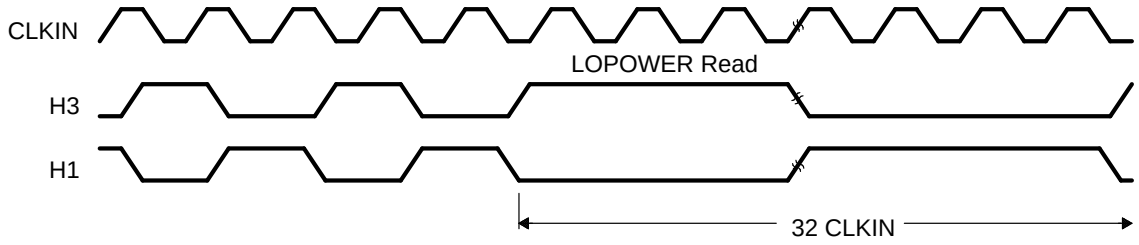
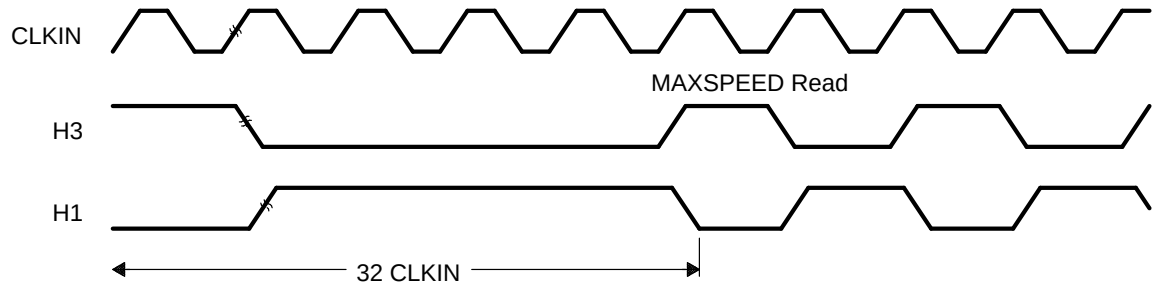


Figure 6–9. MAXSPEED Timing



Chapter 7

External Bus Operation

Memories and external peripheral devices are accessible through two external interfaces on the TMS320C30:

- the primary bus, and
- the expansion bus.

On the TMS320C31, 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.

Major topics discussed in this chapter are listed below.

Topic **Page**

7.1 External Interface Control Registers

The TMS320C30 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. Both buses support software-controlled wait states and an external ready input signal, and both buses are useful for data, program, and I/O accesses.

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 TMS320C30 supports two types of accesses:

- Memory access signalled 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 7–1.

Figure 7–1. Memory-Mapped External Interface Control Registers

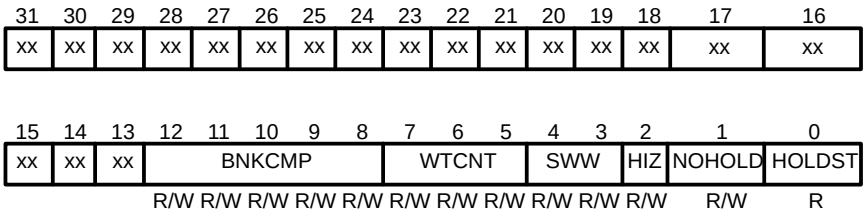
Register	Peripheral Address
Expansion-Bus Control (see subsection 7.1.2)†	808060h
Reserved	808061h
Reserved	808062h
Reserved	808063h
Primary-Bus Control (see subsection 7.1.1)	808064h
Reserved	808065h
Reserved	808066h
Reserved	808067h
Reserved	808068h
Reserved	808069h
Reserved	80806Ah
Reserved	80806Bh
Reserved	80806Ch
Reserved	80806Dh
Reserved	80806Eh
Reserved	80806Fh

† Reserved on the TMS320C31

7.1.1 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 7–2). Table 7–1 lists the register bits with the bit names and functions.

Figure 7–2. Primary-Bus Control Register



NOTE: xx = reserved bit, read as 0.
R = read, W = write.

Table 7–1. Primary-Bus Control Register Bits Summary

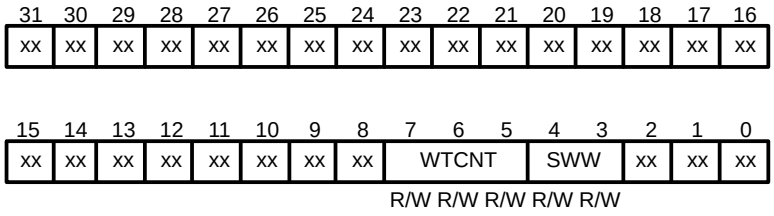
Bit	Name	Reset Value	Function
0	HOLDST	x †	Hold status bit. This bit signals whether the port is being held (HOLDST = 1) or is not being held (HOLDST = 0). This status bit is valid whether the port has been held via hardware or software.
1	NOHOLD	0	Port hold signal. NOHOLD allows or disallows the port to be held by an external HOLD signal. When NOHOLD = 1, the TMS320C3x 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 is received. However, it is asserted if an internal hold is generated (HIZ = 1). NOHOLD is set to 0 at reset.
2	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 TMS320C3x can relinquish the external memory port through software. HOLDA goes low when the port is placed in the high-impedance state. HIZ is set to 0 at reset.
4–3	SWW	11	Software wait mode. In conjunction with WTCNT, this two-bit field defines the mode of wait-state generation. It is set to 1 1 at reset.
7–5	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. It is set to 1 1 1 at reset.
12–8	BNKCMP	10000	Bank compare. This five-bit field specifies the number of MSBs of the address to be used to define the bank size. It is set to 1 0 0 0 0 at reset.
31–13	Reserved	0–0	Read as 0.

† x = 0 or 1

7.1.2 Expansion-Bus Control Register

The expansion-bus control register is a 32-bit register that contains control bits for the expansion bus (see Figure 7–3 and Table 7–2).

Figure 7–3. Expansion-Bus Control Register



NOTE: xx = reserved bit, read as 0.
R = read, W = write.

Table 7–2. Expansion-Bus Control Register Bits Summary

Bit	Name	Reset Value	Function
2–0	Reserved	000	Read as 0.
4–3	SWW	11	Software wait-state generation. In conjunction with the WTCNT, this two-bit field defines the mode of wait-state generation. It is set to 1 1 at reset.
7–5	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 clock cycles. It is set to 1 1 1 at reset.
31–8	Reserved	0–0	Read as 0.

7.2 External Interface Timing

This section discusses functional timing of operations on the primary bus and the expansion bus, the TMS320C3x's two independent parallel buses. Detailed timing specifications for all TMS320C3x signals are contained in Section 13.5 on page 13-30.

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 $\overline{\text{IOSTRB}}$ 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 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 acronym ($\overline{\text{M}}$)$\overline{\text{STRB}}$ 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 ($\overline{\text{X}}$)$\overline{\text{RDY}}$ 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.

7.2.1 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 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. Recall that, internally (from the perspective of the CPU and DMA), writes require only one cycle if no accesses to that interface are in progress. The following discussions pertain to zero wait-state accesses unless otherwise specified.

The ($\overline{\text{M}}$)$\overline{\text{STRB}}$ 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 (($\overline{\text{M}}$)$\overline{\text{STRB}}$ low) of writes only, there is a transition cycle of H1. This transition cycle consists of the following sequence:

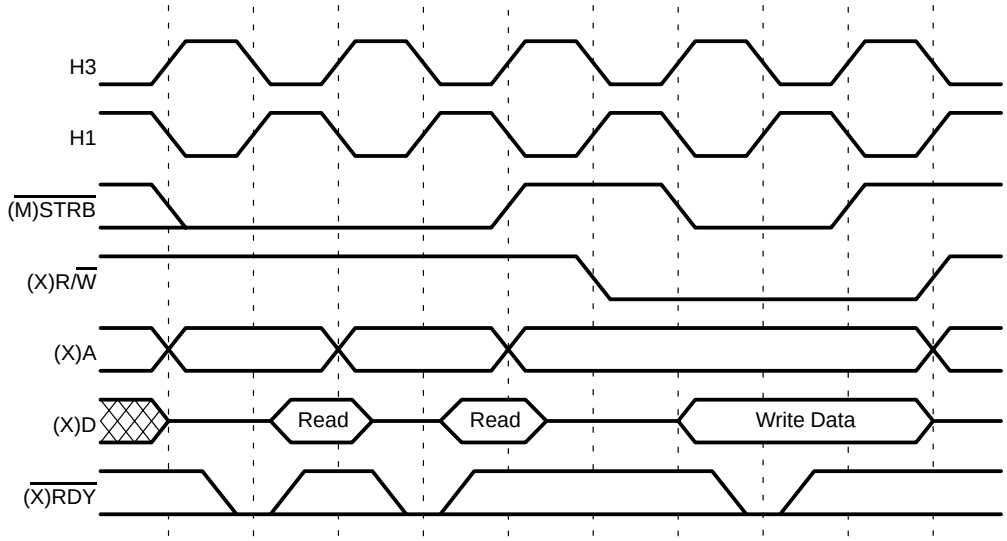
1) ($\overline{\text{M}}$)$\overline{\text{STRB}}$ is high.

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 H1 falling.

Figure 7–4 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. Note that 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 7–4. Read-Read-Write for $\overline{\text{(M)STRB}}$ = 0



Note: Back-to-Back Read Operations

$\overline{\text{(M)STRB}}$ will remain low during back-to-back read operations.

Figure 7–5 illustrates a write-write-read sequence for (M)STRB active and no wait states. The address and data written are held valid approximately one-half cycle after (M)STRB changes.

Figure 7–5. Write-Write-Read for (M)STRB = 0

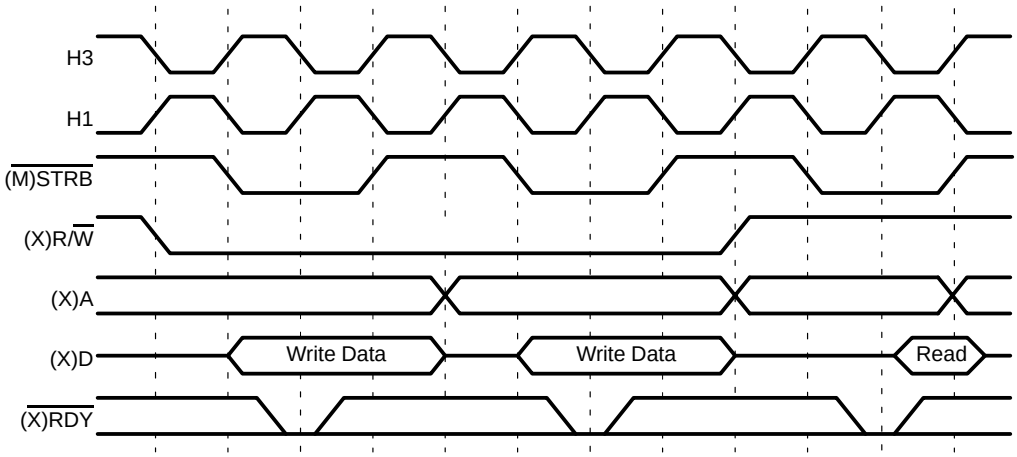


Figure 7–6 illustrates a read cycle with one wait state. Since $\overline{(X)RDY}$ = 1, the read cycle is extended. $\overline{(M)STRB}$, (X)R/$\overline{W}$, and (X)A are also extended one cycle. The next time $\overline{(X)RDY}$ is sampled, it is 0.

Figure 7–6. Use of Wait States for Read for $\overline{(M)STRB}$ = 0

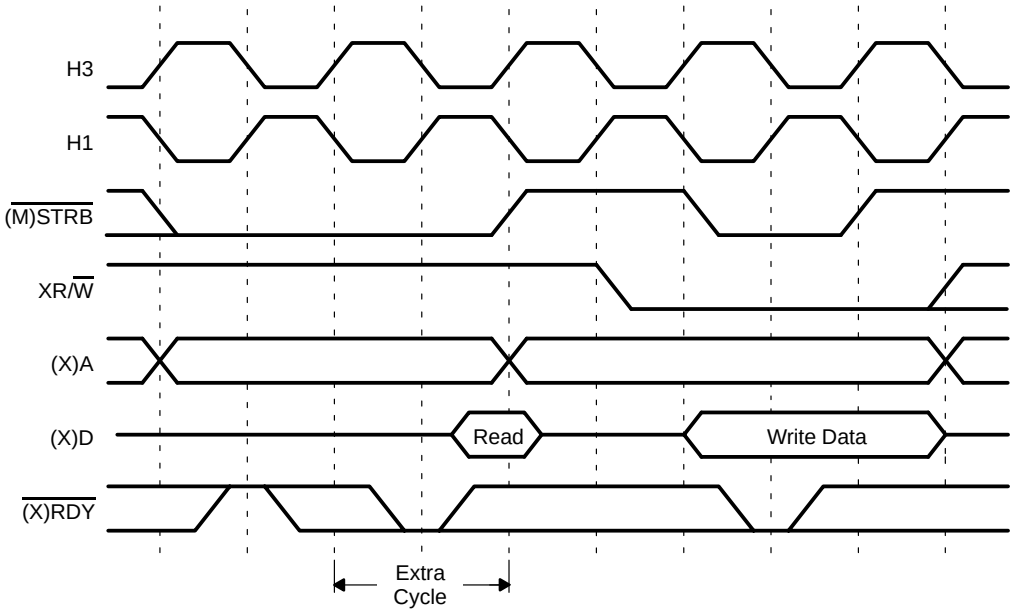
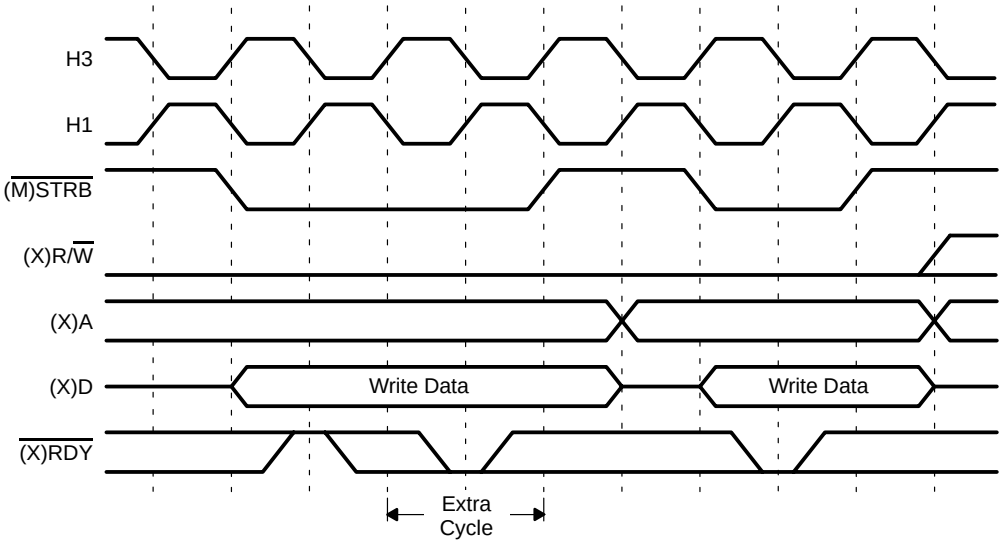


Figure 7–7 illustrates a write cycle with one wait state. Since initially $\overline{(X)RDY}$ = 1, the write cycle is extended. ($\overline{M}$)$\overline{STRB}$, (X)R/$\overline{W}$, and (X)A are extended one cycle. The next time $\overline{(X)RDY}$ is sampled, it is 0.

Figure 7–7. Use of Wait States for Write for $\overline{(M)STRB}$ = 0



7.2.2 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 7–8 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 7–8. Read and Write for $\overline{\text{IOSTRB}}$ = 0

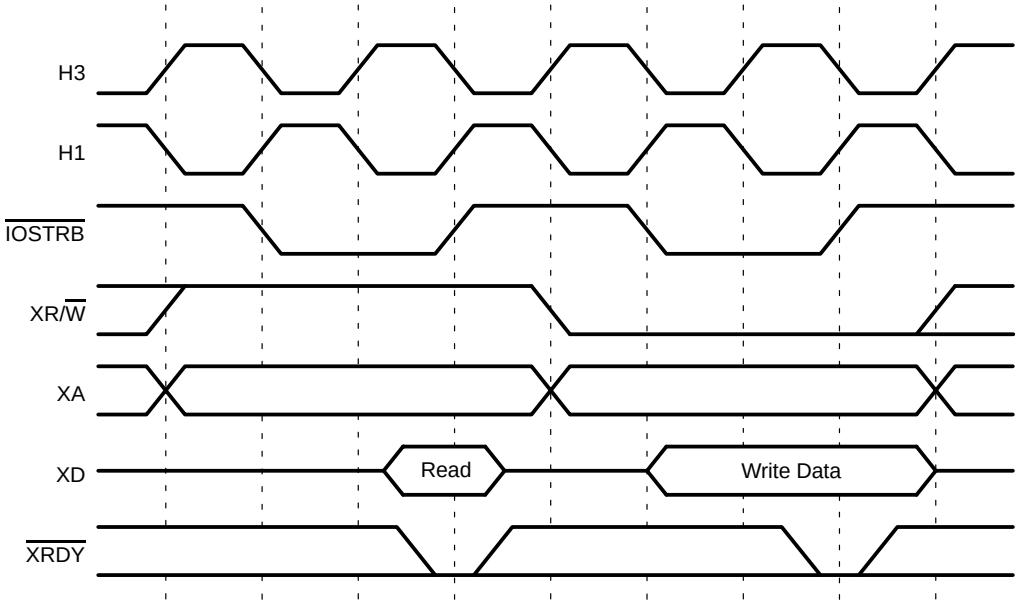


Figure 7–9 illustrates a read with one wait state when $\overline{\text{IOSTRB}}$ is active, and Figure 7–10 illustrates a write with one wait state when $\overline{\text{IOSTRB}}$ is active. For each wait state added, $\overline{\text{IOSTRB}}$, XR/$\overline{\text{W}}$, and XA are extended one clock cycle. Writes hold the data on the bus one additional cycle. The sampling of $\overline{\text{XRDY}}$ is repeated each cycle.

Figure 7–9. Read With One Wait State for $\overline{\text{IOSTRB}}$ = 0

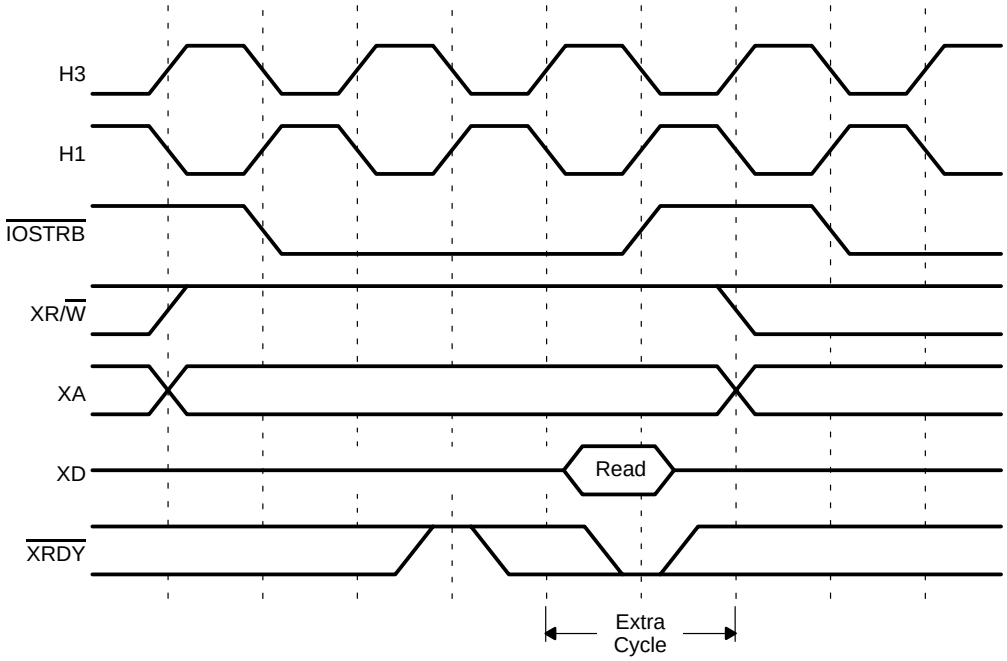


Figure 7–10. Write With One Wait State for $\overline{IOSTRB}$ = 0

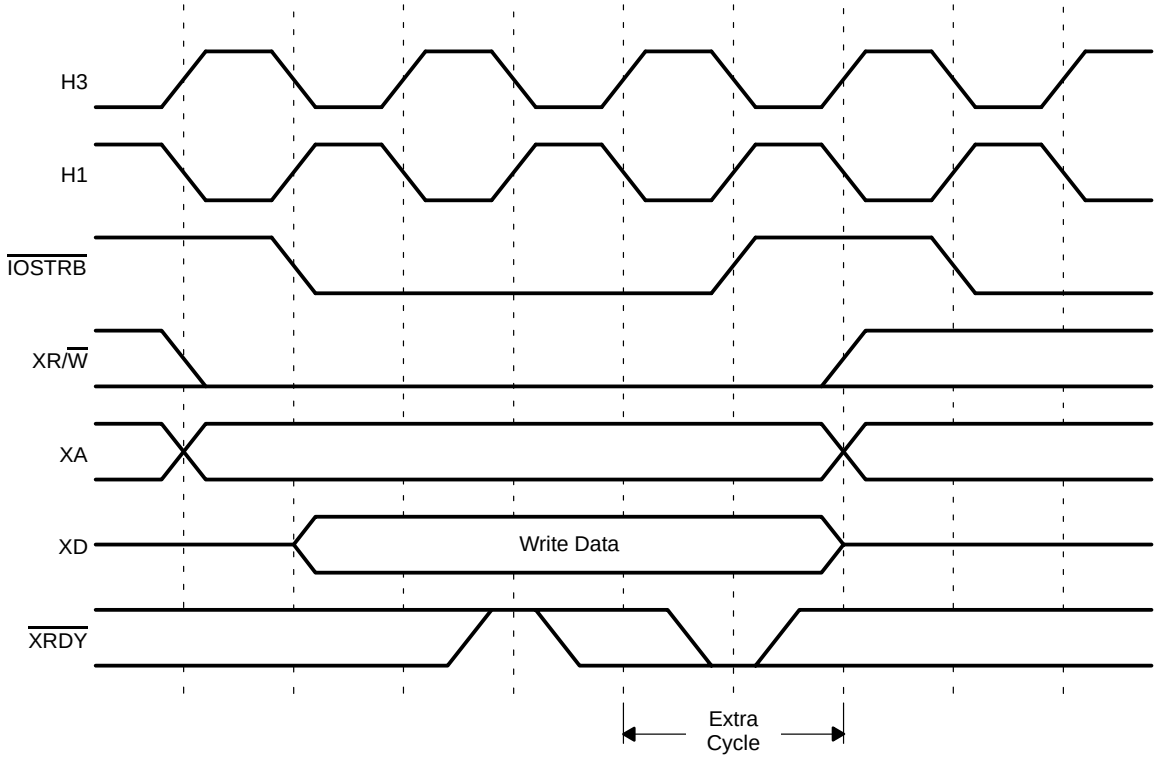


Figure 7–11, Figure 7–12, Figure 7–13, Figure 7–14, Figure 7–15, Figure 7–16, Figure 7–17, Figure 7–18, Figure 7–19, Figure 7–20, and Figure 7–21 illustrate the various transitions between memory reads and writes, and I/O writes over the expansion bus.

Figure 7–11. Memory Read and I/O Write for Expansion Bus

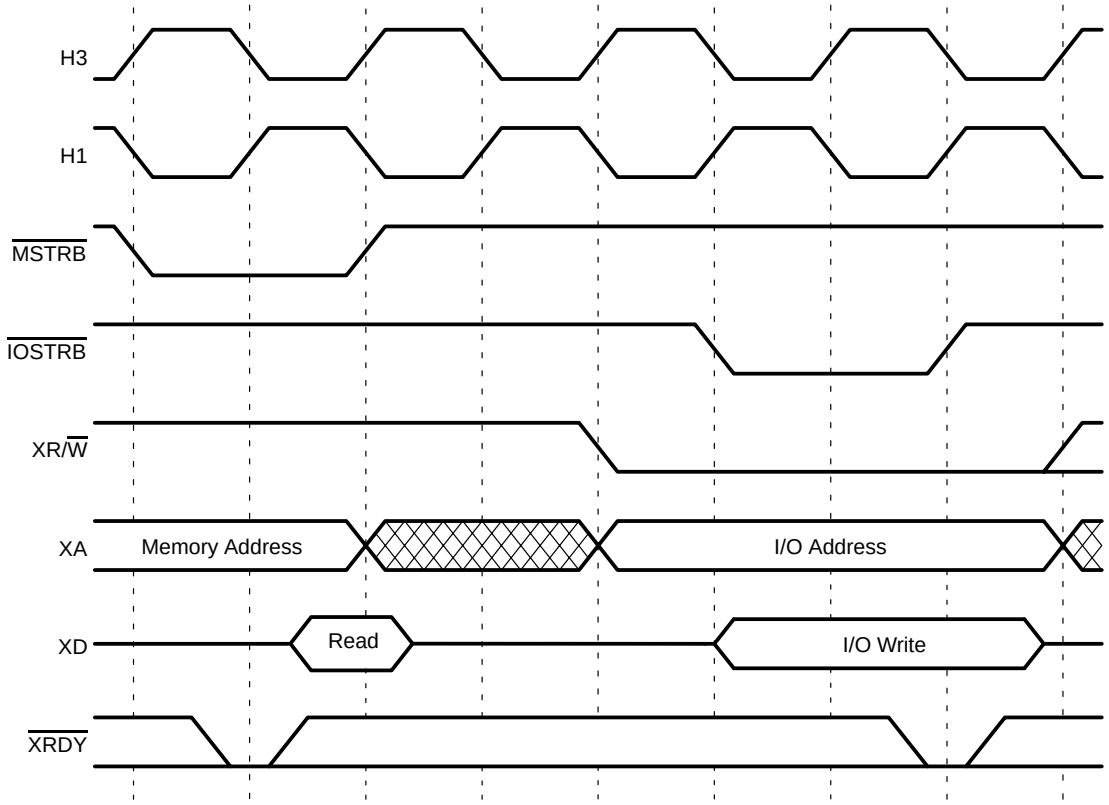


Figure 7–12. Memory Read and I/O Read for Expansion Bus

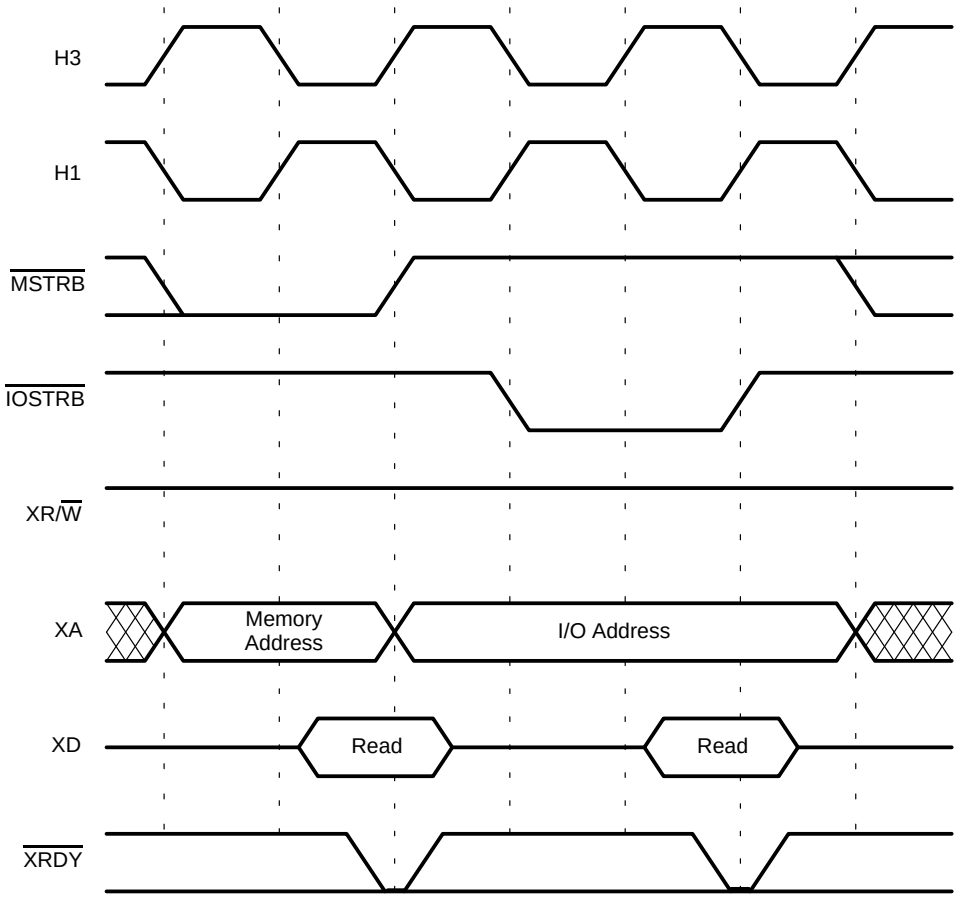


Figure 7–13. Memory Write and I/O Write for Expansion Bus

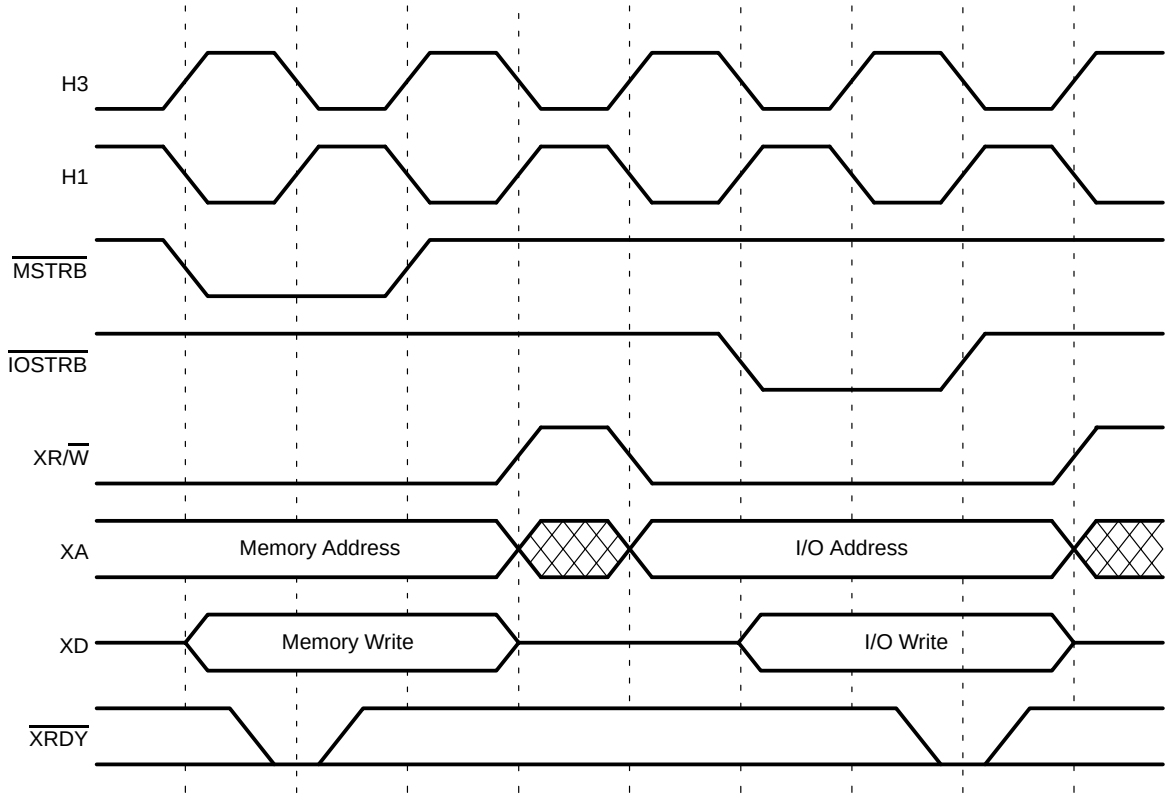


Figure 7–14. Memory Write and I/O Read for Expansion Bus

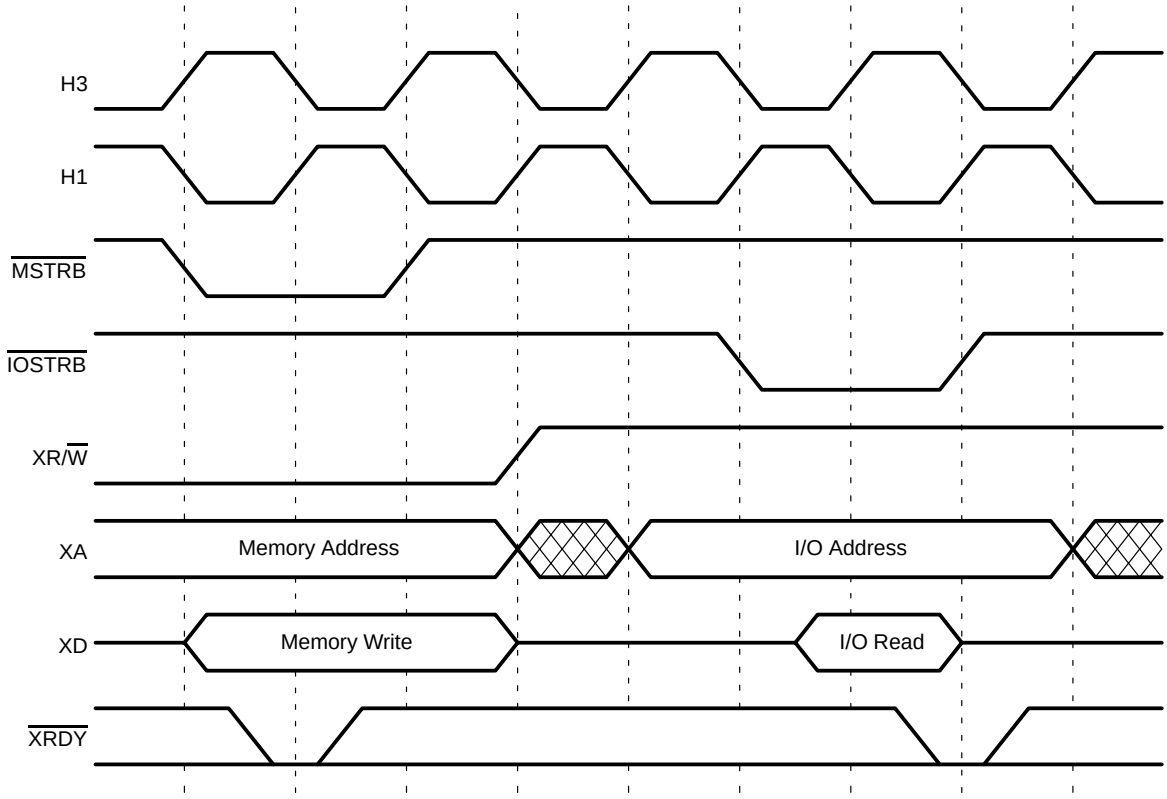


Figure 7–15. I/O Write and Memory Write for Expansion Bus

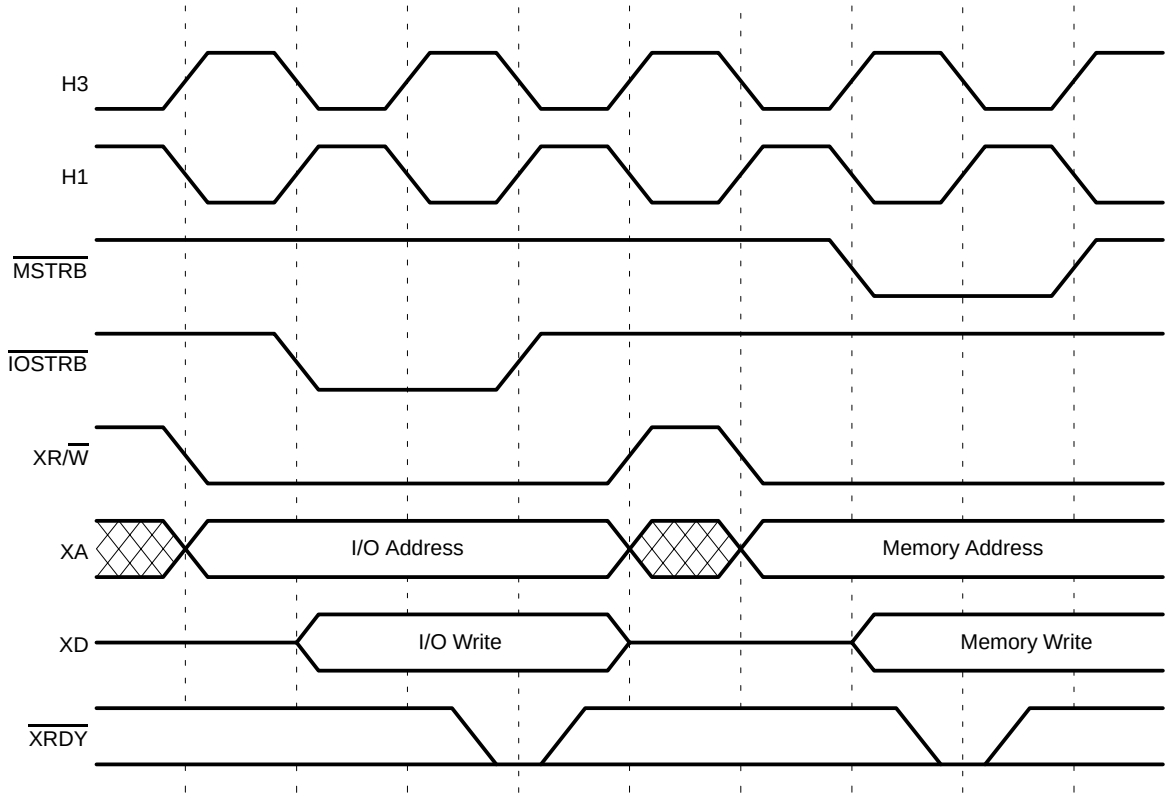


Figure 7–16. I/O Write and Memory Read for Expansion Bus

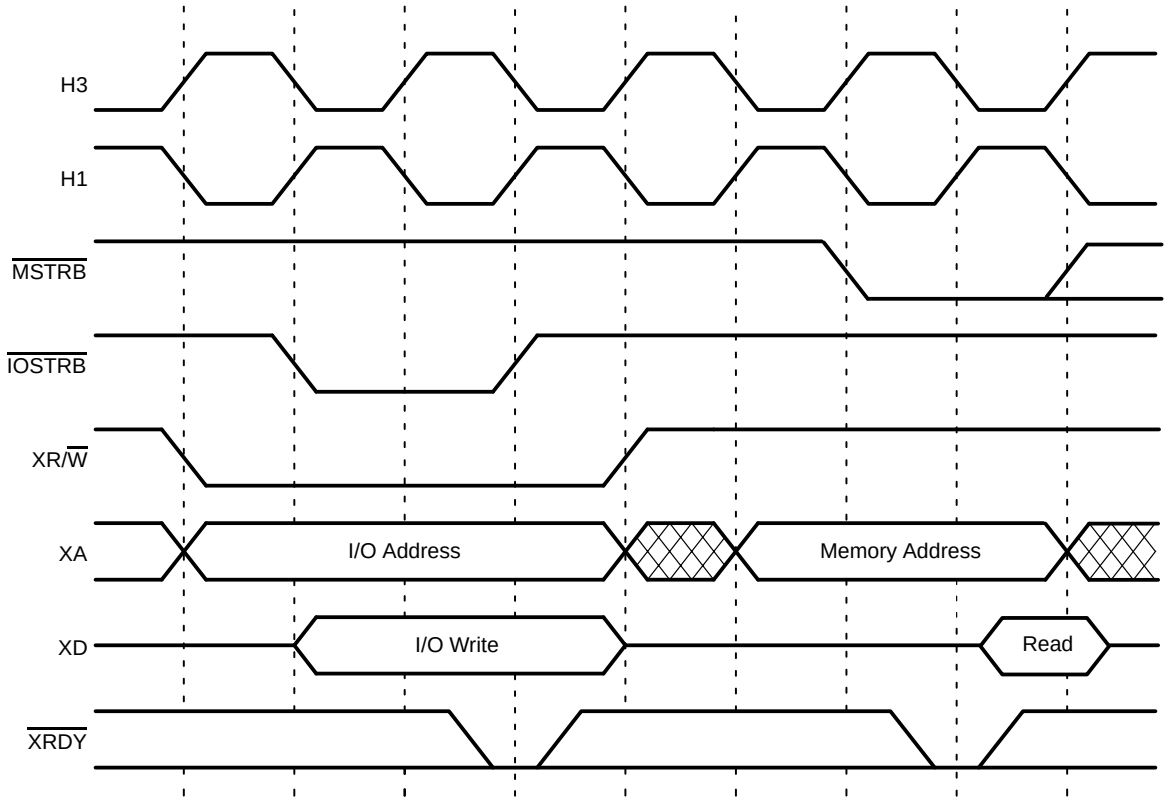


Figure 7–17. I/O Read and Memory Write for Expansion Bus

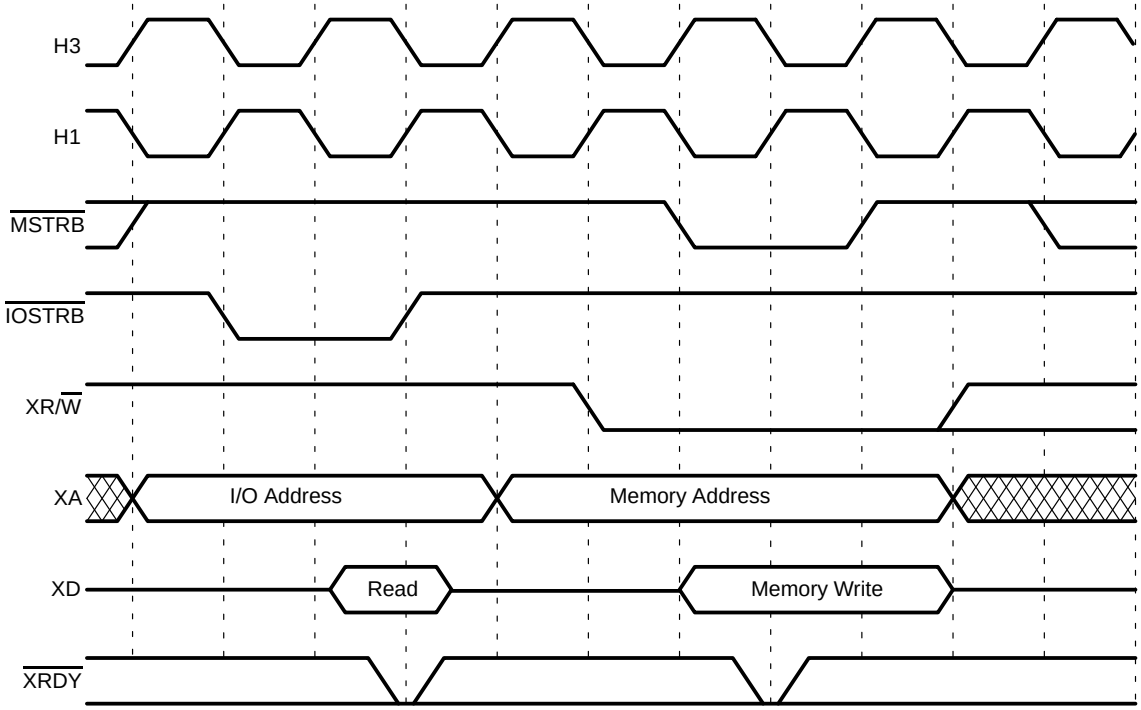


Figure 7–18. I/O Read and Memory Read for Expansion Bus

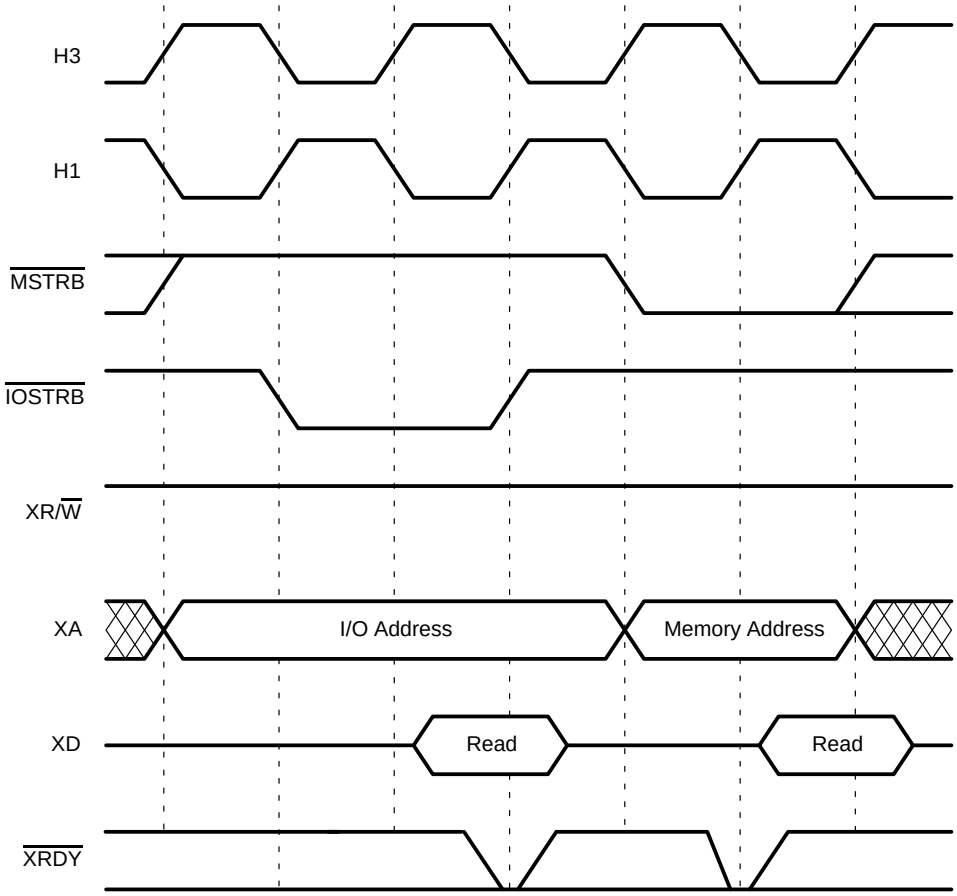


Figure 7–19. I/O Write and I/O Read for Expansion Bus

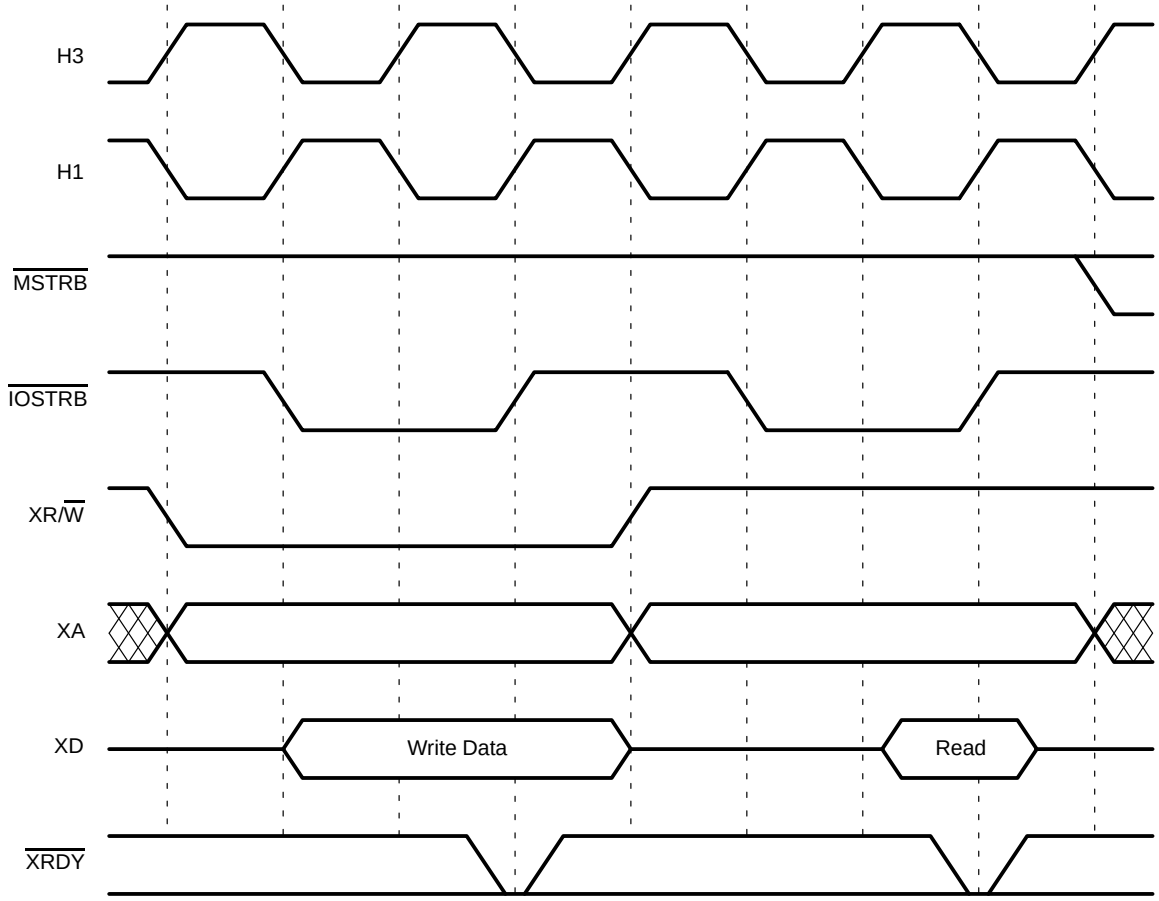


Figure 7–20. I/O Write and I/O Write for Expansion Bus

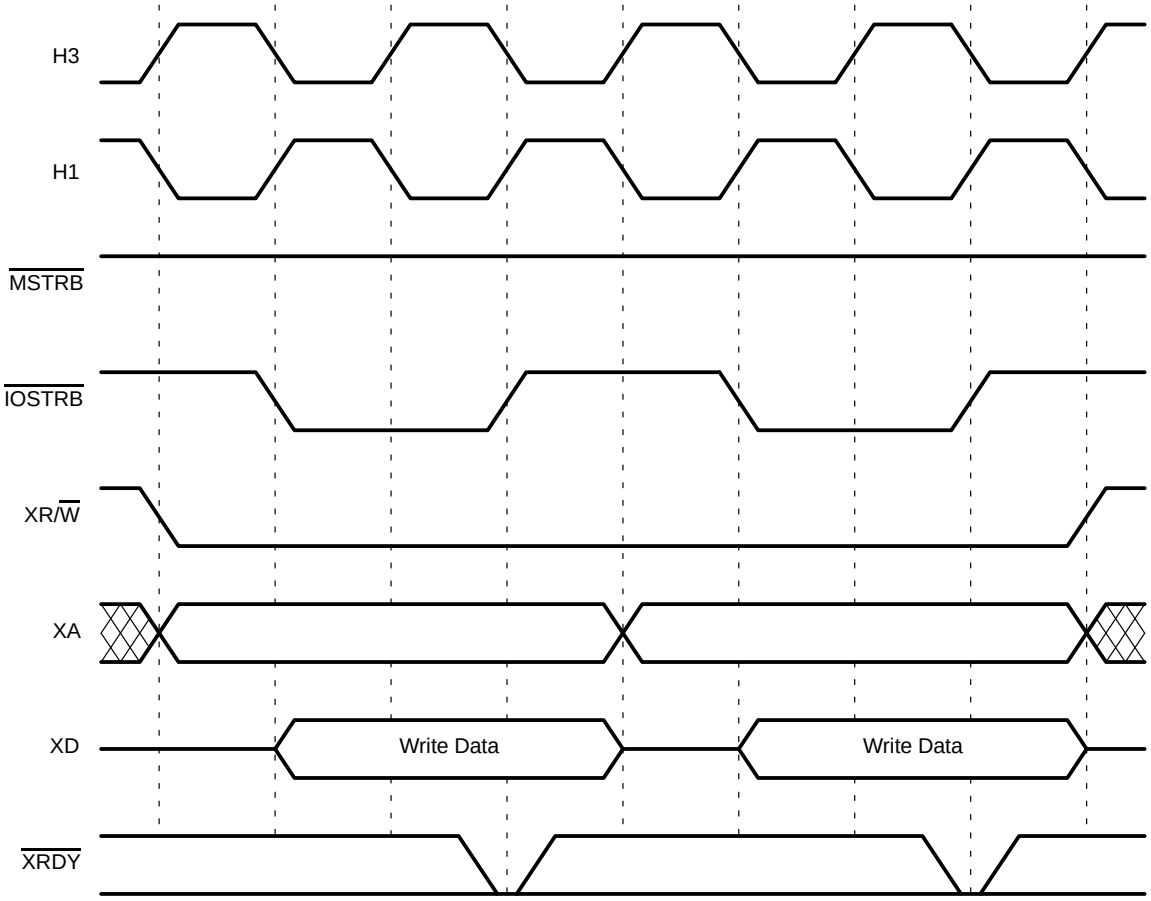


Figure 7–21. I/O Read and I/O Read for Expansion Bus

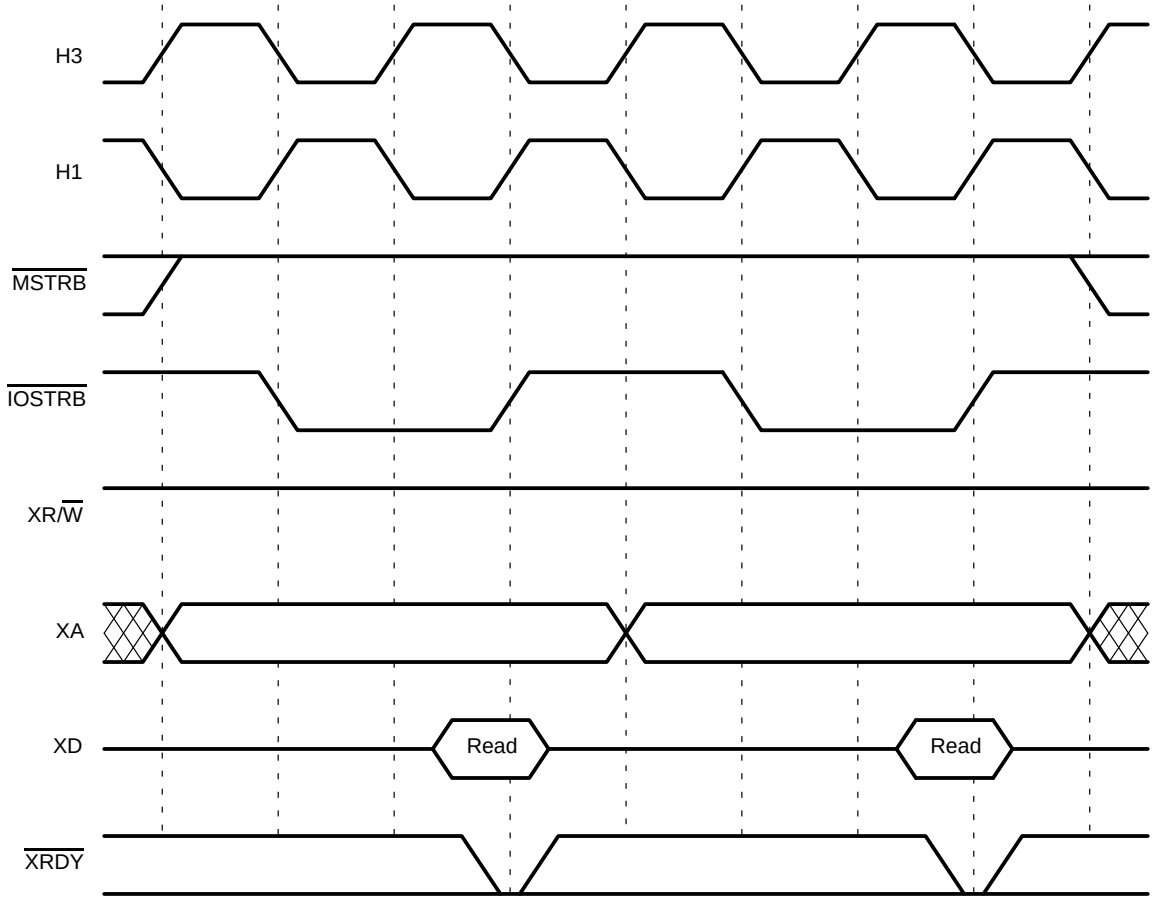


Figure 7–22 and Figure 7–23 illustrate the signal states when a bus is inactive (after an $\overline{\text{IOSTRB}}$ or (M)$\overline{\text{STRB}}$ access, respectively). The strobes ($\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$) and (X)R/$\overline{\text{W}}$) go to 1. The address is undefined, and the ready signal ($\overline{\text{XRDY}}$ or $\overline{\text{RDY}}$) is ignored.

Figure 7–22. Inactive Bus States for $\overline{\text{IOSTRB}}$

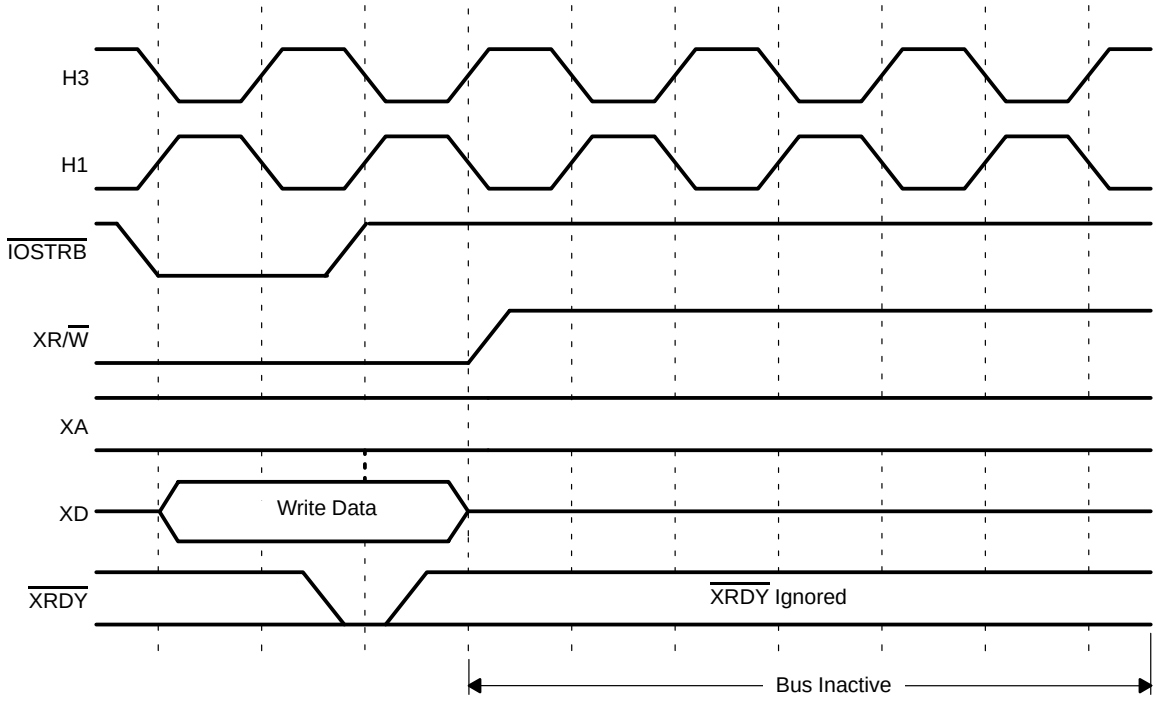


Figure 7–23. Inactive Bus States for $\overline{STRB}$ and $\overline{MSTRB}$

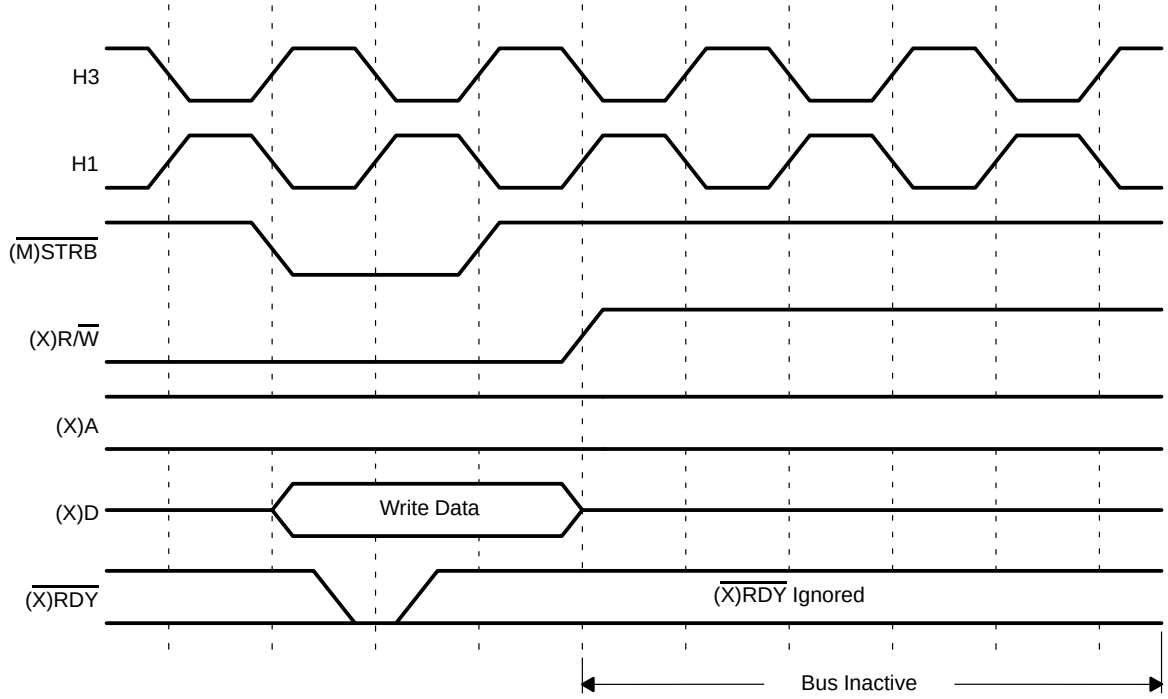
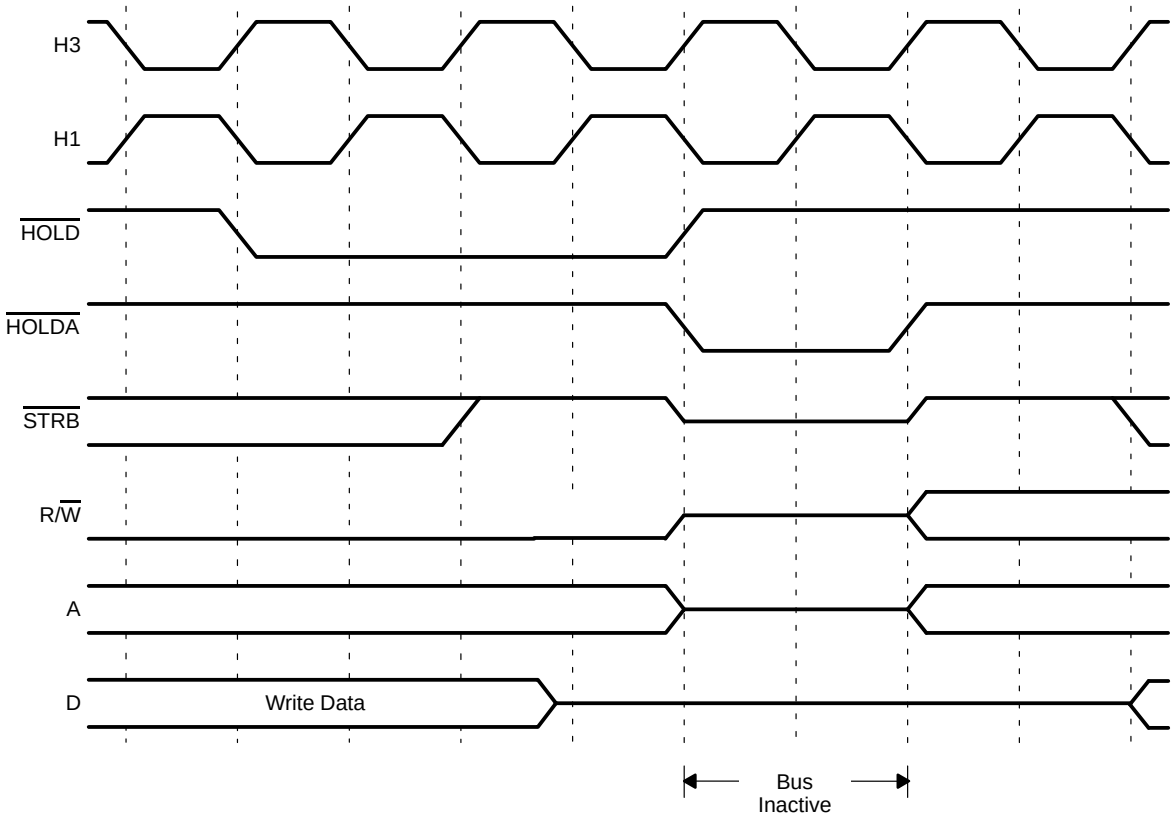


Figure 7–24 illustrates the timing for $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$. $\overline{\text{HOLD}}$ is an external asynchronous input. There is a minimum of one cycle delay from the time when the processor recognizes $\overline{\text{HOLD}}$ = 0 until $\overline{\text{HOLDA}}$ = 0. 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 complete before a hold is acknowledged.

Figure 7–24. $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ Timing



7.3 Programmable Wait States

You can control wait-state generation by manipulating memory-mapped control registers associated with both the primary and expansion interfaces. Use the WTCNT field to load an internal timer, and use the SWW field to select one of the following four modes of wait-state generation:

- External $\overline{RDY}$
- WTCNT-generated $\overline{RDY}_{wtcnt}$
- Logical-AND of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$
- Logical-OR of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$

The four modes are used to generate the internal ready signal, $\overline{RDY}_{int}$, that controls accesses. As long as $\overline{RDY}_{int}$ = 1, the current external access is delayed. When $\overline{RDY}_{int}$ = 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.

$\overline{RDY}_{wtcnt}$ 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, $\overline{RDY}_{wtcnt}$ = 1. While the counter is 0, $\overline{RDY}_{wtcnt}$ = 0.

When SWW = 0 0, $\overline{RDY}_{int}$ depends only on $\overline{RDY}$. $\overline{RDY}_{wtcnt}$ is ignored. Table 7–3 is the truth table for this mode.

Table 7–3. Wait-State Generation When SWW = 0 0

RDY	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	0
1	0	1
1	1	1

When SWW = 0 1, $\overline{RDY}_{int}$ depends only on $\overline{RDY}_{wtcnt}$. $\overline{RDY}$ is ignored. Table 7–4 is the truth table for this mode.

Table 7–4. Wait-State Generation When SWW = 0 1

RDY	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	1
1	0	0
1	1	1

When SWW = 1 0, $\overline{RDY}_{int}$ is the logical-OR (electrical-AND, since these signals are low true) of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$ (see Table 7–5).

Table 7–5. Wait-State Generation When SWW = 1 0

RDY	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	0
1	0	0
1	1	1

When SWW = 1 1, $\overline{RDY}_{int}$ is the logical-AND (electrical-OR, since these signals are low true) of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$. The truth table for this mode is Table 7–6.

Table 7–6. Wait-State Generation When SWW = 1 1

RDY	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	1
1	0	1
1	1	1

7.4 Programmable Bank Switching

Programmable bank switching allows you to switch between external memory banks without externally inserting wait states due to memories that require several cycles to turn off. Bank switching is implemented on the primary bus and not on the expansion bus.

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 (see Table 7–1 on page 7-4). For example (see Figure 7–25), if BNKCMP = 16, the 16 MSBs of the address are used to define a bank. 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 7–7 summarizes the relationship between BNKCMP, the address bits used to define a bank, and the resulting bank size.

Figure 7–25. BNKCMP Example

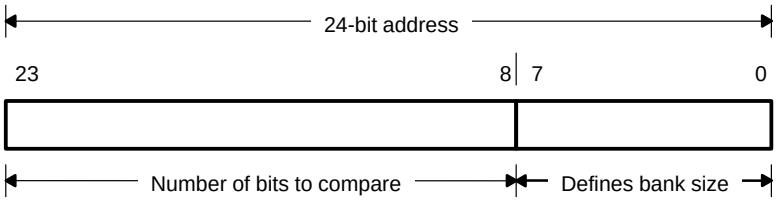


Table 7–7. BNKCMP and Bank Size

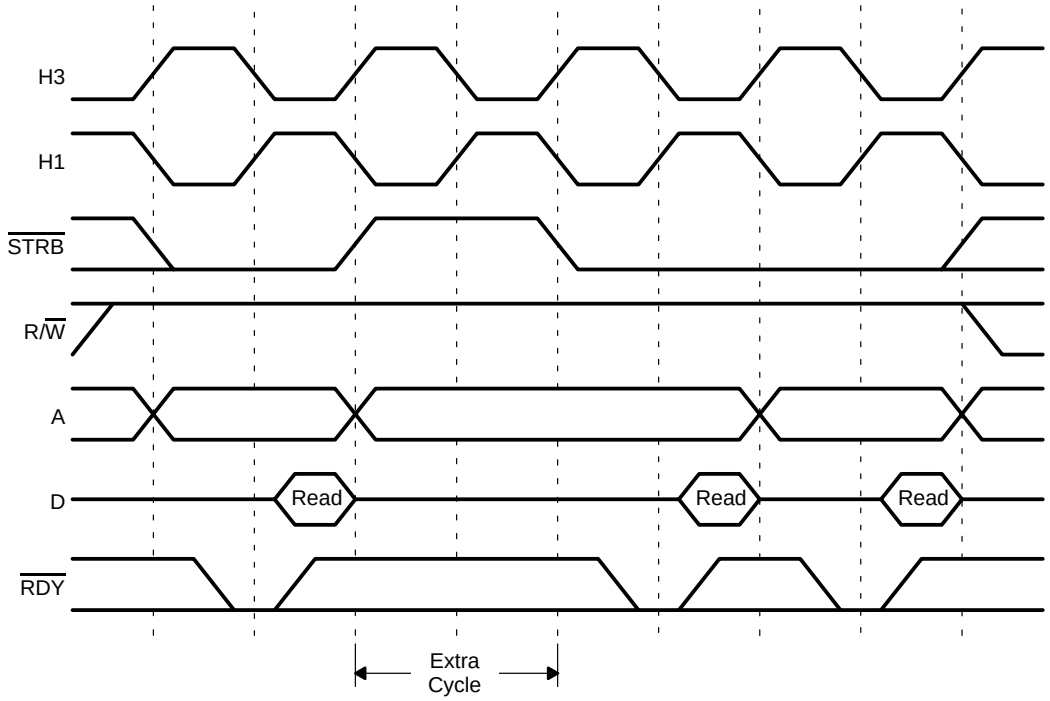
BNKCMP	MSBs Defining a Bank	Bank Size (32-Bit Words)
00000	None	2^{24}= 16M
00001	23	2^{23}= 8M
00010	23—22	2^{22}= 4M
00011	23—21	2^{21}= 2M
00100	23—20	2^{20}= 1M
00101	23—19	2^{19}= 512K
00110	23—18	2^{18}= 256K
00111	23—17	2^{17}= 128K
01000	23—16	2^{16}= 64K
01001	23—15	2^{15}= 32K
01010	23—14	2^{14}= 16K
01011	23—13	2^{13}= 8K
01100	23—22	2^{12}= 4K
01101	23—11	2^{11}= 2K
01110	23—12	2^{10}= 1K
01111	23—9	2^9 =512
10000	23—8	2^8 = 256
10000—11111	Reserved	Undefined

The TMS320C3x 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 7–26 illustrates the addition of an inactive cycle when switches between banks of memory occur.

Figure 7–26. Bank-Switching Example



Chapter 8

Peripherals

The TMS320C3x features two timers, two serial ports (one on the TMS320C31), and an on-chip direct memory access (DMA) controller. These peripheral modules are controlled through memory-mapped registers located on the dedicated peripheral bus.

The DMA controller is used to perform input/output operations without interfering with the operation of the CPU. Therefore, it is possible to interface the TMS320C3x 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.

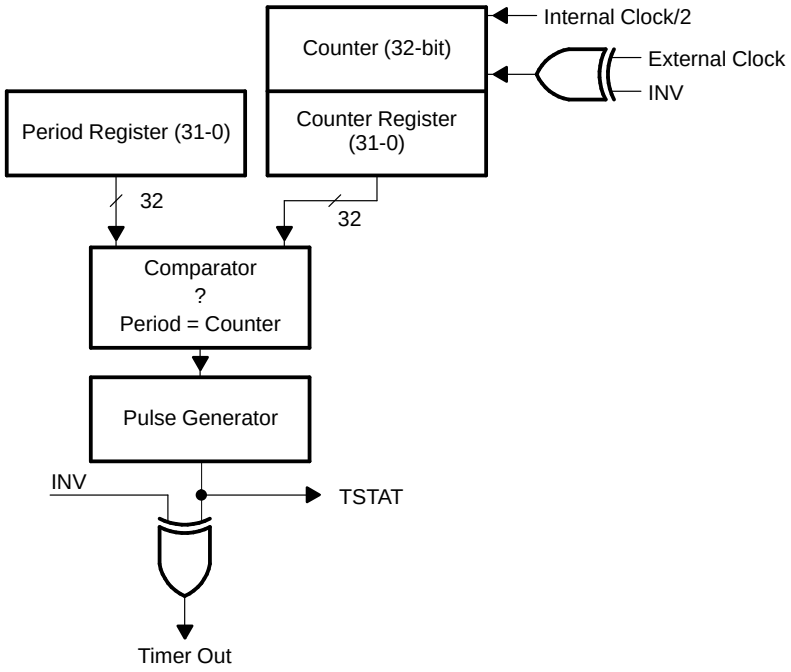
Major topics discussed in this chapter on peripherals are listed below.

Topic **Page**

8.1 Timers

The TMS320C3x timer modules are general-purpose, 32-bit, timer/event counters, with two signaling modes and internal or external clocking (see Figure 8–1). You can use the timer modules to signal to the TMS320C3x or the external world at specified intervals or to count external events. With an internal clock, you can use the timer to signal an external A/D converter to start a conversion, or it can interrupt the TMS320C3x 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, an output clock signal, or a general-purpose I/O pin.

Figure 8–1. Timer Block Diagram



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 two types of external clock signals: pulse or clock. The memory map for the timer modules is shown in Figure 8–2.

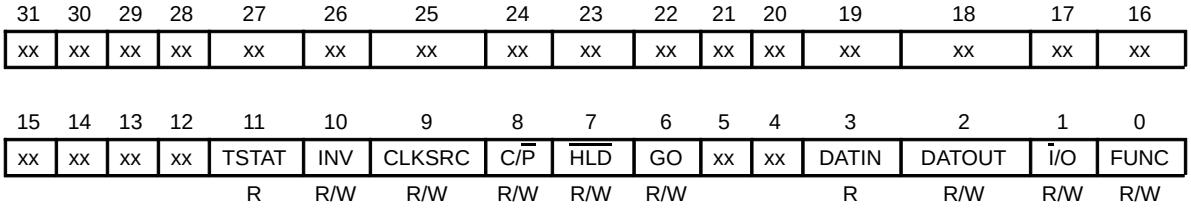
Figure 8–2. Memory-Mapped Timer Locations

Register	Peripheral Address	
	Timer 0	Timer 1
Timer Global Control (See Table 8–1)	808020h	808030h
Reserved	808021h	808031h
Reserved	808022h	808032h
Reserved	808023h	808033h
Timer Counter (See subsection 8.1.2)	808024h	808034h
Reserved	808025h	808035h
Reserved	808026h	808036h
Reserved	808027h	808037h
Timer Period (See subsection 8.1.2)	808028h	808038h
Reserved	808029h	808039h
Reserved	80802Ah	80803Ah
Reserved	80802Bh	80803Bh
Reserved	80802Ch	80803Ch
Reserved	80802Dh	80803Dh
Reserved	80802Eh	80803Eh
Reserved	80802Fh	80803Fh

8.1.1 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. Table 8–1 defines this register's bits, names, and functions. Bits 3–0 are the port control bits; bits 11–6 are the timer global control bits. Figure 8–3 shows the 32-bit register. Note that at reset, all bits are set to 0 except for DATIN (which is set to the value read on TCLK).

Figure 8–3. Timer Global-Control Register



R = Read, *W* = Write, *xx* = reserved bit, read as 0

Table 8–1. Timer Global-Control Register Bits Summary

Bits	Name	Reset Value	Function
0	FUNC	0	FUNC 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 Figure 8–4 for a description of the relationship between FUNC and CLKSRC).
1	$\overline{\text{I/O}}$	0	If FUNC = 0 and CLKSRC = 0, TCLK is configured as a general-purpose I/O pin. In this case, 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.
2	DATOUT	0	DATOUT drives TCLK when the TMS320C3x is in I/O port mode. You can use DATOUT as an input to the timer.
3	DATIN	x †	Data input on TCLK or DATOUT. A write has no effect.
5–4	Reserved	0–0	Read as 0.
6	GO	0	The GO bit 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.
7	$\overline{\text{HLD}}$	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-two 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 $\overline{\text{HLD}}$. Table 8–2 shows the effect of writing to GO and $\overline{\text{HLD}}$.
8	C/$\overline{\text{P}}$	0	Clock/Pulse mode control. When C/$\overline{\text{P}}$ = 1, clock mode is chosen, and the signaling of the TSTAT $\overline{\text{flag}}$ and external output will have a 50 percent duty cycle. When C/$\overline{\text{P}}$ = 0, the status flag and external output will be active for one H1 cycle during each timer period (see Figure 8–5 on page 8-7).

† x = 0 or 1

Table 8–1. Timer Global-Control Register Bits Summary (Continued)

Bits	Name	Reset Value	Function
9	CLKSRC	0	Specifies the source of the timer clock. When CLKSRC = 1, an internal clock with 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 will be less than f(H1)/2. (See Figure 8–4 for a description of the relationship between FUNC and CLKSRC).
10	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 8–1). 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.
11	TSTAT	0	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.
31–12	Reserved	0–0	Read as 0.

† x = 0 or 1

Figure 8–4. Timer Modes as Defined by CLKSRC and FUNC

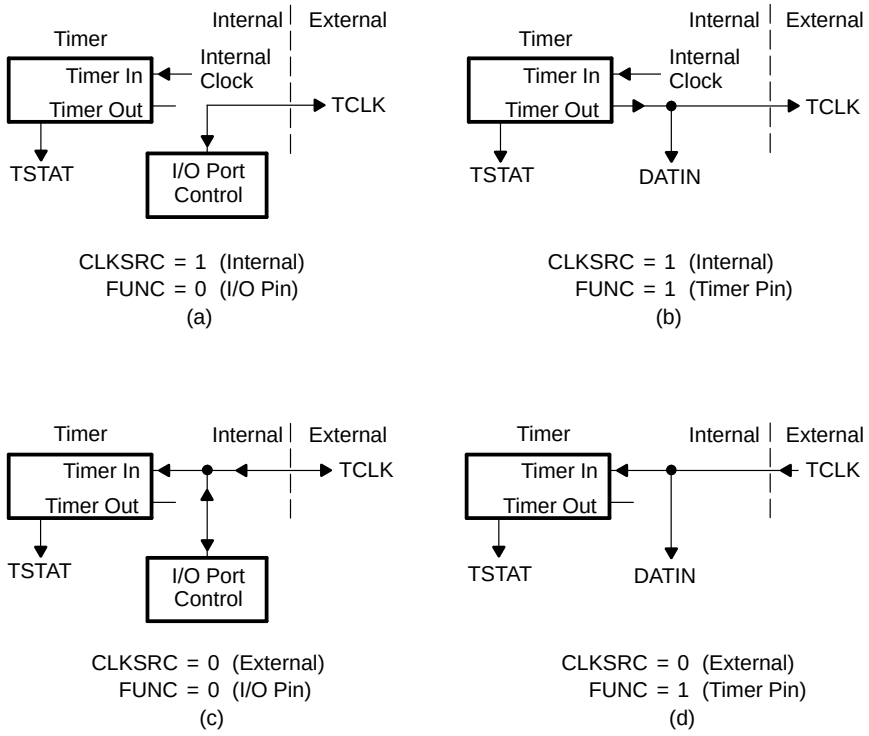
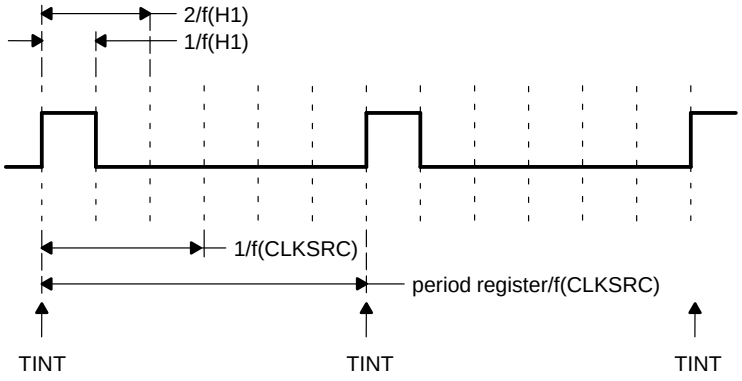
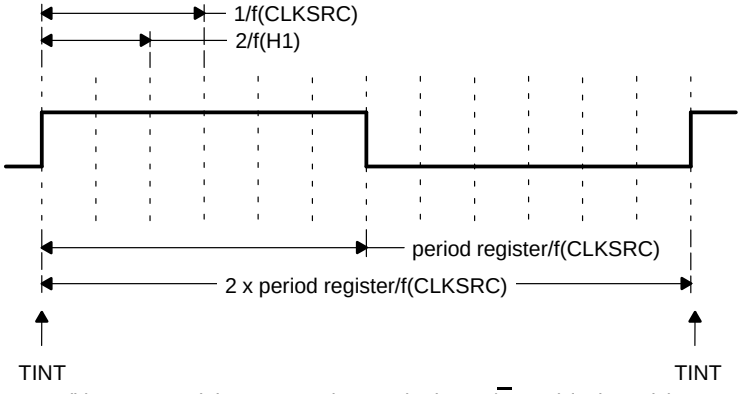


Figure 8–5. Timer Timing



(a) TSTAT and timer output (INV = 0) when C/$\overline{P}$ = 0 (pulse mode)



(b) TSTAT and timer output (INV = 0) when C/$\overline{P}$ = 1 (clock mode)

The rate of 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, refer to Section 8.1.2.

Table 8–2 shows the result of a write using specified values of the GO and $\overline{HLD}$ bits in the global control register.

Table 8–2. Result of a Write of Specified Values of GO and $\overline{HLD}$

GO	$\overline{HLD}$	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.

8.1.2 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 will count, roll over to 0, and then behave as described above.
- When the counter register is set to a value greater than the period register, the counter may overflow when being incremented. Once the counter reaches its maximum 32-bit value (0FFFFFFFFh), it simply clocks over to 0 and continues.

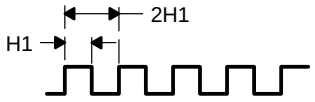
Writes from the peripheral bus override register updates from the counter and new status updates to the control register.

8.1.3 Timer Pulse Generation

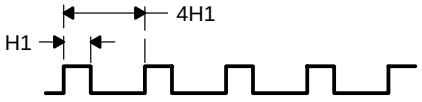
The timer pulse generator (see Figure 8–1 on page 8-2) can generate several external signals. You can invert these signals with the INV bit. The two basic modes are pulse mode and clock mode, as shown in Figure 8–5 on page 8-7. In both modes, an internal clock source f (timer clock) has a frequency of f(H1)/2, and an externally generated clock source f (timer clock) can have a maximum frequency of f(H1)/2.6. Refer to timer timing in subsection 13.5.16 on page 13-66. In pulse mode (C/$\overline{P}$ = 0), the width of the pulse is 1/f(H1).

Figure 8–6 provides some examples of the TCLKx output when the period register is set to various values and clock or pulse mode is selected.

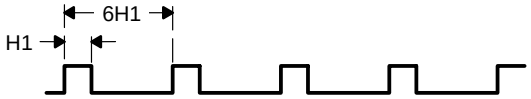
Figure 8–6. Timer Output Generation Examples



(a) INV = 0, C/$\overline{P}$ = 0 (Pulse Mode)
Timer Period = 1
Also,
INV = 0, C/$\overline{P}$ = 1 (Clock Mode)
Timer Period = 0

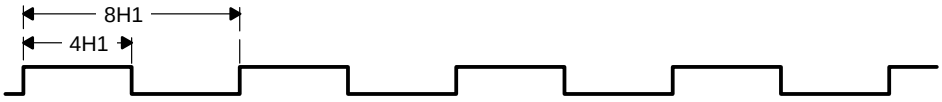


(b) INV = 0, C/$\overline{P}$ = 0 (Pulse Mode)
Timer Period = 2

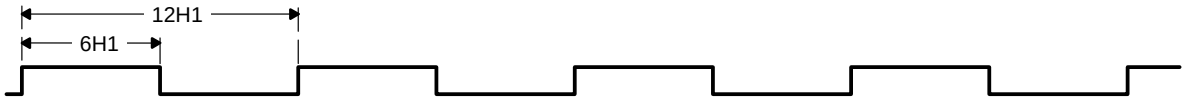


(c) INV = 0, C/$\overline{P}$ = 0 (Pulse Mode)
Timer Period = 3

(d) INV = 0, C/$\overline{P}$ = 1 (Clock Mode)
Timer Period = 1



(e) INV = 0, C/$\overline{P}$ = 1 (Clock Mode)
Timer Period = 2



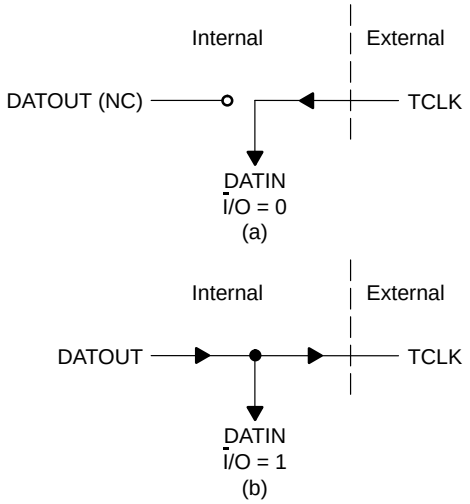
(f) INV = 0, C/$\overline{P}$ = 1 (Clock Mode)
Timer Period = 3

8.1.4 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 $\overline{I}$/O. The four timer modes of operation are defined as follows:

- 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 this mode, TCLK is connected to the I/O port control, and you use TCLK as a general-purpose I/O pin (see Figure 8–7). If $\overline{I}$/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. If $\overline{I}$/O = 1, TCLK is configured as a general-purpose output pin. DATOUT is placed on TCLK and can be read in DATIN.

Figure 8–7. Timer I/O Port Configurations



- If CLKSRC = 1 and FUNC = 1, 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.
- If CLKSRC = 0 and FUNC = 0, the timer is driven according to the status of the $\overline{I}$/O bit. If $\overline{I}$/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. If $\overline{I}$/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.
- If CLKSRC = 0 and FUNC = 1, 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 8–4 on page 8-6 shows the four timer modes of operation.

8.1.5 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:

$$f_{(interrupt)} = \frac{f_{(timer\ clock)}}{period\ register}, \text{ where}$$

$f_{(interrupt)}$ = timer frequency
$f_{(timer\ clock)}$ = interrupt frequency

- In clock mode, the interrupt frequency is determined by the following equation:

$$f_{(interrupt)} = \frac{f_{(timer\ clock)}}{2 \times period\ register}, \text{ where}$$

$f_{(interrupt)}$ = timer frequency
$f_{(timer\ clock)}$ = interrupt 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 subsection 3.1.8 on page 3-7 for more information on the CPU/DMA interrupt enable register.

When a timer interrupt occurs, a change in the state of the corresponding TCLK pin will be 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/$\overline{P}$ bit.

8.1.6 Timer Initialization/Reconfiguration

The timers are controlled through memory-mapped registers located on the dedicated peripheral bus. Following is the general procedure for initializing and/or reconfiguring the timers:

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 via 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.

8.2 Serial Ports

The TMS320C30 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 TMS320C31. 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, via the serial port timer and period registers, or externally, via 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 8–8 on page 8-14, and the memory map of the serial ports is shown in Figure 8–9 on page 8-15.

Figure 8–8. Serial-Port Block Diagram

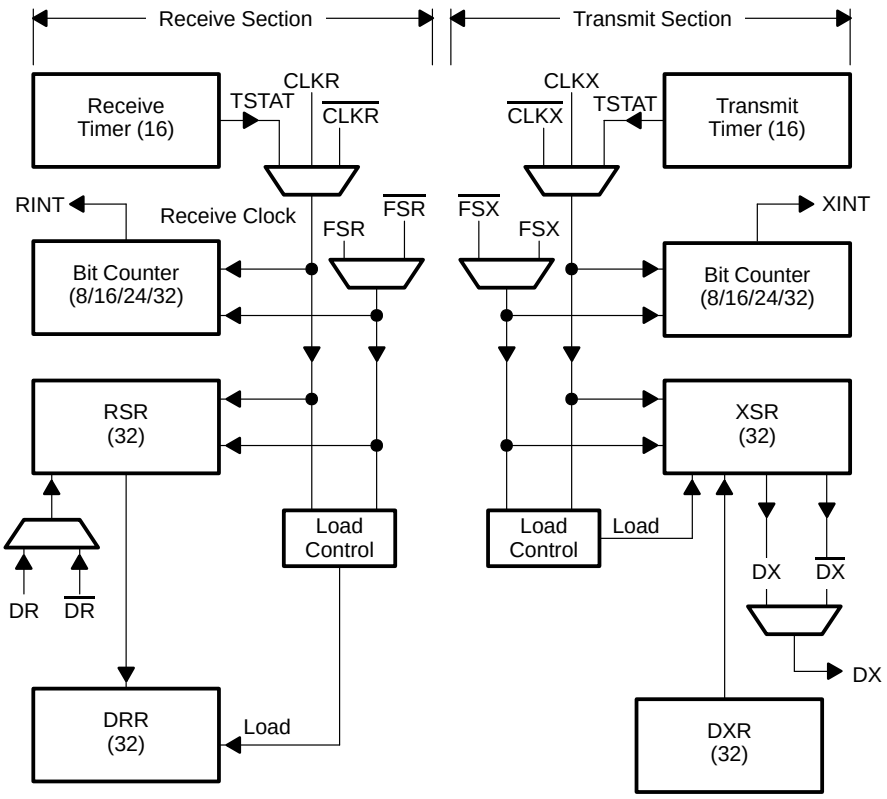


Figure 8–9. Memory-Mapped Locations for the Serial Ports

Register	Peripheral Address	
	Serial Port 0	Serial Port 1†
Serial-Port Global Control (See Figure 8–10)	808040h	808050h
Reserved	808041h	808051h
FSX/DX/CLKX Port Control (See Figure 8–11)	808042h	808052h
FSR/DR/CLKR Port Control (See Figure 8–12)	808043h	808053h
R/X Timer Control (See Figure 8–13)	808044h	808054h
R/X Timer Counter (See Figure 8–14)	808045h	808055h
R/X Timer Period (See Figure 8–15)	808046h	808056h
Reserved	808047h	808057h
Data Transmit (See Figure 8–16)	808048h	808058h
Reserved	808049h	808059h
Reserved	80804Ah	80805Ah
Reserved	80804Bh	80805Bh
Data Receive (See Figure 8–17)	80804Ch	80805Ch
Reserved	80804Dh	80805Dh
Reserved	80804Eh	80805Eh
Reserved	80804Fh	80805Fh

† Reserved locations on the TMS320C31

8.2.1 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. Table 8–3 defines the register bits, bit names, and bit functions. The register is shown in Figure 8–10.

Table 8–3. Serial-Port Global-Control Register Bits Summary

Bit	Name	Reset Value	Function
0	RRDY	0	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 at reset, the receive buffer does not have new data since the last read. RRDY = 0 at reset and after the receive buffer is read.
1	XRDY	1	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. XRDY = 1 at reset.
2	FSXOUT	0	This bit configures the FSX pin as an input (FSXOUT = 0) or an output (FSXOUT = 1).

Table 8–3. Serial-Port Global-Control Register Bits Summary (Continued)

Bit	Name	Reset Value	Function
3	XSREMPTY	0	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.
4	RSRFULL	0	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 will dominate, and RSRFULL is set to 1. If RSRFULL = 0, no overrun of the receiver has occurred.
5	HS	0	If HS = 1, the handshake mode is enabled. If HS = 0, the handshake mode is disabled.
6	XCLKSRCE	0	If XCLKSRCE = 1, the internal transmit clock is used. If XCLKSRCE = 0, the external transmit clock is used.
7	RCLKSRCE	0	If RCLKSRCE = 1, the internal receive clock is used. If RCLKSRCE = 0, the external receive clock is used.
8	XVAREN	0	This bit specifies fixed (XVAREN = 0) or variable (XVAREN = 1) data rate signaling 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.
9	RVAREN	0	This bit specifies fixed (RVAREN = 0) or variable (RVAREN = 1) data rate signaling when receiving. With a fixed data rate, FSR is active for at least one RCLK cycle and then goes inactive before the reception begins. With variable data rate, FSR is active while all bits are being received.
10	XFSM	0	Transmit frame sync mode. Configures the port for continuous mode operation(XFSM = 1) or standard mode (XFSM = 0). In continuous mode, only the first word of a block generates a sync pulse, and the rest are simply transmitted continuously to the end of the block. In standard mode, each word has an associated sync pulse.
11	RFSM	0	Receive frame sync mode. Configures the port for continuous mode (RFSM =1) or standard mode (RFSM = 0) operation. In continuous mode, only the first word of a block generates a sync pulse, and the rest are simply received continuously without expectation of another sync pulse. In standard mode, each word received has an associated sync pulse.
12	CLKXP	0	CLKX polarity. If CLKXP = 0, CLKX is active high. If CLKXP = 1, CLKX is active low.

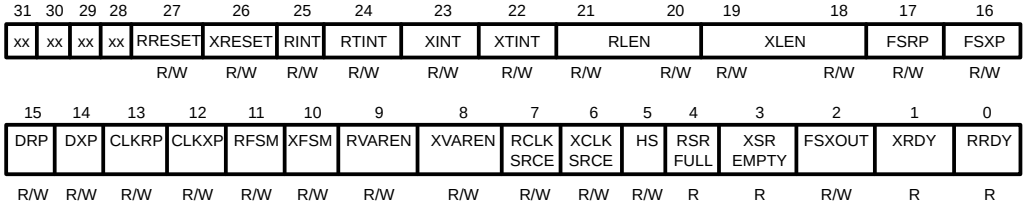
Table 8–3. Serial-Port Global-Control Register Bits Summary (Continued)

Bit	Name	Reset Value	Function
13	CLKRP	0	CLKR polarity. If CLKRP = 0, CLKR is active (high). If CLKRP =1, CLKR is active (low).
14	DXP	0	DX polarity. If DXP = 0, DX is active (high). If DXP = 1, DX is active (low).
15	DRP	0	DR polarity. If DRP = 0, DR is active (high). If DRP = 1, DR is active (low).
16	FSXP	0	FSX polarity. If FSXP = 0, FSX is active (high). If FSXP = 1, FSX is active (low).
17	FSRP	0	FSR polarity. If FSRP = 0, FSR is active (high). If FSRP = 1, FSR is active (low).
19–18	XLEN	00	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
21–20	RLEN	00	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
22	XTINT	0	Transmit timer interrupt enable. If XTINT = 0, the transmit timer interrupt is disabled. If XTINT = 1, the transmit timer interrupt is enabled.
23	XINT	0	Transmit interrupt enable. If XINT = 0, the transmit interrupt is disabled. If XINT= 1, the transmit interrupt is enabled. Note that 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.
24	RTINT	0	Receive timer interrupt enable. If RTINT = 0, the receive timer interrupt is disabled. If RTINT = 1, the receive timer interrupt is enabled.
25	RINT	0	Receive interrupt enable. If RINT = 0, the receive interrupt is disabled. If RINT= 1, the receive interrupt is enabled. Note that the CPU receive flag RINT and the serial-port-to-DMA interrupt (ERINT0 in the IE register) is the OR of the enabled receive timer interrupt and the enabled receive interrupt.
26	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. However, do not set XRESET to 1 until at least three cycles after XRESET 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.

Table 8–3. Serial-Port Global-Control Register Bits Summary (Concluded)

Bit	Name	Reset Value	Function
27	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. 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. RFSM can be toggled without resetting the global-control register.
31–28	Reserved	0–0	Read as 0.

Figure 8–10. Serial-Port Global-Control Register



R = Read, *W* = Write, *xx* = reserved bit, read as 0

8.2.2 FSX/DX/CLKX Port-Control Register

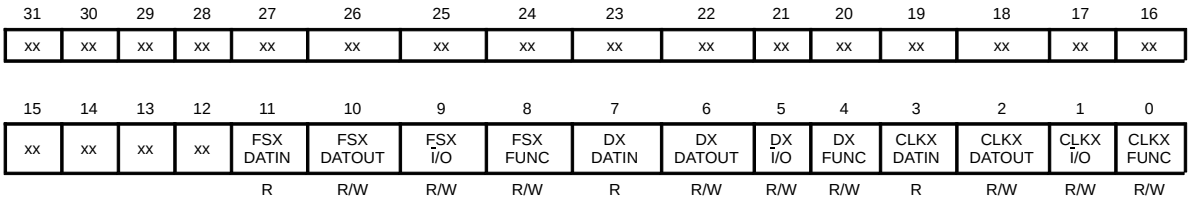
This 32-bit port control register controls the function of the serial port FSX, DX, and CLKX pins. At reset, all bits are set to 0. Table 8–4 defines the register bits, bit names, and functions. Figure 8–11 shows this port control register.

Table 8–4. FSX/DX/CLKX Port-Control Register Bits Summary

Bit	Name	Reset Value	Function
0	CLKXFUNC	0	CLKXFUNC controls the function of CLKX. If CLKXFUNC = 0, CLKX is configured as a general-purpose digital I/O port. If CLKXFUNC = 1, CLKX is a serial port pin.
1	CLKX $\overline{\text{I}}$/O	0	If CLKX $\overline{\text{I}}$/O = 0, CLKX is configured as a general-purpose input pin. If CLKX $\overline{\text{I}}$/O = 1, CLKX is configured as a general-purpose output pin.
2	CLKXDATOUT	0	Data output on CLKX.
3	CLKXDATIN	x	Data input on CLKX. A write has no effect.
4	DXFUNC	0	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 a serial port pin.
5	DX $\overline{\text{I}}$/O	0	If DX $\overline{\text{I}}$/O = 0, DX is configured as a general-purpose input pin. If DX $\overline{\text{I}}$/O = 1, DX is configured as a general-purpose output pin.
6	DXDATOUT	0	Data output on DX.
7	DXDATIN	x†	Data input on DX. A write has no effect.
8	FSXFUNC	0	FSXFUNC controls the function of FSX. If FSXFUNC = 0, FSX is configured as a general-purpose digital I/O port. If FSXFUNC = 1, FSX is a serial port pin.
9	FSX $\overline{\text{I}}$/O	0	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.
10	FSXDATOUT	0	Data output on FSX.
11	FSXDATIN	x†	Data input on FSX. A write has no effect.
31–12	Reserved	0–0	Read as 0.

† x = 0 or 1

Figure 8–11. FSX/DX/CLKX Port-Control Register



R = Read, *W* = Write, *xx* = reserved bit, read as 0

8.2.3 FSR/DR/CLKR Port-Control Register

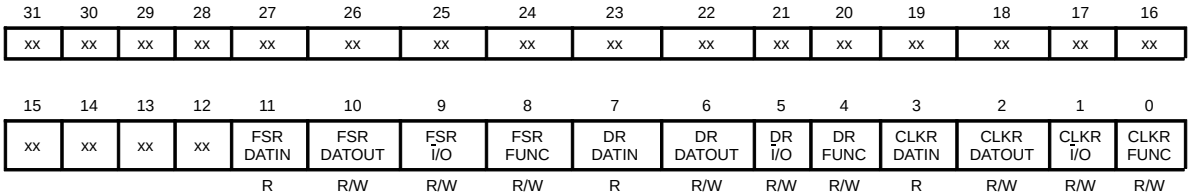
This 32-bit port control register is controlled by the function of the serial port FSR, DR, and CLKR pins. At reset, all bits are set to 0. Table 8–5 defines the register bits, the bit names, and functions. Figure 8–12 illustrates this port control register.

Table 8–5. FSR/DR/CLKR Port-Control Register Bits Summary

Bit	Name	Reset Value	Function
0	CLKRFUNC	0	CLKRFUNC controls the function of CLKR. If CLKRFUNC = 0, CLKR is configured as a general-purpose digital I/O port. If CLKRFUNC = 1, CLKR is a serial port pin.
1	CLKR$\overline{\text{I}}$/O	0	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.
2	CLKRDATOUT	0	Data output on CLKR.
3	CLKRDATIN	x	Data input on CLKR. A write has no effect.
4	DRFUNC	0	DRFUNC controls the function of DR. If DRFUNC = 0, DR is configured as a general-purpose digital I/O port. If DRFUNC = 1, DR is a serial port pin.
5	DR $\overline{\text{I}}$/O	0	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.
6	DRDATOUT	0	Data output on DR
7	DRDATIN	x †	Data input on DR. A write has no effect.
8	FSRFUNC	0	FSRFUNC controls the function of FSR. If FSRFUNC = 0, FSR is configured as a general-purpose digital I/O port. If FSRFUNC = 1, FSR is a serial port pin.
9	FSR $\overline{\text{I}}$/O	0	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.
10	FSRDATOUT	0	Data output on FSR
11	FSRDATIN	x	Data input on FSR. A write has no effect.
31–12	Reserved	0–0	Read as 0.

† x = 0 or 1

Figure 8–12. FSR/DR/CLKR Port-Control Register



R = Read, *W* = Write, *xx* = reserved bit, read as 0

8.2.4 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. Table 8–6 lists the register bits, bit names, and functions. Bits 5–0 control the transmitter timer. Bits 11–6 control the receiver timer. Figure 8–13 shows the register. The serial port receive/transmit timer function is similar to timer module operation. It can be considered a 16-bit-wide timer. Refer to Section 8.1 on page 8-2 for more information on timers.

Table 8–6. Receive/Transmit Timer-Control Register

Bit	Name	Reset Value	Function
0	XGO	0	The XGO bit resets and starts the transmit timer counter. When XGO 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 XGO bit is cleared on the same rising edge. Writing 0 to XGO has no effect on the transmit timer.
1	$\overline{\text{XHLD}}$	0	Transmit counter hold signal. When this bit is set to 0, the counter is disabled and held in its current state. The internal divide-by-two counter is also held so that the counter will continue where it left off when $\overline{\text{XHLD}}$ is set to 1. You can read and modify the timer registers while the timer is being held. $\overline{\text{RESET}}$ has priority over $\overline{\text{XHLD}}$.
2	XC/$\overline{\text{P}}$	0	XClock/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.
3	XCLKSRC	0	This bit 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.
4	Reserved	0	Read as zero.
5	XTSTAT	0	This bit 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.
6	RGO	0	The RGO bit 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.
7	$\overline{\text{RHLD}}$	0	Receive counter hold signal. When this bit is set to 0, the counter is disabled and held in its current state. The internal divide-by-two counter is also held so that the counter will continue where it left off when $\overline{\text{RHLD}}$ is set to 1. You can read and modify the timer registers while the timer is being held. $\overline{\text{RESET}}$ has priority over $\overline{\text{RHLD}}$.

Table 8–6. Receive/Transmit Timer-Control Register (Concluded)

Bit	Name	Reset Value	Function
8	RC/$\overline{P}$	0	RClock/Pulse mode control. When RC/$\overline{P}$ = 1, the clock mode is chosen. The signaling of the status flag and external output has a 50 percent duty cycle. When RC/$\overline{P}$ = 0, the status flag and external output are active for one CLKOUT cycle during each timer period.
9	RCLKSRC	0	This bit 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, 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.
10	Reserved	0	Read as zero.
11	RTSTAT	0	This bit 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.
31—12	Reserved	0–0	Read as 0.

Figure 8–13. Receive/Transmit Timer-Control Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

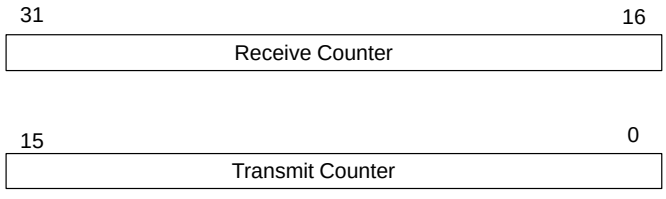
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	RTSTAT	xx	RCLKSRC	RC/$\overline{P}$	$\overline{RHLD}$	RGO	XTSTAT	xx	XCLKSRC	XC/$\overline{P}$	$\overline{XHLD}$	XGO
				R		R/W	R/W	R	R/W	R/W		R	R/W	R/W	R/W

R = Read, *W* = Write, *xx* = reserved bit, read as 0

8.2.5 Receive/Transmit Timer-Counter Register

The receive/transmit timer counter register is a 32-bit register (see Figure 8–14). 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 8.2.6). It is also set to 0 at reset.

Figure 8–14. Receive/Transmit Timer Counter Register

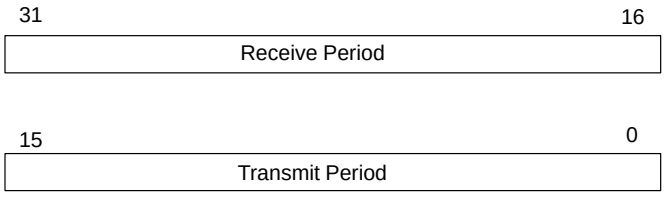


NOTE: All bits are read/write.

8.2.6 Receive/Transmit Timer-Period Register

The receive/transmit timer period register is a 32-bit register (see Figure 8–15). Bits 15–0 are the timer transmit period, and bits 31–16 are the receive period. Each register is used to specify the period of the timer. It is also cleared to 0 at reset.

Figure 8–15. Receive/Transmit Timer-Period Register

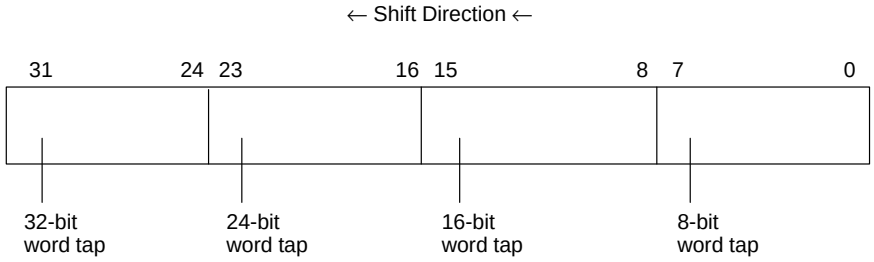


Note: All bits are read/write.

8.2.7 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 8–16. The shift is a left-shift (LSB to MSB) with the data shifted out of the MSB corresponding to the appropriate tap point.

Figure 8–16. Transmit Buffer Shift Operation

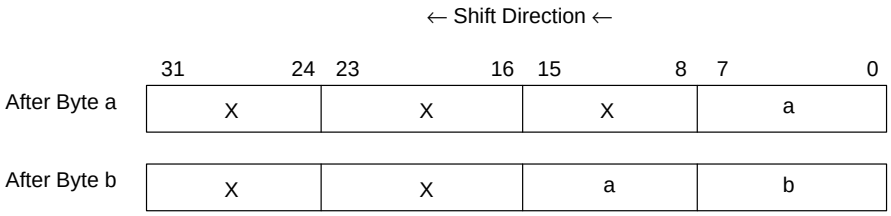


8.2.8 Data-Receive Register

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 will 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 8–17 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 8–17. Receive Buffer Shift Operation



8.2.9 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 8–18 shows serial port clocking in the I/O mode (CLKRFUNC = 0) when CLKX is either an input or an output. Figure 8–19 shows clocking in the serial-port mode (CLKRFUNC=1). Both figures use a transmit section for an example. The same relationship holds for a receive section.

Figure 8–18. Serial-Port Clocking in I/O Mode

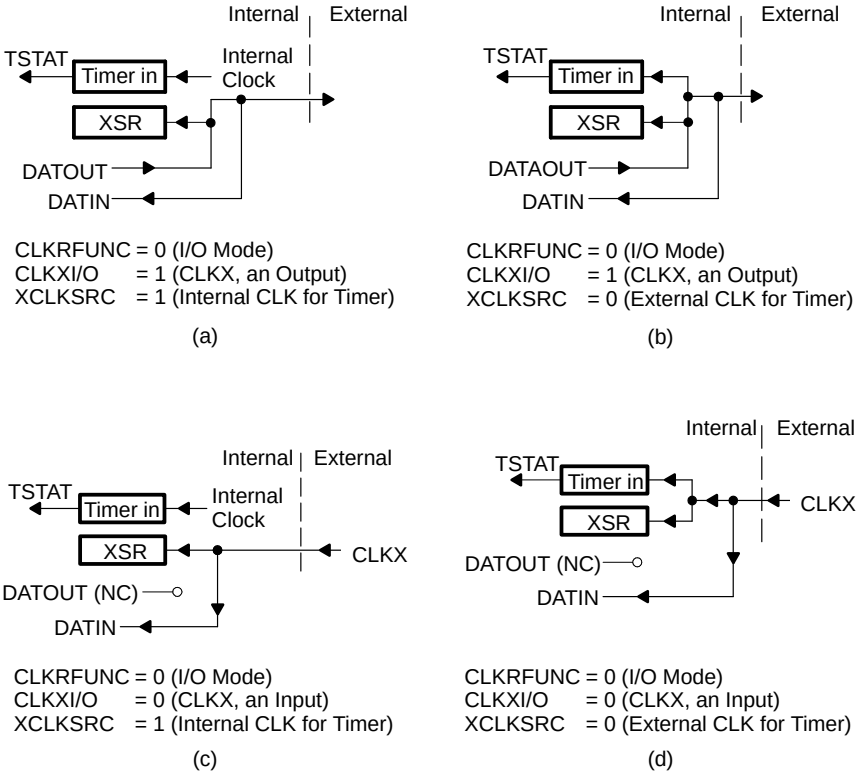
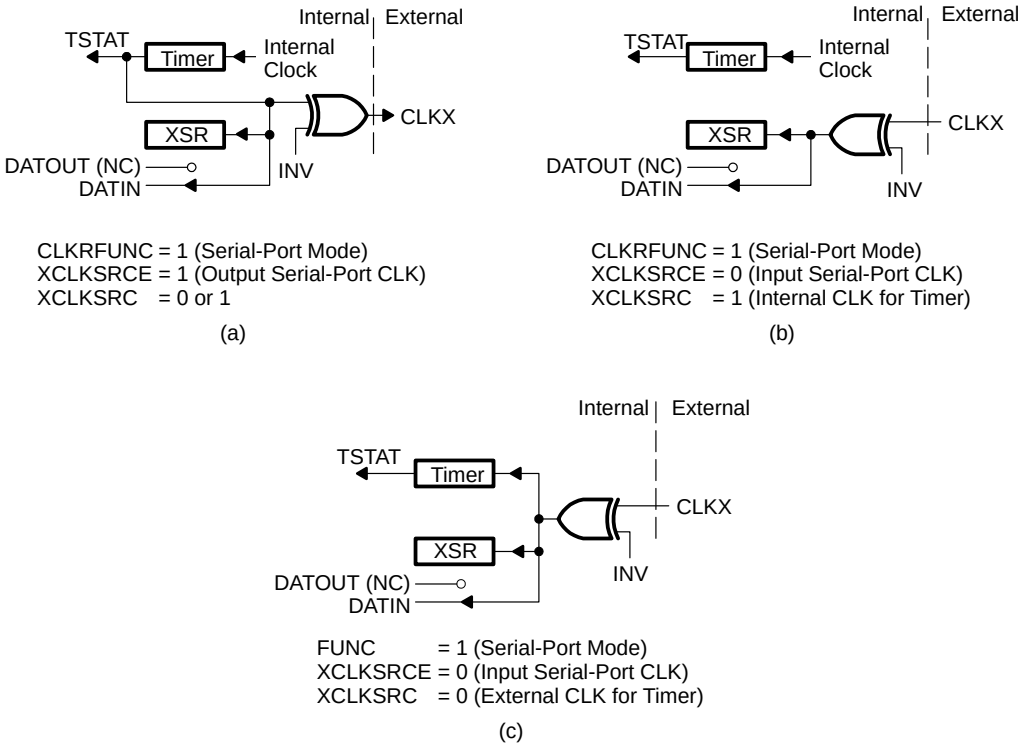


Figure 8–19. Serial-Port Clocking in Serial-Port Mode



8.2.10 Serial-Port Timing

The formula for calculating the frequency of the serial-port clock with an internally generated clock is dependent 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 serial port timing in Table 13–27 on page 13-57. Also, see subsection 8.1.3 on page 8-8 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 transmitted and loaded MSB first and right-justified. If fewer than 32 bits are transferred, the data are right-justified in the 32-bit transmit and receive buffers. Therefore, the LSBs of the transmit buffer are the bits that are transmitted.

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 solely by the internal state of the serial port. If a fixed data rate is specified, FSX goes active when DXR is loaded into XSR to be transmitted out. 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 should 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 LEN bits. In the variable data-rate mode, the transmitter begins sending from the time FSX goes active until the number of specified bits has 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 will change DXR, but a DXR-to-XSR transfer will not occur. XSR begins shifting when the external FSX is received, or when it is reset using XRESET.

Continuous Transmit and Receive Modes

When continuous mode is chosen, 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 continues to be 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.

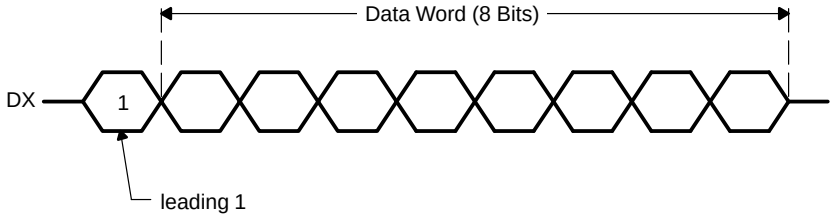
Similarly with FSR, the receiver continues shifting in new data and loading DRR. If the data-receive buffer is not read before the next word is shifted in, you will lose subsequent incoming data. You can use the RFSM bit to terminate the receive-continuous mode.

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 8–20). For example, if an eight-bit word is to be transmitted, the first bit sent is a 1, followed by the eight-bit data word.

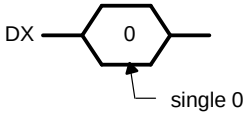
In this mode, once the serial port transmits a word, it will not transmit another word until it receives a separately transmitted zero bit. Therefore, the 1 bit that precedes every data word is, in effect, a request bit.

Figure 8–20. Data Word Format in Handshake Mode



After a serial port receives a word (with the leading 1) and that word has been read from the DRR, the receiving serial port sends a single 0 to the transmitting serial port. Thus, the single 0 bit acts as an acknowledge bit (see Figure 8–21). This single acknowledge bit is sent every time the DRR is read, even if the DRR does not contain new data.

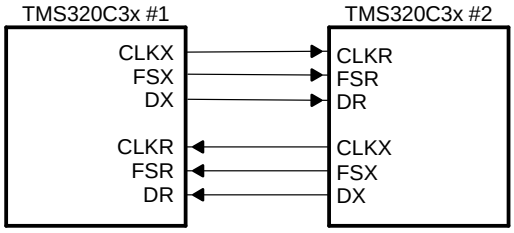
Figure 8–21. Single Zero Sent as an Acknowledge Bit



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 8–22 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 8–22. Direct Connection Using Handshake Mode



8.2.11 Serial-Port Interrupt Sources

A serial port has the following interrupt sources:

- **The 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.
- **The 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.
- **The 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.
- **The 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.

8.2.12 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. Therefore, 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.

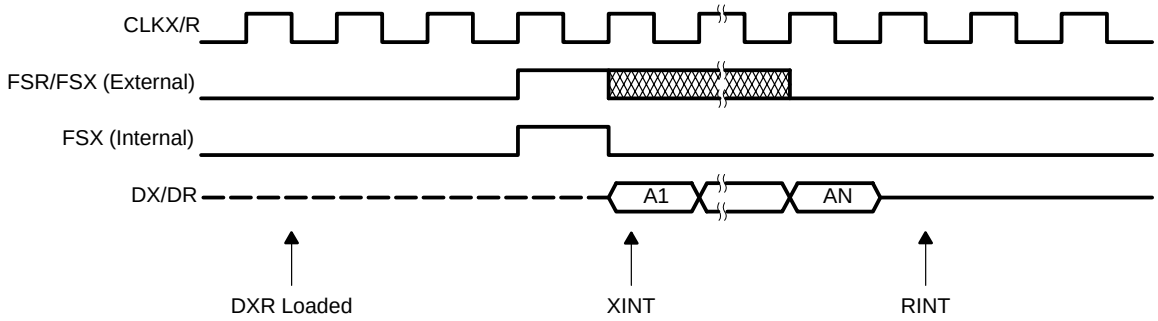
All of the serial-port operating configurations can be broadly classified in two categories: fixed data-rate timing and variable data-rate timing. The following paragraphs discuss fixed and variable data-rate operation and all of their variations.

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.

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 8–23), 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. Therefore, 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.

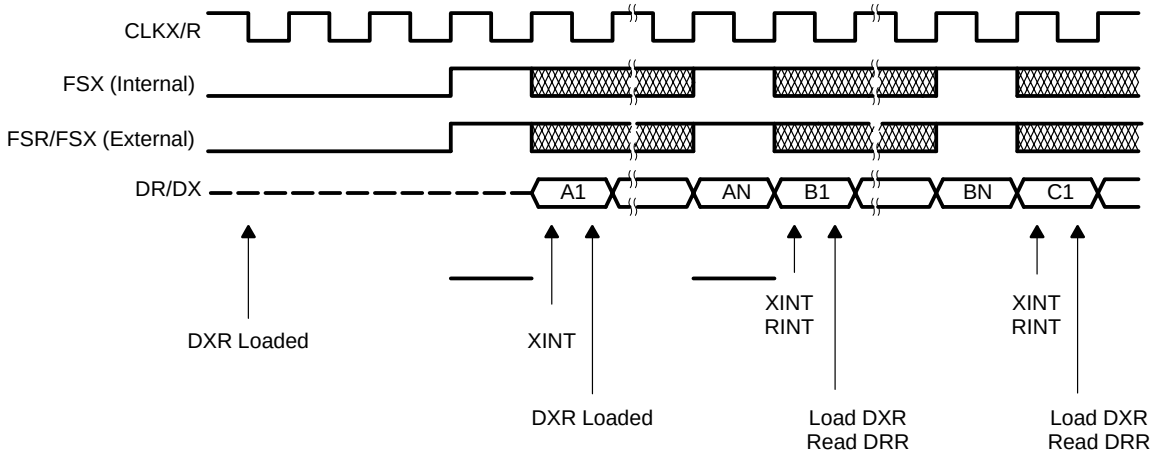
Figure 8–23. Fixed Burst Mode



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.

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 8–24).

Figure 8–24. Fixed Continuous Mode With 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. Therefore, continuous transfers will occur if 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 may vary, you should incorporate a conservative margin of safety in allowing for this delay.

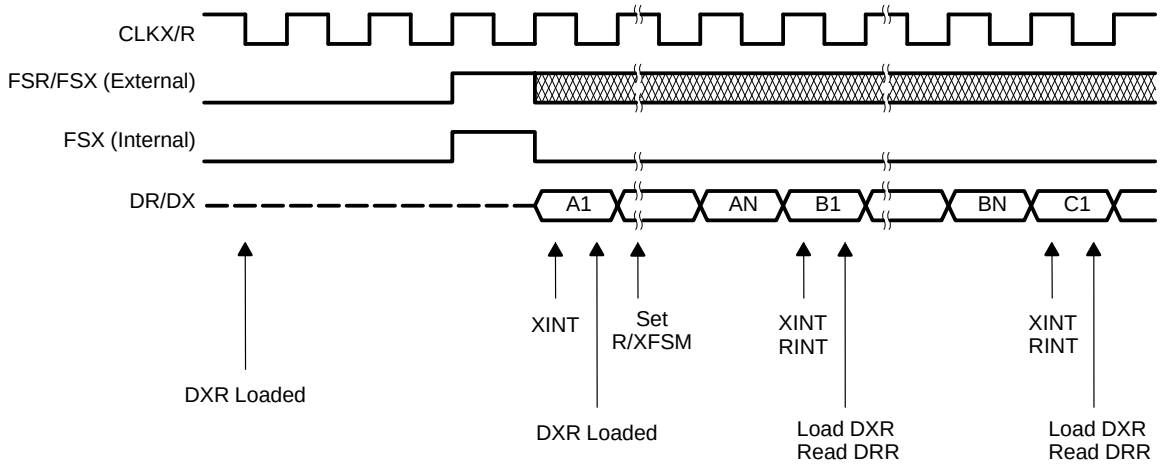
Once the process begins, an XINT and an RINT are generated at the beginning of each transfer. The XINT indicates that the XSR has been loaded from DXR and can be used to cause DXR to be reloaded. To maintain continuous transmission in fixed rate mode with frame sync, especially with an internally generated 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 is therefore commonly used to indicate an appropriate time to read DRR.

Continuous transfers are terminated by discontinuing frame sync pulses or, in the case of internally generated FSX, not reloading DXR.

You can accomplish continuous serial-port transfers without the use of frame sync pulses if R/XFSM are 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 8–25), 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.

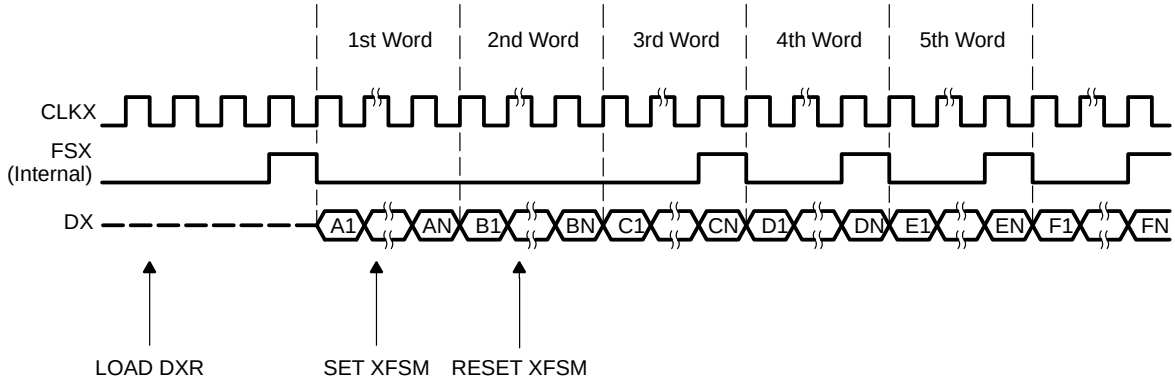
Figure 8–25. Fixed Continuous Mode Without Frame Sync



Timing of RINT and XINT and data transfers to and from DXR and DRR, respectively, are the same as in fixed data-rate continuous mode with frame sync. 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.

When you use continuous operation in fixed data-rate mode, R/XFSM can be set and cleared as desired, even during active transfers, to enable or disable the use of frame sync pulses as dictated by system requirements. Under most conditions, the effect of 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 one-word delay occurs before frame sync pulse generation resumes when clearing XFSM to 0 (see Figure 8–26). Therefore, in this case, one additional word is transferred before the next FSX pulse is generated. Also note that, as discussed previously, 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 8–26. Exiting Fixed Continuous Mode Without Frame Sync, FSX Internal

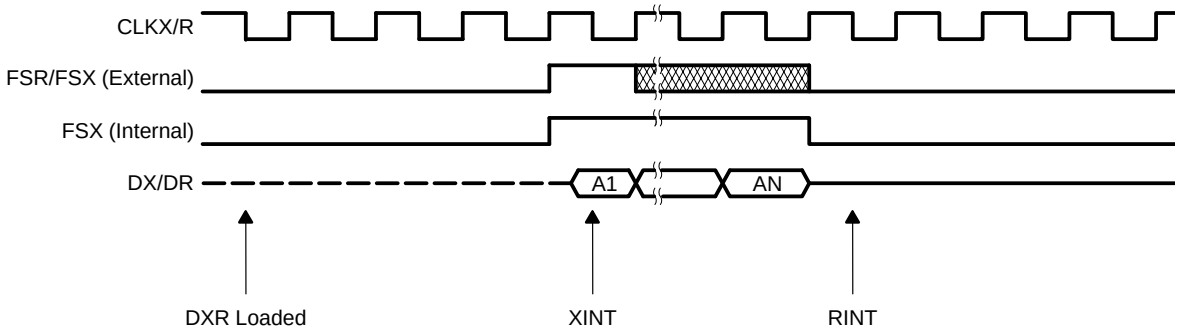


Variable Data-Rate Timing Operation

Variable data-rate timing also supports 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 (see Figure 8–27), 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:

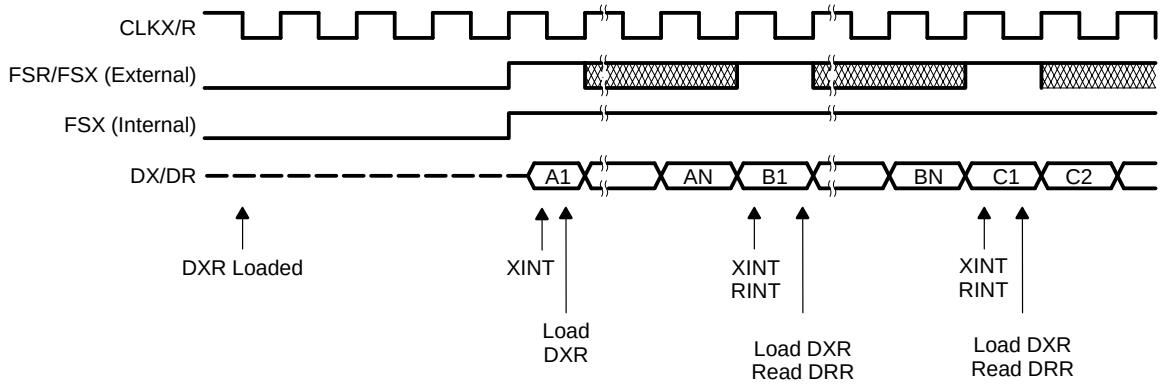
- FSX/R pulses typically last for the entire transfer interval, although FSR and external FSX are ignored after the first bit transferred. FSX/R pulses in fixed data-rate mode typically last only one CLKX/R cycle but can last longer.
- Data transfer begins during the CLKX/R cycle in which FSX/R occurs, rather than the CLKX/R cycle following FSX/R, as is the case with fixed data-rate timing.
- With variable data-rate timing, frame sync inputs are ignored until the end of the last bit transferred, rather than the beginning of the last bit transferred, as is the case with fixed data-rate timing.

Figure 8–27. Variable Burst Mode



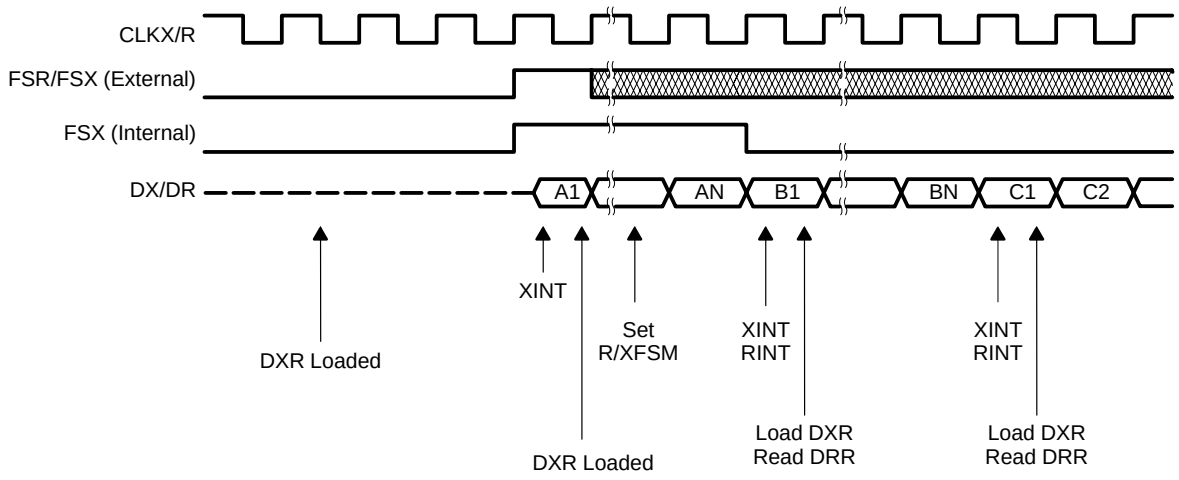
When you transmit continuously in variable data-rate mode with frame sync, timing is the same as for fixed data-rate mode, except for the differences between these two modes as described under *Variable Data-Rate Timing Operation*. The only other exception is that you must reload DXR no later than the *N*–4 bit to maintain continuous operation of the variable data-rate mode (see Figure 8–28); you must reload DXR no later than the *N*–3 bit to maintain continuous operation of the fixed data-rate mode.

Figure 8–28. Variable Continuous Mode With Frame Sync



Continuous operation in variable data-rate mode without frame sync (see Figure 8–29) is also similar to continuous operation without frame sync in fixed data-rate mode. As with variable data-rate mode continuous operation with frame sync, 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 8–29. Variable Continuous Mode Without Frame Sync



8.2.13 Serial-Port Initialization/Reconfiguration

The serial ports are controlled through memory-mapped registers on the dedicated peripheral bus. Following is a general procedure for initializing and/or reconfiguring the serial ports.

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. Note that 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 subsection 8.2.14 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.

8.2.14 TMS320C3x Serial-Port Interface Examples

In addition to the examples presented in this section, DMA/serial port initialization examples can be found in Example 8–6 and Example 8–7 on pages 8-59 and 8-61, respectively.

8.2.14.1 Handshake Mode Example

When handshake mode is used, the transmit (FSX/DS/CLKX) and receive (FSR/DR/CLKR) signals transmit and receive data, respectively. In other words, even if the TMS320C3x serial port is receiving data only with handshake mode, the transmit signals are still needed to transmit the acknowledge signal. This is the serial port register setup for the TMS320C3x serial port handshake communication, as shown in Figure 8–22 on page 8-29:

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)

x = user-configurable

Since the FSX is set as an output and continuous mode is disabled when handshake mode is selected, you should set the XFSM and RFSM bits to 0 and the FSXOUT bit to 1 in the global control register. You should set the XRESET, RRESET, and HS bits to 1 in order to start the handshake communication. You should set the polarity of the serial port pins active (high) for simplification. Although the CLKX/CLKR can be set as either input or output, you should 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, you should set the timer control register to 0Fh. When the serial port timer is used, you should also set the serial timer register to the proper value for the clock speed. The serial port timer clock speed setup is similar to the TMS320C3x timer. Refer to Section 8.1 on page 8-2 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. Therefore, if two TMS320C3xs 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 8–1 and Example 8–2 are serial port register setups for the above case. (Assume two TMS320C3xs have the same system clock.)

Example 8–1. 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 8–2. 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 TMS320C3x serial port can distinguish between the data and the acknowledge signal. Therefore, even if the TMS320C3x serial port receives the data before the acknowledge signal, the data will not be misinterpreted as the acknowledge signal and be lost. In addition, the acknowledge signal is not generated until the data is read from the data receive register (DRR). Therefore, the TMS320C3x will not transmit the data and the acknowledge signal simultaneously.

8.2.14.2 CPU Transfer With Serial-Port Transmit Polling Method

Example 8–3 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.

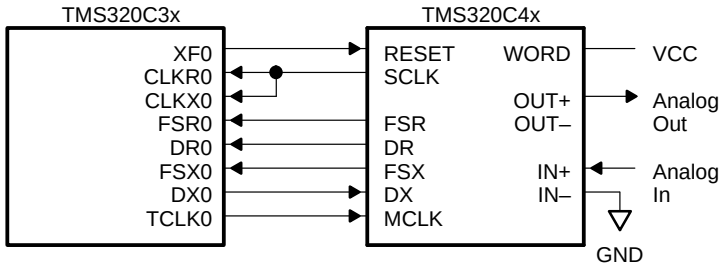
Example 8–3.CPU Transfer With Serial-Port Transmit Polling Method

```
* 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 INITIALIZATION
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
```

8.2.14.3 Serial AIC Interface Example

The TLC320C4x analog interface chips (AIC) from Texas Instruments offer a zero-glue-logic interface to the TMS320C3x family of DSPs. The interface is shown in Figure 8–30 as an example of the TMS320C3x serial-port configuration and operation.

Figure 8–30. TMS320C3x Zero-Glue-Logic Interface to TLC3204x Example



The TMS320C3x 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 TMS320C3x 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

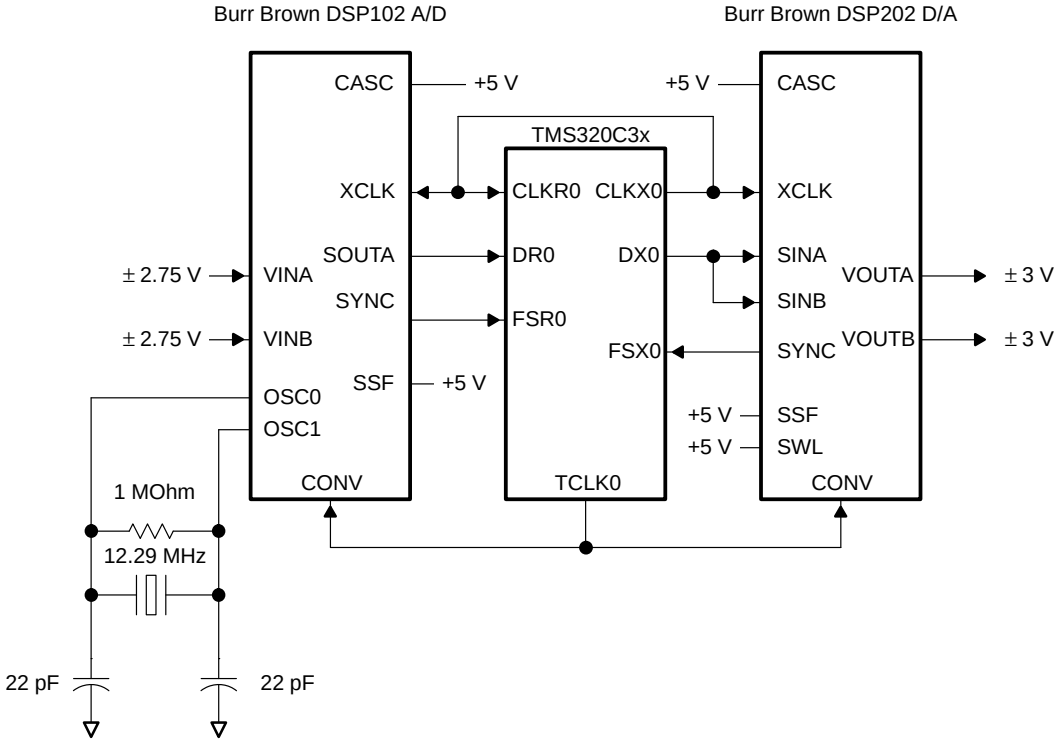
Timer:

Timer global control register	000002C1h
Timer period register	00000001h

8.2.14.4 Serial A/D and 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 TMS320C3x family of DSPs. The interface is shown in Example 8–4. This interface is used as an example of the TMS320C3x serial-port configuration and operation.

Example 8–4. TMS320C3x Zero-Glue-Logic Interface to Burr Brown A/D and D/A



The DSP102 A/D is interfaced to the TMS320C3x 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 TMS320C3x 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. The A/D signals the TMS320C3x via the A/D's SYNC signal (connected to the TMS320C3x FSR0 pin) that serial data is to be transmitted. The 32-bit word is then serially transmitted, MSB first, out the SOUTA serial pin of the DSP102 to the DR0 pin of the TMS320C3x serial port. The TMS320C3x is programmed to drive the analog interface bit clock from the CLKX0 pin of the TMS320C3x. The bit clock drives both the A/D's and D/A's XCLK input. The TMS320C3x transmit clock also acts as the input clock on the receive side of the TMS320C3x serial port. Since the receive clock is synchronous to the internal clock of the TMS320C3x, 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 TMS320C3x and signals the TMS320C3x via the SYNC signal (connected to the TMS320C3x FSX0 pin) to begin transmitting a 32-bit word representing the two channels of data to be converted. The data transmitted from the TMS320C3x DX0 pin is input to both the SINA and SINB inputs of the D/A as shown in the figure.

The TMS320C3x is set up to transfer bits at the maximum rate of about eight 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

8.3 DMA Controller

The TMS320C3x has 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 TMS320C3x 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.

A DMA transfer consists of two operations: a read from a memory location and a write to a memory location. The DMA controller can read from and write to any location in the TMS320C3x memory map. This includes all memory-mapped peripherals. The operation of the DMA is controlled with the following set of memory-mapped registers:

- DMA global-control register
- DMA source-address register
- DMA destination-address register
- DMA transfer-counter register

Table 8–7 shows these registers, their memory-mapped addresses, and their functions. Each of these DMA registers is discussed in the succeeding subsections.

Table 8–7. Memory-Mapped Locations for a DMA Channel

Register	Peripheral Address
DMA Global Control (See Table 8–8)	808000h
Reserved	808001h
Reserved	808002h
Reserved	808003h
DMA Source Address (see subsection 8.3.2)	808004h
Reserved	808005h
DMA Destination Address (see subsection 8.3.2)	808006h
Reserved	808007h
DMA Transfer Counter (see subsection 8.3.3)	808008h
Reserved	808009h
Reserved	80800Ah
Reserved	80800Bh
Reserved	80800Ch
Reserved	80800Dh
Reserved	80800Eh
Reserved	80800Fh

Table 8–8. DMA Global-Control Register Bits

Bit	Name	Reset Value	Function
1–0	START	0–0	These bits control the state in which the DMA starts and stops. The DMA may be stopped without any loss of data (see Table 8–9).
3–2	STAT	0–0	These bits indicate the status of the DMA and change every cycle (see Table 8–10).
4	INCSRC	0	If INCSRC = 1, the source address is incremented after every read.
5	DECSRC	0	If DECSRC = 1, the source address is decremented after every read. If INCSRC = DECSRC, the source address is not modified after a read.
6	INCDST	0	If INCDST = 1, the destination address is incremented after every write.
7	DECDST	0	If DECDST = 1, the destination address is decremented after every write. If INCDST = DECDST, the destination address is not modified after a write.
9–8	SYNC	0–0	The SYNC bits determine the timing synchronization between the events initiating the source and the destination transfers. The interpretation of the SYNC bits is shown in Table 8–11.
10	TC	0	The TC bit 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.
11	TCINT	0	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.
31–12	Reserved	0–0	Read as 0.

Note: 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, or
- The TC bit is set to 0.

Table 8–9. START Bits and Operation of the DMA (Bits 0–1)

START	Function
0 0	DMA read or write cycles in progress will be completed; any data read will be ignored. Any pending read or write will be cancelled. The DMA is reset so that when it starts a new transaction begins; that is, a read is performed. (Reset value)
0 1	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.
1 0	If a DMA transfer has begun, the entire transfer is completed (including both read and write operations) before stopping. If a transfer has not begun, none is started.
1 1	DMA starts from reset or restarts from the previous state.

Table 8–10. STAT Bits and Status of the DMA (Bits 2–3)

STAT	Function
0 0	DMA is being held between DMA transfer (between a write and read). This is the value at reset. (Reset value)
0 1	DMA is being held in the middle of a DMA transfer, that is, between a read and a write.
1 0	Reserved.
1 1	DMA busy; that is, DMA is performing a read or write or waiting for a source or destination synchronization interrupt.

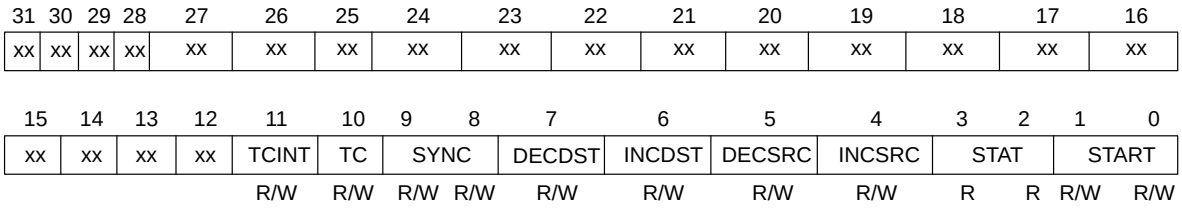
Table 8–11. SYNC Bits and Synchronization of the DMA (Bits 8–9)

SYNC	Function
0 0	No synchronization. Enabled interrupts are ignored. (Reset value)
0 1	Source synchronization. A read is performed when an enabled interrupt occurs.
1 0	Destination synchronization. A write is performed when an enabled interrupt occurs.
1 1	Source and destination synchronization. A read is performed when an enabled interrupt occurs. A write is then performed when the next enabled interrupt occurs.

8.3.1 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. Table 8–8 on page 8-45 lists the register bits, names, and functions. Figure 8–31 shows the bit configuration of the global-control register.

Figure 8–31. DMA Global-Control Register



R = Read, *W* = Write, *xx* = reserved bit, read as 0

8.3.2 Destination- and Source-Address Registers

The DMA destination-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 and decremented at the end of the corresponding memory access, that is, the source register for a read and the destination register for a write. On system reset, 0 is written to these registers.

8.3.3 Transfer-Counter Register

The transfer-counter register is a 24-bit register, controlled by a 24-bit counter that counts down. The counter 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 DMA global-control register is set, the transfer-counter register will cause a DMA interrupt flag to be set upon count down to 0.

8.3.4 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.

Table 8–12 lists the bits, names, and functions of the CPU/DMA interrupt enable register. Figure 8–32 shows the IE register. The priority and decoding schemes of CPU and DMA interrupts are identical. Note that when the DMA receives an interrupt, this interrupt is acted upon according to the SYNC field of the DMA control register. Also note that an interrupt can affect the DMA but not the CPU and can affect the CPU but not the DMA. Refer to subsection 3.1.8 on page 3-7 and to Chapter 6.

Table 8–12. CPU/DMA Interrupt-Enable Register Bits

Bit	Name	Function
0	EINT0	Enable external interrupt 0 (CPU)
1	EINT1	Enable external interrupt 1 (CPU)
2	EINT2	Enable external interrupt 2 (CPU)
3	EINT3	Enable external interrupt 3 (CPU)
4	EXINT0	Enable serial-port 0 transmit interrupt (CPU)
5	ERINT0	Enable serial-port 0 receive interrupt (CPU)
6	EXINT1	Enable serial-port 1 transmit interrupt (CPU)
7	ERINT1	Enable serial-port 1 receive interrupt (CPU)
8	ETINT0	Enable timer 0 interrupt (CPU)
9	ETINT1	Enable timer 1 interrupt (CPU)
10	EDINT	Enable DMA controller interrupt (CPU)
15–11	Reserved	Read as 0
16	EINT0	Enable external interrupt 0 (DMA)
17	EINT1	Enable external interrupt 1 (DMA)
18	EINT2	Enable external interrupt 2 (DMA)
19	EINT3	Enable external interrupt 3 (DMA)
20	EXINT0	Enable serial-port 0 transmit interrupt (DMA)
21	ERINT0	Enable serial-port 0 receive interrupt (DMA)
22	EXINT1	Enable serial-port 1 transmit interrupt (DMA)
23	ERINT1	Enable serial-port 1 receive interrupt (DMA)
24	ETINT0	Enable timer 0 interrupt (DMA)
25	ETINT1	Enable timer 1 interrupt (DMA)
26	EDINT	Enable DMA controller interrupt (DMA)
31–27	Reserved	Read as 0

Figure 8–32. 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

Note: xx = Reserved bit, read as 0
R = read, W = write

8.3.5 DMA Memory Transfer Operation

Each DMA memory transfer consists of two parts:

- Read data from the address specified by the DMA source register
- Write data that has been read to the address specified by the DMA destination register

A transfer is complete only when the read and write are complete. You can stop a transfer by setting the START bits to the desired value. When the DMA is restarted (START = 1 1), it completes any pending transfer.

At the end of a DMA read, the source address is modified as specified by the SRCINC and SRCDEC bits of the DMA global-control register. At the end of a DMA write, the destination address is modified as specified by the DSTINC and DSTDEC bits of the DMA global control register. At the end of every DMA write, the DMA transfer counter is decremented.

DMA on-chip reads and writes (reads and writes from on-chip memory and peripherals) are single-cycle. DMA off-chip reads are two cycles. The first cycle is the external read, and the second cycle loads the DMA register. The external read cycle is identical to a CPU read cycle. DMA off-chip writes are identical to CPU off-chip writes. If the DMA has been started and is transferring data over either external bus, you should not modify the bus-control register associated with that bus. If you must modify the bus-control register (see Chapter 7), stop the DMA, make the modification, and then restart the DMA. Failure to do this may produce an unexpected zero-wait-state bus access.

Through the 24-bit source and destination registers, the DMA is capable of accessing any memory-mapped location in the TMS320C3x memory map. Table 8–13, Table 8–14, and Table 8–15 show the number of cycles a DMA transfer requires, depending on whether the source and destination are on-chip memory and peripherals, the external port, or the I/O port. *T* represents the number of transfers to be performed, C_r represents the number of wait-states for the source read, and C_w represents the number of wait-states for the destination write. Each entry in the table represents the total cycles required to do the *T* transfers, assuming that there are no pipeline conflicts.

Accompanying each table is a figure illustrating the timing of the DMA transfer. |R| and |W| represent single-cycle reads and writes, respectively. |R.R| and |W.W| represent multicycle reads and writes. $|C_r|$ and $|C_w|$ show the number of wait cycles for a read and write.

Table 8–13. 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	19
Source On-Chip	R		R		R														
Destination On-Chip		W		W		W													
Source Primary Bus	R.R.R:I				R.R.R:I					R.R.R:I									
		C_r				C_r					C_r								
Destination On-Chip				W				W				W							
Source Expansion Bus	R.R.R:I				R.R.R:I					R.R.R:I									
		C_r				C_r					C_r								
Destination On-Chip				W				W				W							

Source	Destination On-Chip
On-Chip	$(1 + 1)T$
Primary Bus	$(2 + C_r + 1)T$
Expansion Bus	$(2 + C_r + 1)T$

Legend:

T = Number of transfers
C_r = Source-read wait states
C_w = Destination-write wait states
|R| = Single-cycle reads
|W| = Single-cycle writes
|R.R| = Multicycle reads
|W.W| = Multicycle writes
| I | = Internal register cycle

Table 8–14. DMA Timing When Destination Is a Primary Bus

Cycles (H1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Source On-Chip	R		R				R												
Destination Primary Bus				W	W	W	W	W	W	W	W	W	W	W	W	W			
					C_W			C_W				C_W							
Source Primary Bus	R	R	R	I				R	R	R	I								
		C_r							C_r										
Destination Primary Bus					W	W	W	W				W	W	W	W				
						C_W									C_W				
Source Expansion Bus	R	R	R	I		R	R	R	I		R	R	R	I					
		C_r				C_r			C_r			C_r							
Destination Primary Bus					W	W	W	W		W	W	W	W		W	W	W	W	
						C_W			C_W				C_W				C_W		

Source	Destination Primary Bus
On-Chip	$1 + (2 + C_W)T$
Primary Bus	$(2 + C_r + 2 + C_W)T$
Expansion Bus	$(2 + C_r + 2 + C_W) + (2 + C_W + \max(1, C_r - C_W + 1))(T - 1)$

Legend:

T = Number of transfers
C_r = Source-read wait states
C_W = Destination-write wait states
|R| = Single-cycle reads
|W| = Single-cycle writes
|R.R| = Multicycle reads
|W.W| = Multicycle writes
| I | = Internal register cycle

Table 8–15. DMA Timing When Destination Is an Expansion Bus

Cycles (H1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Source On-Chip	R		R				R												
Destination Expansion Bus				W	W	W	W	W	W	W	W	W	W						
					C_w				C_w				C_w						
Source Primary Bus	R	R	R	I			R	R	R	I			R	R	R	I			
					C_r				C_r				C_r						
Destination Expansion Bus						W	W	W	W				W	W	W	W			
						C_w				C_w						C_w			
Source Expansion Bus	R	R	R	I				R	R	R	I								
					C_r					C_r									
Destination Expansion Bus						W	W	W	W				W	W	W	W			
						C_w										C_w			

Source	Destination Expansion Bus
On-Chip	$1 + (2 + C_w)T$
Primary Bus	$(2 + C_r + 2 + C_w)$ $+ (2 + C_w + \max(1, C_r - C_w + 1))(T - 1)$
Expansion Bus	$(2 + C_r + 2 + C_w)T$

Legend:

T = Number of transfers
C_r = Source-read wait states
C_w = Destination-write wait states
|R| = Single-cycle reads
|W| = Single-cycle writes
|R.R| = Multicycle reads
|W.W| = Multicycle writes
| I | = Internal register cycle

Table 8–16 shows the maximum DMA transfer rates, assuming that there are no wait states ($C_r = C_w = 0$). Table 8–17 shows the maximum DMA transfer rates, assuming there is one wait state for the read ($C_r = 1$) and no wait states for the write ($C_w = 0$). Table 8–18 shows the maximum DMA transfer rates, assuming there is one wait state for the read ($C_r = 1$) and one wait state for the write ($C_w = 1$).

In each table, the time for the complete transfer (the read and the write) is considered. Since one bus access is required for the read and another for the write, internal bus transfer rates will be twice the DMA transfer rate. It is also assumed that no conflicts with the CPU exist. Rates are listed in Mwords/sec. A word is 32 bits (4 bytes).

Table 8–16. Maximum DMA Transfer Rates When $C_r = C_w = 0$

	Destination		
Source	**Internal**	**Primary**	**Expansion**
Internal	8.33 Mwords/sec	8.33 Mwords/sec	8.33 Mwords/sec
Primary	5.56 Mwords/sec	4.17 Mwords/sec	5.56 Mwords/sec
Expansion	5.56 Mwords/sec	5.56 Mwords/sec	4.17 Mwords/sec

Table 8–17. Maximum DMA Transfer Rates When $C_r = 1$, $C_w = 0$

	Destination		
Source	**Internal**	**Primary**	**Expansion**
Internal	8.33 Mwords/sec	8.33 Mwords/sec	8.33 Mwords/sec
Primary	4.17 Mwords/sec	3.33 Mwords/sec	4.17 Mwords/sec
Expansion	4.17 Mwords/sec	4.17 Mwords/sec	3.33 Mwords/sec

Table 8–18. Maximum DMA Transfer Rates When $C_r = 1$, $C_w = 1$

	Destination		
Source	**Internal**	**Primary**	**Expansion**
Internal	8.33 Mwords/sec	5.56 Mwords/sec	5.56 Mwords/sec
Primary	4.17 Mwords/sec	2.78 Mwords/sec	4.17 Mwords/sec
Expansion	4.17 Mwords/sec	4.17 Mwords/sec	2.78 Mwords/sec

8.3.6 Synchronization of DMA Channels

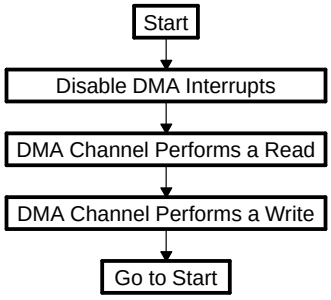
You can synchronize a DMA channel with interrupts. Refer to Table 8–11 on page 8-46 for the relationship between the SYNC bits of the DMA global control register and the synchronization performed. This section describes the following four synchronization mechanisms:

- No synchronization (SYNC = 0 0)
- Source synchronization (SYNC = 0 1)
- Destination synchronization (SYNC = 1 0)
- Source and destination synchronization (SYNC = 1 1)

No Synchronization

When SYNC = 0 0, no synchronization is performed. The DMA performs reads and writes whenever there are no conflicts. All interrupts are ignored and therefore are considered to be globally disabled. However, no bits in the DMA interrupt-enable register are changed. Figure 8–33 shows the synchronization mechanism when SYNC = 0 0.

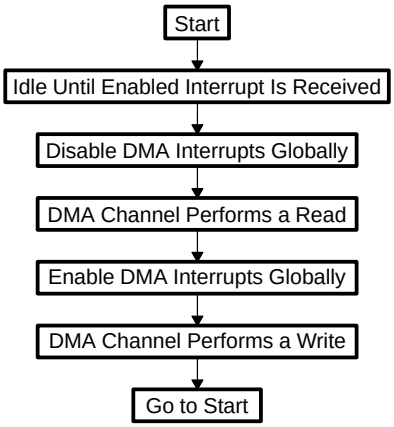
Figure 8–33. No DMA Synchronization



Source Synchronization

When SYNC = 0 1, the DMA is synchronized to the source (see Figure 8–34). A read will not be 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.

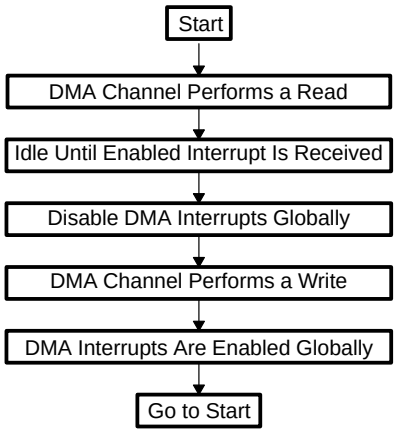
Figure 8–34. DMA Source Synchronization



Destination Synchronization

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 will not be performed until an interrupt is received by the DMA. Figure 8–35 shows the synchronization mechanism when SYNC=1 0.

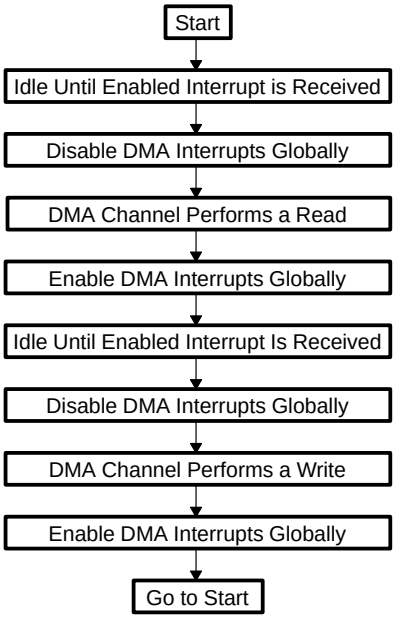
Figure 8–35. DMA Destination Synchronization



Source and Destination Synchronization

When SYNC = 1 1, the DMA is synchronized to both the source and destination. A read is performed when an interrupt is received. A write is performed on the following interrupt. Source and destination synchronization when SYNC = 1 1 is shown in Figure 8–36.

Figure 8–36. DMA Source and Destination Synchronization



8.3.7 DMA Interrupts

You can generate a DMA interrupt to the CPU whenever the transfer count reaches 0, indicating that the last transfer has taken place. The TCINT bit in the DMA global control register determines whether the interrupt will be generated. If TCINT = 1, the DMA interrupt is generated. If TCINT = 0, the DMA interrupt is not generated. If the DMA interrupt is generated, the EDINT bit, bit 10 in the interrupt enable register, must also be set to enable the CPU to be interrupted by the DMA.

A second bit in the DMA global control register, the TC bit, is also generally associated with the state of the TCINT bit and the interrupt operation. The TC bit determines whether transfers are terminated when the transfer counter becomes 0 or whether they are allowed to continue. If TC = 1, transfers are terminated when the transfer count becomes 0. If TC = 0, transfers are not terminated when the transfer count becomes 0.

In general, if TCINT is 0, TC should also be cleared to 0. Otherwise, the DMA transfer will terminate, and the CPU will not be notified. If TCINT is 1, TC should also be 1 in most cases. In this case, the CPU will be notified when the transfer completes, and the DMA will be halted and ready to start a new transfer.

8.3.8 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. Note that the DMA is halted on $\overline{\text{RESET}}$.
2) Configure the DMA via the DMA global-control register (with START = 00), as well as the DMA source, destination, and transfer-counter registers, if necessary. Refer to subsection 8.3.10 on page 8-58 for more information.
3) Start the DMA by setting the START bits of the DMA global-control register as necessary.

8.3.9 Hints for DMA Programming

The following hints help you 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. Refer to Chapter 9 for details on CPU accesses.

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 IIF register is set to 1 (interrupt polling). This requires that the TCINT bit in the DMA control register be set first. This interrupt-polling method does not cause any additional CPU-DMA access conflict.

- The transfer counter has a zero value. However, notice that 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.

8.3.10 DMA Programming Examples

Example 8–5, Example 8–6, and Example 8–7 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, the enabling of the cache helps if the .text section is in external memory. For example, when linking the code in Example 8–5, Example 8–6, and Example 8–7, the .text section can be allocated into RAM0, .data into RAM1, and .bss into RAM1, where RAM0 and RAM1 correspond to on-chip RAM block 0 and block 1, respectively.

In Example 8–5, 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 8–5.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 8–6 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; therefore, no CPU serial port interrupt vector setting is required.

Example 8–6.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 8–7 sets up the DMA to transfer data (128 words) from an array buffer to the serial port 0 output register with serial port transmit interrupt XINT0. The DMA sends an interrupt to the CPU when the data transfer completes.

Serial port 0 is initialized to transmit 32-bit data words with an internally generated frame sync and a bit-transfer rate of 8H1 cycles/bit. The receive-bit clock is internally generated and equal in frequency to one-half of the ’C3x H1 frequency.

This program assumes previous initialization of the CPU interrupt vector table (specifically the DMA-to-CPU interrupt). The serial port interrupt directly affects only the DMA; therefore, no CPU serial port interrupt vector setting is required.

Note: Serial Port Transmit Synchronization

The DMA uses serial port transmit interrupt XINT0 to synchronize transfers. Because the XINT0 is generated when the transmit buffer has written the last bit of data to the shifter, an initial CPU write to the serial port is required to trigger XINT0 to enable the first DMA transfer.

Example 8–7.DMA Transfer With Serial-Port Transmit Interrupt

```
* TITLE: DMA TRANSFER WITH SERIAL PORT TRANSMIT INTERRUPT
*         .GLOBAL START
          .DATA
DMA       .WORD  808000H      ; DMA GLOBAL CONTROL REG ADDRESS
CONTROL   .WORD  0E13H        ; DMA GLOBAL CONTROL REG INITIALIZATION
SOURCE    .WORD  (_ARRAY+1)   ; DATA SOURCE ADDRESS
DESTIN    .WORD  80804CH      ; DATA DESTIN ADDRESS: SERIAL-PORT OUTPUT REG
COUNT     .WORD  127          ; NUMBER OF WORDS TO TRANSFER =(MSG LENGHT–1)
IEVAL     .WORD  00100400H    ; IE REGISTER VALUE
          .BSS   _ARRAY,128   ; DATA ARRAY LOCATED IN .BSS SECTION
                              ; THE UNDERSCORE USED IS JUST TO MAKE IT
                              ; ACCESSIBLE FROM C (OPTIONAL)
RESET1    .WORD  0E10H        ; DMA RESET
SPORT     .WORD  808040H      ; SERIAL-PORT GLOBAL CONTROL REG ADDRESS
SGCCTRL   .WORD  04880044H    ; SERIAL-PORT GLOBAL CONTROL REG INITIALIZATION
SXCTRL    .WORD  111H         ; SERIAL-PORT TX PORT CONTROL REG 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

* 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. The order of memory is to be maintained.

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. The order of memory is to be inverted; that is, 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. The transfer is to be synchronized 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. The transfer is to be synchronized 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. The transfer is to be synchronized 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 9

Pipeline Operation

Two characteristics of the TMS320C3x that contribute to its high performance are:

- Pipelining, and
- Concurrent I/O and CPU operation.

Five functional units control TMS320C3x operation:

- Fetch
- Decode
- Read
- Execute
- Direct memory access (DMA)

Pipelining is the overlapping or parallel operations of the fetch, decode, read, and execute levels of a basic instruction.

By performing input/output operations, the DMA controller reduces the need for the CPU to do so, thereby decreasing pipeline interference and enhancing the CPU's computational throughput.

Major topics discussed in this chapter are as follows:

Topic **Page**

9.1 Pipeline Structure

The five major units of the TMS320C3x pipeline structure and their functions are as follows:

- **Fetch Unit (F)**

  This unit fetches the instruction words from memory and updates the program counter (PC).

- **Decode Unit (D)**

  This unit decodes the instruction word and performs address generation. The unit also controls any modifications to the auxiliary registers and the stack pointer.

- **Read Unit (R)**

  This unit, if required, reads the operands from memory.

- **Execute Unit (E)**

  This unit, if required, reads the operands from the register file, performs any necessary operation, and writes results to the register file. If required, the unit writes results of previous operations to memory.

- **DMA Channel (DMA)**

  The DMA channel reads and writes to memory.

A basic instruction has four levels:

- Fetch
- Decode
- Read
- Execute

Figure 9–1 illustrates these four levels of the pipeline structure. The levels are indexed according to instruction and execution cycle. The perfect overlap in the pipeline, where all four units operate in parallel, occurs at cycle (m). Those levels about to be executed are at $m + 1$, and those just executed are at $m - 1$. The TMS320C3x pipeline control allows a high-speed execution rate of one execution per cycle. It also manages pipeline conflicts so that they are transparent to the user. You do not need to take any special precautions to guarantee correct operation.

Figure 9–1. TMS320C3x Pipeline Structure

CYCLE	F	D	R	E	
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	

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read; *W, X, Y, Z* = Instruction Representations

Priorities from highest to lowest have been assigned to each of the functional units as follows:

1) Execute (highest)
2) Read
3) Decode
4) Fetch
5) DMA (lowest)

When the processing of an instruction is ready to pass to the next higher pipeline level, but that level is not ready to accept a new input, a pipeline conflict occurs. In this case, the lower-priority unit waits until the higher-priority unit completes its currently executing function.

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.

9.2 Pipeline Conflicts

The pipeline conflicts of the TMS320C3x 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 TMS320C3x compete for memory resources.

Each of these three categories is discussed in the following sections. Examples are included. Note that 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 of not ready, the symbols $\overline{\text{RDY}}$ and RDY are used to indicate not ready and ready, respectively.

9.2.1 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 guarantee 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 9–1 shows the code and pipeline operation for a standard branch.

Note: Dummy Fetch

One dummy fetch (an MPYF instruction) is performed, which affects the cache. After the branch address is available, a new fetch (an OR instruction) is performed.

Example 9–1. Standard Branch

```
        BR     THREE  ; Unconditional branch
        MPYF          ; Not executed
        ADD           ; Not executed
        SUBF          ; Not executed
        AND           ; Not executed

        .
        .
        .
THREE   OR            ; Fetched after BR is fetched

        STI

        .
        .
```

PIPELINE OPERATION

PC	F	D	R	E
n	BR	–	–	–
n + 1	MPYF	BR	–	–
n + 1	(nop)	(nop)	BR	–
n + 1	(nop)	(nop)	(nop)	BR
THREE	OR	(nop)	(nop)	(nop)
	STI	OR	(nop)	(nop)

THREE → PC **Fetch held for new PC value**

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

RPTS and RPTB both flush the pipeline, allowing the RS, RE, and RC registers to be loaded at the proper time relative to the flow of the pipeline. If these registers are loaded without the use of RPTS or RPTB, no flushing of the pipeline occurs. If you are not using any of the repeat modes, then you can use RS, RE, and RC as general-purpose 32-bit registers and not cause any pipeline conflicts. In cases such as the nesting of RPTB due to nested interrupts, it might 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 guarantee the fetching of the next three instructions. The delayed branches include BRD, B*cond*D, and DB*cond*D. Example 9–2 shows the code and pipeline operation for a delayed branch.

Example 9–2. 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	**F**	**D**	**R**	**E**	
n	BRD	—	—	—	
n + 1	MPYF	BRD	—	—	**No execute delay**
n + 2	ADDF	MPYF	BRD	—	
n + 3	SUBF	ADDF	MPYF	BRD	
THREE	MPYF	SUBF	ADDF	MPYF	

THREE → PC

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, PC = Program Counter

9.2.2 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 9.3 on page 9-18.

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, instruction execution is completed. In Example 9–3, 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; that is, 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.

Example 9–3. Write to an AR Followed by an AR for Address Generation

```
        LDI    7,AR1      ; 7 → AR1
NEXT    MPYF   *AR2,R0    ; Decode delayed 2 cycles
        ADDF
        FLOAT
```

PIPELINE OPERATION

PC	F	D	R	E
n	LDI	—	—	—
n + 1	MPYF	LDI	—	—
n + 2	**ADDF**	MPYF	LDI	—
n + 2	ADDF	MPYF	(nop)	LDI 7,AR1
n + 2	ADDF	MPYF	(nop)	(nop)
n + 3	FLOAT	ADDF	MPYF	(nop)

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

The case for reads of these groups is similar to the case 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 therefore 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.

In Example 9–4, 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 9–4.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	F	D	R	E
n	ADDI	—	—	—
n + 1	MPYF	ADDI	—	—
n + 2	ADDF	MPYF	ADDI	—
n + 2	ADDF	MPYF	(nop)	ADDI AR0,AR1,R0
n + 3	FLOAT	ADDF	MPYF	(nop)

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

Loop counter auxiliary registers for the decrement and branch (DBR)) instruction are regarded in the same way as they are for addressing. Therefore, the operation shown in Example 9–3 and Example 9–4 can also occur for this instruction.

9.2.3 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 9.4 on page 9-21 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.

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 9–5 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 9–5. Since no 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 9–5. Program Wait Until CPU Data Access Completes

```
ADDF3  *AR0,*AR1,R0
FIX
MPYF
ADDF3
NEGB
```

PIPELINE OPERATION

PC	F	D	R	E	
n	ADDF3	—	—	—	
n + 1	FIX	ADDF3	—	—	
n + 2	(WAIT)	FIX	ADDF3	—	
n + 2	MPYF	(nop)	FIX	ADDF3	*AR0,AR1,R0
n + 3	ADDF3	MPYF	(nop)	FIX	
n + 4	NEGB	ADDF3	MPYF	(nop)	

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

Example 9–6 shows a program wait due to a multicycle data-data access or a multicycle DMA access. The ADDF, MPYF, and SUBF are fetched from a portion of memory other than the external port that 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 7F FFFFh to 80 0000h, or
  - From 80 9FFFh to 80 A000h.
- 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, or
  - a delayed decrement and branch instruction.

Even though the DMA has the lowest priority, multicycle access cannot be aborted. The program fetch must therefore wait until the DMA access completes.

Example 9–6. Program Wait Due to Multicycle Access

PIPELINE OPERATION

PC	F	D	R	E
n	ADDF	—	—	—
n + 1	MPYF	ADDF	—	—
n + 2	SUBF	MPYF	ADDF	—
n + 3	(WAIT)	SUBF	MPYF	ADDF
n + 3	CALL	(nop)	SUBF	MPYF
n + 4	—	CALL	(nop)	SUBF

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

Program Fetch Incomplete

A program fetch incomplete occurs when a program fetch requires more than one cycle to complete due to wait states. In Example 9–7, the MPYF and ADDF are fetched from memory that supports single-cycle accesses. The SUBF is fetched from memory, which requires one wait state. One example that demonstrates this conflict is a fetch across a bank boundary on the primary port. See Section 7.4 on page 7-30.

Example 9–7. Multicycle Program Memory Fetches

PIPELINE OPERATION

PC	F	D	R	E
n	MPYF	—	—	—
n + 1	ADDF	MPYF	—	—
n + 2 $\overline{\text{RDY}}$	SUBF	ADDF	MPYF	—
n + 2 RDY	SUBF	(nop)	ADDF	MPYF
n + 3	ADDI	SUBF	(nop)	ADDF

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

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 three cases in which this occurs:

- An instruction performs a store and is followed by an instruction that does 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 9–8. 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. Note that a refetch of the next instruction can occur.

Example 9–8. Single Store Followed by Two Reads

```
                STF    R0,*AR1   ; R0 → *AR1
                LDF    *AR2,R1   ; *AR2 → R1 in parallel with
          ||    LDF    *AR3,R2   ; *AR3 → R2
```

PIPELINE OPERATION

PC	F	D	R	E
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 *AR2,R1 and *AR3,R2

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter, *W,X,Y* = Instruction Representations

Example 9–9 shows a parallel store followed by a single load or read. Since the two parallel stores are required, the next CPU data memory read must wait a cycle before beginning. One program memory refetch can occur.

Example 9–9. 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	F	D	R	E
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

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

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 9–10. Note that a program refetch can occur.

Example 9–10. Interlocked Load

```
NOT     R1,R0
LDII    300h,AR2
ADDI    *AR2,R2
CMPI    R0,R2
```

PIPELINE OPERATION

PC	F	D	R	E
n	NOT	–	–	–
n + 1	LDII	NOT	–	–
n + 2	ADDI	LDII	NOT	–
n + 3	CMPI	ADDI	LDII	NOT
n + 3	–	CMPI	ADDI	LDII
n + 4	–	CMPI	ADDI	LDII

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

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 are processed.

The first type of hold everything conflict occurs when one of the external ports is busy due to an access that has started but is not complete. In Example 9–11, the first store is a two-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 9–11. Busy External Port

```
STF     R0,@DMA1
LDF     @DMA2,R0
```

PIPELINE OPERATION

PC	F	D	R	E
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

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter, *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 9–12, the LDF is performed from an external memory that requires several cycles to complete.

Example 9–12. Multicycle Data Reads

```
LDF    @DMA,R0
```

PIPELINE OPERATION

PC	F	D	R	E
n	LDF	—	—	—
n + 1	I	LDF	—	—
n + 2	J	I	LDF	—
n + 3	K,(dummy)	I	LDF	—
n + 3	K_2	J	I	LDF

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter, *I, J, K* = Instruction Representations

The final type of hold everything conflict involves conditional calls and traps, which are different from the other branch instructions. Whereas the other branch instructions are conditional loads, the conditional calls and traps are conditional stores, which require one cycle more than a conditional branch (see Example 9–13). The added cycle is used to push the return address after the call condition is evaluated.

Example 9–13. Conditional Calls and Traps

PIPELINE OPERATION

PC	F	D	R	E
n9	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)

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter, *I,* = Instruction Representation

9.3 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 subsection 9.2 on page 9-4.

Example 9–14, Example 9–15, and Example 9–16 demonstrate either some common uses of these registers that do not produce a conflict or ways that you can avoid the conflict.

Example 9–14. 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	F	D	R	E
n	LDF	—	—	—
n + 1	MPYF	LDF	—	—
n + 2	ADDF	MPYF	LDF	—
n + 3	FIX	ADDF	MPYF	LDF
n + 4	MPYF	FIX	ADDF	MPYF
n + 5	ADDF	MPYF	FIX	ADDF

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter, *W, X, Y, Z* = Instruction Representations

Example 9–15. 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	F	D	R	E	
n	LDI	—	—	—	
n + 1	MPYF	LDI	—	—	
n + 2	ADDF	MPYF	LDI	—	
n + 3	MPYF	ADDF	MPYF	LDI	7, AR2
n + 4	SUBF	MPYF	ADDF	MPYF	
n + 5	STF	SUBF	MPYF	ADDF	

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

Example 9–16. 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	F	D	R	E
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

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

9.4 Resolving Memory Conflicts

If program fetches and data accesses are performed in such a manner that the resources being used cannot provide the necessary bandwidth, the program fetch is delayed until the data access is complete. Certain configurations of program fetch and data accesses yield conditions under which the TMS320C3x can achieve maximum throughput.

Table 9–1 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). As shown in Table 9–1, four cases achieve one-cycle maximization.

Table 9–1. One Program Fetch and One Data Access for Maximum Performance

Case #	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 TMS320C30.

Table 9–2 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 9–2. One Program Fetch and Two Data Accesses for Maximum Performance

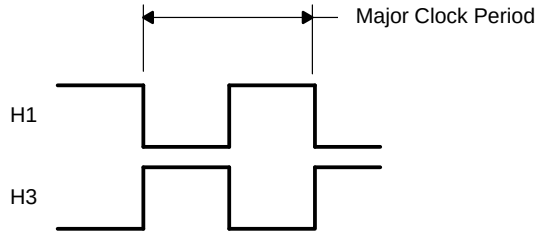
Case #	Primary Bus Accesses	Accesses From Dual-Access Internal Memory	Expansion† Or Peripheral Bus Accesses
1	1	2 from any combination of internal memory	–
2†	1 Program	1 Data	1 Data
3†	1 Data	1 Data	1 Program
4	–	2 from same internal memory block and 1 from a different internal memory block	–
5	–	3 from different internal memory blocks	–
6	–	2 from any combination of internal memory	1

† The expansion bus is available only on the TMS320C30.

9.5 Clocking of Memory Accesses

This section uses the relationships between internal clock phases (H1 and H3) to memory accesses to illustrate how the TMS320C3x handles multiple memory accesses. Whereas 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 60 ns is composed of two minor clock periods of 30 ns, labeled H3 and H1. The active clock period for H3 and H1 is the time when that signal is high.



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.

9.5.1 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 this case, the program fetch occurs during H1, and the data store 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, they are completed with the latching of the instruction word.

9.5.2 Data Loads and Stores

Four types of instructions perform loads, memory reads, and stores:

- Two-operand instructions,
- Three-operand instructions,
- Multiplier/ALU operation with store instructions, and
- Parallel multiply and add instructions.

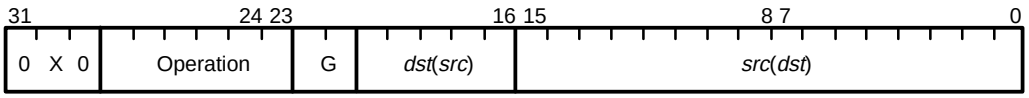
See Chapter 5 for detailed information on addressing modes.

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.

Two-Operand Instruction Memory Accesses

Two-operand instructions include all instructions whose bits 31–29 are 000 or 010 (see Figure 9–2). 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 being presented on the external bus; they complete with the latching of the data word at the end of H1.

Figure 9–2. Two-Operand Instruction Word

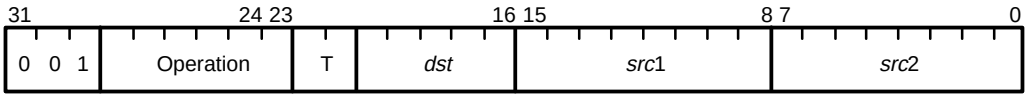


In the case of a data store, bits 15–0 represent the *dst* operand. Internal data stores are performed during H3. External data stores always start at the beginning of H3, with the address and data being presented on the external bus.

Three-Operand Instruction Memory Reads

Three-operand instructions include all instructions whose bits 31–29 are 001 (see Figure 9–3). 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 9–3. Three-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 being 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, several cases occur. If both operands are located in internal memory, the *src1* read is performed during H3 and the *src2* read during H1, thus 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.

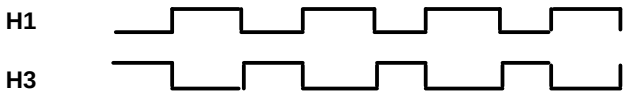
If *src1* is in external memory and *src2* is in internal memory, two cycles are necessary to complete the two reads. In the first cycle, both operands are addressed. Since *src1* takes an entire cycle to be read and latched from external memory, the internal operation on *src2* cannot be completed until 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 9–17 illustrates how the dummy read can occur. Example 9–18 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 9–17. Dummy src2 Read

```
STI    R0,*AR6       ; AR6 points to MSTRB space
ADDI3  *AR3,*AR1,R0  ; AR3 points to on-chip RAM
                     ; AR1 points to MSTRB space
```



PIPELINE OPERATION

PC	F	D	R	E	
n	STI				
n + 1	ADDI3	STI			
n + 2		ADDI3	STI		
n + 3			—	STI	R0,*AR6
n + 4			—	—	The read of src2 cannot start until the store is complete.
n + 5			ADDI3	—	dummy load of src2
n + 6			—	—	second cycle of dummy load
n + 7			ADDI3	—	actual read of src2 and src1
n + 8				ADDI3	*AR3,*AR1,R0

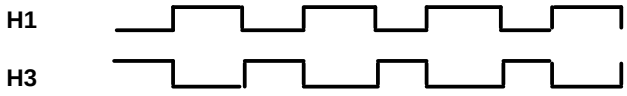
D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

Two cycles are required for the $\overline{\text{MSTRB}}$ store. Two other cycles are required for the dummy $\overline{\text{MSTRB}}$ read of *AR3 (because the read follows a write). One cycle is required for an actual $\overline{\text{MSTRB}}$ read of *AR3.

Example 9–18. Operand Swapping Alternative

Switch the operands of the three-operand instruction so that the internal read is performed first.

```
STI    R0,*AR6       ;AR6 points to MSTRB space
ADDI3  *AR1,*AR3,R0  ;AR3 points to on-chip RAM
                     ;AR1 points to MSTRB space
```



PIPELINE OPERATION

PC	F	D	R	E	
n	STI				
n + 1	ADDI3	STI			
n + 2		ADDI3	STI		
n + 3			—	STI	R0,*AR6
n + 4			—	—	The read of src2 cannot start until the store is complete.
n + 5			ADDI3	—	actual read of src2 and src1
n + 6			—	—	second cycle of src2 read
n + 7			—	ADDI3	*AR1,*AR3,R0

D = Decode, *E* = Execute, *F* = Fetch, *R* = Read, *PC* = Program Counter

Operations with Parallel Stores

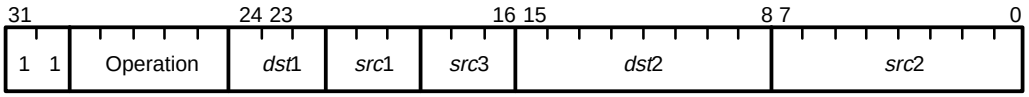
The next class of instructions includes every instruction that has a store in parallel with another instruction. Bits 31 and 30 for these instructions are equal to 1 1.

The instruction word format for those operations that perform a multiply or ALU operation in parallel with a store is shown in Figure 9–4. 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 per-

formed 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 9–4. Multiply or CPU Operation With a Parallel Store



The instruction word format for those instructions that have parallel stores to memory is shown in Figure 9–5. 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.

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.

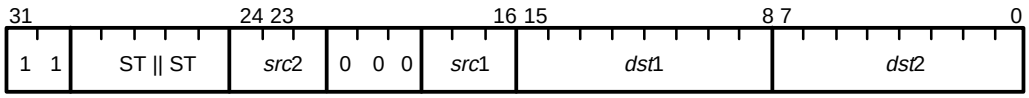
If *dst1* and *dst2* are both written to external memory, a single CPU cycle is still all that is necessary to complete the stores. In this case, four bus cycles are required.

1) In the first cycle, both *dst1* and *dst2* are written to the port, and the external bus access for *dst1* begins.

2) The store for *dst1* is completed on the second cycle, and the store for *dst2* begins on the third external bus cycle.

3) Finally, the store for *dst2* is completed on the fourth external bus cycle.

Figure 9–5. Two Parallel Stores



Parallel Multiplies and Adds

Memory addressing for parallel multiplies and adds is similar to that for three-operand instructions. The parallel multiplies and adds include all instructions whose bits 31–30 = 10 (see Figure 9–6).

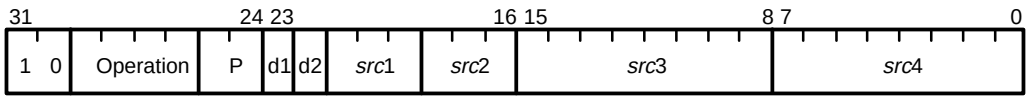
For these operations, *src3* and *src4* are both located in memory. If both operands are located in internal memory, *src3* is performed during H3, and *src4* is performed during H1, thus completing two memory reads in a single cycle.

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 one 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.

Figure 9–6. Parallel Multiplies and Adds



Chapter 10

Assembly Language Instructions

The TMS320C3x assembly language instruction set supports numeric-intensive, signal-processing, and general-purpose applications. The instructions are organized into major groups consisting of load-and-store, two- or three-operand arithmetic/logical, parallel, program-control, and interlocked operations instructions. The addressing modes used with the instructions are described in Chapter 5.

The TMS320C3x 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 10.3.2 on page 10-16). Example instructions demonstrate the special format and explain its content.

This chapter discusses the following major topics:

Topic **Page**

10.1 Instruction Set

All of the instructions in the TMS320C3x instruction set are one machine word long. Most require one cycle to execute. All instructions are a single machine word long, and most instructions require one cycle to execute. In addition to multiply and accumulate instructions, the TMS320C3x possesses a full complement of general-purpose instructions.

The instruction set contains 113 instructions organized into the following functional groups:

- Load-and-store
- Two-operand arithmetic/logical
- Three-operand arithmetic/logical
- Program control
- Interlocked operations
- Parallel operations

Each of these groups is discussed in the succeeding subsections.

10.1.1 Load-and-Store Instructions

The TMS320C3x supports 12 load-and-store instructions (see Table 10–1). These instructions can:

- Load a word from memory into a register,
- Store a word from a register into memory, or
- 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 10.2 on page 10-10 for detailed information on condition codes.

Table 10–1. 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		

10.1.2 Two-Operand Instructions

The TMS320C3x supports 35 two-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 10–2, these instructions provide integer, floating-point, or logical operations, and multiprecision arithmetic.

Table 10–2. Two-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		

† Two- and three-operand versions

10.1.3 Three-Operand Instructions

Most instructions have only two operands; however, some arithmetic and logical instructions have three-operand versions. The 17 three-operand instructions allow the TMS320C3x to read two operands from memory or the CPU register file in a single cycle and store the results in a register. The following factors differentiate the two- and three-operand instructions:

- Two-operand instructions have a single source operand (or shift count) and a destination operand.
- Three-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 three-operand instruction is always a register.

Table 10–3 lists the instructions that have three-operand versions. Note that you can omit the 3 in the mnemonic from three-operand instructions (see subsection 10.3.2 on page 10-16).

Table 10–3. Three-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		

10.1.4 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 of the program control instructions are capable of conditional operations (see Section 11.2 on page 11-6 for detailed information on condition codes). Table 10–4 lists the program control instructions.

Table 10–4. 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		

10.1.5 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 10–5 lists the low-power control instructions.

Table 10–5. Low-Power Control Instructions

Instruction	Description	Instruction	Description
IDLE2	Low-power idle	MAXSPEED	Restore clock to regular speed
LOPOWER	Divide clock by 16		

10.1.6 Interlocked-Operations Instructions

The interlocked operations instructions (Table 10–6) support multiprocessor communication and the use of external signals to allow for powerful synchronization mechanisms. The instructions also guarantee the integrity of the communication and result in a high-speed operation. Refer to Chapter 6 for examples of the use of interlocked instructions.

Table 10–6. 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		

10.1.7 Parallel-Operations Instructions

The parallel-operations instructions group makes a high degree of parallelism possible. Some of the TMS320C3x instructions can occur in pairs that will be executed in parallel. These instructions offer the following features:

- Parallel loading of registers,
- Parallel arithmetic operations, or
- 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 10–7 lists the valid instruction pairs.

Table 10–7. Parallel Instructions

Mnemonic	Description
	Parallel Arithmetic with Store Instructions
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
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

Table 10–7. Parallel Instructions (Continued)

Mnemonic	Description
	Parallel Arithmetic with Store Instructions (Concluded)
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
	Parallel Load Instructions
LDF || LDF	Load floating-point
LDI || LDI	Load integer
	Parallel Multiply and Add/Subtract Instructions
MPYF3 || ADDF3	Multiply and add floating-point
MPYF3 || SUBF3	Multiply and subtract floating-point
MPYI3 || ADDI3	Multiply and add integer
MPYI3 || SUBI3	Multiply and subtract integer

10.1.8 Illegal Instructions

The TMS320C3x 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 can cause the generation of an illegal opcode:

- Misuse of the tools
- An error in the ROM code
- Defective RAM

10.2 Condition Codes and Flags

The TMS320C3x 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. Note that 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 can be modified by most instructions when either of the preceding conditions is established and either of the following two cases occurs:

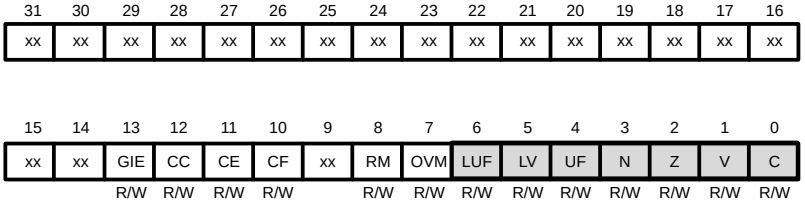
- A result is generated when the specified operation is performed to infinite precision. This is appropriate for compare and test instructions that do not store results in a register. It is also appropriate for arithmetic instructions that produce underflow or overflow.
- The output is written to the destination register, as shown in Table 10–8. This is appropriate for other instructions that modify the condition flags.

Table 10–8. 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 10–1 on page 10-11 shows the condition flags in the low-order bits of the status register. Following the figure is a list of status register condition flags and descriptions 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 subsection 10.3.3 on page 10-18.

Figure 10–1. Status Register



NOTE: xx = reserved bit
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 Condition Flag**

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 Condition Flag**

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 integer and floating-point operations, N is set if the result is negative, and cleared otherwise. Zero 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, $-2^{32} \leq$ result $\leq 2^{32} - 1$). 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 final value shifted out; for a 0 shift count, this is set to 0.

Table 10–9 lists the condition mnemonic, code, description, and flag for each of the 20 condition codes.

Table 10–9. Condition Codes and Flags

Condition	Code	Description	Flag†
		Unconditional Compares	
U	00000	Unconditional	Don't care
		Unsigned Compares	
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
		Signed Compares	
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
		Compare to Zero	
Z	00101	Zero	Z
NZ	00110	Not zero	~Z
P	01001	Positive	~N AND ~Z
N	00111	Negative	N
NN	01010	Nonnegative	~N
		Compare to Condition Flags	
NN	01010	Nonnegative	~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)

10.3 Individual Instructions

This section contains the individual assembly language instructions for the TMS320C3x. 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 10.1 on page 10-2. Appendix A lists the opcodes for all of the instructions. Refer to Chapter 5 for information on memory addressing. Code examples using many of the instructions are provided in Chapter 11.

10.3.1 Symbols and Abbreviations

Table 10–10 lists the symbols and abbreviations used in the individual instruction descriptions.

Table 10–10. 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	Three-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
ARn	Auxiliary register n
IRn	Index register n
Rn	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

10.3.2 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. Following is a list of these optional syntax forms.

- 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 three-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 three-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 three-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 MSBs of a relocatable address:

  LDP addr,REG or LDP @addr,REG

  The @ sign is optional.

  If the destination REG 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 three-operand parallel instructions that look like normal two-operand instructions. For example,

  ADDI *AR0,R2,R2 can be written as ADD *AR0,R2
  || MPYI *AR1,R0,R0 || MPYI *AR1,R0.

  Instructions (applies to all parallel instructions that have a register second operand) affected: ADDI, ADDF, AND, MPYI, MPYF, OR, SUBI, SUBF, and 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: parallel instructions containing any of ADDI, ADDF, MPYI, MPYF, AND, OR, and XOR.

- Use the syntax in Table 10–11 to designate CPU registers in operands. Note the alternate notation Rn, $0 \leq n \leq 27$, which is used to designate any CPU register.

Table 10–11. 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

10.3.3 Individual Instruction Descriptions

Each assembly language instruction for the TMS320C3x 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, $0 \leq n \leq 27$)
0 1	direct
1 0	indirect
1 1	immediate

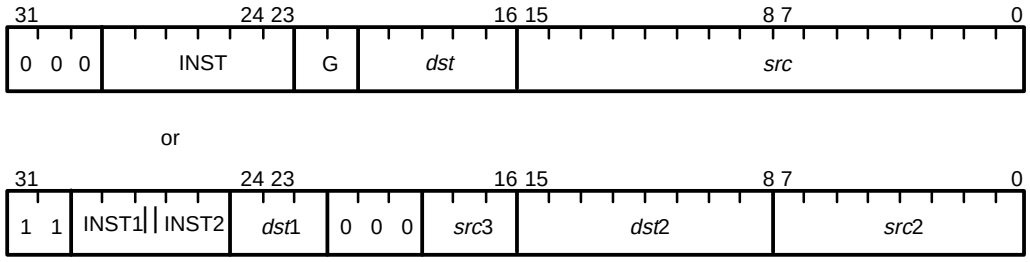
dst register (Rn, $0 \leq n \leq 27$)

or

src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn1, $0 \leq n1 \leq 7$)
src3	register (Rn2, $0 \leq n2 \leq 7$)
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. Refer to Chapter 5 for detailed information on addressing.

Encoding



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:

DP = 80h
R5 = 0766900000h = 2.30562500e+02
Memory at 8098AEh = 5CDFh = 1.00001107e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R5 = 0066900000h = 1.80126953e + 00
Memory at 8098AEh = 5CDFh = 1.00001107e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

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 10.2 on page 10-10 and Table 10–9 on page 10-13 for further information on these seven status bits).

Syntax **ABSF** *src*, *dst*

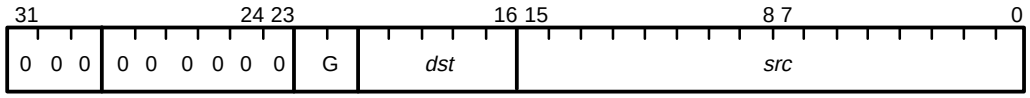
Operation |*src*| → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, ≤ 0 n ≤ 7)

Encoding



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:

R4 = 05C8000F971h = –9.90337307e + 27
R7 = 07D251100AEh = 5.48527255e + 37
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 05C8000F971h = –9.90337307e + 27
R7 = 05C7FFF068Fh = 9.90337307e + 27
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

```
    ABSF  src2, dst1
||  STF   src3, dst2
```

Operation

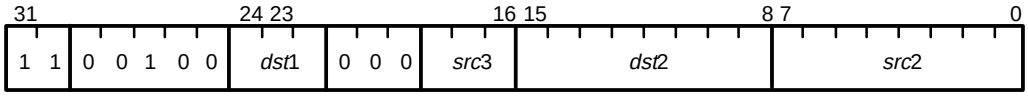
$|src2| \rightarrow dst1$

$src3 \rightarrow dst2$

Operands

src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn1, $0 \leq n1 \leq 7$)
src3	register (Rn2, $0 \leq n2 \leq 7$)
dst2	indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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 as input the contents of the register 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:

AR3 = 809800h
IR1 = 0AFh
R4 = 733C00000h = 1.79750e + 02
AR7 = 8098C5h
Data at 8098AFh = 58B4000h = – 6.118750e + 01
Data at 8098C4h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 8098AFh
IR1 = 0AFh
R4 = 574C00000h = 6.118750e + 01
AR7 = 8098C5h
Data at 8098AFh = 58B4000h = –6.118750e + 01
Data at 8098C4h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **ABSI** *src*, *dst*

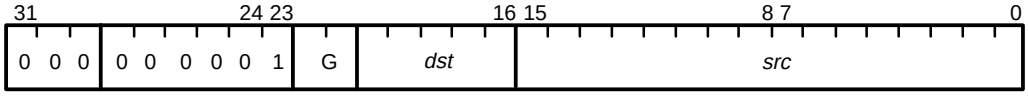
Operation |*src*| → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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:

R0 = 0FFFFFFCBh = – 53

After Instruction:

R0 = 035h = 53

Example 2

```
ABSI  *AR1,R3
```

Before Instruction:

AR1 = 20h
R3 = 0h
Data at 20h = 0FFFFFFCBh = – 53

After Instruction:

AR1 = 20h
R3 = 35h = 53
Data at 20h = 0FFFFFFCBh = – 53

Syntax

ABSI *src2*, *dst1*

|| **STI** *src3, dst2*

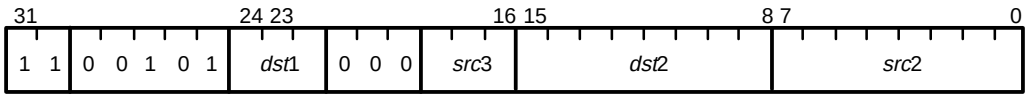
Operation

|*src2*| → *dst1*
|| *src3* → *dst2*

Operands

src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn1, 0 ≤ 1 ≤ 7)
src3	register (Rn2, 0 ≤ n2 ≤ 7)
dst2	indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that, 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 as input the contents of the register 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:

AR5 = 8099E2h
R5 = 0h
R1 = 42h = 66
AR2 = 8098FFh
IR1 = 0Fh
Data at 8099E1h = 0FFFFFFCBh = – 53
Data at 8098FFh = 2h = 2
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 8099E2h
R5 = 35h = 53
R1 = 42h = 66
AR2 = 8098F0h
IR1 = 0Fh
Data at 8099E1h = 0FFFFFFCBh = – 53
Data at 8098FFh = 42h = 66
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **ADDC** *src*, *dst*

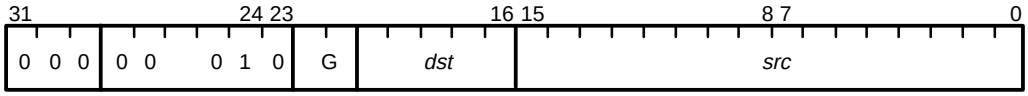
Operation *dst* + *src* + C → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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:

R1 = 00FFFF5C25h = – 41,947
R5 = 00FFFF019Eh = – 65,122
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 00FFFF5C25h = – 41,947
R5 = 00FFFE5DC4h = – 107,068
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **ADDC3** *src2*, *src1*, *dst*

Operation *src1* + *src2* + C → *dst*

Operands

src1 three-operand addressing modes (T):

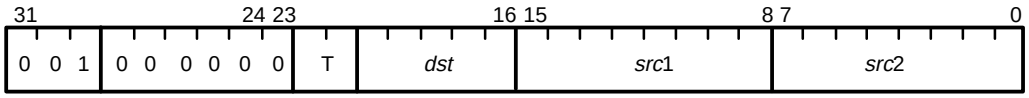
0 0 any CPU register
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 any CPU register
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-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

Encoding



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:

AR5 = 809908h
IR0 = 10h
R5 = 066h = 102
R2 = 0h
Data at 809908h = 0FFFFFFCBh = – 53
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

AR5 = 809918h
IR0 = 10h
R5 = 066h = 102
R2 = 032h = 50
Data at 809908h = 0FFFFFFCBh = – 53
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Example 2

```
ADDC3  R2, R7, R0
```

Before Instruction:

R2 = 02BCh = 700
R7 = 0F82h = 3970
R0 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R2 = 02BCh = 700
R7 = 0F82h = 3970
R0 = 0123Fh = 4671
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **ADDF** *src*, *dst*

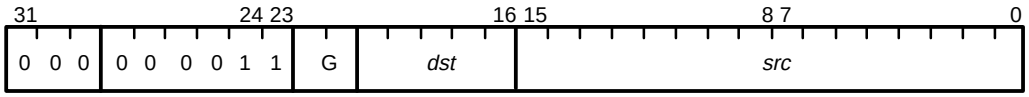
Operation *dst* + *src* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

AR4 = 809800h
IR1 = 12Bh
R5 = 0579800000h = 6.23750e+01
Data at 809800h = 86B2800h = 4.7031250e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 80992Bh
IR1 = 12Bh
R5 = 09052C0000h = 5.3268750e+02
Data at 809800h = 86B2800h = 4.7031250e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **ADDF3** *src2*, *src1*, *dst*

Operation *src1* + *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

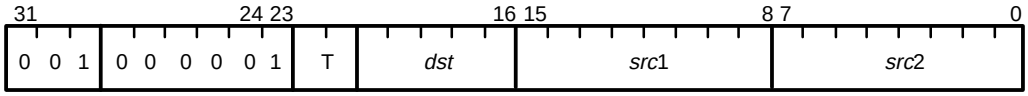
0 0 register (Rn1, 0 ≤ n1 ≤ 7)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 7)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 7)
0 1 register (Rn2, 0 ≤ n2 ≤ 7)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

R6 = 086B280000h = 4.7031250e + 02
R5 = 0579800000h = 6.23750e+01
R1 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R6 = 086B280000h = 4.7031250e + 02
R5 = 0579800000h = 6.23750e + 01
R1 = 09052C0000h = 5.3268750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
ADDF3  *+AR1(1),*AR7++(IR0),R4
```

Before Instruction:

AR1 = 809820h
AR7 = 8099F0h
IR0 = 8h
R4 = 0h
Data at 809821h = 700F000h = 1.28940e + 02
Data at 8099F0h = 34C2000h = 1.27590e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809820h
AR7 = 8099F8h
IR0 = 8h
R4 = 070DB20000h = 1.41695313e + 02
Data at 809821h = 700F000h = 1.28940e + 02
Data at 8099F0h = 34C2000h = 1.27590e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
    ADDF3   src2, src1, dst1
||  STF     src3, dst2
```

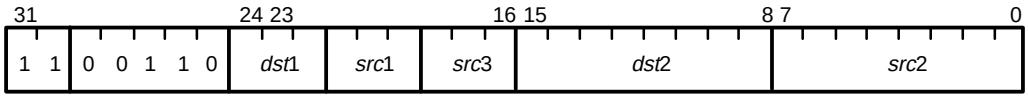
Operation

src1 + *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (Rn1, 0 ≤ n1 ≤ 7)
src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn2, 0 ≤ n2 ≤ 7)
src3	register (Rn3, 0 ≤ n3 ≤ 7)
dst2	indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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:

AR3 = 809800h
IR1 = 0A5h
R2 = 070C800000h = 1.4050e + 02
R5 = 0h
R4 = 057B400000h = 6.281250e + 01
AR2 = 8098F3h
Data at 8098A5h = 733C000h = 1.79750e + 02
Data at 8098F3h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809800h
IR1 = 0A5h
R2 = 070C800000h = 1.4050e+02
R5 = 0820200000h = 3.20250e + 02
R4 = 057B400000h = 6.281250e + 01
AR2 = 8098F3h
Data at 8098A5h = 733C000h = 1.79750e + 02
Data at 8098F3h = 57B4000h = 6.28125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **ADDI** *src, dst*

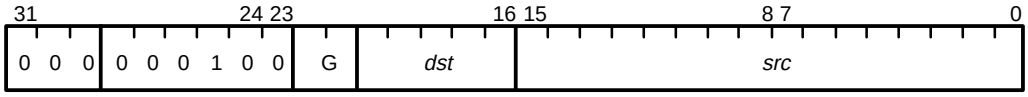
Operation *dst* + *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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:

R3 = 0FFFFFFCBh = – 53
R7 = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 0FFFFFFCBh = – 53
R7 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **ADDI3** *<src2 >,<src1 >,<dst >*

Operation *src1* + *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

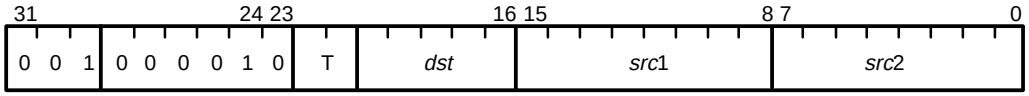
0 0 any CPU register
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 any CPU register
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-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

Encoding



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:

R4 = 0DCh = 220
R7 = 0A0h = 160
R5 = 10h = 16
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 0DCh = 220
R7 = 0A0h = 160
R5 = 017Ch = 380
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
ADDI3  *-AR3(1),*AR6--(IR0),R2
```

Before Instruction:

AR3 = 809802h
AR6 = 809930h
IR0 = 18h
R2 = 10h = 16
Data at 809801h = 2AF8h = 11,000
Data at 809930h = 3A98h = 15,000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809802h
AR6 = 809918h
IR0 = 18h
R2 = 06598h = 26,000
Data at 809801h = 2AF8h = 11,000
Data at 809930h = 3A98h = 15,000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
    ADDI3   src2, src1, dst1
||  STI     src3, dst2
```

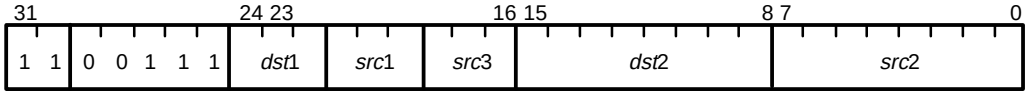
Operation

$src1 + src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src1	register (Rn1, $0 \leq n1 \leq 7$)
src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn2, $0 \leq n2 \leq 7$)
src3	register (Rn3, $0 \leq n3 \leq 7$)
dst2	indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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 as input the contents of the register 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:

AR0 = 80992Ch
IR0 = 0Ch
R5 = 0DCh = 220
R0 = 0h
R3 = 35h = 53
AR7 = 80983Bh
Data at 80992Ch = 12Ch = 300
Data at 80983Bh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 809920h
IR0 = 0Ch
R5 = 0DCh = 220
R0 = 208h = 520
R3 = 35h = 53
AR7 = 80983Bh
Data at 80992Ch = 12Ch = 300
Data at 80983Bh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **AND** *src, dst*

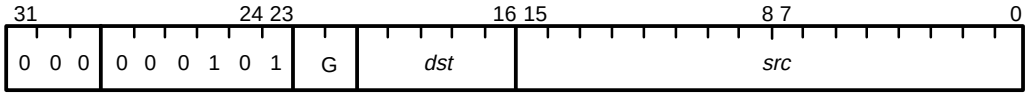
Operands *dst* AND *src* → *dst*

Operands *src* general addressing modes (G):

0 0 any CPU register
0 1 direct
1 0 indirect
1 1 immediate (not sign-extended)

dst any CPU register

Encoding



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:

R1 = 80h
R2 = 0AFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R1 = 80h
R2 = 80h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax **AND3** *src2, src1, dst*

Operation *src1* AND *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

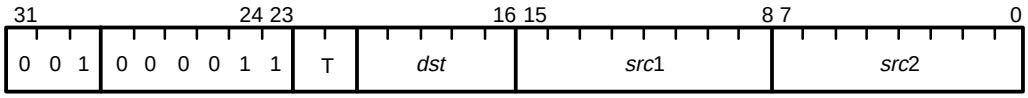
0 0 any CPU register
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 any CPU register
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-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

Encoding



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:

AR0 = 8098F4h
IR0 = 50h
AR1 = 809951h
R4 = 0h
Data at 8098F4h = 30h
Data at 809952h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 8098A4h
IR0 = 50h
AR1 = 809951h
R4 = 020h
Data at 8098F4h = 30h
Data at 809952h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
AND3    *–AR5,R7,R4
```

Before Instruction:

AR5 = 80985Ch
R7 = 2h
R4 = 0h
Data at 80985Bh = 0AFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 80985Ch
R7 = 2h
R4 = 2h
Data at 80985Bh = 0AFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
    AND3  src2, src1, dst1
||  STI   src3, dst2
```

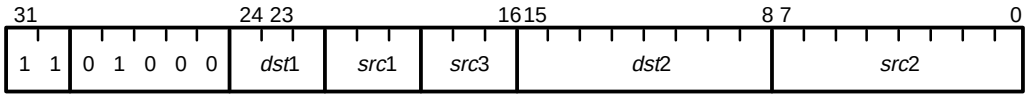
Operation

src1 AND *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (Rn1, 0 ≤ n1 ≤ 7)
src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn2, 0 ≤ n2 ≤ 7)
src3	register (Rn3, 0 ≤ n3 ≤ 7)
dst2	indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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 as input the contents of the register 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:

AR1 = 8099F1h
IR0 = 8h
R4 = 0A323h
R7 = 0h
R3 = 35h = 53
AR2 = 80983Fh
Data at 8099F9h = 5C53h
Data at 80983Fh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 8099F1h
R0 = 8h
R4 = 0A323h
R7 = 03h
R3 = 35h = 53
AR2 = 80983Fh
Data at 8099F9h = 5C53h
Data at 80983Fh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **ANDN** *src*, *dst*

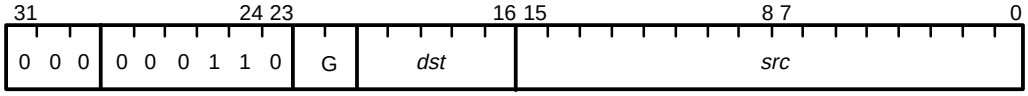
Operation *dst* AND ~*src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate (not sign-extended)

dst any CPU register

Encoding



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:

DP = 80h
R2 = 0C2Fh
Data at 80980Ch = 0A02h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R2 = 042Dh
Data at 80980Ch = 0A02h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **ANDN3** *src2, src1, dst*

Operation *src1* AND ~*src2* → *dst*

Operands *src1* three-operand addressing modes (T):

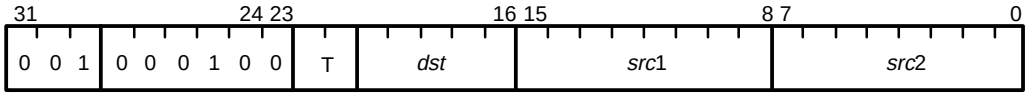
0 0 any CPU register
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 any CPU register
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-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 (Rn, 0 ≤ n ≤ 27)

Encoding



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:

R5 = 0A02h
R3 = 0C2Fh
R7 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0A02h
R3 = 0C2Fh
R7 = 042Dh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
ANDN3 R1,*AR5++(IR0),R0
```

Before Instruction:

R1 = 0CFh
AR5 = 809825h
IR0 = 5h
R0 = 0h
Data at 809825h = 0FFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0CFh
AR5 = 80982Ah
IR0 = 5h
R0 = 0F30h
Data at 809825h = 0FFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **ASH** *count, dst*

Operation

If (count ≥ 0):

dst << *count* → *dst*

Else:

dst >> |*count* | → *dst*

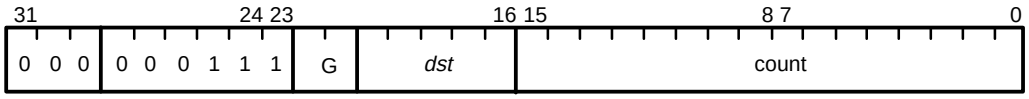
Operands

count general addressing modes (G):

0 0 any CPU register
0 1 direct
1 0 indirect
1 1 immediate

dst any CPU register

Encoding



Description

The seven least significant bits of the *count* operand are used to generate the two's 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 shifted in are 0-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:

R1 = 10h = 16
R3 = 0AE000h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 10h
R3 = 0E0000000h
LUF LV UF N Z V C = 0 1 0 1 0 1 0

Example 2

```
ASH @98C3h,R5
```

Before Instruction:

DP = 80h
R5 = 0AEC00001h
Data at 8098C3h = 0FFE8 = – 24
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R5 = 0FFFFFFAEh
Data at 8098C3h = 0FFE8 = – 24
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax **ASH3** *count, src, dst*

Operation

If (*count* ≥ 0):

src << *count* → *dst*

Else:

src >> |*count* | → *dst*

Operands

count three-operand addressing modes (T):

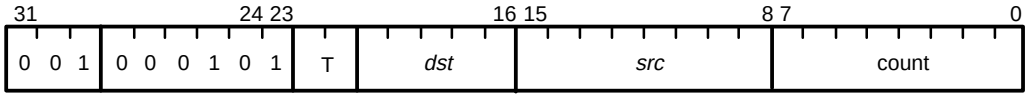
0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

src three-operand addressing modes (T):

0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description

The seven least significant bits of the count operand are used to generate the two's 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 shifted in are 0-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

```
ASH3   *AR3––(1),R5,R0
```

Before Instruction:

AR3 = 809921h
R5 = 02B0h
R0 = 0h
Data at 809921h = 10h = 16
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809920h
R5 = 000002B0h
R0 = 02B00000h
Data at 809921h = 10h = 16
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example ASH3 R1,R3,R5

Before Instruction:

R1 = 0FFFFFFF8h = – 8
R3 = 0FFFFCB00h
R5 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0FFFFFFF8h = – 8
R3 = 0FFFFCB00h
R5 = 0FFFFFFCBh
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

ASH3 *count, src2, dst1*
|| **STI** *src3, dst2*

Operation

If (*count* ≥ 0):
src2 << *count* → *dst1*

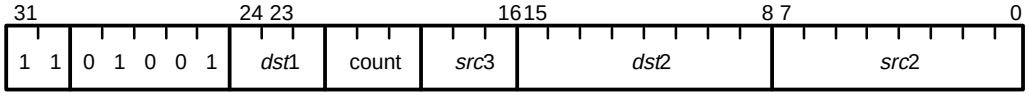
Else:
src2 >> |*count*| → *dst1*

|| *src3* → *dst2*

Operands

count register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

The seven least significant bits of the *count* operand register are used to generate the two's 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 0-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-shifted by the absolute value of the *count* operand. The high-order bits of the *src2* operand are sign-extended as it is 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. This means that, 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 as input the contents of the register 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:

AR6 = 809900h
IR1 = 8Ch
R1 = 0FFE8h = – 24
R0 = 0h
R5 = 35h = 53
AR2 = 8098A2h
Data at 809900h = 0AE000000h
Data at 8098A2h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR6 = 80998Ch
IR1 = 8Ch
R1 = 0FFE8h = – 24
R0 = 0FFFFFFAEh
R5 = 35h = 53
AR2 = 8098A2h
Data at 809900h = 0AE000000h
Data at 8098A2h = 35h = 53
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

B***cond*** *src*

Operation

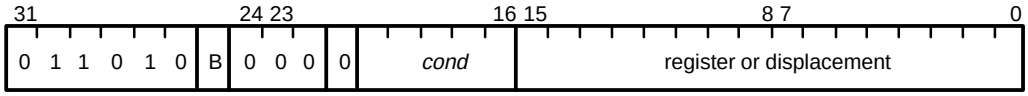
If *cond* is true:
  If *src* is in register-addressing mode (Rn, $0 \leq n \leq 27$),
    src $\rightarrow$ PC.
  If *src* is in PC-relative mode (label or address),
    displacement + PC + 1 $\rightarrow$ PC.
  Else, continue.

Operands

src conditional-branch addressing modes (B):

0	register
1	PC-relative

Encoding



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 9.2 on page 9-4). 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 least significant bits of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The TMS320C3x provides 20 condition codes that you can use with this instruction (see Table 10–9 on page -13 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:

PC = 2B00h
R0 = 0003FF00h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 3FF00h
R0 = 0003FF00h
LUF LV UF N Z V C = 0 0 0 0 0 0 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

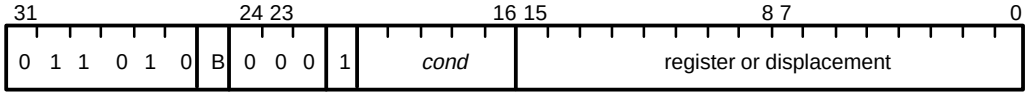
If *cond* is true:
  If *src* is in register-addressing mode (Rn, $0 \le n \le 27$),
    src $\rightarrow$ PC.
  If *src* is in PC-relative mode (label or address),
    displacement + PC + 3 $\rightarrow$ PC.
  Else, continue.

Operands

src conditional-branch addressing modes (B):

0	register
1	PC-relative

Encoding



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 will not affect the *cond*.

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 least significant bits of the branch instruction. This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. The TMS320C3x provides 20 condition codes that you can use with this instruction (see Table 10–9 on page -13 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

```
BNZD 36 (36 = 24h)
```

Before Instruction:

```
PC = 50h
LUF LV  UF  N  Z  V  C = 0  0  0  0  0  0  0
```

After Instruction:

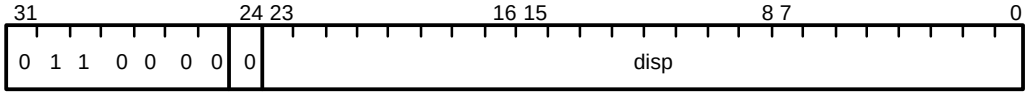
```
PC = 77h
LUF LV  UF  N  Z  V  C = 0  0  0  0  0  0  0
```

Syntax **BR** *src*

Operation *src* → PC or PC + disp → PC, where disp = src – (PC + 1)

Operands *src* long-immediate addressing mode

Encoding



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 9.2 on page 9-4. 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:

PC = 80h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

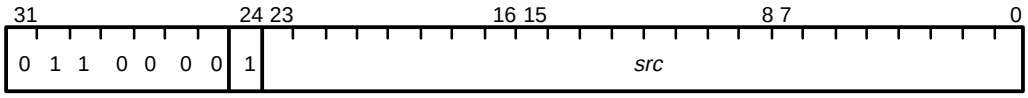
PC = 805Ch
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **BRD** *src*

Operation *src* → PC

Operands *src* long-immediate addressing mode

Encoding



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:

PC = 1Bh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

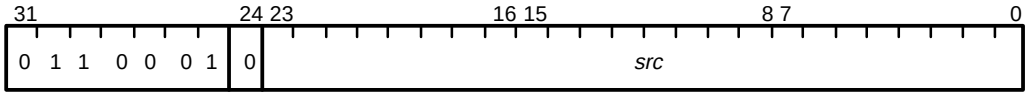
PC = 2Ch
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **CALL** *src*

Operation Next PC → *++SP
src → PC

Operands *src* long-immediate addressing mode

Encoding



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.

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:

PC = 5h
SP = 809801h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 123456h
SP = 809802h
Data at 809802h = 6h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

CALL*cond* *src*

Operation

If *cond* is true:
Next PC → *++SP
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.

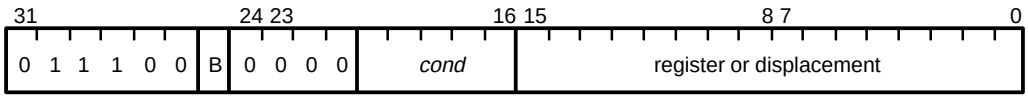
Else, continue.

Operands

src conditional-branch addressing modes (B):

0	register
1	PC-relative

Encoding



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 least significant bits of the call instruction word. This displacement is added to the PC of the call instruction plus 1 to generate the new PC.

The TMS320C3x provides 20 condition codes that can be used with this instruction (see Table 10–9 on page -13 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:

PC = 123h
SP = 809835h
R5 = 789h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 789h
SP = 809836h
R5 = 789h
Data at 809836h = 124h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **CMPF** *src, dst*

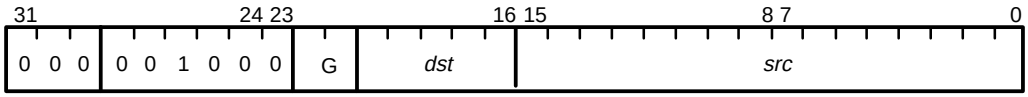
Operation *dst – src*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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 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:

AR4 = 8098F2h
R6 = 070C800000h = 1.4050e+02
Data at 8098F3h = 070C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8098F2h
R6 = 070C800000h = 1.4050e + 02
Data at 8098F3h = 070C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 1 0 0

Syntax **CMPF3** *src2, src1*

Operation *src1 – src2*

Operands

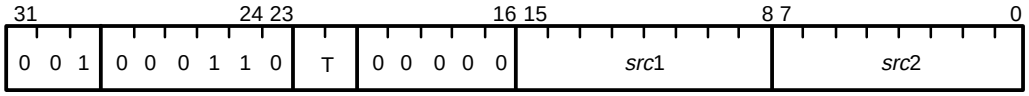
src1 three-operand addressing modes (T):

0 0	register (Rn1, 0 ≤ n1 ≤ 7)
0 1	indirect (disp = 0, 1, IR0, IR1)
1 0	register (Rn1, 0 ≤ n1 ≤ 7)
1 1	indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0	register (Rn2, 0 ≤ n2 ≤ 7)
0 1	register (Rn2, 0 ≤ n2 ≤ 7)
1 0	indirect (disp = 0, 1, IR0, IR1)
1 1	indirect (disp = 0, 1, IR0, IR1)

Encoding



Description The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register, thus allowing 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 three-operand instruction because operands are specified in the three-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:

AR2 = 809831h
AR3 = 809852h
Data at 809831h = 77A7000h = 2.5044e + 02
Data at 809852h = 57A2000h = 6.253125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809831h
AR3 = 809851h
Data at 809831h = 77A7000h = 2.5044e + 02
Data at 809852h = 57A2000h = 6.253125e + 01
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **CMPI** *src, dst*

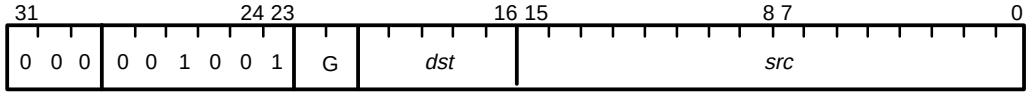
Operation *dst* – *src*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 27)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

R3 = 898h = 2200
R7 = 3E8h = 1000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 898h = 2200
R7 = 3E8h = 1000
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax **CMPI3** *src2, src1*

Operation *src1 – src2*

Operands

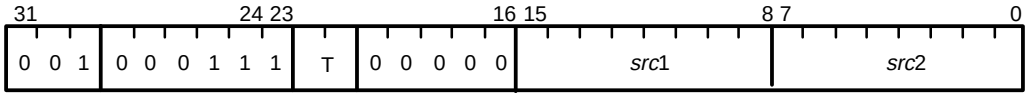
src1 three-operand addressing modes (T):

0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register, thus allowing 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 three-operand instruction because operands are specified in the three-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:

R7 = 03E8h = 1000
R4 = 0898h = 2200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 03E8h = 1000
R4 = 0898h = 2200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

DB*cond* ARn*, 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.

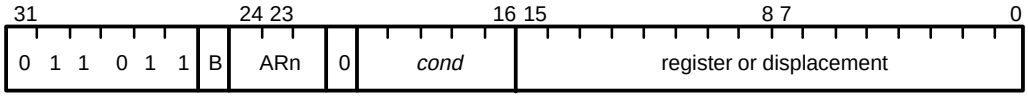
Operands

src conditional-branch addressing modes (B):

0	register
1	PC-relative

ARn register (0 ≤ n ≤ 7)

Encoding



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 most significant eight bits are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 24 least significant bits 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 least significant bits of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The TMS320C3x provides 20 condition codes that can be used with this instruction (see Table 10–9 on page -13 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:

PC = 5Fh
AR3 = 12h
R2 = 9Fh
R3 = 80h
LUF LV UF N Z V C = 0 0 0 1 0 0 0

After Instruction:

PC = 9Fh
AR3 = 11h
R2 = 9Fh
R3 = 80h
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax

DB*cond*D ARn, *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 + 3 → PC.

Else, continue.

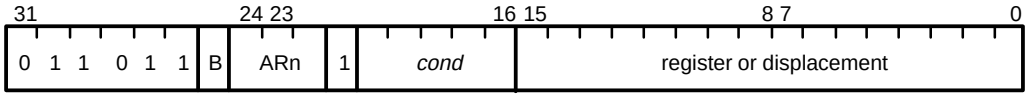
Operands

src conditional-branch addressing modes (B):

0	register
1	PC-relative

ARn register (0 ≤ n ≤ 7)

Encoding



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 *cond*.

The auxiliary register is treated as a 24-bit signed integer. The most significant eight bits are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 24 least significant bits 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* 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 TMS320C3x provides 20 condition codes that you can use with this instruction (see Table 10–9 on page 10-13 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:

PC = 100h
R2 = 26h
AR5 = 67h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 210h
R2 = 26h
AR5 = 66h
LUF LV UF N Z V C = 0 0 0 0 1 0 0

Syntax **FIX** *src, dst*

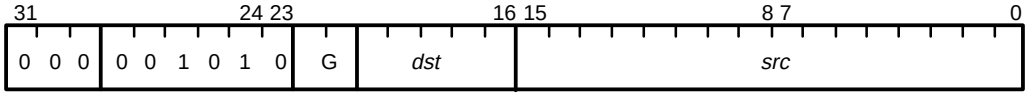
Operation fix(*src*) → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



Description The floating-point operand *src* is converted to the nearest integer less than or equal to it in 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 result register (if it has one) is not modified.

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit two's complement integer. In the case of integer overflow, the result will be 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:

R1 = 0A28200000h = 1.3454e + 3
R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0A28200000h = 13454e + 3
R2 = 541h = 1345
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

FIX *src2, dst1*
|| **STI** *src3*, *dst2*

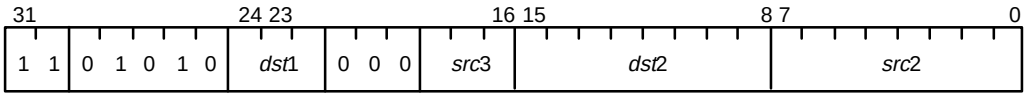
Operation

fix(*src2*) → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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 as input the contents of the register 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 two's complement integer. In the case of integer overflow, the result will be 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:

AR4 = 8098A2h
R1 = 0h
R0 = 0DCh = 220
AR2 = 80983Ch
Data at 8098A3h = 733C000h = 1.7950e + 02
Data at 80983Ch = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8098A3h
R1 = 0B3h = 179
R0 = 0DCh = 220
AR2 = 80983Ch
Data at 8098A3h = 733C000h = 1.79750e + 02
Data at 80983Ch = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **FLOAT** *src, dst*

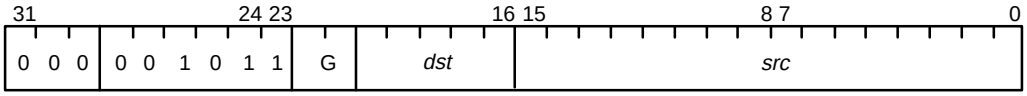
Operation float *(src)* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The integer operand *src* is converted to the floating-point value equal to it, and the result loaded into the *dst* register. The *src* operand is assumed to be a signed integer, and the *dst* operand 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:

AR2 = 809800h
R5 = 034C2000h = 1.27578125e + 01
Data at 809802h = 0AEh = 174
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809802h
R5 = 072E00000h = 1.74e + 02
Data at 809802h = 0AEh = 174
LUF LV UF N Z V C = 0 0 0 0 0 0 0

FLOAT||STF *Parallel FLOAT and STF*

Syntax

```
     FLOAT   src2, dst1
||   STF     src3, dst2
```

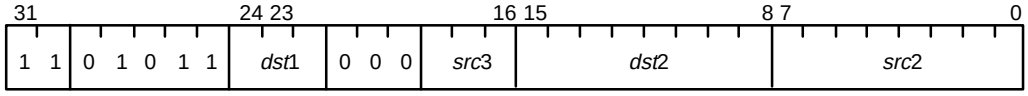
Operation

float(*src2*) → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 3 7)
dst2 register (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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 as input the contents of the register 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:

AR2 = 8098C5h
IR0 = 8h
R6 = 0h
R7 = 034C200000h = 1.27578125e + 01
AR1 = 809933h
Data at 8098CDh = 0AEh = 174
Data at 809933h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8098C5h
IR0 = 8h
R6 = 072E000000h = 1.740e + 02
R7 = 034C200000h = 1.27578125e + 01
AR1 = 809933h
Data at 8098CDh = 0AEh = 174
Data at 809933h = 034C2000h = 1.27578125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

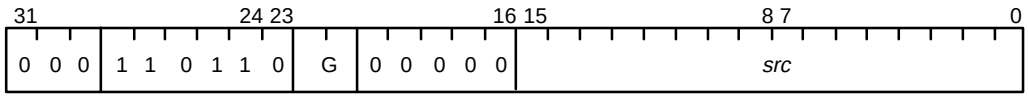
Syntax **IACK** *src*

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

Encoding



Description A dummy read operation is performed. If off-chip memory is specified, $\overline{\text{IACK}}$ is set to 0 at half H1 cycle after the beginning of the decode phase of the $\overline{\text{IACK}}$ instruction. At the first half of the H1 cycle of the dummy read, $\overline{\text{IACK}}$ is set to 1. Because of a multicycle read, the $\overline{\text{IACK}}$ signal will not be extended. 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:

$\overline{\text{IACK}}$ = 1
PC = 300h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

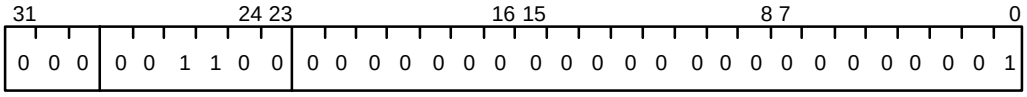
$\overline{\text{IACK}}$ = 1
PC = 301h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **IDLE**

Operation 1 → ST(GIE)
Next PC → PC
Idle until interrupt.

Operands None

Encoding



Description The global interrupt enable bit is set, the next PC value is loaded into the PC, and the CPU idles until an interrupt is received. When the interrupt is received, the contents of the PC are pushed onto the active system stack.

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

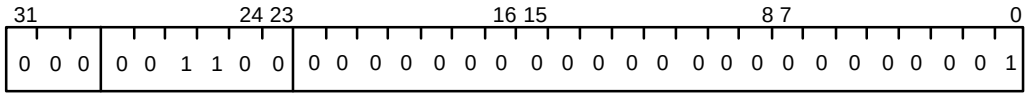
```
IDLE    ; The processor idles until a reset
        ; or unmasked interrupt occurs.
```

Syntax **IDLE2** (TMS320LC31 Only)

Operation
1 → ST(GIE)
Next PC → PC
Idle until interrupt.

Operands None

Encoding



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 ’C31 will behave as follows:

- The CPU, peripherals, and memory will retain their previous states.
- When the device is in the functional (nonemulation) mode, the clocks will stop with H1 high and H3 low.
- The ’LC31 will remain in IDLE2 until one of the four external interrupts ($\overline{INT3}$–$\overline{INT0}$) 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, for one of the four external interrupts to be recognized by the CPU and serviced, it must be asserted for at least two H1 cycles. For the processor to recognize only one interrupt when it restarts operation, the interrupt must be asserted for less than three cycles.
- When the ’LC31 is in emulation mode, the H1 and H3 clocks will continue to run normally, and the CPU will operate as if an IDLE instruction had been executed. The clocks continue to run for correct operation of the emulator.

Delayed Branch

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 unmasked interrupt occurs.
```

Syntax **LDE** *src, dst*

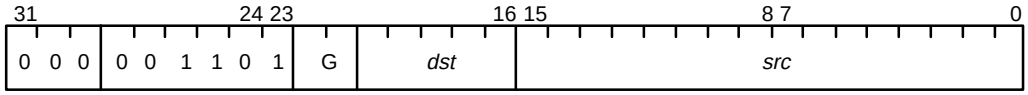
Operation *src*(exp) → *dst*(exp)

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

R0 = 0200056F30h = 4.00066337e + 00
R5 = 0A056FE332h = 1.06749648e + 03
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0200056F30h = 4.00066337e + 00
R5 = 02056FE332h = 4.16990814e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **LDF** *src, dst*

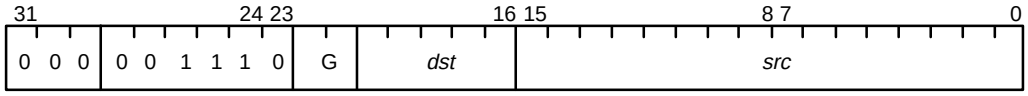
Operation *src* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

DP = 80h
R2 = 0h
Data at 809800h = 10C52A00h = 2.19254303e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R2 = 010C52A00h = 2.19254303e + 00
Data at 809800h = 10C52A00h = 2.19254303e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LDF*cond* *src, dst*

Operation

If *cond* is true:
src → *dst.*

Else:
dst is unchanged.

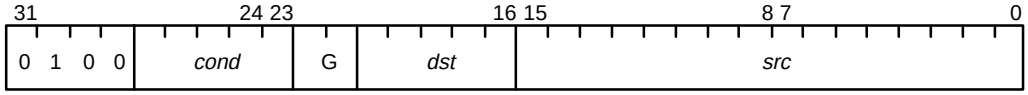
Operands

src general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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 TMS320C3x provides 20 condition codes that can be used with this instruction (see Table 10–9 on page 10-13 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:

R3 = 2CFF2CD500h = 1.77055560e +13
R5 = 5F0000003Eh = 3.96140824e + 28
LUF LV UF N Z V C = 0 0 0 0 1 0 0

After Instruction:

R3 = 2CFF2CD500h = 1.77055560e +13
R5 = 2CFF2CD500h = 1.77055560e +13
LUF LV UF N Z V C = 0 0 0 0 1 0 0

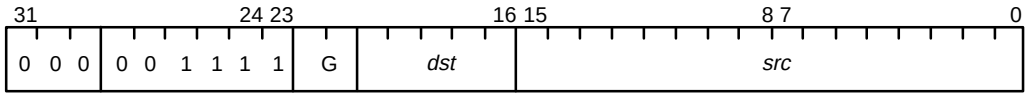
Syntax **LDFI** *src, dst*

Operation Signal interlocked operation
src → *dst*

Operands *src* general addressing modes (G):
0 1 direct
1 0 indirect

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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. Note that only direct and indirect modes are allowed. Refer to Section 6.4 on page 6-12 for detailed description.

Cycles 1 if XF1 = 0 (See Section 6.4 on page 6-12)

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:

AR2 = 8098F1h
R7 = 0h
Data at 8098F2h = 584C000h = – 6.28125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8098F1h
R7 = 0584C00000h = – 6.28125e + 01
Data at 8098F2h = 584C000h = – 6.28125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax

LDF *src2, dst2*
|| **LDF** *src1, dst1*

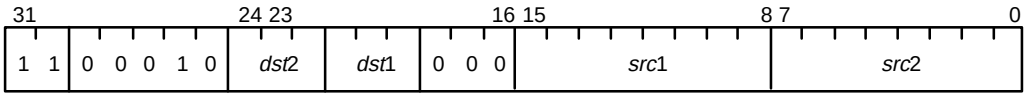
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn1, 0 ≤ n1 ≤ 7)
src2	indirect (disp = 0, 1, IR0, IR1)
dst2	register (Rn2, 0 ≤ n2 ≤ 7)

Encoding



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:

AR1 = 80985Fh
IR0 = 8h
R7 = 0h
AR7 = 80988Ah
R3 = 0h
Data at 809857h = 70C8000h = 1.4050e + 02
Data at 80988Ah = 57B4000h = 6.281250e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809857h
R0 = 8h
R7 = 070C800000h = 1.4050e + 02
AR7 = 80988Bh
R3 = 057B400000h = 6.281250e + 01
Data at 809857h = 70C8000h = 1.4050e + 02
Data at 80988Ah = 57B4000h = 6.281250e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

LDF *src2, dst1*
|| **STF** *src3, dst2*

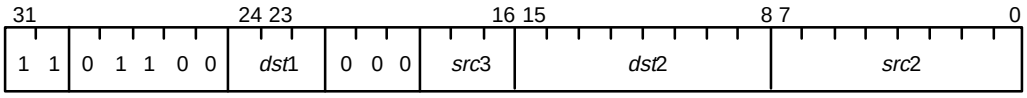
Operation

src2 → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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:

AR2 = 8098E7h
R1 = 0h
R3 = 057B400000h = 6.28125e + 01
AR4 = 809900h
IR1 = 10h
Data at 8098E7h = 70C8000h = 1.4050e + 02
Data at 809900h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8098E6h
R1 = 070C800000h = 1.4050e + 02
R3 = 057B400000h = 6.28125e + 01
AR4 = 809910h
IR1 = 10h
Data at 8098E7h = 70C8000h = 1.4050e + 02
Data at 809900h = 57B4000h = 6.28125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **LDI** *src, dst*

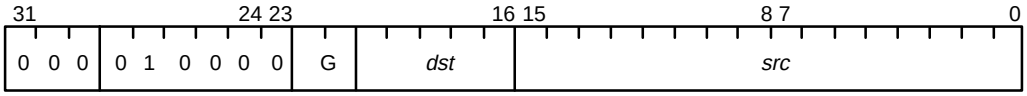
Operation *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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 and subsection 10.3.2 on page 10-16.

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:

AR1 = 2Ch
IR0 = 5h
R5 = 3C5h = 965
Data at 27h = 26h = 38
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 2Ch
IR0 = 5h
R5 = 26h = 38
Data at 27h = 26h = 38
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LDI*cond* *src, dst*

Operation

If *cond* is true:
src → *dst,*

Else:
dst is unchanged.

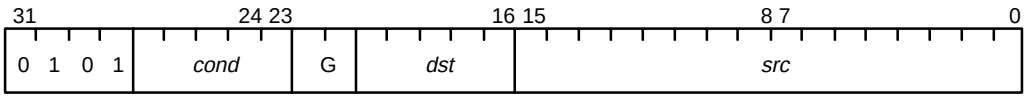
Operands

src general addressing modes (G):

0 0 any CPU register
0 1 direct
1 0 indirect
1 1 immediate

dst any CPU register

Encoding



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, the read of the *src* takes place. The *dst* and *src* operands are assumed to be signed integers.

The TMS320C3x provides 20 condition codes that can be used with this instruction (see Table 10–9 on page 10-13 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:

ARO = 8098FO
Data at 8098FOh = 027Ch = 636
R6 = 0FE2h = 4,066
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

ARO = 8098F1h
Data at 8098FOh = 027Ch = 636
R6 = 0FE2h = 4,066
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Auxiliary Register Arithmetic

The test condition does not affect the auxiliary register arithmetic. (AR modification will always occur.)

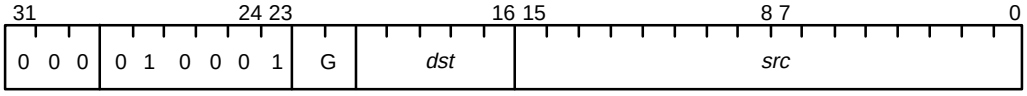
Syntax **LDII** *src, dst*

Operation Signal interlocked operation
src → *dst*

Operands *src* general addressing modes (G):
0 1 direct
1 0 indirect

dst any CPU register

Encoding



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. Refer to Section 6.4 on page 6-12 for detailed description.

Cycles 1 if XF = 0 (See Section 6.4 on page 6-12)

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:

DP = 80
R3 = 0h
Data at 80985Fh = 0DCh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80
R3 = 0DCH
Data at 80985Fh = 0DCh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LDI *src2, dst2*
|| **LDI** *src1, dst1*

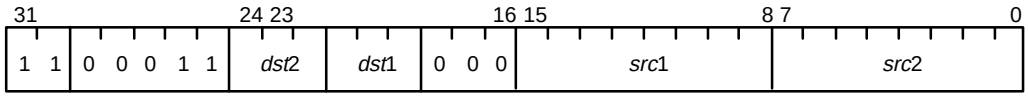
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn1, 0 ≤ n1 ≤ 7)
src2	indirect (disp = 0, 1, IR0, IR1)
dst2	register (Rn2, 0 ≤ n2 ≤ 7)

Encoding



Description

Two integer loads are performed in parallel. A warning is issued by the assembler 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:

AR1 = 809826h
R7 = 0h
AR7 = 8098C8h
IR0 = 10h
R1 = 0h
Data at 809825h = 0FAh = 250
Data at 8098C8h = 2EEh = 750
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809826h
R7 = 0FAh = 250
AR7 = 8098D8h
IR0 = 10h
R1 = 02EEh = 750
Data at 809825h = 0FAh = 250
Data at 8098C8h = 2EEh = 750
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

LDI *src2, dst1*
|| **STI** *src3, dst2*

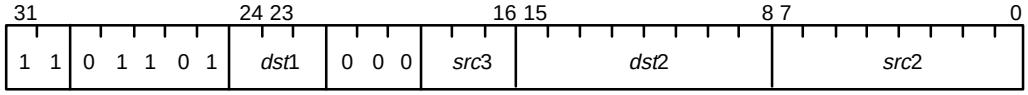
Operation

src2 → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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 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

```
   LDI *–AR1(1),R2
|| STI R7,*AR5++(IR0)
```

Before Instruction:

AR1 = 8098E7h
R2 = 0h
R7 = 35h = 53
AR5 = 80982Ch
IR0 = 8h
Data at 8098E6h = 0DCh = 220
Data at 80982Ch = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 8098E7h
R2 = 0DCh = 220
R7 = 35h = 53
AR5 = 809834h
IR0 = 8h
Data at 8098E6h = 0DCh = 220
Data at 80982Ch = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **LDM** *src, dst*

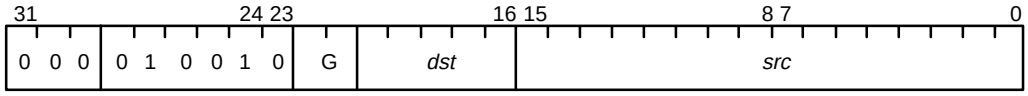
Operation *src* (man) → *dst* (man)

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

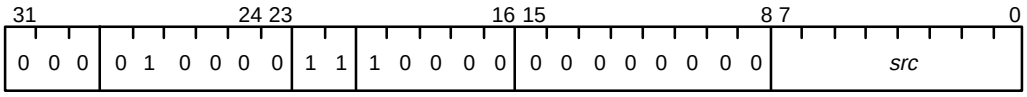
R2 = 001CC00000h = 1.22460938e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

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.

Encoding



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:

DP = 65h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

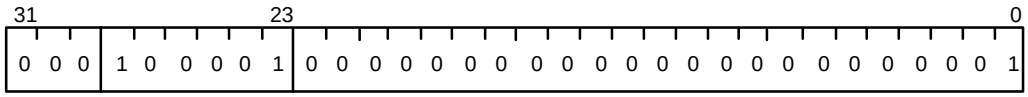
LOPOWER *Divide Clock by 16*

Syntax **LOPOWER** (TMS320LC31 Only)

Operation *H1/16* → H1

Operands None

Encoding



Description Device continues to execute instructions, but at the reduced rate of the CLKIN frequency divided by 16 (that is, in LOPOWER mode, an ’LC31 with a CLKIN frequency of 32 MHz will perform in the same way as a 2-MHz ’LC31, which has an instruction cycle time of 1000 ns). This allows for low-power operation.

The ’LC31 CPU slows 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 ’LC31 resumes full-speed operation during the read phase of the MAXSPEED instruction.

Delayed Branch

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*

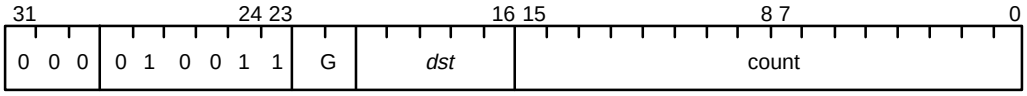
Operands

count general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



Description

The seven least significant bits of the *count* operand are used to generate the two's 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 0-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 0-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:

R4 = 018h = 24
R7 = 02ACh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 018h = 24
R7 = 0AC000000h
LUF LV UF N Z V C = 0 0 0 1 0 1 0

Example 2

```
LSH *–AR5(IR1),R5
```

Before Instruction:

AR5 = 809908h
IR0 = 4h
R5 = 0012C00000h
Data at 809904h = 0FFFFFFF4h = –12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 809908h
IR0 = 4h
R5 = 0000012C00h
Data at 809904h = 0FFFFFFF4h = –12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LSH3 *count, src, dst*

Operation

If *count* ≥ 0:
src << *count* → *dst*

Else:
src >> |*count*| → *dst*

Operands

src three-operand addressing modes (T):

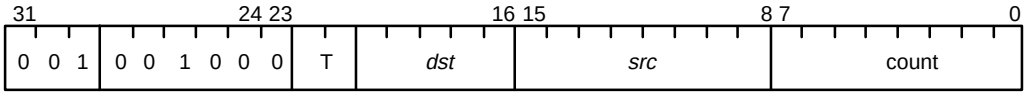
0 0	any CPU register
0 1	indirect (disp = 0, 1, IR0, IR1)
1 0	any CPU register
1 1	indirect (disp = 0, 1, IR0, IR1)

count three-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 (Rn, 0 ≤ n ≤ 27)

Encoding



Description

The seven least significant bits of the *count* operand are used to generate the two's 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 0-filled, and high-order bits are shifted out through the C (carry) 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:

R4 = 018h = 24
R7 = 02ACh
R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 018h = 24
R7 = 02ACh
R2 = 0AC000000h
LUF LV UF N Z V C = 0 0 0 1 0 1 0

Example 2

```
LSH3 *–AR4(IR1),R5,R3
```

Before Instruction:

AR4 = 809908h
IR1 = 4h
R5 = 012C00000h
R3 = 0h
Data at 809904h = 0FFFFFFF4h = –12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 809908h
IR1 = 4h
R5 = 012C00000h
R3 = 0000012C00h
Data at 809904h = 0FFFFFFF4h = –12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

LSH3 c*ount, src2, dst1*
|| **STI** *src3*, *dst2*

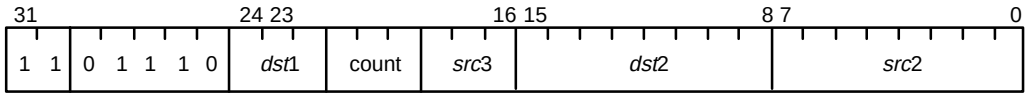
Operation

If *count* ≥ 0:
src2 << *count* → *dst1*
Else:
src2 >> |*count* | → *dst1*
|| *src3* → *dst2*

Operands

count register (Rn1, 0 ≤ n1 ≤ 7)
src1 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn3, 0 ≤ n3 ≤ 7)
src2 register (Rn4, 0 ≤ n4 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

The seven least significant bits of the *count* operand are used to generate the two's 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 0-filled, and high-order bits are shifted out through the C (carry) 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 (carry 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 seven-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 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 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:

R2 = 18h = 24
AR3 = 8098C2h
R0 = 0h
R4 = 0DCh = 220
AR5 = 8098A3h
Data at 8098C3h = 0ACh
Data at 8098A2h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 18h = 24
AR3 = 8098C3h
R0 = 0AC000000h
R4 = 0DCh = 220
AR5 = 8098A3h
Data at 8098C3h = 0ACh
Data at 8098A2h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 1 0 1 0

Example 2

```
   LSH3  R7,*AR2--(1),R2
||STI   R0,*+AR0(1)
```

Before Instruction:

R7 = 0FFFFFFF4h = –12
AR2 = 809863h
R2 = 0h
R0 = 12Ch = 300
AR0 = 8098B7h
Data at 809863h = 2C000000h
Data at 8098B8h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 0FFFFFFF4h = –12
AR2 = 809862h
R2 = 2C000h
R0 = 12Ch = 300
AR0 = 8098B7h
Data at 809863h = 2C000000h
Data at 8098B8h = 12Ch = 300
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

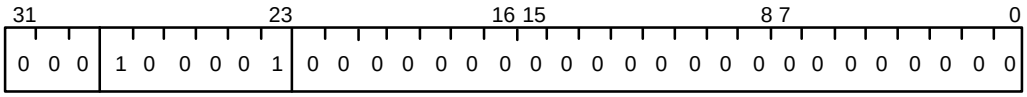
See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **MAXSPEED**

Operation *H1/16 → H1*

Operands None

Encoding



Description Exits LOPOWER power-down mode (invoked by LOPOWER instruction with opcode 10800001h). The ’LC31 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*

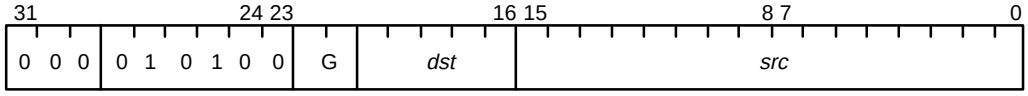
Operation *dst* × *src* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

R0 = 070C800000h = 1.4050e + 02
R2 = 034C200000h = 1.27578125e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 070C800000h = 1.4050e + 02
R2 = 0A600F2000h = 1.79247266e + 03
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **MPYF3** *src2, src1, dst*

Operation *src1* × *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

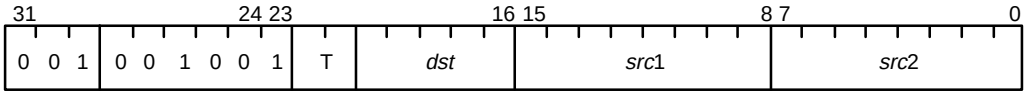
0 0 register (Rn1, 0 ≤ n1 ≤ 7)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 7)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 7)
0 1 register (Rn2, 0 ≤ n2 ≤ 7)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

R0 = 057B400000h = 6.281250e + 01
R7 = 0733C00000h = 1.79750e + 02
R1 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 057B400000h = 6.281250e + 01
R7 = 0733C00000h = 1.79750e + 02
R1 = 0D306A3000h = 1.12905469e + 04
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
MPYF3 *+AR2(IR0),R7,R2
```
or
```
MPYF3 R7,*+AR2(IR0),R2
```

Before Instruction:

AR2 = 809800h
IR0 = 12Ah
R7 = 057B400000h = 6.281250e + 01
R2 = 0h
Data at 80992Ah = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809800h
IR0 = 12Ah
R7 = 057B400000h = 6.281250e + 01
R2 = 0D09E4A000h = 8.82515625e + 03
Data at 80992Ah = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

MPYF3 *srcA, srcB, dst1*
|| **ADDF3** *srcC, srcD, dst2*

Operation

srcA × *srcB* → *dst1*
|| *srcC* + *srcD* → *dst2*

Operands

srcA *srcB*	Any two indirect (disp = 0,1,IR0,IR1)
srcC *srcD*	Any two register ($0 \le Rn \le 7$)

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

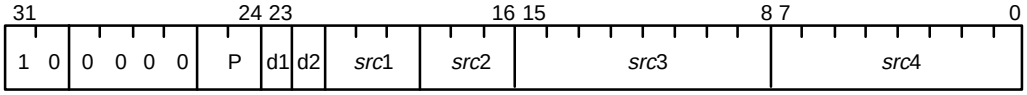
src1	register	(Rn, $0 \le n \le 7$)
src2	register	(Rn, $0 \le n \le 7$)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P parallel addressing modes ($0 \le P \le 3$)

Operation (P Field)

00	*src3* × *src4, src1* + *src2*
01	*src3* × *src1, src4* + *src2*
10	*src1* × *src2, src3* + *src4*
11	*src3* × *src1, src2* + *src4*

Encoding



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. This means that 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 as input the contents of the register 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 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

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
```

Before Instruction:

AR5 = 8098C5h
AR1 = 8098A8h
IR0 = 4h
R0 = 0h
R5 = 0733C00000h = 1.79750e + 02
R7 = 070C800000h = 1.4050e + 02
R3 = 0h
Data at 8098C5h = 34C0000h = 1.2750e + 01
Data at 8098A4h = 1110000h = 2.265625e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

After Instruction:

AR5 = 8098C6h
AR1 = 8098A4h
IR0 = 4h
R0 = 0467180000h = 2.88867188e + 01
R5 = 0733C00000h = 1.79750e + 02
R7 = 070C800000h = 1.4050e + 02
R3 = 0820200000h = 3.20250e + 02
Data at 8098C5h = 34C0000h = 1.2750e + 01
Data at 8098A4h = 1110000h = 2.265625e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYF3 *src2, src1, dst*
|| **STF** *src3, dst2*

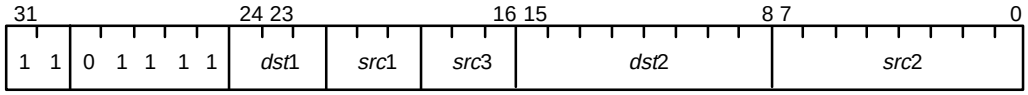
Operation

src1 × *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn3, 0 ≤ n3 ≤ 7)
src3 register (Rn4, 0 ≤ n4 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that if one of the parallel operations (MPYF3) writes to a register and the operation being performed in parallel (STF) reads from the same register, the STF accepts as input the contents of the register 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:

AR2 = 80982Bh
R7 = 057B400000h = 6.281250e + 01
R0 = 0h
R3 = 086B280000h = 4.7031250e + 02
AR0 = 809860h
IR0 = 8h
Data at 80982Ah = 70C8000h = 1.4050e + 02
Data at 809860h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 80982Bh
R7 = 057B400000h = 6.281250e + 01
R0 = 0D09E4A000h = 8.82515625e + 03
R3 = 086B280000h = 4.7031250e + 02
AR0 = 809858h
IR0 = 8h
Data at 80982Ah = 70C8000h = 1.4050e + 02
Data at 809860h = 86B280000h = 4.7031250e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on 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 ≤ Rn ≤ 7)

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

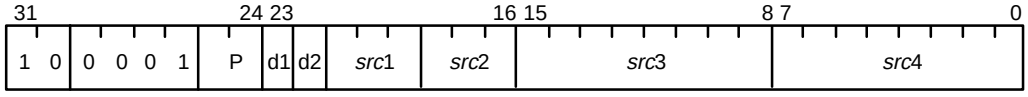
src1	register	(Rn, 0 ≤ n ≤ 7)
src2	register	(Rn, 0 ≤ n ≤ 7)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

Operation (P Field)

00	*src3* × *src4, src1* – *src2*
01	*src3* × *src1, src4* – *src2*
10	*src1* × *src2, src3* – *src4*
11	*src3* × *src1, src2* – *src4*

Encoding



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. This means that 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

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:

R5 = 034C000000h = 1.2750e + 01
AR7 = 809904h
IR1 = 8h
R0 = 0h
R7 = 0733C00000h = 1.79750e + 02
AR3 = 8098B2h
R2 = 0h
Data at 80990Ch = 1110000h = 2.250e + 00
Data at 8098B2h = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 034C000000h = 1.2750e + 01
AR7 = 80990Ch
IR1 = 8h
R0 = 0467180000h = 2.88867188e + 01
R7 = 0733C00000h = 1.79750e + 02
AR3 = 8098B1h
R2 = 05E3000000h = – 3.9250e + 01
Data at 80990Ch = 1110000h = 2.250e + 00
Data at 8098B2h = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **MPYI** *src, dst*

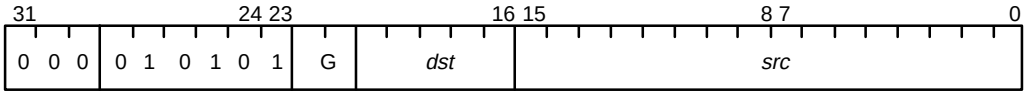
Operation *dst* × *src* → *dst*

Operands *src* general addressing modes (G):

0 0 any CPU register
0 1 direct
1 0 indirect
1 1 immediate

dst any CPU register

Encoding



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 least significant bits of the result.

Integer overflow occurs when any of the most significant 16 bits of the 48-bit result differs from the most significant bit 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:

R1 = 000033C251h = 3,392,081
R5 = 000078B600h = 7,910,912
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 000033C251h = 3,392,081
R5 = 00E21D9600h = –501,377,536
LUF LV UF N Z V C = 0 1 0 1 0 1 0

Syntax **MPYI3** *src2, src1, dst*

Operation *src1* × *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

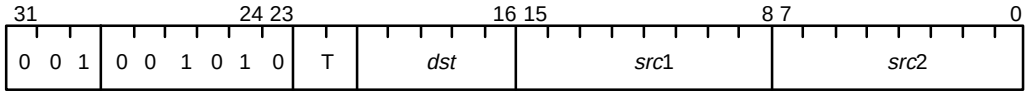
0 0 any CPU register
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 any CPU register
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-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 (Rn, 0 ≤ n ≤ 27)

Encoding



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 least significant bits of the result.

Integer overflow occurs when any of the most significant 16 bits of the 48-bit result differs from the most significant bit 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:

AR4 = 809850h
AR1 = 8098F3h
R2 = 0h
Data at 809850h = 0ADh = 173
Data at 8098F2h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 809850h
AR1 = 8098F3h
R2 = 094ACh = 38,060
Data at 809850h = 0ADh = 173
Data at 8098F2h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
MPYI3   *––AR4(IR0),R2,R7
```

Before Instruction:

AR4 = 8099F8h
IR0 = 8h
R2 = 0C8h = 200
R7 = 0h
Data at 8099F0h = 32h = 50
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8099F0h
IR0 = 8h
R2 = 0C8h = 200
R7 = 02710h = 10,000
Data at 8099F0h = 32h = 50
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

MPYI3 *srcA, srcB, dst1*
|| **ADDI3** *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 ($0 \le Rn \le 7$)

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

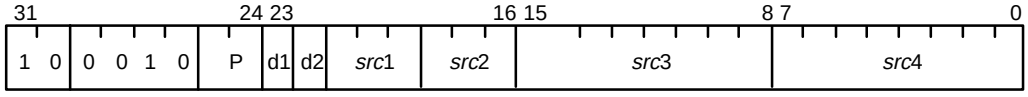
src1	register	(Rn, $0 \le n \le 7$)
src2	register	(Rn, $0 \le n \le 7$)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P parallel addressing modes ($0 \le P \le 3$)

Operation (P Field)

00	*src3* × *src4, src1* + *src2*
01	*src3* × *src1, src4* + *src2*
10	*src1* × *src2, src3* + *src4*
11	*src3* × *src1, src2* + *src4*

Encoding



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, then MPYI3 accepts as input the contents of the register 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

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:

R7 = 14h = 20
R4 = 64h = 100
R0 = 0h
AR3 = 80981Fh
AR5 = 80996Eh
R3 = 0h
Data at 80981Eh = 0FFFFFFCBh = – 53
Data at 80996Eh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 14h = 20
R4 = 64h = 100
R0 = 07D0h = 2000
AR3 = 80981Fh
AR5 = 80996Dh
R3 = 0h
Data at 80981Eh = 0FFFFFFCBh = – 53
Data at 80996Eh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
     MPYI3   src2, src1, dst1
||   STI     src3, dst2
```

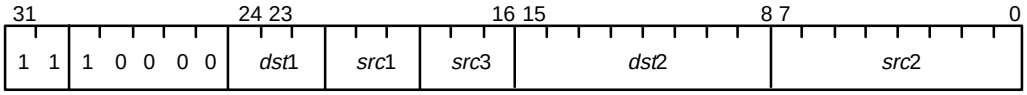
Operation

$src1 \times src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src1 register (Rn1, $0 \leq n1 \leq 7$)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn3, $0 \leq n3 \leq 7$)
src3 register (Rn4, $0 \leq n4 \leq 7$)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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 most significant 16 bits of the 48-bit result differ from the most significant bit 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:

AR0 = 80995Ah
R5 = 32h = 50
R7 = 0h
R2 = 0DCh = 220
AR3 = 80982Fh
Data at 80995Bh = 0C8h = 200
Data at 80982Eh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 80995Bh
R5 = 32h = 50
R7 = 2710h = 10000
R2 = 0DCh = 220
AR3 = 80982Fh
Data at 80995Bh = 0C8h = 200
Data at 80982Eh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

MPYI3 *srcA, srcB, dst1*
|| **SUBI3** *srcC, srcD, dst2*

Operation

srcA × *srcB* → *dst1*
|| *srcD* – *srcC* → *dst2*

Operands

srcA *srcB*	Any two indirect (disp = 0,1,IR0,IR1)
srcC *srcD*	Any two register (0 ≤ Rn ≤ 7)

dst1 register (*d1*):
0 = R0
1 = R1

dst2 register (*d2*):
0 = R2
1 = R3

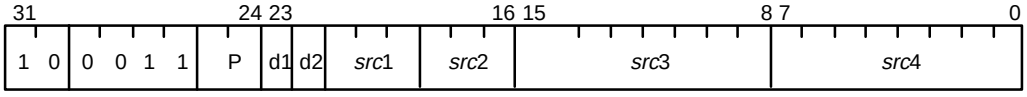
src1	register	(Rn, 0 ≤ n ≤ 7)
src2	register	(Rn, 0 ≤ n ≤ 7)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

Operation (P Field)

00	*src3* × *src4, src1* – *src2*
01	*src3* × *src1, src4* – *src2*
10	*src1* × *src2, src3* – *src4*
11	*src3* × *src1, src2* – *src4*

Encoding



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. This means that 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 as input the contents of the register 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 most significant 16 bits of the 48-bit result differs from the most significant bit 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 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:

R2 = 32h = 50
AR0 = 8098E3h
R0 = 0h
AR5 = 8099FCh
IR1 = 0Ch
R4 = 07D0h = 2000
Data at 8098E4h = 62h = 98
Data at 8099FCh = 4B0h = 1200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 320h = 800
AR0 = 8098E4h
R0 = 01324h = 4900
AR5 = 8099F0h
IR1 = 0Ch
R4 = 07D0h = 2000
Data at 8098E4h = 62h = 98
Data at 8099FCh = 4B0h = 1200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **NEGB** *src, dst*

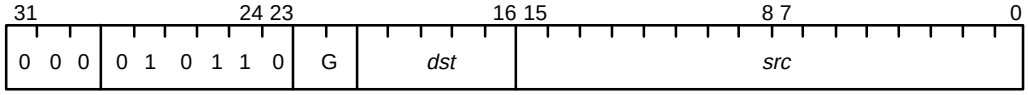
Operation 0 – *src* – C → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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:

R5 = 0FFFFFFCBh = – 53
R7 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R5 = 0FFFFFFCBh = – 53
R7 = 34h = 52
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax **NEGF** *src, dst*

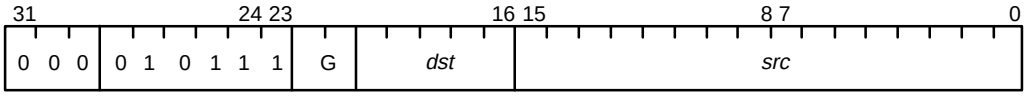
Operation 0 – *src* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

AR3 = 809800h
R1 = 057B400025h = 6.28125006e + 01
Data at 809802h = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809802h
R1 = 07F3800000h = –1.4050e + 02
Data at 809802h = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

```
    NEGF  src2, dst1
||  STF   src3, dst2
```

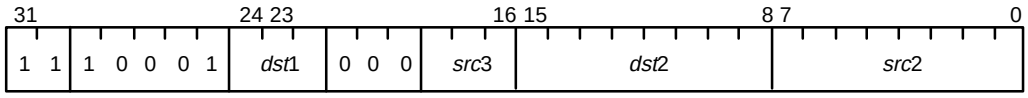
Operation

```
    0 – src2 → dst1
||  src3 → dst2
```

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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 as input the contents of the register 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; 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

```
   NEGF   *AR4––(1),R7
|| STF    R2,*++AR5(1)
```

Before Instruction:

AR4 = 8098E1h
R7 = 0h
R2 = 0733C00000h = 1.79750e + 02
AR5 = 809803h
Data at 8098E1h = 57B400000h = 6.281250e + 01
Data at 809804h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8098E0h
R7 = 0584C00000h = – 6.281250e + 01
R2 = 0733C00000h = 1.79750e + 02
AR5 = 809804h
Data at 8098E1h = 57B4000h = 6.281250e + 01
Data at 809804h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **NEGI** *src, dst*

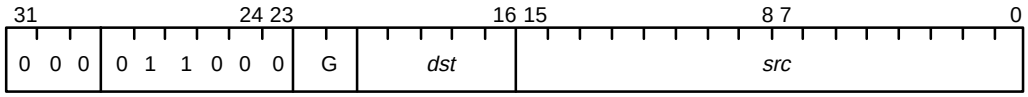
Operation 0 – *src* → *dst*

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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:

R5 = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0FFFFFF52 = –174
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax

NEGI *src2, dst1*
|| **STI** *src3, dst2*

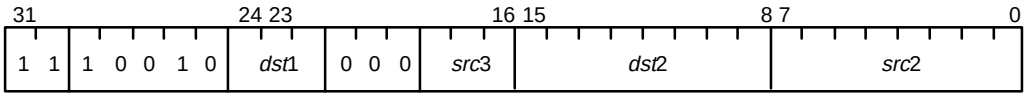
Operation

$0 - src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, $0 \leq n1 \leq 7$)
src3 register (Rn2, $0 \leq n2 \leq 7$)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (NEGI) writes to the same register, then STI accepts as input the contents of the register 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:

AR3 = 80982Fh
R2 = 19h = 25
AR1 = 8098A5h
Data at 80982Eh = 0DCh = 220
Data at 8098A5h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 80982Fh
R2 = 0FFFFFF24h = – 220
AR1 = 8098A6h
Data at 80982Eh = 0DCh = 220
Data at 8098A5h = 19h = 25
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Note: Cycle Count

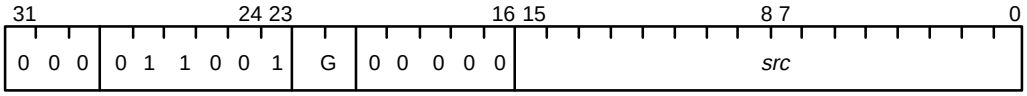
See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **NOP** *src*

Operation No ALU or multiplier operations.
ARn is modified if *src* is specified in indirect mode.

Operands *src* general addressing modes (G):
0 0 register (no operation)
1 0 indirect (modify ARn, $0 \leq n \leq 7$)

Encoding



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:

PC = 3Ah

After Instruction:

PC = 3Bh

Example 2

```
NOP   *AR3--(1)
```

Before Instruction:

PC = 5h
AR3 = 809900h

After Instruction:

PC = 6h
AR3 = 8098FFh

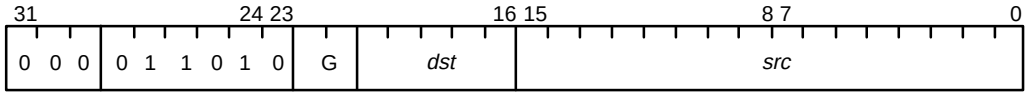
Syntax **NORM** *src, dst*

Operation norm (*src*) → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

Encoding



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 4.6 on page 4-18 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:

R1 = 0400003AF5h
R2 = 070C800000h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0400003AF5h
R2 = F26BD40000h = 1.12451613e – 04
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **NOT** *src, dst*

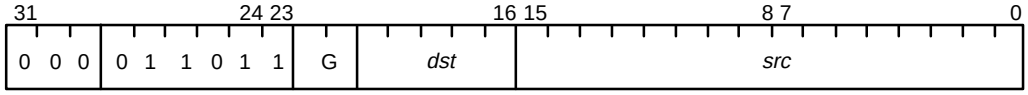
Operation $\sim src \rightarrow dst$

Operands *src* general addressing modes (G):

0 0	any CPU register
0 1	direct
1 0	indirect
1 1	immediate

dst any CPU register

Encoding



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:

DP = 80h
R4 = 0h
Data at 80982Ch = 5E2Fh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R4 = 0FFFFA1D0h
Data at 80982Ch = 5E2Fh
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax

```
    NOT   src2, dst1
||  STI   src3, dst2
```

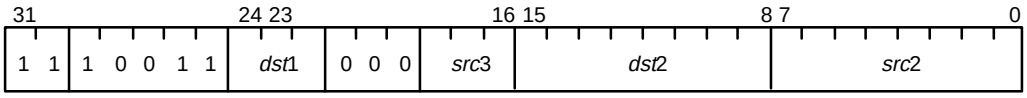
Operation

$\sim src2 \rightarrow dst1$
|| $src3 \rightarrow dst2$

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, $0 \leq n1 \leq 7$)
src3 register (Rn2, $0 \leq n2 \leq 7$)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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 as input the contents of the register 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:

AR2 = 8099CBh
R3 = 0h
R7 = 0DCh = 220
AR4 = 809850h
IR1 = 10h
Data at 8099CCh = 0C2Fh
Data at 809840h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8099CBh
R3 = 0FFFFF3D0h
R7 = 0DCh = 220
AR4 = 809840h
IR1 = 10h
Data at 8099CCh = 0C2Fh
Data at 809840h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **OR** *src, dst*

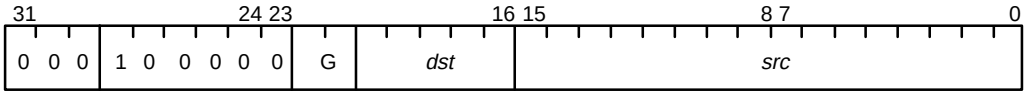
Operation *dst* OR *src* → *dst*

Operands *src* general addressing modes (G):

0 0 any CPU register
0 1 direct
1 0 indirect
1 1 immediate (not sign-extended)

dst any CPU register

Encoding



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:

AR1 = 809800h
IR1 = 4h
R2 = 012560000h
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809804h
IR1 = 4h
R2 = 012562BCDh
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **OR3** *src2, src1, dst*

Operation *src1* OR *src2* → *dst*

Operands

src1 three-operand addressing modes (T):

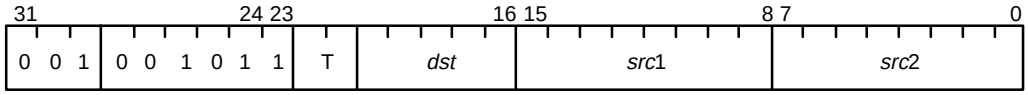
0 0	register (Rn1, 0 n1 ≤ 27)
0 1	indirect (disp = 0, 1, IR0, IR1)
1 0	register (Rn1, 0 ≤ n1 ≤ 27)
1 1	indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0	register (Rn2, 0 ≤ n2 ≤ 27)
0 1	register (Rn2, 0 ≤ n2 ≤ 27)
1 0	indirect (disp = 0, 1, IR0, IR1)
1 1	indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

AR1 = 809800h
IR1 = 4h
R2 = 012560000h
R7 = 0h
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809804h
IR1 = 4h
R2 = 012560000h
R7 = 012562BCDh
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
    OR3   src2, src1, dst1
||  STI   src3, dst2
```

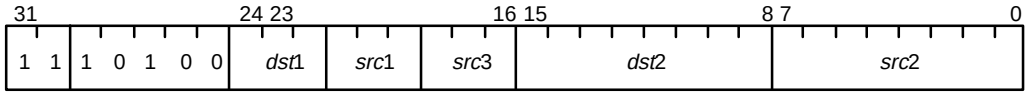
Operation

```
    src1 OR src2 → dst1
|   src3 → dst2
```

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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 as input the contents of the register 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:

AR2 = 809830h
R5 = 800000h
R2 = 0h
R6 = 0DCh = 220
AR1 = 809883h
Data at 809831h = 9800h
Data at 809883h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809831h
R5 = 800000h
R2 = 809800h
R6 = 0DCh = 220
AR1 = 809882h
Data at 809831h = 9800h
Data at 809883h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

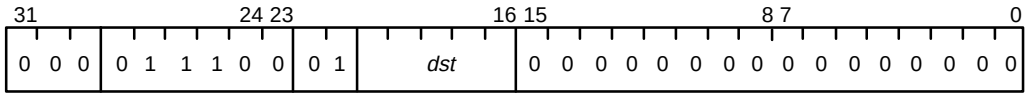
See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **POP** *dst*

Operation *SP– – → *dst*

Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

SP = 809856h
R3 = 012DAh = 4,826
Data at 809856h = FFFF0DA4h = – 62,044
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

SP = 809855h
R3 = 0FFFF0DA4h = –62,044
Data at 809856h = FFFF0DA4h = – 62,044
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax **POPF** *dst*

Operation *SP– – → *dst1*

Operands *dst* register (Rn, 0 ≤ n ≤ 7)

Encoding

31			24 23		16 15	8 7	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 is popped and loaded into the *dst* register (32 MSBs). 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 0 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:

SP = 80984Ah
R4 = 025D2E0123h = 6.91186578e + 00
Data at 80984Ah = 5F2C1302h = 5.32544007e + 28
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

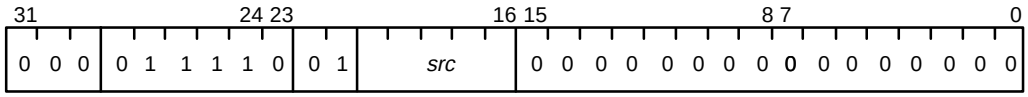
SP = 809849h
R4 = 5F2C130200h = 5.32544007e + 28
Data at 80984Ah = 5F2C1302h = 5.32544007e + 28
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **PUSH** *src*

Operation *src* → *++SP

Operands *src* register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

SP = 8098AEh
R6 = 025C128081h = 633,415,688
Data at 8098AFh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

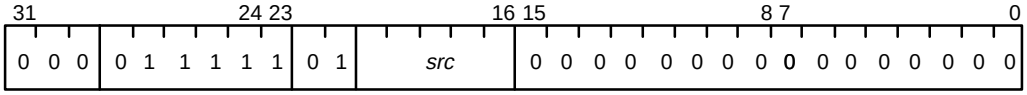
SP = 8098AFh
R6 = 025C128081h = 633,415,688
Data at 8098AFh = 5C128081h = 1,544,716,417
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **PUSHF** *src*

Operation *src* → *++SP

Operands *src* register (Rn, 0 ≤ n ≤ 7)

Encoding



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:

SP = 809801h
R2 = 025C128081h = 6.87725854e + 00
Data at 809802h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

SP = 809802h
R2 = 025C128081h = 6.87725854e + 00
Data at 809802h = 025C1280h = 6.87725830e + 00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **RETI*cond***

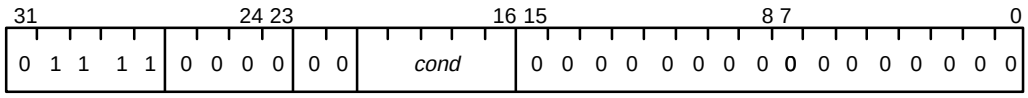
Operation If *cond* is true:

*SP – – → PC
1 → ST (GIE).

Else, continue.

Operands None

Encoding



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 TMS320C3x provides 20 condition codes that can be used with this instruction (see Table 10–9 on page 10-13 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:

PC = 456h
SP = 809830h
ST = 0h
Data at 809830h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

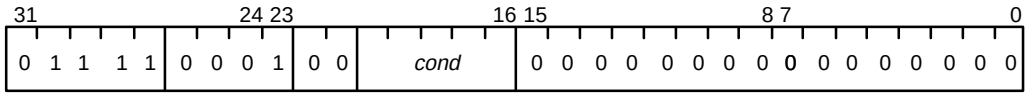
PC = 123h
SP = 80982Fh
ST = 2000h
Data at 809830h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **RETS*cond***

Operation If *cond* is true:
*SP– – → PC.
Else, continue.

Operands None

Encoding



Description A conditional return is performed. If the condition is true, the top of the stack is popped to the PC.

The TMS320C3x provides 20 condition codes that you can use with this instruction (see Table 10–9 on page -13 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:

PC = 123h
SP = 80983Ch
Data at 80983Ch = 456h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 456h
SP = 80983Bh
Data at 80983Ch = 456h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **RND** *src, dst*

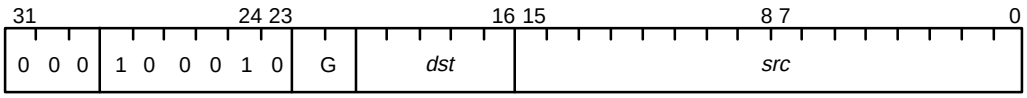
Operation rnd(*src*) → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



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 half-way 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:

R5 = 0733C16EEFh = 1.79755599e + 02
R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0733C16EEFh = 1.79755599e + 02
R2 = 0733C16F00h = 1.79755600e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: BZUF Instruction

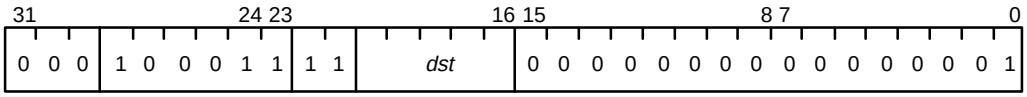
If a BZ instruction is executed immediately following an RND instruction with a 0 operand, the branch is not performed because the zero flag is not set. To circumvent this problem, execute a BZUF instruction instead of a BZ instruction.

Syntax **ROL** *dst*

Operation *dst* left-rotated 1 bit → *dst*

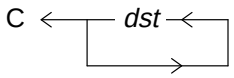
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



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 transferred into 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:

R3 = 80025CD4h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

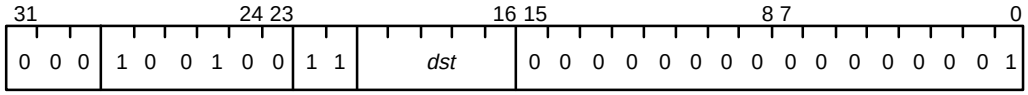
R3 = 0004B9A9h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax **ROLC** *dst*

Operation *dst* left-rotated one bit through carry bit → *dst*

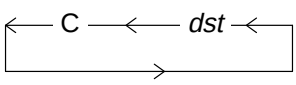
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *dst* operand are left-rotated one bit through the carry 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:

R3 = 00000420h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R3 = 000000841h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
ROLC R3
```

Before Instruction:

R3 = 80004281h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

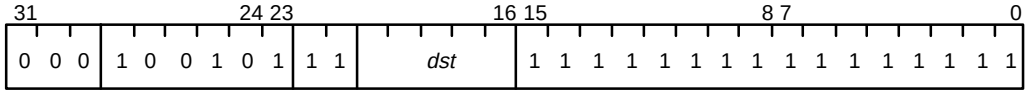
R3 = 00008502h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax **ROR** *dst*

Operation *dst* right-rotated one bit through carry bit → *dst*

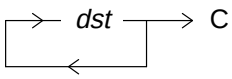
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



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 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:

R7 = 00000421h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

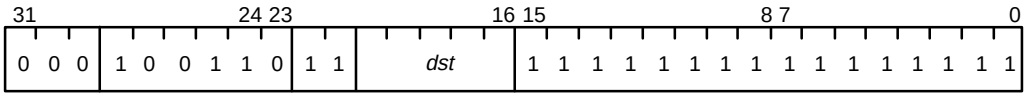
R7 = 80000210h
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax **RORC** *dst*

Operation *dst* right-rotated one bit through carry bit → *dst*

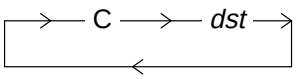
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *dst* operand are right-rotated one bit through the status register's carry 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:

R4 = 80000081h
LUF LV UF N Z V C = 0 0 0 1 0 0 0

After Instruction:

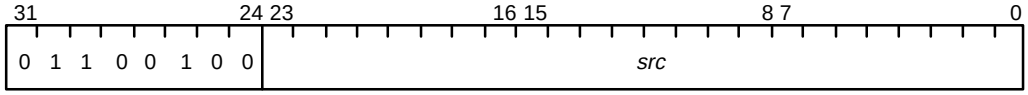
R4 = 40000040h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax **RPTB** *src*

Operation
src → RE
1 → ST (RM)
Next PC → RS

Operands *src* long-immediate addressing mode

Encoding



Description RPTB allows a block of instructions to be repeated a number of 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.

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:

PC = 123h
ST = 0h
RE = 0h
RS = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 124h
ST = 100h
RE = 127h
RS = 124h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

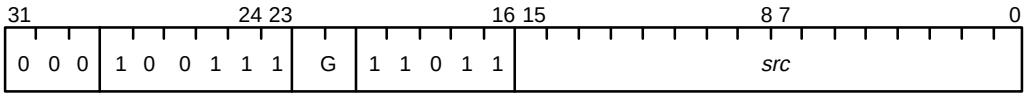
Syntax **RPTS** *src*

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
1 1	immediate

Encoding



Description The RPTS instruction allows you to repeat a single instruction a number of 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:

PC = 123h
ST = 0h
RS = 0h
RE = 0h
RC = 0h
AR5 = 0FFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

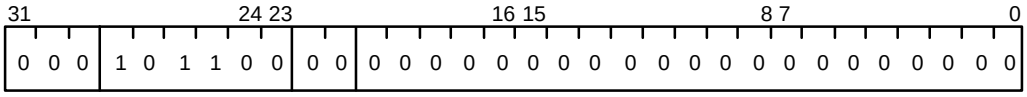
PC = 124h
ST = 100h
RS = 124h
RE = 124h
RC = 0FFh
AR5 = 0FFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SIGI**

Operation Signal interlocked operation.
Wait for interlock acknowledge.
Clear interlock.

Operands None

Encoding



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 6.4 on page 6-12 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.
```

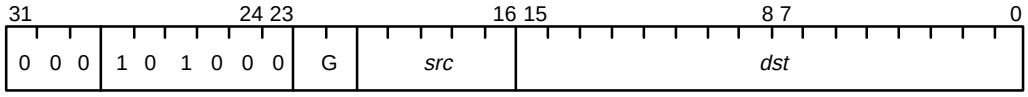
Syntax **STF** *src, dst*

Operation *src* → *dst*

Operands *src* register (Rn, 0 ≤ n ≤ 7)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



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:

DP = 80h
R2 = 052C501900h = 4.30782204e + 01
Data at 8098A1h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R2 = 052C501900h = 4.30782204e + 01
Data at 8098A1h = 52C5019h = 4.30782204e + 01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

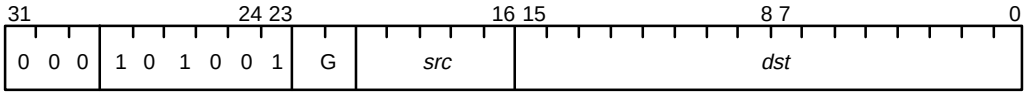
Syntax **STFI** *src, dst*

Operation *src* → *dst*
Signal end of interlocked operation.

Operands *src* register (Rn, 0 ≤ n ≤ 7)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



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 6.4 on page 6-12 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:

R3 = 0733C00000h = 1.79750e + 02
AR4 = 80993Ch
Data at 80993Bh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 0733C00000h = 1.79750e + 02
AR4 = 80993Ch
Data at 80993Bh = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

```
    STF   src2, dst2
||  STF   src1, dst1
```

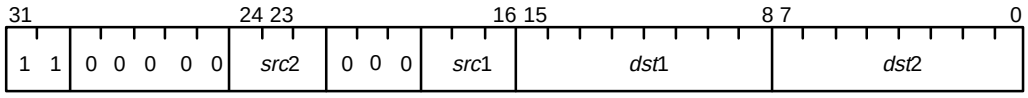
Operation

$src2 \rightarrow dst2$
$||\ src1 \rightarrow dst1$

Operands

src1 register (Rn1, $0 \leq n1 \leq 7$)
dst1 indirect (disp = 0, 1, IR0, IR1)
src2 register (Rn2, $0 \leq n2 \leq 7$)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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:

R4 = 070C800000h = 1.4050e + 02
AR3 = 809835h
R3 = 0733C00000h = 1.79750e + 02
AR5 = 8099D2h
Data at 809835h = 0h
Data at 8099D3h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 070C800000h = 1.4050e + 02
AR3 = 809834h
R3 = 0733C00000h = 1.79750e + 02
AR5 = 8099D3h
Data at 809835h = 070C8000h = 1.4050e + 02
Data at 8099D3h = 0733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

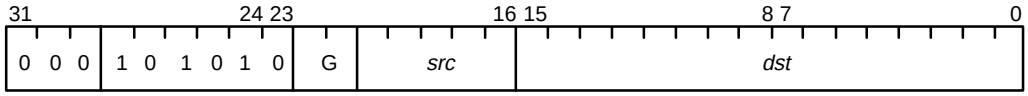
Syntax **STI** *src, dst*

Operation *src* → *dst*

Operands *src* register (Rn, 0 ≤ n ≤ 27)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



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:

DP = 80h
R4 = 42BD7h = 273,367
Data at 80982Bh = 0E5FCh = 58,876
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R4 = 42BD7h = 273,367
Data at 80982Bh = 42BD7h = 273,367
LUF LV UF N Z V C = 0 0 0 0 0 0 0

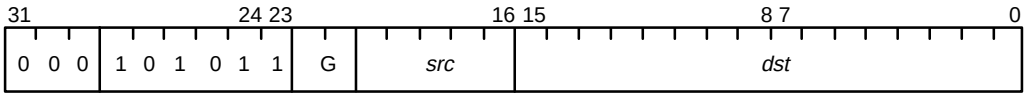
Syntax **STII** *src, dst*

Operation *src* → *dst*
Signal end of interlocked operation

Operands *src* register (Rn, 0 ≤ n ≤ 27)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



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 6.4 on page 6-12 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:

DP = 80h
R1 = 78Dh
Data at 8098AEh = 25Ch

After Instruction:

DP = 80h
R1 = 78Dh
Data at 8098AEh = 78Dh

Syntax

```
    STI   src2, dst2
||  STI   src1, dst1
```

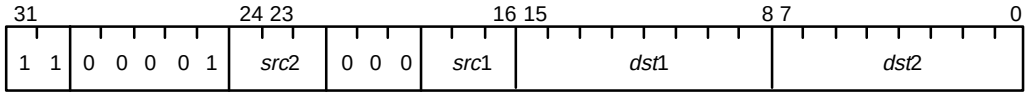
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
dst1 indirect (disp = 0, 1, IR0, IR1)
src2 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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:

R0 = 0DCh = 220
AR2 = 809830h
IR0 = 8h
R5 = 35h = 53
AR0 = 8098D3h
Data at 809838h = 0h
Data at 8098D3h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0DCh = 220
AR2 = 809838h
IR0 = 8h
R5 = 35h = 53
AR0 = 8098D3h
Data at 809838h = 0DCh = 220
Data at 8098D3h = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **SUBB** *src, dst*

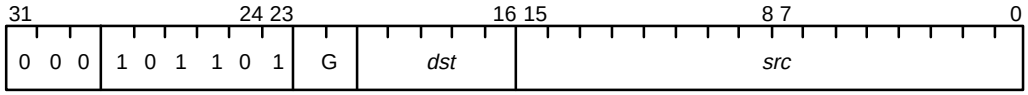
Operation *dst* – *src* – C → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

AR5 = 809800h
R5 = 0FAh = 250
Data at 809800h = 0C7h = 199
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

AR5 = 809804h
R5 = 032h = 50
Data at 809800h = 0C7h = 199
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SUBB3** *src2, src1, dst*

Operation *src1* – *src2* – C → *dst*

Operands

src1 three-operand addressing modes (T):

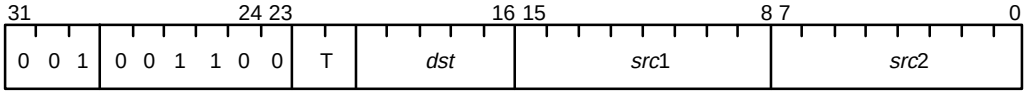
0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of 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:

AR5 = 809800h
IR0 = 4h
R5 = 0C7h = 199
R0 = 0h
Data at 809800h = 0FAh = 250
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

AR5 = 809804h
IR0 = 4h
R5 = 0C7h = 199
R0 = 32h = 50
Data at 809800h = 0FAh = 250
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

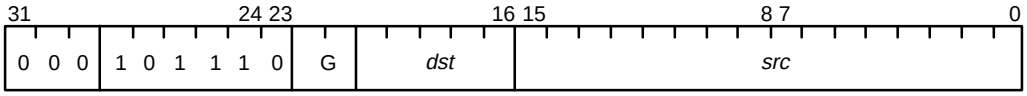
Syntax **SUBC** *src, dst*

Operation
If (*dst* – *src* ≥ 0):
(*dst* – *src* << 1) OR 1 → *dst*
Else:
dst << 1 → *dst*

Operands
src general addressing modes (G):
- 0 0 register (Rn, 0 ≤ n ≤ 27)
- 0 1 direct
- 1 0 indirect
- 1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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 multibit integer division. See subsection 11.3.4 on page 11-26 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:

DP = 80h
R1 = 04F6h = 1270
Data at 8098C5h = 492h = 1170
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R1 = 0C9h = 201
Data at 8098C5h = 492h = 1170
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
SUBC 3000,R0  (3000 = 0BB8h)
```

Before Instruction:

R0 = 07D0h = 2000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0FA0h = 4000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SUBF** *src*, *dst*

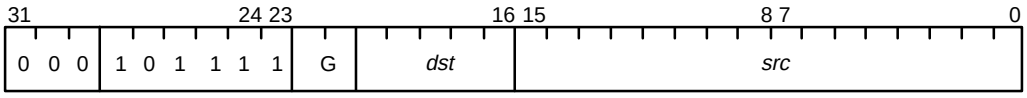
Operation *dst* – *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The difference of *the dst* operand minus 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:

AR0 = 809888h
IR0 = 80h
R5 = 0733C00000h = 1.79750000e + 02
Data at 809888h = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 809808h
IR0 = 80h
R5 = 051D000000h = 3.9250e + 01
Data at 809888h = 70C8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SUBF3** *src2, src1, dst*

Operation *src1* – *src2* → *dst*

Operands

src1 three-operand addressing modes (T):

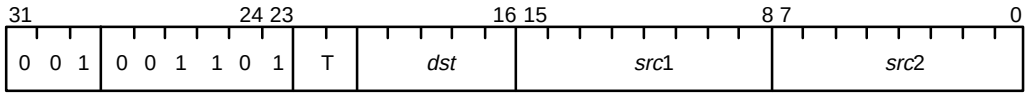
0 0	register (Rn1, ≤ n1 ≤ 7)
0 1	indirect (disp = 0, 1, IR0, IR1)
1 0	register (Rn1, ≤ n1 ≤ 7)
1 1	indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0	register (Rn2, ≤ n2 ≤ 7)
0 1	register (Rn2, ≤ n2 ≤ 7)
1 0	indirect (disp = 0, 1, IR0, IR1)
1 1	indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The difference 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

```
SUBF3  *AR0––(IR0),*AR1,R4
```

Before Instruction:

AR0 = 809888h
IR0 = 80h
AR1 = 809851h
R4 = 0h
Data at 809888h = 70C8000h = 1.4050e + 02
Data at 809851h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 809808h
IR0 = 80h
AR1 = 809851h
R4 = 51D000000h = 3.9250e + 01
Data at 809888h = 70C8000h = 1.4050e + 02
Data at 809851h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
SUBF3  R7,R0,R6
```

Before Instruction:

R7 = 57B400000h = 6.281250e + 01
R0 = 34C200000h = 1.27578125e + 01
R6 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 57B400000h = 6.281250e + 01
R0 = 34C200000h = 1.27578125e + 01
R6 = 5B7C80000h = –5.00546875e + 01
LUF LV UF N Z V C = 0 0 0 0 1 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

SUBF3||STF *Parallel SUBF3 and STF*

Syntax

SUBF3 *src1, src2, dst1*
|| **STF** *src3, dst2*

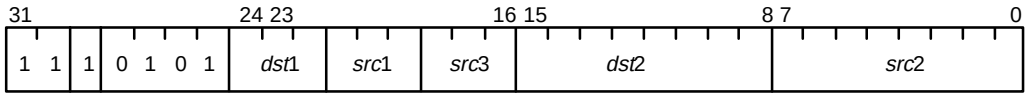
Operation

src2 – *src1* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that 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 as input the contents of the register 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:

R1 = 057B400000h = 6.28125e + 01
AR4 = 8098B8h
IR1 = 8h
R0 = 0h
R7 = 0733C00000h = 1.79750e + 02
AR5 = 809850h
IR0 = 10h
Data at 8098B0h = 70C8000h = 1.4050e + 02
Data at 809860h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 057B400000h = 6.28125e + 01
AR4 = 8098B8h
IR1 = 8h
R0 = 061B600000h = 7.768750e + 01
R7 = 0733C00000h = 1.79750e + 02
AR5 = 809850h
IR0 = 10h
Data at 8098B0h = 70C8000h = 1.4050e + 02
Data at 809860h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **SUBI** *src, dst*

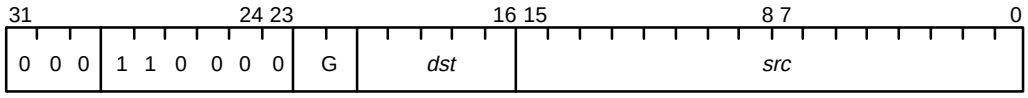
Operation *dst* – *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 27)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the *dst* operand minus 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:

R7 = 226h = 550
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 14Ah = 330
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SUBI3** *src2, src1, dst*

Operation *src1* – *src2* → *dst*

Operands

src1 three-operand addressing modes (T):

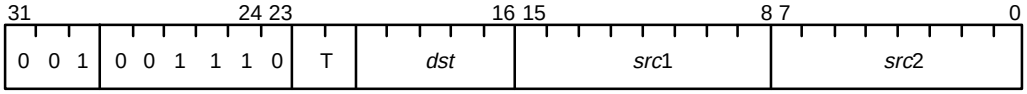
0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the *src1* operand minus 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:

R2 = 0866h = 2150
R7 = 0834h = 2100
R0 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 0866h = 2150
R7 = 0834h = 2100
R0 = 032h = 50
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Example 2

```
SUBI3 *–AR2(1),R4,R3
```

Before Instruction:

AR2 = 80985Eh
R4 = 0226h = 550
R3 = 0h
Data at 80985Dh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 80985Eh
R4 = 0226h = 550
R3 = 014Ah = 330
Data at 80985Dh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
    SUBI3  src1, src2, dst1
||  STI    src3, dst2
```

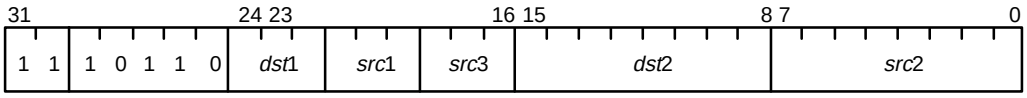
Operation

src2 – src1 → dst1
|| *src3 → dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



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 as input the contents of the register 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:

R7 = 14h = 20
AR2 = 80982Fh
IR0 = 10h
R1 = 0h
R3 = 35h = 53
AR7 = 80983Bh
Data at 80983Fh = 0DCh = 220
Data at 80983Ch = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 14h = 20
AR2 = 80982Fh
IR0 = 10h
R1 = 0C8h = 200
R3 = 35h = 53
AR7 = 80983Ch
Data at 80983Fh = 0DCh = 220
Data at 80983Ch = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **SUBRB** *src, dst*

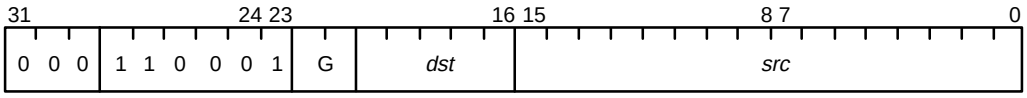
Operation *src* – *dst* – C → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

R4 = 03CBh = 971
R6 = 0258h = 600
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R4 = 03CBh = 971
R6 = 0172h = 370
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SUBRF** *src, dst*

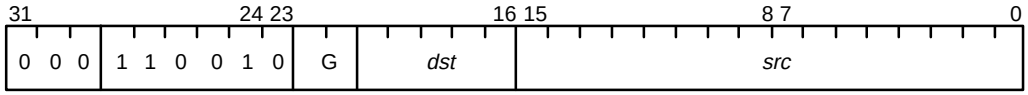
Operation *src* – *dst* → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 7)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The difference of the *src* operand minus 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:

DP = 80h
R5 = 057B400000h = 6.281250e + 01
Data at 809905h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R5 = 0669E00000h = 1.16937500e + 02
Data at 809905h = 733C000h = 1.79750e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SUBRI** *src, dst*

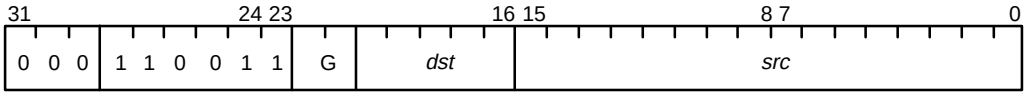
Operation *src* – *dst* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the *src* operand minus 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:

AR5 = 809900h
IR0 = 8h
R3 = 0DCh = 220
Data at 809900h = 226h = 550
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

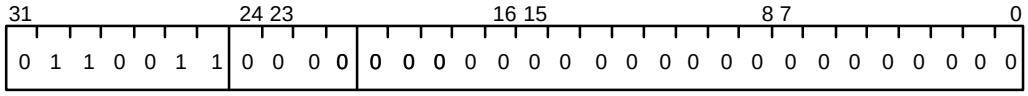
AR5 = 809908h
IR0 = 8h
R3 = 014Ah = 330
Data at 809900h = 226h = 550
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **SWI**

Operation Performs an emulation interrupt

Operands None

Encoding



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

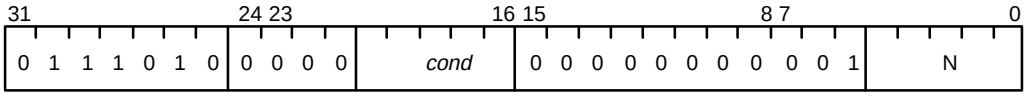
0 → ST(GIE)
If *cond* is true:
    Next PC → *++SP,
    Trap vector N → PC.

Else:

Set ST(GIE) to original state.
Continue.

Operands N (0 ≤ N ≤ 31)

Encoding



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 TMS320C3x provides 20 condition codes that can be used with this instruction (see Table 10–9 on page 10-13 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:

PC = 123h
SP = 809870h
ST = 0h
Trap Vector 16 = 10h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 10h
SP = 809871h
Data at 809871h = 124h
ST = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **TSTB** *src, dst*

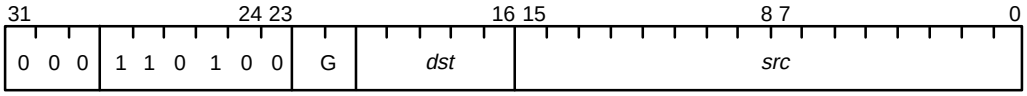
Operation *dst* AND *src*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

AR4 = 8099C5h
R5 = 898h = 2200
Data at 8099C4h = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8099C5h
R5 = 898h = 2200
Data at 8099C4h = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 1 0 0

Syntax **TSTB3** *src2, src1*

Operation *src1* AND *src2*

Operands

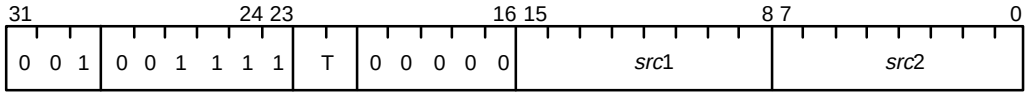
src1 three-operand addressing modes (T):

0 0	register (Rn1, 0 ≤ n1 ≤ 27)
0 1	indirect (disp = 0, 1, IR0, IR1)
1 0	register (Rn1, 0 ≤ n1 ≤ 27)
1 1	indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0	register (Rn2, 0 ≤ n2 ≤ 27)
0 1	register (Rn2, 0 ≤ n2 ≤ 127)
1 0	indirect (disp = 0, 1, IR0, IR1)
1 1	indirect (disp = 0, 1, IR0, IR1)

Encoding



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 three-operand instruction because operands are specified in the three-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:

AR5 = 809885h
IR0 = 80h
AR0 = 80992Ch
Data at 809885h = 898h = 2200
Data at 80992Dh = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 809805h
IR0 = 80h
AR0 = 80992Ch
Data at 809885h = 898h = 2200
Data at 80992Dh = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 1 0 0

Example 2

```
TSTB3  R4,*AR6––(IR0)
```

Before Instruction:

R4 = 0FBC4h
AR6 = 8099F8h
IR0 = 8h
Data at 8099F8h = 1568h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 0FBC4h
AR6 = 8099F0h
IR0 = 8h
Data at 8099F8h = 1568h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax **XOR** *src, dst*

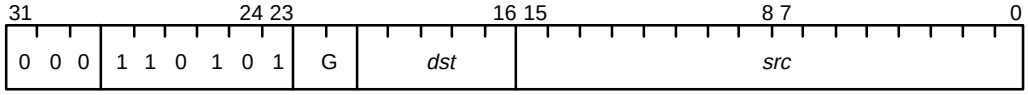
Operation *dst* XOR *src* → *dst*

Operands *src* general addressing modes (G):

0 0	register (Rn, 0 ≤ n ≤ 27)
0 1	direct
1 0	indirect
1 1	immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

R1 = 0FFA32h
R2 = 0FF5C1h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0FF412h
R2 = 000FF3h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax **XOR3** *src2, src1, dst*

Operation *src1* XOR *src2* → *dst*

Operands

src1 three-operand addressing modes (T):

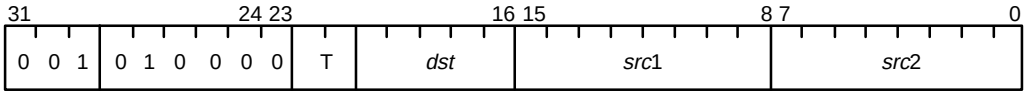
0 0	register (Rn1, 0 ≤ n1 ≤ 27)
0 1	indirect (disp = 0, 1, IR0, IR1)
1 0	register (Rn1, 0 ≤ n1 ≤ 27)
1 1	indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0	register (Rn2, 0 ≤ n2 ≤ 27)
0 1	register (Rn2, 0 ≤ n2 ≤ 27)
1 0	indirect (disp = 0, 1, IR0, IR1)
1 1	indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



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:

AR3 = 809800h
IR0 = 10h
R7 = 0FFFFh
R4 = 0h
Data at 809800h = 5AC3h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809810h
IR0 = 10h
R7 = 0FFFFh
R4 = 0A53Ch
Data at 809800h = 5AC3h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example 2

```
XOR3  R5,*–AR1(1),R1
```

Before Instruction:

R5 = 0FFA32h
AR1 = 809826h
R1 = 0h
Data at 809825h = 0FF5C1h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0FFA32h
AR1 = 809826h
R1 = 000F33h
Data at 809825h = 0FF5C1h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Syntax

```
    XOR3   src2, src1, dst1
||  STI    src3, dst2
```

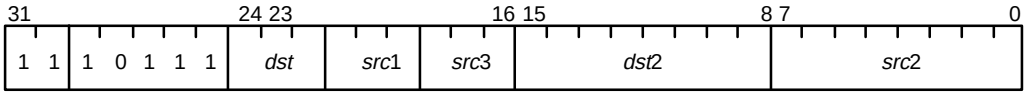
Operation

src1 XOR *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1	register (Rn1, 0 ≤ n1 ≤ 7)
src2	indirect (disp = 0, 1, IR0, IR1)
dst1	register (Rn2, 0 ≤ n2 ≤ 7)
src3	register (Rn3, 0 ≤ n3 ≤ 7)
dst2	indirect (disp = 0, 1, IR0, IR1)

Encoding



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. This means that, 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:

AR1 = 80987Eh
R3 = 85h
R6 = 0DCh = 220
AR2 = 8098B4h
IR0 = 8h
Data at 80987Eh = 85h
Data at 8098ACh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 80987Fh
R3 = 0h
R6 = 0DCh = 220
AR2 = 8098B4h
IR0 = 8h
Data at 80987Eh = 85h
Data at 8098ACh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Note: Cycle Count

See subsection 9.5.2 on page 9-24 for operand ordering effects on cycle count.

Chapter 11

Software Applications

The TMS320C3x is a powerful digital signal processor with an architecture and instruction set designed to find simple solutions to DSP problems. There are instructions specifically designed for efficient implementation of DSP algorithms as well as general-purpose instructions that make the device suitable for more general tasks, like any microprocessor. The floating-point and integer arithmetic supported by the device let you concentrate on the algorithm and pay less attention to scaling, dynamic range, and overflows.

The purpose of this chapter is to explain how to use the instruction set, the architecture, and the interface of the TMS320C3x processor. It presents coding examples for frequently used applications and discusses more involved examples and applications. This chapter defines the principles involved in the applications and provides 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.

Major topics discussed in this chapter are listed below.

Topic **Page**

11.1 Processor Initialization

Before you execute a digital signal processing algorithm, you must initialize the processor. Initialization usually occurs any time the processor is reset.

You can reset the processor by applying a low level to the $\overline{\text{RESET}}$ input for several cycles. At this time, the TMS320C3x terminates execution and puts the reset vector (that is, the contents of memory location 0) in the program counter. The reset vector normally contains the address of the system-initialization routine. The hardware reset also initializes various registers and status bits.

After reset, you can further initialize the processor by executing instructions that set up operational modes, memory pointers, interrupts, and the remaining functions needed to meet system requirements.

To configure the processor at reset, you should initialize the following internal functions:

- Memory-mapped registers
- Interrupt structure

In addition to the initialization performed during the hardware reset (for conditions after hardware reset, see Chapter 12), Example 11–1 shows coding for initializing the TMS320C3x to the following machine state:

- All interrupts are enabled.
- The overflow mode is disabled.
- The data memory page pointer is set to 0.
- The internal memory is filled with 0s.

Note that all constants larger than 16 bits should be placed in memory and accessed through direct or indirect addressing.

Example 11–1. TMS320C3x Processor Initialization

```
*
*   TITLE  PROCESSOR INITIALIZATION
*
    .global   RESET,INIT,BEGIN
    .global   INT0,INT1,INT2,INT3
    .global   ISR0,ISR1,ISR2,ISR3
    .global   DINT,DMA
    .global   TINT0,TINT1,XINT0,RINT0,XINT1,RINT1
    .global   TIME0,TIME1,XMT0,RCV0,XMT1,RCV1
    .global   TRAP0,TRAP1,TRAP2,TRP0,TRP1,TRP2
*
*   PROCESSOR INITIALIZATION FOR THE TMS320C3x
*
*   RESET AND INTERRUPT VECTOR SPECIFICATION. THIS
*   ARRANGEMENT ASSUMES THAT DURING LINKING, THE FOLLOWING
*   TEXT SEGMENT WILL BE PLACED TO START AT MEMORY
*   LOCATION 0.
*
    .sect  “init”        ;  Named section
RESET  .word  INIT       ;  RS± load address INIT to PC
INT0   .word  ISR0       ;  INT0±  loads address ISR0 to PC
INT1   .word  ISR1       ;  INT1±  loads address ISR1 to PC
INT2   .word  ISR2       ;  INT2±  loads address ISR2 to PC
INT3   .word  ISR3       ;  INT3±  loads address ISR3 to PC
*
*   XINT0  .word  XMT0       ;  Serial port 0 transmit interrupt processing
*   RINT0  .word  RCV0       ;  Serial port 0 receive interrupt processing
*   XINT1  .word  XMT1       ;  Serial port 1 transmit interrupt processing
*   RINT1  .word  RCV1       ;  Serial port 1 receive interrupt processing
TINT0  .word  TIME0      ;  Timer 0 interrupt processing
TINT1  .word  TIME1      ;  Timer 1 interrupt processing
DINT   .word  DMA        ;  DMA interrupt processing
       .space 20         ;  Reserved space
TRAP0  .word  TRP0       ;  Trap 0 vector processing begins
TRAP1  .word  TRP1       ;  Trap 1 vector processing begins
TRAP2  .word  TRP2       ;  Trap 2 vector processing begins
       .space 29         ;  Leave space for the other 29 traps
*
*   IN THE FOLLOWING SECTION, CONSTANTS THAT CANNOT BE REPRESENTED
*   IN THE SHORT FORMAT ARE INITIALIZED. THE NUMBERS IN PARENTHESIS
*   AT THE END OF THE COMMENTS REPRESENT THE OFFSET OF A
*   PARTICULAR CONTROL REGISTER FROM
*   CTRL (808000H)
```

```
          .data
MASK      .word  0FFFFFFFFH
BLK0      .word  0809800H  ;  Beginning address of RAM block 0
BLK1      .word  0809C00H  ;  Beginning address of RAM block 1
STCK      .word  0809F00H  ;  Beginning of stack
CTRL      .word  0808000H  ;  Pointer for peripheral±bus memory map
DMACTL    .word  0000000H  ;  Init for DMA control (0)
TIM0CTL   .word  0000000H  ;  Init of timer 0 control (32)
TIM1CTL   .word  0000000H  ;  Init of timer 1 control (48)
SERGLOB0  .word  0000000H  ;  Init of serial 0 glbl control (64)
SERPRTX0  .word  0000000H  ;  Init of serial 0 xmt port control (66)
SERPRTR0  .word  0000000H  ;  Init of serial 0 rcv port control (67)
SERTIM0   .word  0000000H  ;  Init of serial 0 timer control (68)
SERGLOB1  .word  0000000H  ;  Init of serial 1 glbl control (80)
SERPRTX1  .word  0000000H  ;  Init of serial 1 xmt port control (82)
SERPRTR1  .word  0000000H  ;  Init of serial 1 rcv port control (83)
SERTIM1   .word  0000000H  ;  Init of serial 1 timer control (84)
PARINT    .word  0000000H  ;  Init of parallel interface control (100)
IOINT     .word  0000000H  ;  Init of I/O interface control (96)
*
          .text

*
*   THE ADDRESS AT MEMORY LOCATION 0 DIRECTS EXECUTION TO BEGIN HERE
*   FOR RESET PROCESSING THAT INITIALIZES THE PROCESSOR. WHEN RESET
*   IS APPLIED, THE FOLLOWING REGISTERS ARE INITIALIZED TO 0:
*


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

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

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


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

*
    INTERNAL DATA MEMORY INITIALIZATION TO FLOATING POINT 0
*


    LDI@BLK0,AR0        ;  AR0 points to block 0
    LDI@BLK1,AR1        ;  AR1 points to block 1
    LDF0.0,R0           ;  0 register R0
    RPTS   1023         ;  Repeat 1024 times ...
    STFR0,*AR0++(1)     ;  Zero out location in RAM block 0 and ...
||  STFR0,*AR1++(1)     ;  Zero out location in RAM block 1
```

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


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

    .end
```

11.2 Program Control

One group of TMS320C3x instructions provides program control and facilitates all types of high-speed processing. These instructions directly handle:

- subroutine calls
- software stack
- interrupts
- zero-overhead branches
- single- and multiple-instruction loops without any overhead

11.2.1 Subroutines

The TMS320C3x has a 24-bit program counter (PC) and a practically unlimited software stack. The CALL and CALL*cond* subroutine calls cause the stack pointer to increment and store the contents of the next value of the PC counter on the stack. At the end of the subroutine, RETS*cond* performs a conditional return.

Example 11–2 illustrates the use of a subroutine to determine the dot product between two vectors. Given two vectors of length N, represented by the arrays a [0], a [1],..., a [N –1] and b [0], b [1],..., b [N –1], the dot product is computed from the expression

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

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

Example 11–2. Subroutine Call (Dot Product)

```
*
*   TITLE  SUBROUTINE CALL (DOT PRODUCT)
*
*
*   MAIN ROUTINE THAT CALLS THE SUBROUTINE 'DOT' TO COMPUTE THE
*   DOT PRODUCT OF TWO VECTORS

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

*       CALL   DOT
*       .
*       .
*       .

*
*   SUBROUTINE       DOT
*
*
*   EQUATION: d = a(0) * b(0) + a(1) * b(1) + ... + a(N±1) * b(N±1)
*
*   THE DOT PRODUCT OF a AND b IS PLACED IN REGISTER R0. N MUST
*   BE GREATER THAN OR EQUAL TO 2.
*

*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   AR0       |   ADDRESS OF a(0)
*   AR1       |   ADDRESS OF b(0)
*   RC        |   LENGTH OF VECTORS (N)
*

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

        .global   DOT

*
DOT     PUSH   ST     ;  Save status register
        PUSH   R2     ;  Use the stack to save R2's
        PUSHF  R2     ;  Lower 32 and upper 32 bits
        PUSH   AR0    ;  Save AR0
        PUSH   AR1    ;  Save AR1
        PUSH   RC     ;  Save RC
```

```
*                                       ;  Initialize R0:
      MPYF3  *AR0,*AR1,R0               ;  a(0) * b(0) ±> R0
      LDF 0.0,R2                        ;  Initialize R2
      SUBI   2,RC                       ;  Set RC = N±2
*
*   DOT PRODUCT (1 <= i < N)
*
      RPTS   RC                         ;  Setup the repeat single
      MPYF3  *++AR0(1),*++AR1(1),R0     ; a(i) * b(i) ±> R0
||    ADDF3  R0,R2,R2                   ;  a(i±1)*b(i±1) + R2 ±> R2
*
   ADDF3  R0,R2,R0                      ;  a(N±1)*b(N±1) + R2 ±> R0
*
*   RETURN SEQUENCE
*
      POP       RC                      ;  Restore RC
      POP       AR1                     ;  Restore AR1
      POP       AR0                     ;  Restore AR0
      POPF      R2                      ;  Restore top 32 bits of R2
      POP R2                            ;  Restore bottom 32 bits of R2
      POP ST                            ;  Restore ST
      RETS                              ;  Return
*
*   end
*
   .end
```

11.2.2 Software Stack

The TMS320C3x has a software stack whose location is determined by the contents of the stack pointer register (SP). The stack pointer increments from low to high values, and provisions should be made to accommodate the anticipated storage requirements. The stack can be used not only during the subroutines CALL and RETS, but also inside the subroutine as a place of temporary storage of the registers, as shown in Example 11–2. SP always points to the last value pushed on the stack.

The CALL and CALL*cond* instructions and the interrupt routines push the value of the PC onto the stack. RETS*cond* and RETI*cond* then pop the stack and place the value in the program counter. You can also use the PUSH and POP instructions to maneuver the integer value of any register onto and off the stack, respectively. There are two additional instructions, PUSHF and POPF, for floating point numbers. You can push and pop floating point numbers to registers R7–R0. This feature makes it easy to save all 40 bits of the extended precision registers (see Example 11–2). Using PUSH and PUSHF on the same register saves the lower 32 and upper 32 bits. PUSH saves the lower 32; PUSHF, the upper 32. POPF, followed by POP, will recover this extended precision number. It is important to perform the integer and floating-point PUSH and POP in the order given above. POPF forces the least significant eight bits of the extended-precision registers to 0 and therefore must be performed first.

You can easily read and write to the SP to create multiple stacks for different program segments. SP is not initialized by the hardware during reset. It is therefore important to remember to initialize its value so that SP points to a predetermined memory location. This avoids the problem of SP attempting to write into ROM or otherwise write over useful data.

11.2.3 Interrupt Service Routines

Interrupts on the TMS320C3x are prioritized and vectored. When an interrupt occurs, the corresponding flag is set in the interrupt flag register IF. If the corresponding bit in the interrupt enable register (IE) is set, and interrupts are enabled by having the GIE bit in the status register set to 1, interrupt processing begins. You can also write to the interrupt flag register, allowing you to force an interrupt by software or to clear interrupts without processing them.

Even when the interrupt is disabled, you can read the interrupt flag register (IF) and take appropriate action, depending on whether the interrupt has occurred. This is true even when the interrupt is disabled. This can be useful when an interrupt-driven interface is not implemented. Example 11–3 shows the case in which a subroutine is called when interrupt 1 has not occurred.

Example 11–3. Use of Interrupts for Software Polling

```
*   TITLE INTERRUPT POLLING
        .
        .
        .
        TSTB 2,IF           ; Test if interrupt 1 has occurred
        CALLZ SUBROUTINE    ; If not, call subroutine
        .
        .
        .
```

When interrupt processing begins, the PC is pushed onto the stack, and the interrupt vector is loaded in the PC. Interrupts are then disabled by setting the GIE = 0, and the program continues from the address loaded in the PC. Since all interrupts are disabled, interrupt processing can proceed without further interruption, unless the interrupt service routine re-enables interrupts.

Except for very simple interrupt service routines, it is important to ensure that the processor context is saved during execution of this routine. You must save the context before you execute the routine itself and restore it after the routine is finished. The procedure is called context switching. Context switching is also useful for subroutine calls, especially during extensive use of the auxiliary and the extended precision registers. This section contains code examples of context switching and an interrupt service routine.

11.2.3.1 Context Switching

Context switching is commonly required during the processing of subroutine calls or interrupts. It might be quite extensive or it might be simple, depending on system requirements. On the TMS320C3x, the program counter is automatically pushed onto the stack. Important information in other TMS320C3x registers, such as the status, auxiliary, or extended-precision registers, must be saved by special commands. In order to preserve the state of the status register, you should push it first and pop it last. This keeps the restoration of the extended precision registers from affecting the status register.

Example 11–4 and Example 11–5 show saving and restoring of the TMS320C3x state. In both examples, the stack is used for saving the registers, and it expands towards higher addresses. If you don't want to use the stack pointed at by SP, you can create a separate stack by using an auxiliary register as the stack pointer. Registers saved in these examples are:

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

Example 11–4. Context Save for the TMS320C3x

```
*   TITLE CONTEXT SAVE FOR THE TMS320C3x
*
*
        .global    SAVE
*
*   CONTEXT SAVE ON SUBROUTINE CALL OR INTERRUPT
*
SAVE:
        PUSH   ST        ;  Save status register
*
*   SAVE THE EXTENDED PRECISION REGISTERS
*
        PUSH   R0        ;  Save the lower 32 bits
        PUSHF  R0        ;     and the upper 32 bits of R0
        PUSH   R1        ;  Save the lower 32 bits
        PUSHF  R1        ;     and the upper 32 bits of R1
        PUSH   R2        ;  Save the lower 32 bits
        PUSHF  R2        ;     and the upper 32 bits of R2
        PUSH   R3        ;  Save the lower 32 bits
        PUSHF  R3        ;     and the upper 32 bits of R3
        PUSH   R4        ;  Save the lower 32 bits
        PUSHF  R4        ;     and the upper 32 bits of R4
        PUSH   R5        ;  Save the lower 32 bits
        PUSHF  R5        ;     and the upper 32 bits of R5
        PUSH   R6        ;  Save the lower 32 bits
        PUSHF  R6        ;     and the upper 32 bits of R6
        PUSH   R7        ;  Save the lower 32 bits
        PUSHF  R7        ;     and the upper 32 bits of R7
*
*   SAVE THE AUXILIARY REGISTERS
*
        PUSH   AR0       ;  Save AR0
        PUSH   AR1       ;  Save AR1
        PUSH   AR2       ;  Save AR2
        PUSH   AR3       ;  Save AR3
        PUSH   AR4       ;  Save AR4
        PUSH   AR5       ;  Save AR5
        PUSH   AR6       ;  Save AR6
        PUSH   AR7       ;  Save AR7
*
```

```
                    *   SAVE THE REST REGISTERS FROM THE REGISTER FILE
                    *
    PUSH   DP       ;   Save data page pointer
    PUSH   IR0      ;   Save index register IR0
    PUSH   IR1      ;   Save index register IR1
    PUSH   BK       ;   Save block±size register
    PUSH   IE       ;   Save interrupt enable register
    PUSH   IF       ;   Save interrupt flag register
    PUSH   IOF      ;   Save I/O flag register
    PUSH   RS       ;   Save repeat start address
    PUSH   RE       ;   Save repeat end address
    PUSH   RC       ;   Save repeat counter
*
*   SAVE IS COMPLETE
*
```

Example 11–5. Context Restore for the TMS320C3x

```
*
*   TITLE CONTEXT RESTORE FOR THE TMS320C3x
*
       .global RESTR
*
*   CONTEXT RESTORE AT THE END OF A SUBROUTINE CALL OR INTERRUPT
*
RESTR:
*
*   RESTORE THE REST REGISTERS FROM THE REGISTER FILE
*
       POP RC           ;  Restore repeat counter
       POP RE           ;  Restore repeat end address
       POP RS           ;  Restore repeat start address
       POP IOF          ;  Restore I/O flag register
       POP IF           ;  Restore interrupt flag register
       POP IE           ;  Restore interrupt enable register
       POP BK           ;  Restore block±size register
       POP IR1          ;  Restore index register IR1
       POP IR0          ;  Restore index register IR0
       POP DP           ;  Restore data page pointer
*
*   RESTORE THE AUXILIARY REGISTERS
*
       POP AR7          ;  Restore AR7
       POP AR6          ;  Restore AR6
       POP AR5          ;  Restore AR5
       POP AR4          ;  Restore AR4
       POP AR3          ;  Restore AR3
       POP AR2          ;  Restore AR2
       POP AR1          ;  Restore AR1
       POP AR0          ;  Restore AR0
*
*   RESTORE THE EXTENDED PRECISION REGISTERS
*
```

```
        POPF   R7           ;  Restore the upper 32 bits and
        POP R7              ;     the lower 32 bits of R7
        POPF   R6           ;  Restore the upper 32 bits and
        POP R6              ;     the lower 32 bits of R6
        POPF   R5           ;  Restore the upper 32 bits and
        POP R5              ;     the lower 32 bits of R5
        POPF   R4           ;  Restore the upper 32 bits and
        POP R4              ;     the lower 32 bits of R4
        POPF   R3           ;  Restore the upper 32 bits and
        POP R3              ;     the lower 32 bits of R3
        POPF   R2           ;  Restore the upper 32 bits and
        POP R2              ;     the lower 32 bits of R2
        POPF   R1           ;  Restore the upper 32 bits and
        POP R1              ;     the lower 32 bits of R1
        POPF   R0           ;  Restore the upper 32 bits and
        POP R0              ;     the lower 32 bits of R0
        POP ST              ;  Restore status register
*
*   RESTORE IS COMPLETE
*
```

11.2.3.2 Interrupt Priority

Interrupts on the TMS320C3x are automatically prioritized. This allows interrupts that occur simultaneously to be serviced in a predefined order. Infrequent but lengthy interrupt service routines might need to be interrupted by more frequently occurring interrupts. In Example 11–6, the interrupt service routine for INT2 temporarily modifies the IE to permit interrupt processing when an interrupt to INT0 (but no other interrupt) occurs. When the routine has finished processing, the IE register is restored to its original state. Notice that the RETI*cond* instruction not only pops the next program counter address from the stack, but also sets the GIE bit of the status register. This enables all interrupts that have their interrupt enable bit set.

Example 11–6. Interrupt Service Routine

```
*   TITLE INTERRUPT SERVICE ROUTINE
*   .global   ISR2
ENABLE .set   2000h
MASK   .set   1
*
*   INTERRUPT PROCESSING FOR EXTERNAL INTERRUPT INT2±
*
ISR2:
       PUSH   ST           ;  Save status register
       PUSH   DP           ;  Save data page pointer
       PUSH   IE           ;  Save interrupt enable register
       PUSH   R0           ;  Save lower 32 bits and
       PUSHF  R0           ;     upper 32 bits of R0
       PUSH   R1           ;  Save lower 32 bits and
       PUSHF  R1           ;     upper 32 bits of R1
       LDI MASK,IE         ;  Unmask only INT0
       OR  ENABLE,ST       ;  Enable all interrupts
*
*   MAIN PROCESSING SECTION FOR ISR2
       .
       .
       .
       XOR ENABLE,ST       ;  Disable all interrupts
       POPF   R1           ;  Restore upper 32 bits and
       POP R1              ;     lower 32 bits of R1
       POPF   R0           ;  Restore upper 32 bits and
       POP R0              ;     lower 32 bits of R0
       POP IE              ;  Restore interrupt enable register
       POP DP              ;  Restore data page register
       POP ST              ;  Restore status register
*
       RETI                ;  Return and enable interrupts
```

11.2.4 Delayed Branches

The TMS320C3x uses delayed branches to create single-cycle branching. The delayed branches operate like regular branches but do not flush the pipeline. Instead, the three instructions following a delayed branch are also executed. As discussed in Chapter 6, the only limitations are that none of the three instructions following a delayed branch can be a:

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

Conditional delayed branches use the conditions that exist at the end of the instruction immediately preceding the delayed branch. Sometimes a branch is necessary in the flow of a program, but fewer than three instructions can be placed after a delayed branch. For faster execution, it is still advantageous to use a delayed branch. This is shown in Example 11–7, with NOPs taking the place of the unused instructions. The trade-off is more instruction words for less execution time.

Example 11–7. 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
```

11.2.5 Repeat Modes

The TMS320C3x supports looping without any overhead. For that purpose, there are two instructions: RPTB repeats a block of code, and RPTS repeats a single instruction. There are three control registers: repeat start address (RS), (repeat end address (RE), and repeat counter (RC). These contain the parameters that specify loop execution (refer to Section 6.1 on page 6-2 for a complete description of RPTB and RPTS). RS and RE are automatically set from the code, while you must set RC, as shown in the examples below.

11.2.5.1 Block Repeat

Example 11–8 shows an application of the block repeat construct. In this example, an array of 64 elements is flipped over by exchanging the elements that are equidistant from the end of the array. In other words, if the original array is

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

the final array after the rearrangement will be

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

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

Example 11–8. Loop Using Block Repeat

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

Subsection 6.1.2 on page 6-3 specifies restrictions in the block-repeat construct. Because the program counter is modified at the end of the loop according to the contents of the registers RS, RE, and RC, no operation should attempt to modify the repeat counter or the program counter at the end of the loop in a different way.

In principle, it is possible to nest repeat blocks. However, there is only one set of control registers: RS, RE, and RC. It is therefore necessary to save these registers before entering an inside loop. It might be more practical to implement a nested loop by the more traditional method of using a register as a counter and then using a delayed branch rather than using the nested repeat block approach.

Example 11–9 shows another example of using the block repeat to find a maximum of 147 numbers.

Example 11–9. Use of Block Repeat to Find a Maximum

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

        .
        .
        .
        LDI    146,RC        ;  Initialize repeat counter to 147±1
        LDI    @ADDR,AR0     ;  AR0 points to beginning of array
        LD     *AR0++(1),R0  ;  Initialize MAX to the first value
*
        RPTB   LOOP
        CMPF   *AR0++(1),R0  ;  Compare number to the maximum
LOOP    LDFLT  *±AR0(1),R0   ;  If greater, this is a new maximum
        .
        .
        .
```

11.2.5.2 Single-Instruction Repeat

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

Example 11–10 shows an application of the single-repeat construct. In this example, the sum of the products of two arrays is computed. The arrays are not necessarily different. If the arrays are a(i) and b(i), each of length N = 512, register R0 will contain, after computation, this quantity:

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

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

Example 11–10. Loop Using Single Repeat

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

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

11.2.6 Computed GOTOs

It is occasionally convenient to select during run time (and not during assembly) the subroutine to be executed. The TMS320C3x's computed GOTO supports this selection. The computed GOTO is implemented using the CALL*cond* instruction in the register-addressing mode. This instruction uses the contents of the register as the address of the call. Example 11–11 shows a computed GOTO for a task controller.

Example 11–11. Computed GOTO

```
*   TITLE COMPUTED GOTO
*
*   TASK CONTROLLER
*
*   THIS MAIN ROUTINE CONTROLS THE ORDER OF TASK EXECUTION (6 TASKS
*   IN THE PRESENT EXAMPLE). TASK0 THROUGH TASK5 ARE THE NAMES OF
*   SUBROUTINES TO BE CALLED. THEY ARE EXECUTED IN ORDER, TASK0,
*   TASK1, . . .TASK5. WHEN AN INTERRUPT OCCURS, THE INTERRUPT
*   SERVICE ROUTINE IS EXECUTED, AND THE PROCESSOR CONTINUES
*   WITH THE INSTRUCTION FOLLOWING THE IDLE INSTRUCTION. THIS
*   ROUTINE SELECTS THE TASK APPROPRIATE FOR THE CURRENT CYCLE,
*   CALLS THE TASK AS A SUBROUTINE, AND BRANCHES BACK TO THE IDLE
*   TO WAIT FOR THE NEXT SAMPLE INTERRUPT WHEN THE SCHEDULED TASK
*   HAS COMPLETED EXECUTION. R0 HOLDS THE OFFSET FROM THE BASE
*   ADDRESS OF THE TASK TO BE EXECUTED.
*
*

       LDI    5,R0          ;  Initialize R0
       LDI    @ADDR,AR1     ;  AR1 holds base address of the table
WAIT   IDLE                 ;  Wait for the next interrupt
       ADDI3  *AR1,R0,AR2   ;  Add the base address to the table
*                           ;  Entry number
       SUBI   1,R0          ;  Decrement R0
       LDILT  5,R0          ;  If R0<0, reinitialize it to 5
       LDI    *AR2,R1       :  Load the task address
       CALLU  R1            ;  Execute appropriate task
       BR     WAIT
*
TSKSEQ .word  TASK5         ;  Address of TASK5
       .word  TASK4         ;  Address of TASK4
       .word  TASK3         ;  Address of TASK3
       .word  TASK2         ;  Address of TASK2
       .word  TASK1         ;  Address of TASK1
       .word  TASK0         ;  Address of TASK0
ADDR   .word  TSKSEQ
```

11.3 Logical and Arithmetic Operations

The TMS320C3x instruction set supports both integer and floating-point arithmetic and logical operations. The basic functions of such instructions can be combined to form more complex operations. This section examines examples of these operations:

- Bit manipulation
- Block moves
- Bit-reversed addressing
- Integer and floating-point division
- Square root
- Extended-precision arithmetic
- Floating-point format conversion between IEEE and TMS320C3x formats

11.3.1 Bit Manipulation

Instructions for logical operations, such as AND, OR, NOT, ANDN, and XOR can be used together with the shift instructions for bit manipulation. A special instruction, TSTB, tests bits. TSTB performs the same operation as AND, but the result of the logical AND is only used to set the condition flags and is not written anywhere. Example 11–12 and Example 11–13 demonstrate the use of the several instructions for bit manipulation and testing.

Example 11–12. Use of TSTB for Software-Controlled Interrupt

```
*   TITLE USE OF TSTB FOR SOFTWARE±CONTROLLED INTERRUPT
*
*   IN THIS EXAMPLE, ALL INTERRUPTS HAVE BEEN DISABLED BY
*   RESETTING THE GIE BIT OF THE STATUS REGISTER. WHEN AN
*   INTERRUPT ARRIVES, IT IS STORED IN THE IF REGISTER. THE
*   PRESENT EXAMPLE ACTIVATES THE INTERRUPT SERVICE ROUTINE INTR
*   WHEN IT DETECTS THAT INT2± HAS OCCURRED.
        .
        .
        .
        TSTB   0100b,IF  ;  Check if bit 2 of IF is set,
        CALLNZ INTR      ;  and, if so, call subroutine INTR
        .
        .
        .
```

Example 11–13. Copy a Bit From One Location to Another

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

11.3.2 Block Moves

Since the TMS320C3x directly addresses a large amount of memory, blocks of data or program code can be stored off-chip in slow memories and then loaded on-chip for faster execution. Data can also be moved from on-chip to off-chip memory for storage or for multiprocessor data transfers.

You can use direct memory access (DMA) in parallel with CPU operations to accomplish such data transfers. The DMA operation is explained in detail in subsection 8.3 on page 8-43. An alternative to DMA is to perform data transfers under program control using load and store instructions in a repeat mode. Example 11–14 shows the transfer of a block of 512 floating-point numbers from external memory to block 1 of the on-chip RAM.

Example 11–14. Block Move Under Program Control

```
*   TITLE BLOCK MOVE UNDER PROGRAM CONTROL
*

extern .word     01000H
block1 .word     0809C00H
       .
       .
       .
       LDI    @extern,AR0   ;  Source address
       LDI    @block1,AR1   ;  Destination address

       LDF    *AR0++,R0     ;  Load the first number

       RPTS   510           ;  Repeat following instruction 511 times
       LDF    *AR0++,R0     ;  Load the next number, and...
||     STF    R0,*AR1++     ;  store the previous one

       STF    R0,*AR1       ;  Store the last number
       .
       .
       .
```

11.3.3 Bit-Reversed Addressing

The TMS320C3x can implement fast Fourier transforms (FFT) with bit-reversed addressing. If the data to be transformed is in the correct order, the final result of the FFT is scrambled in bit-reversed order. To recover the frequency-domain data in the correct order, you must swap certain memory locations. The bit-reversed addressing mode makes swapping unnecessary. The next time data needs to be accessed, the access is performed in a bit-reversed manner rather than sequentially. The base address of bit-reversed addressing must be located on a boundary of the size of the table. For example, if IR0 = 2^{n-1}, the n LSBs of the base address must be 0.

In bit-reversed addressing, IR0 holds a value equal to one-half the size of the FFT, if real and imaginary data are stored in separate arrays. During accessing, the auxiliary register is indexed by IR0, but with reverse carry propagation. Example 11–15 illustrates a 512-point complex FFT being moved from the place of computation (pointed at by AR0) to a location pointed at by AR1. In this example, real and imaginary parts XR(i) and XI(i) of the data are not stored in separate arrays, but they are interleaved XR(0), XI(0), XR(1), XI(1), ..., XR(N-1), XI(N-1). Because of this arrangement, the length of the array is 2N instead of N, and IR0 is set to 512 instead of 256.

Example 11–15. Bit-Reversed Addressing

```
*
*   TITLE BIT±REVERSED ADDRESSING
*
*   THIS EXAMPLE MOVES THE RESULT OF THE 512±POINT FFT
*   COMPUTATION POINTED AT BY AR0 TO A LOCATION POINTED AT
*   BY AR1. REAL AND IMAGINARY POINTS ARE ALTERNATING.

        .
        .
        .

        LDI    512,IR0
        LDI    2,IR1
        LDI    511,RC           ;  Repeat 511+1 times
        LDF    *+AR0(1),R1      ;  Load first imaginary point
        RPTB   LOOP
*
        LDF    *AR0++(IR0)B,R0  ;  Load real value (and point
||      STF    R1,*+AR1(1)      :     to next location) and store
*                               ;     the imaginary value
LOOP    LDF    *+AR0(1),R1      ;  Load next imaginary point and store
||      STF    R0,*AR1++(IR1)   ;     previous real value
        .
        .
        .
```

11.3.4 Integer and Floating-Point Division

Although division is not implemented as a single instruction in the TMS320C3x, the instruction set has the capacity to perform an efficient division routine. Integer and floating-point division are examined separately because different algorithms are used.

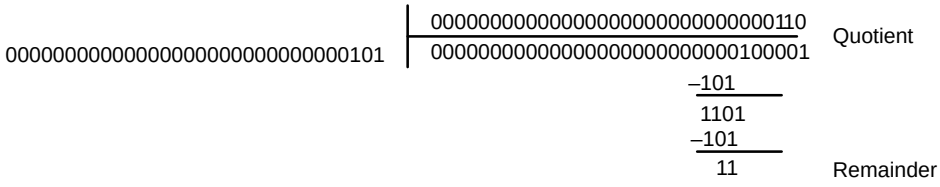
11.3.4.1 Integer Division

Division is implemented on the TMS320C3x by repeated subtractions using SUBC, a special conditional subtract instruction. Consider the case of a 32-bit positive dividend with i significant bits (and 32 – i sign bits) and a 32-bit positive divisor with j significant bits (and 32 – j sign bits). The repetition of the SUBC command i – j + 1 times produces a 32-bit result in which the lower i – j + 1 bits are the quotient and the upper 31 – i + j bits are the remainder of the division.

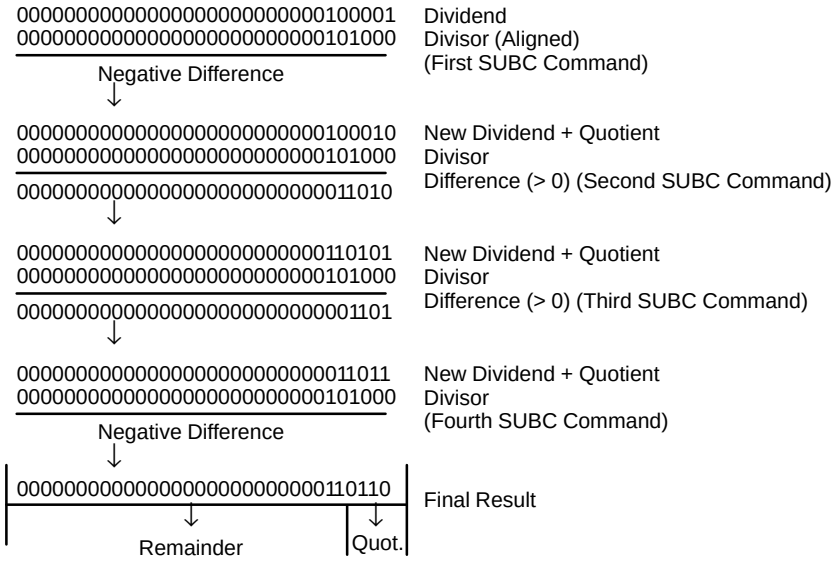
SUBC implements binary division in the same manner that long division implements it. The divisor which is assumed to be smaller than the dividend) is shifted left i – j times to be aligned with the dividend. Then, using SUBC, the shifted divisor is subtracted from the dividend. For each subtraction that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

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

Long Division:



SUBC Method:



When the SUBC command is used, both the dividend and the divisor must be positive. Example 11–16 shows an example of a realization of the integer division in which the sign of the quotient is properly handled. The last instruction before returning modifies the condition flag in case subsequent operations depend on the sign of the result.

Example 11–16. Integer Division

```
*
*   TITLE INTEGER DIVISION
*
    SUBROUTINE DIVI
*
*
*      INPUTS:   SIGNED INTEGER DIVIDEND IN R0,
*         SIGNED INTEGER DIVISOR IN R1
*
*      OUTPUT:   R0/R1 into R0
*
*      REGISTERS USED:  R0±R3, IR0, IR1
*
*      OPERATION:    1.  NORMALIZE DIVISOR WITH DIVIDEND
*                    2.  REPEAT SUBC
*                    3.  QUOTIENT IS IN LSBs OF RESULT
*
*      CYCLES:       31±62 (DEPENDS ON AMOUNT OF NORMALIZATION)
*

       .globl    DIVI

SIGN   .set      R2
TEMPF  .set      R3
TEMP   .set      IR0
COUNT  .set      IR1

*   DIVI ± SIGNED DIVISION


DIVI:
*
*   DETERMINE SIGN OF RESULT. GET ABSOLUTE VALUE OF OPERANDS.
*

       XOR    R0,R1,SIGN    ;  Get the sign
       ABSI   R0
       ABSI   R1


       CMPI   R0,R1         ;  Divisor > dividend ?
       BGTD   ZERO          ;  If so, return 0
*

*   NORMALIZE OPERANDS. USE DIFFERENCE IN EXPONENTS AS SHIFT COUNT
*   FOR DIVISOR AND AS REPEAT COUNT FOR 'SUBC'.
*

       FLOAT  R0,TEMPF      ;  Normalize dividend
       PUSHF  TEMPF         ;  PUSH as float
       POP    COUNT         ;  POP as int
       LSH±24,COUNT         ;  Get dividend exponent
```

```
        FLOAT  R1,TEMPF      ;  Normalize divisor
        PUSHF  TEMPF         ;  PUSH as float
        POP    TEMP          ;  POP as int
        LSH    ±24,TEMP      ;  Get divisor exponent
        SUBI   TEMP,COUNT    ;  Get difference in exponents
        LSH    COUNT,R1      ;  Align divisor with dividend
*
*   DO COUNT+1 SUBTRACT & SHIFTS.
        RPTS   COUNT
        SUBC   R1,R0
*
*   MASK OFF THE LOWER COUNT+1 BITS OF R0.
*
        SUBRI  31,COUNT      ;  Shift count is (32 ± (COUNT+1))
        LSH    COUNT,R0      ;  Shift left
        NEGI   COUNT
        LSH    COUNT,R0      ;  Shift right to get result
*
*   CHECK SIGN AND NEGATE RESULT IF NECESSARY.
*
        NEGI   R0,R1         ;  Negate result
        ASH    ±31,SIGN      ;  Check sign
        LDINZ  R1,R0         ;  If set, use negative result
        CMPI   0,R0          ;  Set status from result
        RETS
*
*   RETURN 0.
*
0:
        LDI    0,R0
        RETS
        .end
```

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

11.3.4.2 Computation of Floating-Point Inverse and Division

This section presents a method of implementing floating-point division on the TMS320C3x. Since the algorithm outlined here computes the inverse of a number v, to perform y / v, multiply y by the inverse of v.

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

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

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

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

Example 11–17 shows the implementation of this algorithm on the TMS320C3x, where the iteration has been applied five times. Both accuracy and speed are affected by the number of iterations. The accuracy offered by the single-precision floating-point format is $2^{-23} = 1.192E-7$. If you want more accuracy, use more iterations. If you want less accuracy, reduce the number of iterations to increase the execution speed.

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

Example 11–17. Inverse of a Floating-Point Number

```
*
*   TITLE INVERSE OF A FLOATING±POINT NUMBER
*
*
*   SUBROUTINE INVF
*
*
*   THE FLOATING-POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, 1/v IS ALSO STORED IN R0.
*
*   TYPICAL CALLING SEQUENCE:
*      LDF v, R0
*      CALL   INVF
*

*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   v = NUMBER TO FIND THE RECIPROCAL OF (UPON THE CALL)
*   R0        |   1/v (UPON THE RETURN)
*
*   REGISTER USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, R3
*   REGISTER CONTAINING RESULT: R0

*
*      CYCLES: 35   WORDS: 32
*
*
       .global   INVF
*
INVF:  LDF R0,R3         ;  v is saved for later
       ABSF   R0         ;  The algorithm uses v = |v|
*
*   EXTRACT THE EXPONENT OF v.
*
       PUSHF  R0
       POP    R1
       ASH    ±24,R1     ;  The 8 LSBs of R1 contain the exponent
*                        ;  of v
*
*   x[0] FORMATION IS GIVEN THE EXPONENT OF v.
*
```

```
            NEGI   R1
            SUBI   1,R1       ;  Now we have ±e±1, the exponent of x[0]
            ASH    24,R1
            PUSH   R1
            POPF   R1         ;  Now R1 = x[0] = 1.0 * 2**(±e±1)
      *
      *   NOW THE ITERATIONS BEGIN.
      *
            MPYF   R1,R0,R2   ;  R2 = v * x[0]
            SUBRF  2.0,R2     ;  R2 = 2.0 ± v * x[0]
            MPYF   R2,R1      ;  R1 = x[1] = x[0] * (2.0 ± v * x[0])
      *
            MPYF   R1,R0,R2   ;  R2 = v * x[1]
            SUBRF  2.0,R2     ;  R2 = 2.0 – v * x[1]
            MPYF   R2,R1      ;  R1 = x[2] = x[1] * (2.0 ± v * x[1])
      *
            MPYF   R1,R0,R2   ;  R2 = v * x[2]
            SUBRF  2.0,R2     ;  R2 = 2.0 ± v * x[2]
            MPYF   R2,R1      ;  R1 = x[3] = x[2] * (2.0 ± v * x[2])
      *
            MPYF   R1,R0,R2   ;  R2 = v * x[3]
            SUBRF  2.0,R2     ;  R2 = 2.0 ± v * x[3]
            MPYF   R2,R1      ;  R1 = x[4] = x[3] * (2.0 ± v * x[3])
      *
            RND    R1         ;  This minimizes error in the LSBs
      *
      *   FOR THE LAST ITERATION WE USE THE FORMULATION:
      *   x[5] = (x[4] * (1.0 ± (v * x[4]))) + x[4]
      *
            MPYF   R1,R0,R2   ;  R2 =  v * x[4] = 1.0..01.. => 1
            SUBRF  1.0,R2     ;  R2 =  1.0 ± v * x[4] = 0.0..01... => 0
            MPYF   R1,R2      ;  R2 =  x[4] * (1.0 ± v * x[4])
            ADDF   R2,R1      ;  R2 =  x[5] = (x[4]*(1.0±(v*x[4])))+x[4]
      *
            RND R1,R0         ;  Round since this is followed by a MPYF
      *
      *   NOW THE CASE OF v < 0 IS HANDLED.
      *
            NEGF   R0,R2
            LDF    R3,R3      ;  This sets condition flags
            LDFN   R2,R0      ;  If v < 0, then R0 = ±R0
      *
            RETS
      *
*     END
*
            .end
```

11.3.5 Square Root

An iterative algorithm computes square root on the TMS320C3x and is similar to the one used for the computation of the inverse. This algorithm computes the inverse of the square root of a number v, 1 / SQRT(v). To derive SQRT(v), multiply this result by v. Since in many applications, division by the square root of a number is desirable, the output of the algorithm saves the effort to compute the inverse of the square root.

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

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

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

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

Example 11–18 shows the implementation of this algorithm on the TMS320C3x, where the iteration has been applied five times. Both accuracy and speed are affected by the number of iterations. If you want more accuracy, use more iterations. If you want less accuracy, reduce the number of iterations to increase the execution speed.

Example 11–18. Square Root of a Floating-Point Number

```
*
*   TITLE  SQUARE ROOT OF A FLOATING±POINT NUMBER
*
*
*   SUBROUTINE SQRT
*
*   THE FLOATING POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, SQRT(v) IS ALSO STORED IN R0. NOTE
*   THAT THE ALGORITHM ACTUALLY COMPUTES 1/SQRT(v).
*
*
*   TYPICAL CALLING SEQUENCE:
*
*      LDF v, R0
*      CALL SQRT
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-----------------------------------
*   R0        |   v = NUMBER TO FIND THE SQUARE ROOT OF
*             |   (UPON THE CALL)
*   R0        |   SQRT(v) (UPON THE RETURN)
*
*   REGISTER USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, R3
*   REGISTER CONTAINING RESULT: R0
*
*   CYCLES: 50 WORDS: 39
*
          .global SQRT
*
*   EXTRACT THE EXPONENT OF V.
*
```

```
SQRT:  LDF R0,R3          ;  Save v
       RETSLE             ;  Return if number is non±positive
       PUSHF  R0
       POP R1
       ASH ±24,R1         ;  The 8 LSBs of R1 contain exponent of v
       ADDI   1,R1        ;  Add a rounding bit in the exponent
       ASH    -1,R1       ;  e/2
*
*   X[0] FORMATION GIVEN THE EXPONENT OF V.
*
       NEGI   R1
       ASH    24,R1
       PUSH   R1
       POPF   R1          ;  Now R1 = x[0] = 1.0 * 2**(±e/2)
*
*   GENERATE V/2.
*
       MPYF   0.5,R0      ;  V/2 and take rounding bit out
*
*   NOW THE ITERATIONS BEGIN.
*
       MPYF   R1,R1,R2    ;  R2 = x[0] * x[0]
       MPYF   R0,R2       ;  R2 = (v/2) * x[0] * x[0]
       SUBRF  1.5,R2      ;  R2 = 1.5 ± (v/2) * x[0] * x[0]

       MPYF   R2,R1       ;  R1 = x[1] = x[0] *
*                         ;     (1.5 ± (v/2)*x[0]*x[0])
       RND    R1
       MPYF   R1,R1,R2    ;  R2 = x[1] * x[1]
       MPYF   R0,R2       ;  R2 = (v/2) * x[1] * x[1]
       SUBRF  1.5,R2      ;  R2 = 1.5 ± (v/2) * x[1] * x[1]
       MPYF   R2,R1       ;  R1 = x[2] = x[1] *
*                         ;     (1.5 ± (v/2)*x[1]*x[1])
       RND    R1
       MPYF   R1,R1,R2    ;  R2 = x[2] * x[2]
       MPYF   R0,R2       ;  R2 = (v/2) * x[2] * x[2]
       SUBRF  1.5,R2      ;  R2 = 1.5 ± (v/2) * x[2] * x[2]
       MPYF   R2,R1       ;  R1 = x[3] = x[2]
*                         ;     *(1.5 ± (v/2)*x[2]*x[2])
       RND    R1
*
```

```
        MPYF   R1,R1,R2  ;  R2 = x[3] * x[3]
        MPYF   R0,R2     ;  R2 = (v/2) * x[3] * x[3]
        SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[3] * x[3]
        MPYF   R2,R1     ;  R1 = x[4] = x[3]
*                        ;     * (1.5 ± (v/2) * x[3] * x[3])
        RND    R1
*
        MPYF   R1,R1,R2  ;  R2 = x[4] * x[4]
        MPYF   R0,R2     ;  R2 = (v/2) * x[4] * x[4]
        SUBRF  1.5,R2    ;  R2 = 1.5 ± (v/2) * x[4] * x[4]
        MPYF   R2,R1     ;  R1 = x[5] = x[4]
*                        ;     * (1.5 ± (v/2) * x[4] * x[4])
*
*
        RND    R1,R0     ;  Round
*
        MPYF   R3,R0     ;  Sqrt(v) from sqrt(v**(±1))
*
        RETS
*
*   end
*
        .end
```

11.3.6 Extended-Precision Arithmetic

The TMS320C3x offers 32 bits of precision for integer arithmetic and 24 bits of precision in the mantissa for floating-point arithmetic. For higher precision in floating-point operations, the eight extended-precision registers R7 to R0 contain eight additional bits of accuracy. Since no comparable extension is available for fixed-point arithmetic, this section shows how you can achieve fixed-point double precision by using the capabilities of the processor. The technique consists of performing the arithmetic by parts (which is similar to performing longhand arithmetic).

In the instruction set, operations ADDC (add with carry) and SUBB (subtract with borrow) use the status carry bit for extended-precision arithmetic. The carry bit is affected by the arithmetic operations of the ALU and by the rotate and shift instructions. It can also be manipulated directly by setting the status register to certain values. For proper operation, the overflow mode bit should be reset (OVM = 0) so that the accumulator results are not loaded with the saturation values. Example 11–19 and Example 11–20 show 64-bit addition and 64-bit subtraction. The first operand is stored in the registers R0 (low word) and R1 (high word). The second operand is stored in R2 and R3. The result is stored in R0 and R1.

Example 11–19. 64-Bit Addition

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

Example 11–20. 64-Bit Subtraction

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

When two 32-bit numbers are multiplied, a 64-bit product results. The procedure for multiplication is to split the 32-bit magnitude values of the multiplicand X and the multiplier Y into two parts (X1,X0) and (X3,X2), respectively, with 16 bits each. The operation is done on unsigned numbers, and the product is adjusted for the sign bit. Example 11–21 shows the implementation of a 32-bit by 32-bit multiplication.

Example 11–21. 32-Bit-by-32-Bit Multiplication

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

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

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

11.3.7 IEEE/TMS320C3x Floating-Point Format Conversion

The fast version of the IEEE-to-TMS320C3x conversion routine was originally developed by Keith Henry of Apollo Computer, Inc. The other routines were based on this initial input.

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. For example, if a 32-bit number has the binary point after the most significant bit (which is also the sign bit), only fractional numbers (numbers with absolute values less than 1), can be represented. In other words, there is a number called a Q31 number, which is a number with 31 fractional bits. All operations assume that the binary point is fixed at this location. The fixed-point system, although simple to implement in hardware, imposes limitations in the dynamic range of the represented number, which causes scaling problems in many applications. You can avoid this difficulty by using floating-point numbers.

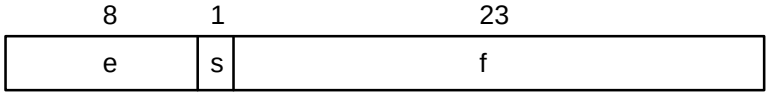
A floating-point number consists of a mantissa m multiplied by base b raised to an exponent e:

$$m * b^e$$

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

To achieve higher efficiency in hardware implementation, the TMS320C3x uses a floating-point format that differs from the IEEE standard. This section briefly describes the two formats and presents software routines to convert between them.

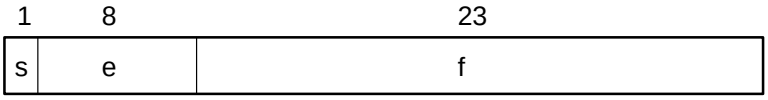
TMS320C3x floating-point format:



In a 32-bit word representing a floating-point number, the first eight bits correspond to the exponent expressed in two's-complement format. There is one bit for sign and 23 bits for the mantissa. The mantissa is expressed in two's-complement form, with the binary point after the most significant nonsign bit. Since this bit is the complement of the sign bit s, it is suppressed. In other words, the mantissa actually has 24 bits. A special case occurs when e = –128. In this case, the number is interpreted as 0, independently of the values of s and f (which are set to 0 by default). To summarize, the values of the represented numbers in the TMS320C3x floating-point format are as follows:

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

IEEE floating-point format:



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

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

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

Special cases:

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

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.

11.3.7.1 IEEE-to-TMS320C3x Floating-Point Format Conversion

Example 11–22 shows the fast conversion from IEEE to TMS320C3x 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.

Example 11–22. IEEE-to-TMS320C3x Conversion (Fast Version)

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

```
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 11–23 shows the complete conversion between the IEEE and TMS320C3x 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 TMS320C3x format. Otherwise, an underflow occurs, and the number is set to 0.

Example 11–23. 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


*   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
```

11.3.7.2 TMS320C3x-to-IEEE Floating-Point Format Conversion

The vast majority of the numbers represented by the TMS320C3x floating-point format are covered by the general IEEE format and the representation of 0s. The only special case is e = –127 in the TMS320C3x 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 11–24 shows the fast version, and Example 11–25 shows the complete version of the TMS320C3x-to-IEEE conversion.

Example 11–24. 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.
*
*
```

```
*  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 11–25. 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
```

```
*
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
```

11.4 Application-Oriented Operations

Certain features of the TMS320C3x architecture and instruction set facilitate the solution of numerically intensive problems. This section presents examples of applications using these features, such as companding, filtering, FFTs, and matrix arithmetic.

11.4.1 Companding

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

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

Example 11–26 and Example 11–27 show μ-law compression and expansion (that is, linear to μ-law and μ-law to linear conversion), while Example 11–28 and Example 11–29 show A-law compression and expansion. For expansion, using a look-up table is an alternative approach. A look-up table trades memory space for speed of execution. Since the compressed data is eight bits long, you can construct a table with 256 entries containing the expanded data. If the compressed data is stored in the register AR0, the following two instructions will put the expanded data in register R0:

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

You could use the same look-up table approach for compression, but the required table length would then be 16,384 words for μ-law or 8,192 words for A-law. If this memory size is not acceptable, use the subroutines presented in Example 11–26 or Example 11–28.

Example 11–26. μ-Law Compression

```
*
*   TITLE U±LAW COMPRESSION
*
*
*   SUBROUTINE MUCMPR
*
*

*   ARGUMENT  ASSIGNMENTS:

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

       .global  MUCMPR
*
MUCMPR    LDI    R0,R1        ;  Save sign of number
          ABSI   R0,R0
          CMPI   1FDEH,R0     ;  If R0>0x1FDE,
          LDIGT  1FDEH,R0     ;  saturate the result
          ADDI   33,R0        ;  Add bias


          FLOAT  R0           ;  Normalize: (seg+5)0WXYZx...x
          MPYF   0.03125,R0   ;  Adjust segment number by 2**(±5)
          LSH    1,R0         ;  (seg)WXYZx...x
          PUSHF  R0
          POP    R0           ;  Treat number as integer
          LSH    ±20,R0       ;  Right-justify


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

Example 11–27. μ-Law Expansion

```
*
*   TITLE U-LAW EXPANSION
*
*
*   SUBROUTINE  MUXPND
*
*

*   ARGUMENT  ASSIGNMENTS:
*
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*
*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*
*
*   CYCLES: 20 (WORST CASE)WORDS: 14
*
*

    .global MUXPND
*
MUXPND    NOT    R0,R0     ;  Complement bits
          LDI    R0,R1
          AND    0FH,R1    ;  Isolate quantization bin
          LSH    1,R1
          ADDI   33,R1     ;  Add bias to introduce 1xxxx1
          LDI    R0,R2     ;  Store for sign bit
          LSH    ±4,R0
          AND    7,R0      ;  Isolate segment code
          LSH3   R0,R1,R0  ;  Shift and put result in R0
          SUBI   33,R0     ;  Subtract bias
          TSTB   80H,R2    ;  Test sign bit
          RETSZ
          NEGI   R0        ;  Negate if a negative number
          RETS
```

Example 11–28. A-Law Compression

```
*
*   TITLE  A±LAW COMPRESSION
*
*
*   SUBROUTINE  ACMPR
*
*

*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*

*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*

*   NOTE:  SINCE THE STACK POINTER 'SP' IS USED IN THE COMPRESSION
*   ROUTINE 'ACMPR', MAKE SURE TO INITIALIZE IT IN THE
*   CALLING PROGRAM.
*
*

*   CYCLES:22 WORDS: 19

*
       .global  ACMPR
*
ACMPR     LDI       R0,R1        ;  Save sign of number
          ABSI      R0,R0
          CMPI      1FH,R0       ;  If R0<0x20,
          BLED      END          ;     do linear coding
          CMPI      0FFFH,R0     ;  If R0>0xFFF,
          LDIGT     0FFFH,R0     ;     saturate the result
          LSH       ±1,R0        ;  Eliminate rightmost bit


          FLOAT     R0           ;  Normalize: (seg+3)0WXYZx...x
          MPYF      0.125,R0     ;  Adjust segment number by 2**(±3)
          LSH       1,R0         ;  (seg)WXYZx...x
          PUSHF     R0
          POP       R0           ;  Treat number as integer
          LSH       ±20,R0       ;  Right±justify


END       LDI       0,R2
          LDI       R1,R1        ;  If number is negative,
          LDILT     80H,R2       ;     set sign bit
          ADDI      R2,R0        ;  R0 = compressed number
          XOR       0D5H,R0      ;  Invert even bits
                                 ;     for transmission
          RETS
*
```

Example 11–29. A-Law Expansion

```
*
*   TITLE A-LAW EXPANSION
*
*
*
*   SUBROUTINE  AXPND
*
*

*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R0        |   NUMBER TO BE CONVERTED
*

*   REGISTERS USED AS INPUT: R0
*   REGISTERS MODIFIED: R0, R1, R2, SP
*   REGISTER CONTAINING RESULT: R0
*
*

*      CYCLES: 25 (WORST CASE)WORDS: 16
*
*
       .global AXPND
*

AXPND  XOR    D5H,R0    ;  Invert even bits
       LDI    R0,R1
       AND    0FH,R1    ;  Isolate quantization bin
       LSH    1,R1
       LDI    R0,R2     ;  Store for bit sign
       LSH    ±4,R0
       AND    7,R0      ;  Isolate segment code
       BZ     SKIP1
       SUBI   1,R0
       ADDI   32,R1     ;  Create 1xxxx1
SKIP1  ADDI   1,R1      ;  OR 0xxxx1
       LSH3   R0,R1,R0  ;  Shift and put result in R0
       TSTB   80H,R2    ;  Test sign bit
       RETSZ
       NEGI   R0        ;  Negate if a negative number
       RETS
```

11.4.2 FIR, IIR, and Adaptive Filters

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

11.4.2.1 FIR Filters

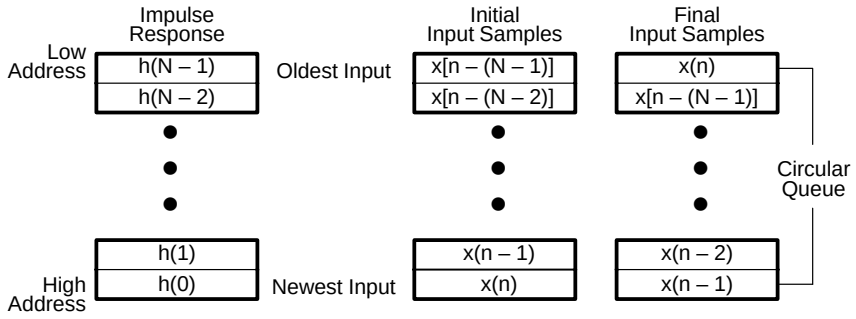
If the FIR filter has an impulse response h [0], h [1],..., h [N –1], and x[n] represents the input of the filter at time n, the output y [n] at time n is given by this equation:

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

Two features of the TMS320C3x that facilitate the implementation of the FIR filters are parallel multiply/add operations and circular addressing. The former permits the performance of a multiplication and an addition in a single machine cycle, while the latter makes a finite buffer of length N sufficient for the data x.

Figure 11–1 shows the arrangement of the memory locations necessary to implement circular addressing, while Example 11–30 presents the TMS320C3x assembly code for an FIR filter.

Figure 11–1. Data Memory Organization for an FIR Filter



To set up circular addressing, initialize the block-size register BK to block length N. Also, the locations for signal x should start from a memory location whose address is a multiple of the smallest power of 2 that is greater than N. For instance, if N = 24, the first address for x should be a multiple of 32 (the lowest five bits of the beginning address should be 0). See Section 5.3 on page 5-24 for more information.

In Example 11–30, the pointer to the input sequence x is incremented and is assumed to be moving from an older input to a newer input. At the end of the subroutine, AR1 will be pointing to the position for the next input sample.

Example 11–30. FIR Filter

```
*
*   TITLE  FIR FILTER
*
*
*   SUBROUTINE  FIR
*
*   EQUATION: y(n) = h(0) * x(n) + h(1) * x(n±1) +
*      ... + h(N±1) * x(n±(N±1))
*
*   TYPICAL CALLING SEQUENCE:
*
*   LOAD   AR0
*   LOAD   AR1
*   LOAD   RC
*   LOAD   BK
*   CALL   FIR
*
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |  FUNCTION
*   ----------+-------------------------------------
*   AR0       |  ADDRESS OF h(N±1)
*   AR1       |  ADDRESS OF x(n–(N±1))
*   RC        |  LENGTH OF FILTER ± 2 (N±2)
*   BK        |  LENGTH OF FILTER (N)
*
*   REGISTERS USED AS INPUT: AR0, AR1, RC, BK
*   REGISTERS MODIFIED: R0, R2, AR0, AR1, RC
*   REGISTER CONTAINING RESULT: R0
*
*
*   CYCLES: 11 + (N±1)  WORDS: 6
*
*
      .global FIR
*                                    ;  Initialize R0:
   FIR   MPYF3  *AR0++(1),*AR1++(1)%,R0
*                                    ;  h(N±1) * x(n±(N±1)) ±> R0
      LDF   0.0,R2                   ;  Initialize R2
*
*   FILTER (1 <= i < N)
*
      RPTS  RC                       ;  Set up the repeat cycle
      MPYF3 *AR0++(1),*AR1++(1)%,R0  ;  h(N±1±i)*x(n±(N±1±i))±>R0
||    ADDF3 R0,R2,R2                 ;  Multiply and add operation
```

```
*
        ADDF   R0,R2,R0                 ;  Add last product
*
*   RETURN SEQUENCE
*
        RETS                            ;  Return
*
*   end
*
    .end
```

11.4.2.2 IIR Filters

The transfer function of the IIR filters has both poles and 0s. Its output depends on both the input and the past output. As a rule, the filters need less computation than an FIR with similar frequency response, but the filters have the drawback of being sensitive to coefficient quantization. Most often, the IIR filters are implemented as a cascade of second-order sections, called biquads. Example 11–31 and Example 11–32 show the implementation for one biquad and for any number of biquads, respectively.

This is the equation for a single biquad:

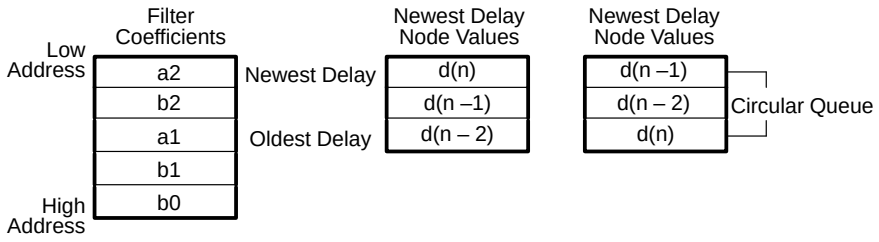
$$y[n] = a1\ \ y[n-1] + a2\ \ y[n-2] + b0\ x[n] + b1\ x[n-1] + b2\ x[n-2]$$

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

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

Figure 11–2 shows the memory organization for this two-equation approach, and Example 11–31 is an implementation of a single biquad on the TMS320C3x.

Figure 11–2. Data Memory Organization for a Single Biquad



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

Example 11–31. IIR Filter (One Biquad)

```
*
*   TITLE IIR FILTER
*
*
*   SUBROUTINE  IIR 1
*
*   IIR1 == IIR FILTER (ONE BIQUAD)
*
*
*   EQUATIONS:   d(n) = a2 * d(n±2) + a1 * d(n±1) + x(n)
*      y(n) = b2 * d(n±2) + b1 * d(n±1) + b0 * d(n)
*
*   OR y(n) = a1*y(n±1) + a2*y(n±2) + b0*x(n)
*         + b1*x(n±1) + b2*x(n±2)
*
*   TYPICAL CALLING SEQUENCE:
*
*      load   R2
*      load   AR0
*      load   AR1
*      load   BK
*      CALL   IIR1
*
*
*   ARGUMENT  ASSIGNMENTS:
*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R2        |   INPUT SAMPLE X(N)
*   AR0       |   ADDRESS OF FILTER COEFFICIENTS (A2)
*   AR1       |   ADDRESS OF DELAY MODE VALUES (D(N±2))
*   BK        |   BK = 3
*
*   REGISTERS USED AS INPUT: R2, AR0, AR1, BK
*   REGISTERS MODIFIED: R0, R1, R2, AR0, AR1
*   REGISTER CONTAINING RESULT: R0
*
*   CYCLES: 11   WORDS: 8
*
*
*   FILTER
```

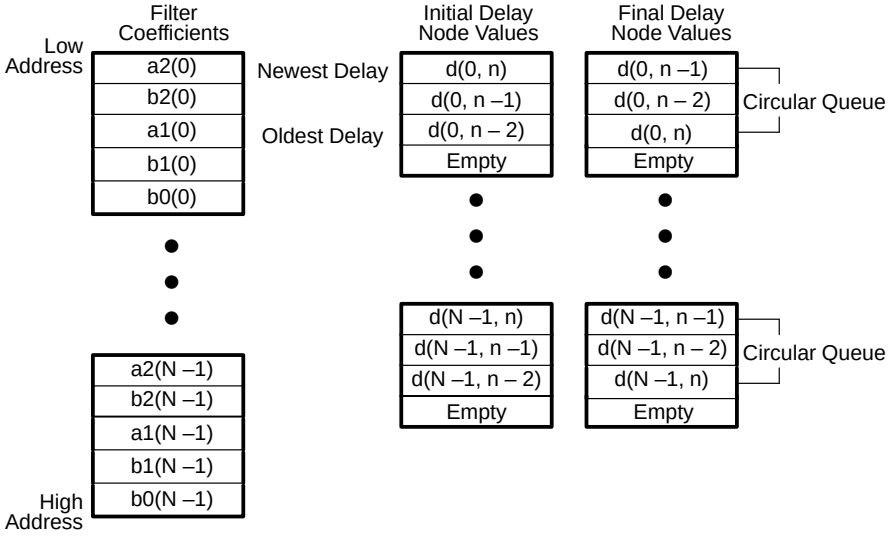
```
*
       .global  IIR1
*
IIR1   MPYF3  *AR0,*AR1,R0
*                                    ;  a2 * d(n±2) ±> R0
       MPYF3  *++AR0(1),*AR1––(1) % ,R1
*                                    ;  b2 * d(n±2) ±> R1
*
       MPYF3  *++AR0(1),*AR1,R0      ;  a1 * d(n±1) ±> R0
||     ADDF3  R0,R2,R2               ;  a2*d(n±2)+x(n) ±> R2
*
       MPYF3  *++AR0(1),*AR1––(1)%,R0;  b1 * d(n±1) ±> R0
||     ADDF3  R0,R2,R2               ;  a1*d(n±1)+a2*d(n±2)+x(n) ±> R2
*
       MPYF3  *++AR0(1),R2,R2        ;  b0 * d(n) ±> R2
||     STF    R2,*AR1++(1)%
*
*                                    ;  Store d(n)and point to d(n±1)
*
       ADDF   R0,R2                  ;  b1*d(n±1)+b0*d(n) ±> R2
       ADDF   R1,R2,R0               ;  b2*d(n±2)+b1*d(n±1)
                                     ;      +b0*d(n) ±> R0
*
*      RETURN SEQUENCE
*
       RETS                          ;  Return
*

*   end
*

    .end
```

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

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

Figure 11–3 shows the corresponding memory organization, while Example 11–32 shows the TMS320C3x assembly-language code.

Figure 11–3. Data Memory Organization for N Biquads



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

Example 11–32. IIR Filters (N > 1 Biquads)

```
*
*   TITLE IIR FILTERS (N > 1 BIQUADS)

*
*
*   SUBROUTINE IIR2
*
*
*
*   EQUATIONS: y(0,n) = x(n)
*
*   FOR (i = 0; i < N; i++)

*      {

*      d(i,n) = a2(i) * d(i,n±2) + a1(i) * d(i,n±1) * y(i±1,n)
*      y(i,n) = b2(i) * d(i,n±2) + b1(i) * d(i,n±1) * b0(i) * d(i,n)

*      TYPICAL CALLING SEQUENCE:
*      }

*   y(n) = y(N±1,n)
*
*   TYPICAL CALLING SEQUENCE:
*

*      load   R2
*      load   AR0
*      load   AR1
*      load   IR0
*      load   IR1
*      load   BK
*      load   RC
*      CALL   IIR2
*
*

*   ARGUMENT  ASSIGNMENT:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R2        |   INPUT SAMPLE x(n)
*   ARO       |   ADDRESS OF FILTER COEFFICIENTS (a2(0))
*   AR1       |   ADDRESS OF DELAY NODE VALUES (d(0,n±2))
*   BK        |   BK = 3
*   IR0       |   IR0 = 4
*   IR1       |   IR1 = 4*N±4
*   RC        |   NUMBER OF BIQUADS (N) ±2
*

*   REGISTERS USED AS INPUT; R2, AR0, AR1, IR0, IR1, BK, RC
*   REGISTERS MODIFIED; R0, R1, R2, AR0, AR1, RC
*   REGISTERS CONTAINING RESULT: R0
*
```

```
*   CYCLES: 17 + 6N  WORDS: 17
*
*
*
*
        .global IIR2
*
IIR2    MPYF3  *AR0, *AR1, R0
*                                       ;   a2(0) * d(0,n±2) ±> R0
        MPYF3  *AR0++(1), *AR1––(1)%, R1
*                                       ;   b2(0) * d(0,n±2) ±> R1


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

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

11.4.2.3 Adaptive Filters (LMS Algorithm)

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

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

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

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

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

β is a constant for the computation. You can interleave the updating of the filter coefficients with the computation of the filter output so that it takes three cycles per filter tap to do both. The updated coefficients are written over the old filter coefficients. Example 11–33 shows the implementation of an adaptive FIR filter on the TMS320C3x. The memory organization and the positioning of the data in memory should follow the same rules that apply to the FIR filter described in subsection 11.4.2.1 on page 11-58.

Example 11–33. Adaptive FIR Filter (LMS Algorithm)

```
*   TITLE ADAPTIVE FIR FILTER (LMS ALGORITHM)


*
*   SUBROUTINE LMS


*   LMS == LMS ADAPTIVE FILTER
*
*
*
*   EQUATIONS: y(n) = h(n,0)*x(n) + h(n,1)*x(n±1) + ...

*                  + h(n,N±1)*x(n±(N±1))

*            FOR (i = 0; i < N; i++)

*                  h(n+1,i) = h(n,i) + tmuerr * x(n±i)

*
*   TYPICAL CALLING SEQUENCE:
*

*   load   R4
*   load   AR0
*   load   AR1
*   load   RC
*   load   BK
*   CALL   LMS
*
*

*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   R4        |   SCALE FACTOR (2 * mu * err)
*   AR0       |   ADDRESS OF h(n,N±1)
*   AR1       |   ADDRESS OF x(n±(N±1))
*   RC        |   LENGTH OF FILTER ± 2 (N±2)
*   BK        |   LENGTH OF FILTER (N)
*
```

```
    *   REGISTERS USED AS INPUT: R4, AR0, AR1, RC, BK
    *   REGISTERS MODIFIED: R0, R1, R2, AR0, AR1, RC
    *   REGISTER CONTAINING RESULT: R0
    *
    *   PROGRAM SIZE: 10 words
    *
    *   EXECUTION CYCLES: 14 + 3(N±1)
    *
    *
    *   SETUP (i = 0)
    *

        .global LMS

    *                                   ;   Initialize R0:
        LMS     MPYF3  *AR0, *AR1, R0
    *                                   ;   h(n,N±1) * x(n±(N±1)) ±> R0
        LDF     0.0,R2                  ;   Initialize R2
    *
    *                                   ;   Initialize R1:
        MPYF3  *AR1++(1)%, R4, R1
    *                                   ;   x(n±(N±1)) * tmuerr ±> R1
        ADDF3  *AR0++(1), R1, R1
    *                                   ;   h(n,N±1) + x(n±(N±1)) *
    *                                   ;   tmuerr ±> R1
    *

    *   FILTER AND UPDATE (1 <= I < N)
    *

        RPTB   LOOP                     ;   Set up the repeat block
    *
    *                                   ;   Filter:
        MPYF3  *AR0−−(1),*AR1,R0        ;   h(n,N±1±i)
                                        ;       * x(n±(N±1±i)) ±> R0
||      ADDF3  R0,R2,R2                 ;   Multiply and add operation
    *
    *                                   ;   UPDATE:
        MPYF3  *AR1++(1)%,R4,R1         ;   x(n,N±(N±1±i)) * tmuerr ±> R1
||      STF    R1,*AR0++(1)             ;   R1 ±> h(n+1,N±1±(i±1))
*
LOOP    ADDF3  *AR0++(1), R1, R1
*                                       ;   h(n,N±1±i) + x(n±(N±1±i))
                                        ;       *tmuerr ±> R1
*
        ADDF3  R0,R2,R0                 ;   Add last product
        STF    R1,*±AR0(1)              ;   h(n,0) + x(n)
                                        ;       * tmuerr ±> h(n+1,0)
*

*       RETURN SEQUENCE
```

```
*
        RETS                        ;   Return
*

*   end
*
    .end
```

11.4.3 Matrix-Vector Multiplication

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

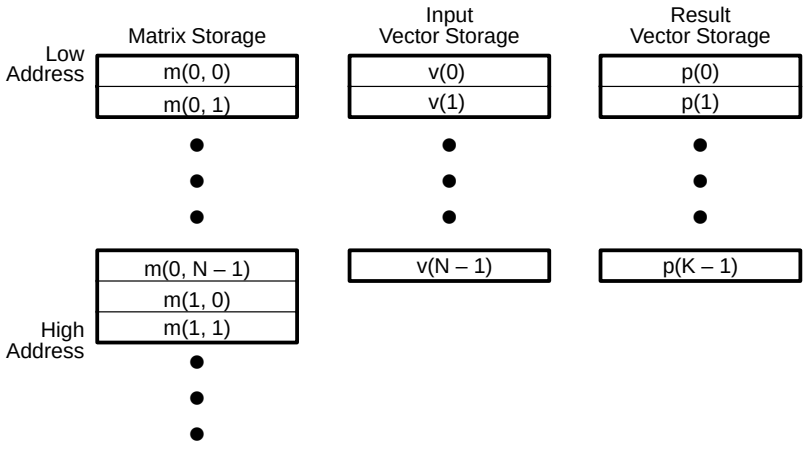
$$p(i) = m(i,0)\,v(0) + m(i,1)\,v(1) + \ldots + m(i,N-1)\,v(N-1) \quad i = 0,1,\ldots,K-1$$

This is essentially a dot product, and the matrix-vector multiplication contains, as a special case, the dot product presented in Example 11–2 on page 11-7. In pseudo-C format, the computation of the matrix multiplication is expressed by

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

Figure 11–4 shows the data memory organization for matrix-vector multiplication, and Example 11–34 shows the TMS320C3x assembly code that implements it. Note that in Example 11–34, K (number of rows) should be greater than 0, and N (number of columns) should be greater than 1.

Figure 11–4. Data Memory Organization for Matrix-Vector Multiplication



Example 11–34. Matrix Times a Vector Multiplication

```
*
*   TITLE MATRIX TIMES A VECTOR MULTIPLICATION
*
*
*   SUBROUTINE MAT
*
*   MAT == MATRIX TIMES A VECTOR OPERATION
*
*
*   TYPICAL CALLING SEQUENCE:*
*   load   AR0
*   load   AR1
*   load   AR2
*   load   AR3
*   load   R1
*   CALL   MAT
*
*


*   ARGUMENT  ASSIGNMENTS:

*   ARGUMENT  |   FUNCTION
*   ----------+-------------------------------------
*   AR0       |   ADDRESS OF M(0,0)
*   AR1       |   ADDRESS OF V(0)
*   AR2       |   ADDRESS OF P(0)
*   AR3       |   NUMBER OF ROWS ± 1 (K±1)
*   R1        |   NUMBER OF COLUMNS ± 2 (N±2)
*

*   REGISTERS USED AS INPUT: AR0, AR1, AR2, AR3, R1
*   REGISTERS MODIFIED: R0, R2, AR0, AR1, AR2, AR3, IR0,
*      RC, RSA, REA
*
*
*   PROGRAM SIZE: 11
*
*   EXECUTION CYCLES: 6 + 10 * K + K * (N ± 1)
*
*
*


      .global  MAT

*
*   SETUP

*
MAT   LDI    R1,IR0                ;   Number of columns±2 ±> IR0
      ADDI   2,IR0                 ;   IR0 = N
```

```
        *
        *   FOR (i = 0; i < K; i++) LOOP OVER THE ROWS
        *

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

        *   FOR (j = 1; j < N; j++) DO DOT PRODUCT OVER COLUMNS
        *

               RPTS   R1                      ;  Multiply a row by a column
        *
               MPYF3  *AR0++(1),*AR1++(1),R0  ;  m(i,j) * v(j) ±> R0
        ||     ADDF3  R0,R2,R2                ;  m(i,j±1) * v(j±1) + R2 ±> R2
        *
               DBD    AR3,ROWS                ;  Counts the no. of rows left
        *
        *
               ADDF   R0,R2                   ;  Last accumulate
               STF    R2,*AR2++(1)            ;  Result ±> p(i)

               NOP    *−−AR1(IR0)             ;  Set AR1 to point to v(0)

        *   !!! DELAYED BRANCH HAPPENS HERE !!!
        *
        *   RETURN SEQUENCE
        *

RETS                                          ;  Return

        *   end
        *
            .end
```

11.4.4 Fast Fourier Transforms (FFT)

Fourier transforms are an important tool often used in digital signal processing systems. The purpose of the transform is to convert information from the time domain to the frequency domain. The inverse Fourier transform converts information back to the time domain from the frequency domain. Implementation of Fourier transforms that are computationally efficient are known as fast Fourier transforms (FFTs). The theory of FFTs can be found in books such as *DFT/FFT and Convolution Algorithms* by C.S. Burrus and T.W. Parks (John Wiley, 1985) and *Digital Signal Processing Applications with the TMS320 Family* by Texas Instruments (literature number SPRA012A).

Fast Fourier transform is a label for a collection of algorithms that implement efficient conversion from time to frequency domain. There are several types of FFTs:

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

Certain TMS320C3x features that increase efficient implementation of numerically intensive algorithms are particularly well-suited for FFTs. The high speed of the device (33-ns cycle time) makes implementation of real-time algorithms easier, while floating-point capability eliminates the problems associated with dynamic range. The powerful indirect-addressing indexing scheme facilitates the access of FFT butterfly legs with different spans. The repeat block implemented by the RPTB instruction reduces the looping overhead in algorithms heavily dependent on loops (such as the FFTs). This construct provides the efficiency of in-line coding in loop form. The FFT will reverse the bit order of the output; therefore, the output must be reordered. This reordering does not require extra cycles, because the device has a special mode of indirect addressing (bit-reversed addressing) for accessing the FFT output in the original order.

The examples in this subsection were based on programs contained in the Burrus and Parks book and in the paper *Real-Valued Fast Fourier Transform Algorithms* by H.V. Sorensen, et al (IEEE Transform on ASSP, June 1987).

Example 11–35 and Example 11–36 show the implementation of a complex radix-2, DIF FFT on the TMS320C3x. Example 11–35 contains the generic code of the FFT, which can be used with a number of any length. However, for the complete implementation of an FFT, you need a table of twiddle factors (sines/cosines); the length of the table depends on the size of the transform. To retain the generic form of Example 11–35, the table with the twiddle factors (containing 1-1/4 complete cycles of a sine) is presented separately in Example 11–36 for the case of a 64-point FFT. A full cycle of a sine should have a number of points equal to the FFT size. Example 11–36 uses two variables: N, which is the FFT length, and M, which is the logorithm of N to a base equal to the radix. In other words, M is the number of stages of the FFT. For example, in a 64-point FFT, M = 6 when using a radix-2 algorithm, and M = 3 when using a radix-4 algorithm. If the table with the twiddle factors and the FFT code are kept in separate files, they should be connected at link time.

Example 11–35. Complex, Radix-2, DIF FFT

```
*
*   TITLE COMPLEX, RADIX–2, DIF FFT
*
*   GENERIC PROGRAM FOR LOOPED±CODE RADIX±2 FFT COMPUTATION IN TMS320C3x
*
*   THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 111.
*   THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY.  THE COMPUTATION
*   IS DONE IN  PLACE, BUT THE RESULT IS MOVED TO ANOTHER MEMORY
*   SECTION TO DEMONSTRATE THE BIT±REVERSED ADDRESSING.
*
*   THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE THAT IS PUT IN A .DATA
*   SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE
*   GENERIC NATURE OF THE PROGRAM.  FOR THE SAME PURPOSE, THE SIZE OF
*   THE FFTN AND LOG2(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED
*   DURING LINKING.
*
*

       .globl    FFT             ;  Entry point for execution
       .globl    N               ;  FFT size
       .globl    M               ;  LOG2(N)
       .globl    SINE            ;  Address of sine table


INP    .usect “IN”,1024           ;  Memory with input data
.BSS   OUTP,1024                  ;  Memory with output data


       .text

*   INITIALIZE

FFTSIZ .word  N
LOGFFT .word  M
SINTAB .word  SINE
INPUT  .word  INP
OUTPUT .word  OUTP


FFT:       LDP    FFTSIZ          ;  Command to load data page pointer


LDI        @FFTSIZ,IR1
LSH        ±2,IR1                 ;  IR1 = N/4, pointer for SIN/COS table
LDI        0,AR6                  ;  AR6 holds the current stage number
LDI        @FFTSIZ,IR0
LSH        1,IR0                  ;  IR0 = 2*N1 (because of real/imag)
LDI        @FFTSIZ,R7             ;  R7 = N2
LDI        1,AR7                  ;  Initialize repeat counter
                                  ;      of first loop
LDI        1,AR5                  ;  Initialize IE index (AR5 = IE)
```

```
*   OUTER LOOP

LOOP:     NOP    *++AR6(1)          ;  Current FFT stage
          LDI    @INPUT,AR0         ;  AR0 points to X(I)
          ADDI   R7,AR0,AR2         ;  AR2 points to X(L)
          LDI    AR7,RC
          SUBI   1,RC               ;  RC should be one less than desired #


*   FIRST LOOP

          RPTB   BLK1
          ADDF   *AR0,*AR2,R0       ;  R0 = X(I)+X(L)
          SUBF   *AR2++,*AR0++,R1   ;  R1 = X(I)±X(L)
          ADDF   *AR2,*AR0,R2       ;  R2 = Y(I)+Y(L)
          SUBF   *AR2,*AR0,R3       ;  R3 = Y(I)±Y(L)
          STF    R2,*AR0--          ;  Y(I) = R2 and...
   ||     STF    R3,*AR2--          ;  Y(L) = R3
   BLK1   STF    R0,*AR0++(IR0)     ;  X(I) = R0 and...
   ||     STF    R1,*AR2++(IR0)     ;  X(L) = R1 and AR0,2 = AR0,2 + 2*n

*  IF THIS IS THE LAST STAGE, YOU ARE DONE

          CMPI   @LOGFFT,AR6
          BZD    END

*   MAIN INNER LOOP

          LDI    2,AR1              ;  Init loop counter for
                                    ;     inner loop
          LDI    @SINTAB,AR4        ;  Initialize IA index (AR4 = IA)
INLOP:    ADDI   AR5,AR4            ;  IA = IA+IE; AR4 points to
                                    ;     cosine
          LDI    AR1,AR0
          ADDI   2,AR1              ;  Increment inner loop counter
          ADDI   @INPUT,AR0         ;  (X(I),Y(I)) pointer
          ADDI   R7,AR0,AR2         ;  (X(L),Y(L)) pointer
          LDI    AR7,RC
          SUBI   1,RC               ;  RC should be 1 less than
                                    ;     desired #
          LDF    *AR4,R6            ;  R6 = SIN

*   SECOND LOOP

          RPTB      BLK2
          SUBF      *AR2,*AR0,R2    ;  R2 = X(I)±X(L)
          SUBF      *+AR2,*+AR0,R1
*                                   ;  R1 = Y(I)±Y(L)
          MPYF      R2,R6,R0        ;  R0 = R2*SIN and...
||        ADDF      *+AR2,*+AR0,R3
*                                   ;  R3 = Y(I)+Y(L)
          MPYF      R1,*+AR4(IR1),R3;  R3 = R1*COS and ...
||        STF       R3,*+AR0        ;  Y(I) = Y(I)+Y(L)
```

```
       SUBF      R0,R3,R4          ;  R4 = R1 * COS±R2 * SIN
       MPYF      R1,R6,R0          ;  R0 = R1 * SIN and...
||     ADDF      *AR2,*AR0,R3      ;  R3 = X(I) + X(L)
       MPYF      R2,*+AR4(IR1),R3  ;  R3 = R2 * COS and...
||     STF       R3,*AR0++(IR0)
*
*                                  ;  X(I) = X(I)+X(L) and AR0 = AR0+2*N1
       ADDF      R0,R3,R5          ;  R5 = R2*COS+R1*SIN
BLK2   STF R5,*AR2++(IR0)          ;  X(L) = R2 * COS+R1 * SIN,
                                   ;      incr AR2 and...
||     STF R4,*+AR2                ;  Y(L) = R1*COS±R2*SIN


       CMPI      R7,AR1
       BNE       INLOP             ;  Loop back to the inner loop


       LSH       1,AR7             ;  Increment loop counter for next time
       BRD       LOOP              ;  Next FFT stage (delayed)
       LSH       1,AR5             ;  IE = 2*IE
       LDI       R7,IR0            ;  N1 = N2
       LSH       ±1,R7             ;  N2 = N2/2

*   STORE RESULT OUT USING BIT-REVERSED ADDRESSING


END:   LDI       @FFTSIZ,RC        ;  RC = N
       SUBI      1,RC              ;  RC should be one less than desired #
       LDI       @FFTSIZ,IR0       ;  IR0 = size of FFT = N
       LDI       2,IR1
       LDI       @INPUT,AR0
       LDI       @OUTPUT,AR1


       RPTB      BITRV
       LDF       *+AR0(1),R0
||     LDF       *AR0++(IR0)B,R1
BITRV  STF       R0,*+AR1(1)
||     STF       R1,*AR1++(IR1)


SELF   BR        SELF              ;  Branch to itself at the end
       .end
```

Example 11–36. Table With Twiddle Factors for a 64-Point FFT

```
*
*TITLE TABLE WITH TWIDDLE FACTORS FOR A 64±POINT FFT
*
*   FILE TO BE LINKED WITH THE SOURCE CODE FOR A 64–POINT, RADIX±2 FFT

*

          .globl SINE
          .globl N
          .globl M


N         .set   64
M         .set   6


          .data

SINE
          .float       0.000000
          .float       0.098017
          .float       0.195090
          .float       0.290285
          .float       0.382683
          .float       0.471397
          .float       0.555570
          .float       0.634393
          .float       0.707107
          .float       0.773010
          .float       0.831470
          .float       0.881921
          .float       0.923880
          .float       0.956940
          .float       0.980785
          .float       0.995185
COSINE
          .float       1.000000
          .float       0.995185
          .float       0.980785
          .float       0.956940
          .float       0.923880
          .float       0.881921
          .float       0.831470
          .float       0.773010
          .float       0.707107
          .float       0.634393
          .float       0.555570
          .float       0.471397
          .float       0.382683
          .float       0.290285
          .float       0.195090
```

```
.float       0.098017
.float       0.000000
.float ±     0.098017
.float ±     0.195090
.float ±     0.290285
.float ±     0.382683
.float –     0.471397
.float      –0.555570
.float –     0.634393
.float –     0.707107
.float –     0.773010
.float –     0.831470
.float –     0.881921
.float –     0.923880
.float –     0.956940
.float –     0.980785
.float –     0.995185
.float      –1.000000
.float –     0.995185
.float –     0.980785
.float –     0.956940
.float –     0.923880
.float –     0.881921
.float –     0.831470
.float –     0.773010
.float –     0.707107
.float –     0.634393
.float –     0.555570
.float –     0.471397
.float –     0.382683
.float –     0.290285
.float –     0.195090
.float –     0.098017
```

```
.float      0.000000
.float      0.098017
.float      0.195090
.float      0.290285
.float      0.382683
.float      0.471397
.float      0.555570
.float      0.634393
.float      0.707107
.float      0.773010
.float      0.831470
.float      0.881921
.float      0.923880
.float      0.956940
.float      0.980785
.float      0.995185
```

The radix-2 algorithm has tutorial value, because the functioning of the FFT algorithm is relatively easy to understand. However, radix-4 implementation can increase execution speed by reducing the amount of arithmetic required. Example 11–37 shows the generic implementation of a complex, DIF FFT in radix-4. A companion table, such as the one in Example 11–36, should have a value of M equal to the logN, where the base of the logarithm is 4.

Example 11–37. Complex, Radix-4, DIF FFT

```
*
*   TITLE COMPLEX, RADIX-4, DIF FFT
*
*   GENERIC PROGRAM TO PERFORM A LOOPED±CODE RADIX±4 FFT COMPUTATION
*   IN THE TMS320C3x
*
*   THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 117.
*   THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY, AND THE COMPUTATION
*   IS DONE IN PLACE.
*
*   THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE THAT IS PUT IN A .DATA
*   SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE
*   GENERIC NATURE OF THE PROGRAM. FOR THE SAME PURPOSE, THE SIZE OF
*   THE FFT N AND LOG4(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND
*   SPECIFIED DURING LINKING.
*
*   IN ORDER TO HAVE THE FINAL RESULT IN BIT±REVERSED ORDER, THE TWO
*   MIDDLE BRANCHES OF THE RADIX±4 BUTTERFLY ARE INTERCHANGED DURING
*   STORAGE. NOTE THIS DIFFERENCE WHEN COMPARING WITH THE PROGRAM IN
*   P. 117 OF THE BURRUS AND PARKS BOOK.
*
*
        .globl    FFT       ;  Entry point for execution
        .globl    N         ;  FFT size
        .globl    M         ;  LOG4(N)
        .globl    SINE      ;  Address of sine table

        .usect    "IN",1024 ;  Memory with input data

        .text


*   INITIALIZE


TEMP    .word     $+2
STORE   .word     FFTSIZ    ;  Beginning of temp storage area
        .word     N
        .word     M
        .word     SINE
        .word     INP


        .BSS      FFTSIZ,1  ;  FFT size
        .BSS      LOGFFT,1  ;  LOG4(FFTSIZ)
        .BSS      SINTAB,1  ;  Sine/cosine table base
        .BSS      INPUT,1   ;  Area with input data to process
        .BSS      STAGE,1   ;  FFT stage #
        .BSS      RPTCNT,1  ;  Repeat counter
        .BSS      IEINDX,1  ;  IE index for sine/cosine
```

```
        .BSS    LPCNT,1   ;  Second±loop count
        .BSS    JT,1      ;  JT counter in program, P. 117
        .BSS    IA1,1     ;  IA1 index in program, P. 117

    FFT:


*   INITIALIZE DATA LOCATIONS

    LDP   TEMP            ;  Command to load data page counter
    LDI   @TEMP,AR0
    LDI   @STORE,AR1
    LDI   *AR0++,R0       ;  Xfer data from one memory to the other
    STI   R0,*AR1++
    LDI   *AR0++,R0
    STI   R0,*AR1++
    LDI   *AR0++,R0
    STI   R0,*AR1++
    LDI   *AR0,R0
    STI   R0,*AR1


    LDP   FFTSIZ          ;  Command to load data page pointer
    LDI   @FFTSIZ,R0
    LDI   @FFTSIZ,IR0
    LDI   @FFTSIZ,IR1
    LDI   0,AR7
    STI   AR7,@STAGE      ;  @STAGE holds the current stage number
    LSH   1,IR0           ;  IR0 = 2*N1 (because of real/imag)
    LSH   ±2,IR1          ;  IR1 = N/4, pointer for SIN/COS table
    LDI   1,AR7
    STI   AR7,@RPTCNT     ;  Init repeat counter of first loop
    STI   AR7,@IEINDX     ;  Init. IE index
    LSH   ±2,R0           ;  JT = R0/2+2
    ADDI  2,R0
    STI   R0,@JT
    SUBI  2,R0
    LSH   1,R0            ;  R0 = N2

*   OUTER LOOP

LOOP:
    LDI   @INPUT,AR0      ;  AR0 points to X(I)
    ADDI  R0,AR0,AR1      ;  AR1 points to X(I1)
    ADDI  R0,AR1,AR2      ;  AR2 points to X(I2)
    ADDI  R0,AR2,AR3      ;  AR3 points to X(I3)
    LDI   @RPTCNT,RC
    SUBI  1,RC            ;  RC should be one less than desired #

*   FIRST LOOP

    RPTB  BLK1
    ADDF  *+AR0,*+AR2,R1
```

```
*                                ;  R1 = Y(I)+Y(I2)
       ADDF   *+AR3,*+AR1,R3
*                                ;  R3 = Y(I1)+Y(I3)
       ADDF   R3,R1,R6           ;  R6 = R1+R3
       SUBF   *+AR2,*+AR0,R4
*                                ;  R4 = Y(I)±Y(I2)
       STF    R6,*+AR0           ;  Y(I) = R1+R3
       SUBF   R3,R1              ;  R1 = R1±R3
       LDF    *AR2,R5            ;  R5 = X(I2)
||     LDF    *+AR1,R7           ;  R7 = Y(I1)
       ADDF   *AR3,*AR1,R3       ;  R3 = X(I1)+X(I3)
       ADDF   R5,*AR0,R1         ;  R1 = X(I)+X(I2)
||     STF    R1,*+AR1           ;  Y(I1) = R1±R3
       ADDF   R3,R1,R6           ;  R6 = R1+R3
       SUBF   R5,*AR0,R2         ;  R2 = X(I)±X(I2)
||     STF    R6,*AR0++(IR0)     ;  X(I) = R1+R3
       SUBF   R3,R1              ;  R1 = R1±R3
       SUBF   *AR3,*AR1,R6       ;  R6 = X(I1)±X(I3)
       SUBF   R7,*+AR3,R3        ;  ±R3 = Y(I1)±Y(I3)
||     STF    R1,*AR1++(IR0)     ;  X(I1) = R1±R3
       SUBF   R6,R4,R5           ;  R5 = R4±R6
       ADDF   R6,R4              ;  R4 = R4+R6
       STF    R5,*+AR2           ;  Y(I2) = R4±R6
||     STF    R4,*+AR3           ;  Y(I3) = R4+R6
       SUBF   R3,R2,R5           ;  R5 = R2±R3
       ADDF   R3,R2              ;  R2 = R2+R3
BLK1   STF    R5,*AR2++(IR0)     ;  X(I2) = R2±R3
||     STF R2,*AR3++(IR0)        ;  X(I3) = R2+R3


*  IF THIS IS THE LAST STAGE, YOU ARE DONE

       LDI    @STAGE,AR7
       ADDI   1,AR7
       CMPI   @LOGFFT,AR7
       BZD    END
       STI    AR7,@STAGE         ;  Current FFT stage

*  MAIN INNER LOOP

       LDI    1,AR7
       STI    AR7,@IA1           ;  Init IA1 index
       LDI    2,AR7
       STI    AR7,@LPCNT         ;  Init loop counter for inner loop
                                 ;     INLOP:
       LDI    2,AR6              ;  Increment inner loop counter
       ADDI   @LPCNT,AR6
       LDI    @LPCNT,AR0
       LDI    @IA1,AR7
       ADDI   @IEINDX,AR7        ;  IA1 = IA1+IE
       ADDI   @INPUT,AR0         ;  (X(I),Y(I)) pointer
       STI    AR7,@IA1
```

```
        ADDI   R0,AR0,AR1       ;  (X(I1),Y(I1)) pointer
        STI    AR6,@LPCNT
        ADDI   R0,AR1,AR2       ;  (X(I2),Y(I2)) pointer
        ADDI   R0,AR2,AR3       ;  (X(I3),Y(I3)) pointer
        LDI    @RPTCNT,RC
        SUBI   1,RC             ;  RC should be one less than desired #
        CMPI   @JT,AR6          ;  If LPCNT = JT, go to
        BZD    SPCL             ;     special butterfly
        LDI    @IA1,AR7
        LDI    @IA1,AR4
        ADDI   @SINTAB,AR4      ;  Create cosine index AR4
        SUBI   1,AR4            ;  Adjust sine table pointer
        ADDI   AR4,AR7,AR5
        SUBI   1,AR5            ;  IA2 = IA1+IA1±1
        ADDI   AR7,AR5,AR6
        SUBI   1,AR6            ;  IA3 = IA2+IA1±1

*   SECOND LOOP

        RPTB   BLK2
        ADDF   *+AR2,*+AR0,R3
*                               ;  R3 = Y(I)+Y(I2)
        ADDF   *+AR3,*+AR1,R5
*                               ;  R5 = Y(I1)+Y(I3)
        ADDF   R5,R3,R6         ;  R6 = R3+R5
        SUBF   *+AR2,*+AR0,R4
*                               ;  R4 = Y(I)±Y(I2)
        SUBF   R5,R3            ;  R3 = R3±R5
        ADDF   *AR2,*AR0,R1     ;  R1 = X(I)+X(I2)
        ADDF   *AR3,*AR1,R5     ;  R5 = X(I1)+X(I3)
        MPYF   R3,*+AR5(IR1),R6 R6 = R3*CO2
||      STF    R6,*+AR0         ;  Y(I) = R3+R5
        ADDF   R5,R1,R7         ;  R7 = R1+R5
        SUBF   *AR2,*AR0,R2     ;  R2 = X(I)±X(I2)
        SUBF   R5,R1            ;  R1 = R1±R5
        MPYF   R1,*AR5,R7       ;  R7 = R1*SI2
||      STF R7,*AR0++(IR0)      ;  X(I) = R1+R5
        SUBF   R7,R6            ;  R6 = R3*CO2±R1*SI2
        SUBF   *+AR3,*+AR1,R5
*                               ;  R5 = Y(I1)±Y(I3)
        MPYF   R1,*+AR5(IR1),R7 ;  R7 = R1*C02
||      STF    R6,*+AR1         ;  Y(I1) = R3*CO2±R1*SI2
        MPYF   R3,*AR5,R6       ;  R6 = R3*SI2
        ADDF   R7,R6            ;  R6 = R1*CO2+R3*SI2
        ADDF   R5,R2,R1         ;  R1 = R2+R5
        SUBF   R5,R2            ;  R2 = R2±R5
        SUBF   *AR3,*AR1,R5     ;  R5 = X(I1)±X(I3)
        SUBF   R5,R4,R3         ;  R3 = R4±R5
        ADDF   R5,R4            ;  R4 = R4+R5
        MPYF   R3,*+AR4(IR1),R6 ;  R6 = R3*CO1
||      STF R6,*AR1++(IR0)      ;  X(I1) = R1*CO2+R3*SI2
```

```
       MPYF   R1,*AR4,R7       ;  R7 = R1*SI1
       SUBF   R7,R6            ;  R6 = R3*CO1±R1*SI1
       MPYF   R1,*+AR4(IR1),R6 ;  R6 = R1*CO1
||     STF R6,*+AR2            ;  Y(I2) = R3*CO1±R1*SI1
       MPYF   R3,*AR4,R7       ;  R7 = R3*SI1
       ADDF   R7,R6            ;  R6 = R1*C O1+R3*SI1
       MPYF   R4,*+AR6(IR1),R6 ;  R6 = R4*CO3
||     STF R6,*AR2++(IR0)      ;  X(I2) = R1*CO1+R3*SI1
       MPYF   R2,*AR6,R7       ;  R7 = R2*SI3
       SUBF   R7,R6            ;  R6 = R4*CO3±R2*SI3
       MPYF   R2,*+AR6(IR1),R6 ;  R6 = R2*CO3
||     STF R6,*+AR3            ;  Y(I3) = R4*CO3±R2*SI3
       MPYF   R4,*AR6,R7       ;  R7 = R4*SI3
       ADDF   R7,R6            ;  R6 = R2*CO3+R4*SI3

BLK2   STF    R6,*AR3++(IR0)
*                              ;  x(i3) = R2*CO3+R4*SI3


       CMPI   @LPCNT,R0
       BP  INLOP               ;  Loop back to the inner loop
       BR  CONT


*   SPECIAL BUTTERFLY FOR W = J


SPCL   LDI IR1,AR4
       LSH ±1,AR4              ;  Point to SIN(45)
       ADDI   @SINTAB,AR4      ;  Create cosine index AR4 = CO21


       RPTB   BLK3
       ADDF   *AR2,*AR0,R1     ;  R1 = X(I)+X(I2)
       SUBF   *AR2,*AR0,R2     ;  R2 = X(I)±X(I2)
       ADDF   *+AR2,*+AR0,R3
*                              ;  R3 = Y(I)+Y(I2)
       SUBF   *+AR2,*+AR0,R4
*                              ;  R4 = Y(I)±Y(I2)
       ADDF   *AR3,*AR1,R5     ;  R5 = X(I1)+X(I3)
       SUBF   R1,R5,R6         ;  R6 = R5±R1
       ADDF   R5,R1            ;  R1 = R1+R5
       ADDF   *+AR3,*+AR1,R5
*                              ;  R5 = Y(I1)+Y(I3)
       SUBF   R5,R3,R7         ;  R7 = R3±R5
       ADDF   R5,R3            ;  R3 = R3+R5
       STF    R3,*+AR0         ;  Y(I) = R3+R5
||     STF    R1,*AR0++(IR0)   ;  X(I) = R1+R5
       SUBF   *AR3,*AR1,R1     ;  R1 = X(I1)±X(I3)
       SUBF   *+AR3,*+AR1,R3
*                              ;  R3 = Y(I1)±Y(I3)
       STF    R6,*+AR1         ;  Y(I1) = R5±R1
```

```
||     STF    R7,*AR1++(IR0)   ;  X(I1) = R3±R5
       ADDF   R3,R2,R5         ;  R5 = R2+R3
       SUBF   R2,R3,R2         ;  R2 = ±R2+R3
       SUBF   R1,R4,R3         ;  R3 = R4±R1
       ADDF   R1,R4            ;  R4 = R4+R1
       SUBF   R5,R3,R1         ;  R1 = R3±R5
       MPYF   *AR4,R1          ;  R1 = R1*CO21
       ADDF   R5,R3            ;  R3 = R3+R5
       MPYF   *AR4,R3          ;  R3 = R3*CO21
||     STF    R1,*+AR2         ;  Y(I2) = (R3±R5)*CO21
       SUBF   R4,R2,R1         ;  R1 = R2±R4
       MPYF   *AR4,R1          ;  R1 = R1*CO21
||     STF    R3,*AR2++(IR0)   ;  X(I2) = (R3+R5)*CO21
       ADDF   R4,R2            ;  R2 = R2+R4
       MPYF   *AR4,R2          ;  R2 = R2*CO21
BLK3   STF    R1,*+AR3         ;  Y(I3) = ±(R4±R2)*CO21
||     STF R2,*AR3++(IR0)      ;  X(I3) = (R4+R2)*CO21


       CMPI   @LPCNT,R0
       BPD    INLOP            ;  Loop back to the inner loop

CONT   LDI    @RPTCNT,AR7
       LDI    @IEINDX,AR6
       LSH    2,AR7            ;  Increment repeat counter for
*                               ;     next time
       STI    AR7,@RPTCNT
       LSH    2,AR6            ;  IE = 4*IE
       STI    AR6,@IEINDX
       LDI    R0,IR0           ;  N1 = N2
       LSH    -3,R0
       ADDI   2,R0
       STI    R0,@JT           ;  JT = N2/2+2
       SUBI   2,R0
       LSH    1,R0             ;  N2 = N2/4
       BR     LOOP             ;  Next FFT stage
*  STORE RESULT USING BIT±REVERSED ADDRESSING
```

```
END:   LDI    @FFTSIZ,RC       ;  RC = N
       SUBI   1,RC             ;  RC should be one less than desired #
       LDI    @FFTSIZ,IR0      ;  IR0 = size of FFT = N
       LDI    2,IR1
       LDI    @INPUT,AR0
       LDP    STORE
       LDI    @STORE,AR1

       RPTB   BITRV
       LDF    *+AR0(1),R0
||     LDF    *AR0++(IR0)B,R1
BITRV  STF    R0,*+AR1(1)
||     STF    R1,*AR1++(IR1)

SELF   BR SELF                 ;  Branch to itself at the end
       .end
```

The data to be transformed is usually a sequence of real numbers. In this case, the FFT demonstrates certain symmetries that permit the reduction of the computational load even further. Example 11–38 shows the generic implementation of a real-valued, radix-2 FFT. For such an FFT, the total storage required for a length-N transform is only N locations; in a complex FFT, 2N are necessary. Recovery of the rest of the points is based on the symmetry conditions.

Example 11–39 shows the implementation of a radix-2 real inverse FFT. The inverse transformation assumes that the input data is given in the order presented at the output of the forward transformation and produces a time signal in the proper order (that is, bit reversing takes place at the end of the program).

Example 11–38. Real, Radix-2 FFT

```
*****************************************************************************
* FILENAME        : ffft_rl.asm
*
* WRITTEN BY      : Alex Tessarolo
*                   Texas Instruments, Australia
*
* DATE            : 23rd July 1991
*
* VERSION         : 2.0
*
*****************************************************************************

*
* VER      DATE            COMMENTS
* ---    ------------      ------------------------------------------------
* 1.0    18th July 91      Original release.
* 2.0    23rd July 91      Most stages modified.
*                          Minimum FFT size increased from 32 to 64.
*                          Faster in place bit reversing algorithm.
*                          Program size increased by about 100 words.
*                          One extra data word required.

*****************************************************************************

* SYNOPSIS:     int     ffft_rl( FFT_SIZE, LOG_SIZE, SOURCE_ADDR, DEST_ADDR,
*                          SINE_TABLE, BIT_REVERSE );
*
*               int     FFT_SIZE       ; 64, 128, 256, 512, 1024, ...
*               int     LOG_SIZE       ; 6,   7,   8,   9,   10, ...
*               float   *SOURCE_ADDR   ; Points to location of source data.
*               float   *DEST_ADDR     ; Points to where data will be
*                                      ; operated on and stored.
*               float   *SINE_TABLE    ; Points to the SIN/COS table.
*               int     BIT_REVERSE    ; =  0, bit reversing is disabled.
*                                      ; <> 0, bit reversing is enabled.
*
*               NOTE:   1) If SOURCE_ADDR  =  DEST_ADDR, then in-place bit
*                          reversing is performed, if enabled (more
*                          processor intensive).
*                       2) FFT_SIZE must be >= 64 (this is not checked).
*
```

```
* DESCRIPTION: Generic function to do a radix–2 FFT computation on the C30.
*              The data array is FFT_SIZE–long with only real data. The out-
*              put is stored in the same locations with real and imaginary
*              points R and I as follows:
*
*              DEST_ADDR[0]              → R(0)
*                                          R(1)
*                                          R(2)
*                                          R(3)
*                                          .
*                                          .
*                                          R(FFT_SIZE/2)
*                                          I(FFT_SIZE/2 – 1)
*                                          .
*                                          .
*                                          I(2)
*              DEST_ADDR[FFT_SIZE – 1]   → I(1)
*
*              The program is based on the FORTRAN program in the
*              paper by Sorensen et al., June 1987 issue of Trans.
*              on ASSP.
*
*              Bit reversal is optionally implemented at the begin-
*              ning of the function.
*
*              The sine/cosine table for the twiddle factors is ex-
*              pected to be supplied in the following format:
*
*              SINE_TABLE[0]             → sin(0*2*pi/FFT_SIZE)
*                                          sin(1*2*pi/FFT_SIZE)
*                                                 .
*                                                 .
*                                          sin((FFT_SIZE/2–2)*2*pi/FFT_SIZE)
*              SINE_TABLE[FFT_SIZE/2 – 1]→ sin((FFT_SIZE/2–1)*2*pi/FFT_SIZE)
*
*              NOTE: The table is the first half period of a sine wave.
*
*              Stack structure upon call:
*
*
*                                 ┌──────────┐
*              –FP(7)             │BIT_REVERSE
*              –FP(6)             │SINE_TABLE
*              –FP(5)             │DEST_ADDR
*              –FP(4)             │SOURCE_ADDR
*              –FP(3)             │LOG_SIZE
*              –FP(2)             │FFT_SIZE
*              –FP(1)             │returne
*              –FP(0)             │addr
*                                 │old FP
*                                 └──────────┘
*
*
*******************************************************************************
```

```
*                NOTE:    Calling C program can be compiled using either large
*                         or small model.
*
*                WARNING: DP initialized only once in the program. Be wary
*                         with interrupt service routines. Make sure interrupt
*                         service routines save the DP pointer.
*
*                WARNING: The DEST_ADDR must be aligned such that the first
*                         LOG_SIZE bits are zero (this is not checked by the
*                         program).
*
*******************************************************************************
*

* REGISTERS USED: R0, R1, R2, R3, R4, R5, R6, R7
*                 AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7
*                 IR0, IR1
*                 RC, RS, RE
*                 DP
*
* MEMORY REQUIREMENTS:     Program  =  405 Words (approximately)
*                          Data     =    7 Words
*                          Stack    =   12 Words
*
*******************************************************************************
*

* BENCHMARKS:   Assumptions     - Program in RAM0
*                               - Reserved data in RAM0
*                               - Stack on primary/expansion bus RAM
*                               - Sine/cosine tables in RAM0
*                               - Processing and data destination in RAM1.
*                               - Primary/expansion bus RAM, 0 wait state.
*
*               FFT Size        Bit Reversing Data Source   Cycles(C30)
*               --------        ------------- -----------   -----------
*               1024            OFF           RAM1          19816 approx.
*               Note: This number does not include the C callable overheads.
*                     Add 57 cycles for these overheads.
*
*******************************************************************************

FP              .set    AR3

                .global _ffft_rl            ; Entry execution point.

FFT_SIZE:       .usect  ".fftdata",1        ; Reserve memory for arguments.
LOG_SIZE:       .usect  ".fftdata",1
SOURCE_ADDR:    .usect  ".fftdata",1
DEST_ADDR:      .usect  ".fftdata",1
SINE_TABLE:     .usect  ".fftdata",1
BIT_REVERSE:    .usect  ".fftdata",1
SEPARATION:     .usect  ".fftdata",1
```

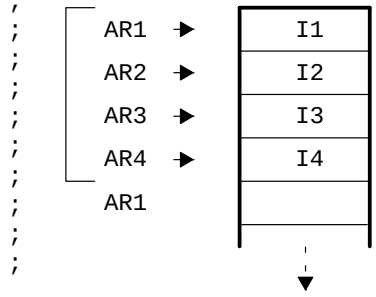
```
                                ;
                                ; Initialize C function.
                                ;
                .sect           ".ffttext"
_ffft_rl:  PUSH     FP          ; Preserve C environment.
           LDI      SP,FP
           PUSH     R4
           PUSH     R5
           PUSH     R6
           PUSHF    R6
           PUSH     R7
           PUSHF    R7
           PUSH     AR4
           PUSH     AR5
           PUSH     AR6
           PUSH     AR7
           PUSH     DP

           LDP      FFT_SIZE    ; Init. DP pointer.

           LDI      *-FP(2),R0  ; Move arguments from stack.
           STI      R0,@FFT_SIZE
           LDI      *-FP(3),R0
           STI      R0,@LOG_SIZE
           LDI      *-FP(4),R0
           STI      R0,@SOURCE_ADDR
           LDI      *-FP(5),R0
           STI      R0,@DEST_ADDR
           LDI      *-FP(6),R0
           STI      R0,@SINE_TABLE
           LDI      *-FP(7),R0
           STI      R0,@BIT_REVERSE

                                ;
                                ; Check bit reversing mode (on or off).
                                ;
                                ; BIT_REVERSING = 0, then OFF
                                ;     (no bit reversing).
                                ; BIT_REVERSING <> 0, Then ON.
                                ;
           LDI      @BIT_REVERSE,R0
           CMPI     0,R0
           BZ       MOVE_DATA

                                ;
                                ; Check bit reversing type.
                                ;
                                ; If SourceAddr = DestAddr, then in place
                                ;     bit reversing.
                                ; If SourceAddr <> DestAddr, then
                                ;     standard bit reversing.
                                ;
```

```
            LDI     @SOURCE_ADDR,R0
            CMPI    @DEST_ADDR,R0
            BEQ     IN_PLACE

                                ;
                                ; Bit reversing Type 1 (from source to
                                ;    destination).
                                ;
                                ; NOTE: abs(SOURCE_ADDR – DEST_ADDR)
                                ;    must be > FFT_SIZE, this is not
                                ;    checked.
                                ;

            LDI     @FFT_SIZE,R0
            SUBI    2,R0
            LDI     @FFT_SIZE,IR0
            LSH     –1,IR0          ; IRO  =  half FFT size.
            LDI     @SOURCE_ADDR,AR0
            LDI     @DEST_ADDR,AR1

            LDF     *AR0++,R1

            RPTS    R0
            LDF     *AR0++,R1
               ||   STF             R1,*AR1++(IR0)B

            STF     R1,*AR1++(IR0)B

            BR      START

                                ;
                                ; In-place bit reversing.
                                ;

                                ; Bit reversing on even locations,
                                ;    1st half only.

IN_PLACE:   LDI     @FFT_SIZE,IR0
            LSH     –2,IR0          ; IRO  =  quarter FFT size.
            LDI     2,IR1

            LDI     @FFT_SIZE,RC
            LSH     –2,RC
            SUBI    3,RC
            LDI     @DEST_ADDR,AR0
            LDI     AR0,AR1
            LDI     AR0,AR2

            NOP     *AR1++(IR0)B
            NOP     *AR2++(IR0)B
            LDF     *++AR0(IR1),R0
            LDF     *AR1,R1
            CMPI    AR1,AR0         ; Xchange locs only if AR0<AR1.
            LDFGT   R0,R1
            LDFGT   *AR1++(IR0)B,R1
```

```
            RPTB    BITRV1
            LDF     *++AR0(IR1),R0
               ||   STF          R0,*AR0
            LDF     *AR1,R1
               ||   STF          R1,*AR2++(IR0)B
            CMPI    AR1,AR0
            LDFGT   R0,R1
BITRV1:     LDFGT   *AR1++(IR0)B,R0

            STF     R0,*AR0
            STF     R1,*AR2

                                ;    Perform bit reversing on odd
                                ;    locations, 2nd half only.

            LDI     @FFT_SIZE,RC
            LSH     -1,RC
            LDI     @DEST_ADDR,AR0
            ADDI    RC,AR0
            ADDI    1,AR0
            LDI     AR0,AR1
            LDI     AR0,AR2
            LSH     -1,RC
            SUBI    3,RC

            NOP     *AR1++(IR0)B
            NOP     *AR2++(IR0)B
            LDF     *++AR0(IR1),R0
            LDF     *AR1,R1
            CMPI    AR1,AR0         ;    Xchange locs only if AR0<AR1.
            LDFGT   R0,R1
            LDFGT   *AR1++(IR0)B,R1

            RPTB    BITRV2
            LDF     *++AR0(IR1),R0
               ||   STF          R0,*AR0
            LDF     *AR1,R1
               ||    STF         R1,*AR2++(IR0)B
            CMPI    AR1,AR0
            LDFGT   R0,R1
BITRV2:     LDFGT   *AR1++(IR0)B,R0

            STF     R0,*AR0
            STF     R1,*AR2

                                ;    Perform bit reversing on odd
                                ;    locations, 1st half only.

            LDI     @FFT_SIZE,RC
            LSH     -1,RC
            LDI     RC,IR0
            LDI     @DEST_ADDR,AR0
            LDI     AR0,AR1
            ADDI    1,AR0
```

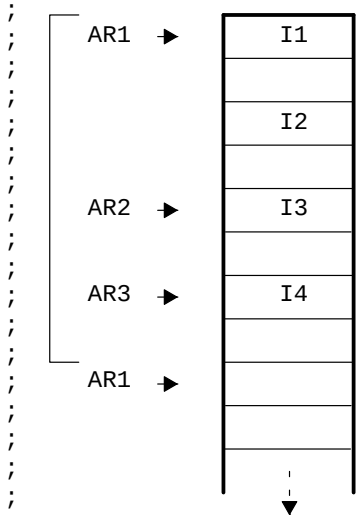
```
            ADDI    IR0,AR1
            LSH     -1,RC
            LDI     RC,IR0
            SUBI    2,RC

            LDF     *AR0,R0
            LDF     *AR1,R1

            RPTB    BITRV3
            LDF     *++AR0(IR1),R0
               ||   STF          R0,*AR1++(IR0)B
BITRV3:     LDF     *AR1,R1
               ||   STF          R1,*-AR0(IR1)

            STF     R0,*AR1
            STF     R1,*AR0

            BR      START

                                ;
                                ;    Check data source locations.
                                ;
                                ;    If SourceAddr = DestAddr, then
                                ;    do nothing.
                                ;    If SourceAddr <> DestAddr, then move
                                     data.
                                ;
MOVE_DATA:  LDI     @SOURCE_ADDR,R0
            CMPI    @DEST_ADDR,R0
            BEQ     START

            LDI     @FFT_SIZE,R0
            SUBI    2,R0
            LDI     @SOURCE_ADDR,AR0
            LDI     @DEST_ADDR,AR1

            LDF     *AR0++,R1

            RPTS    R0
            LDF      *AR0++,R1
               ||   STF          R1,*AR1++

            STF     R1,*AR1
```

```
;
; Perform first and second FFT loops.
;
;     AR1 ->    I1      0 <-  [X(I1)  + X(I2)] + [X(I3) + X(I4)]
;     AR2 ->    I2      1 <-  [X(I1)  - X(I2)]
;     AR3 ->    I3      2 <-  [X(I1)  + X(I2)] - [X(I3) + X(I4)]
;     AR4 ->    I4      3 <- -[X(I3)  - X(I4)]
;     AR1               4
;
;                  |
;                  v
```



```
START:          LDI     @DEST_ADDR,AR1
                LDI     AR1,AR2
                LDI     AR1,AR3
                LDI     AR1,AR4
                ADDI    1,AR2
                ADDI    2,AR3
                ADDI    3,AR4
                LDI     4,IR0
                LDI     @FFT_SIZE,RC
                LSH     -2,RC
                SUBI    2,RC
                LDF     *AR2,R0             ; R0  =  X(I2)
             || LDF     *AR3,R1             ; R1  =  X(I3)
                ADDF3   R1,*AR4,R4          ; R4  =  X(I3) + X(I4)
                SUBF3   R1,*AR4++(IR0),R5   ; R5  =  -[X(I3) - X(I4)]
                SUBF3   R0,*AR1,R6          ; R6  =  X(I1) - X(I2)
                ADDF3   R0,*AR1++(IR0),R7   ; R7  =  X(I1) + X(I2)
                ADDF3   R7,R4,R2            ; R2  =  R7 + R4
                SUBF3   R4,R7,R3            ; R3  =  R7 - R4
                                            ;
                RPTB    LOOP1_2             ;
                LDF     *+AR2(IR0),R0       ;
             || LDF     *+AR3(IR0),R1       ;
                ADDF3   R1,*AR4,R4          ;
             || STF     R3,*AR3++(IR0)      ; X(I3)
                SUBF3   R1,*AR4++(IR0),R5   ;
             || STF     R5,*-AR4(IR0)       ; X(I4)
                SUBF3   R0,*AR1,R6          ;
             || STF     R6,*AR2++(IR0)      ; X(I2)
                ADDF3   R0,*AR1++(IR0),R7   ;
             || STF     R2,*-AR1(IR0)       ; X(I1)
                ADDF3   R7,R4,R2
LOOP1_2:        SUBF3   R4,R7,R3
                STF     R3,*AR3
             || STF     R5,*-AR4(IR0)
                STF     R6,*AR2
             || STF     R2,*-AR1(IR0)
```

```
;
; Perform third FFT loop.
;
; Part A:
;
;    AR1 ->  I1    0 <-   X(I1)  + X(I3)
;            |     1
;            I2    2
;            |     3
;    AR2 ->  I3    4 <-   X(I1)  - X(I3)
;            |     5
;    AR3 ->  I4    6 <-  -X(I4)
;            |     7
;    AR1 ->  |     8
;            |     9
;            |
;            v
```

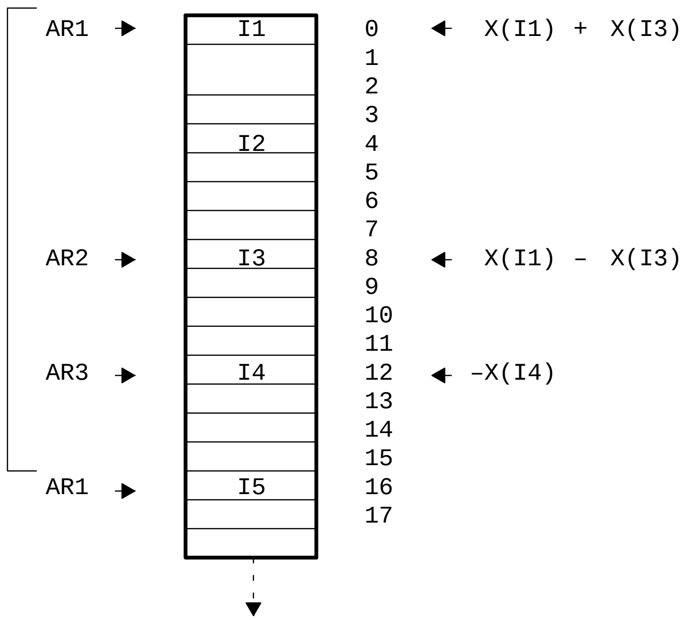


```
                LDI     @DEST_ADDR,AR1
                LDI     AR1,AR2
                LDI     AR1,AR3
                ADDI    4,AR2
                ADDI    6,AR3
                LDI     8,IR0
                LDI     @FFT_SIZE,RC
                LSH     -3,RC
                SUBI    2,RC

                SUBF3   *AR2,*AR1,R1
                ADDF3   *AR2,*AR1,R2
                NEGF    *AR3,R3

                RPTB    LOOP3_A
                LDF     *+AR2(IR0),R0       ; R0  =   X(I3)
             ||  STF    R2,*AR1++(IR0)
                SUBF3   R0,*AR1,R1          ; R1  =   X(I1) - X(I3)
             ||  STF    R1,*AR2++(IR0)      ;
                ADDF3   R0,*AR1,R2          ; R2  =   X(I1) + X(I3)
             ||  STF    R3,*AR3++(IR0)      ;
LOOP3_A:        NEGF    *AR3,R3             ; R3  =  -X(I4)
                                            ;
                STF     R2,*AR1             ; X(I1)
                STF     R1,*AR2             ; X(I3)
                STF     R3,*AR3             ; X(I4)
```

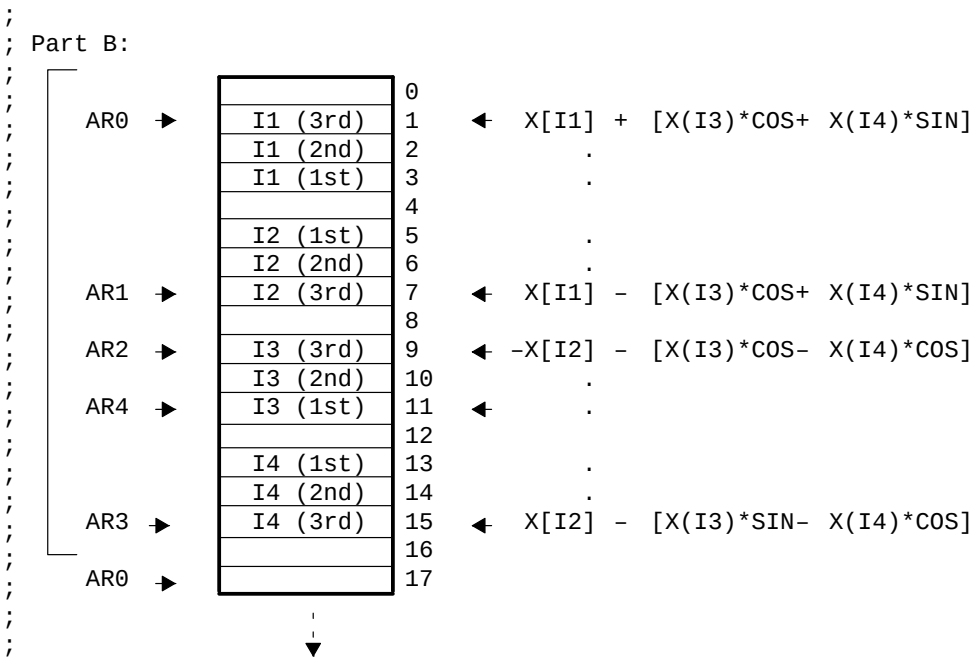
```
;
; Part B:
;
;
;
;                         0
;  AR0 ->  |  I1  |       1   <-  X[I1] + [X(I3)*COS+ X(I4)*COS]
;                         2
;  AR1 ->  |  I2  |       3   <-  X[I1] – [X(I3)*COS+ X(I4)*COS]
;                         4
;  AR2 ->  |  I3  |       5   <- –X[I2] – [X(I3)*COS– X(I4)*COS]
;                         6
;  AR3 ->  |  I4  |       7   <-  X[I2] – [X(I3)*COS– X(I4)*COS]
;                         8
;  AR0 ->  |      |       9       NOTE: COS(2*pi/8) = SIN(2*pi/8)

        LDI     @FFT_SIZE,RC
        LSH     –3,RC
        LDI     RC,IR1
        SUBI    3,RC
        LDI     8,IR0
        LDI     @DEST_ADDR,AR0
        LDI     AR0,AR1
        LDI     AR0,AR2
        LDI     AR0,AR3
        ADDI    1,AR0
        ADDI    3,AR1
        ADDI    5,AR2
        ADDI    7,AR3
        LDI     @SINE_TABLE,AR7       ; Initialize table pointers.
        LDF     *++AR7(IR1),R7        ; R7    =  COS(2*pi/8)
                                      ; *AR7  =  COS(2*pi/8)
        MPYF3   *AR7,*AR2,R0          ; R0  =   X(I3)*COS
        MPYF3   *AR3,R7,R1            ; R5  =   X(I4)*COS
        ADDF3   R0,R1,R2              ; R2 =  [X(I3)*COS + X(I4)*COS]
        MPYF3   *AR7,*+AR2(IR0),R0
||      SUBF3   R0,R1,R3              ; R3 = –[X(I3)*COS – X(I4)*COS]
        SUBF3   *AR1,R3,R4            ; R4 = –X(I2) + R3
        ADDF3   *AR1,R3,R4            ; R4 =  X(I2) + R3
||      STF     R4,*AR2++(IR0)        ; X(I3)<-
        SUBF3   R2,*AR0,R4            ; R4 =  X(I1) – R2
||      STF     R4,*AR3++(IR0)        ; X(I4)<-
        ADDF3   *AR0,R2,R4            ; R4 =  X(I1) + R2
||      STF     R4,*AR1++(IR0)        ; X(I2)<-
                                      ;
        RPTB    LOOP3_B               ;
        MPYF3   *AR3,R7,R1            ;
||      STF     R4,*AR0++(IR0)        ; X(I1)<-
        ADDF3   R0,R1,R2
        MPYF3   *AR7,*+AR2(IR0),R0
```

```
                || SUBF3    R0,R1,R3
                   SUBF3    *AR1,R3,R4
                   ADDF3    *AR1,R3,R4
                || STF      R4,*AR2++(IR0)
                   SUBF3    R2,*AR0,R4
                || STF      R4,*AR3++(IR0)
LOOP3_B:           ADDF3    *AR0,R2,R4
                || STF      R4,*AR1++(IR0)
                   MPYF3    *AR3,R7,R1
                || STF      R4,*AR0++(IR0)
                   ADDF3    R0,R1,R2
                   SUBF3    R0,R1,R3
                   SUBF3    *AR1,R3,R4
                   ADDF3    *AR1,R3,R4
                || STF      R4,*AR2
                   SUBF3    R2,*AR0,R4
                || STF      R4,*AR3
                   ADDF3    *AR0,R2,R4
                || STF      R4,*AR1
                   STF      R4,*AR0
```

```
;
; Perform fourth FFT loop.
;
; Part A:
;
```




```
;
                  LDI     @DEST_ADDR,AR1
                  LDI     AR1,AR2
                  LDI     AR1,AR3
                  ADDI    8,AR2
                  ADDI    12,AR3
                  LDI     16,IR0
                  LDI     @FFT_SIZE,RC
                  LSH     –4,RC
                  SUBI    2,RC
                  SUBF3   *AR2,*AR1,R1
                  ADDF3   *AR2,*AR1,R2
                  NEGF    *AR3,R3
                  RPTB    LOOP4_A
                  LDF     *+AR2(IR0),R0      ; R0  =   X(I3)
               || STF     R2,*AR1++(IR0)
                  SUBF3   R0,*AR1,R1         ; R1  =   X(I1) – X(I3)
               || STF     R1,*AR2++(IR0)     ;
                  ADDF3   R0,*AR1,R2         ; R2  =   X(I1) + X(I3)
               || STF     R3,*AR3++(IR0)     ;
LOOP4_A:          NEGF    *AR3,R3            ; R3  =  –X(I4)
                                             ;
                  STF     R2,*AR1            ; X(I1)
               || STF     R1,*AR2            ; X(I3)
                  STF     R3,*AR3            ; X(I4)
```



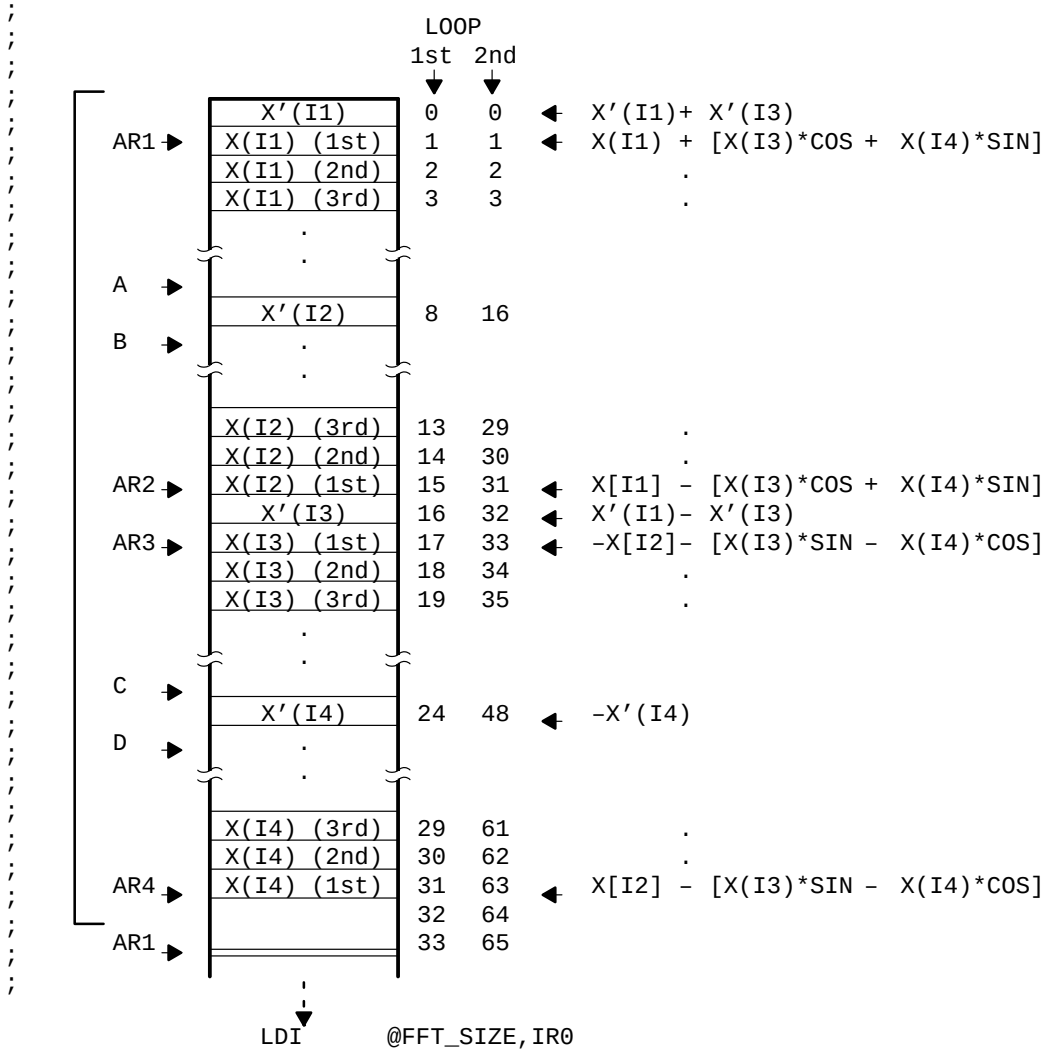

```
        LDI     @FFT_SIZE,RC
        LSH     -4,RC
        LDI     RC,IR1
        LDI     2,IR0
        SUBI    3,RC
        LDI     @DEST_ADDR,AR0
        LDI     AR0,AR1
        LDI     AR0,AR2
        LDI     AR0,AR3
        LDI     AR0,AR4
        ADDI    1,AR0
        ADDI    7,AR1
        ADDI    9,AR2
        ADDI    15,AR3
        ADDI    11,AR4

        LDI     @SINE_TABLE,AR7
        LDF     *++AR7(IR1),R7      ; R7  =  SIN(1*[2*pi/16])
                                    ; *AR7  =  COS(3*[2*pi/16])
        LDI     AR7,AR6
        LDF     *++AR6(IR1),R6      ; R6  =  SIN(2*[2*pi/16])
                                    ; *AR6  =  COS(2*[2*pi/16])
        LDI     AR6,AR5
        LDF     *++AR5(IR1),R5      ; R5  =  SIN(3*[2*pi/16])
                                    ; *AR5  =  COS(1*[2*pi/16])

        LDI     16,IR1
```

```
    MPYF3    *AR7,*AR4,R0          ; R0  =   X(I3)*COS(3)

    MPYF3    *++AR2(IR0),R5,R4     ; R4  =   X(I3)*SIN(3)
    MPYF3    *--AR3(IR0),R5,R1     ; R1  =   X(I4)*SIN(3)
    MPYF3    *AR7,*AR3,R0          ; R0  =   X(I4)*COS(3)
||  ADDF3    R0,R1,R2              ; R2 =  [X(I3)*COS + X(I4)*SIN]
    MPYF3    *AR6,*-AR4,R0
||  SUBF3    R4,R0,R3              ; R3 = -[X(I3)*SIN - X(I4)*COS]
    SUBF3    *--AR1(IR0),R3,R4     ; R4  =  -X(I2) + R3
    ADDF3    *AR1,R3,R4            ; R4  =   X(I2) + R3
    STF      R4,*AR2--             ; X(I3)
    SUBF3    R2,*++AR0(IR0),R4     ; R4  =   X(I1) - R2
    STF      R4,*AR3               ; X(I4)
    ADDF3    *AR0,R2,R4            ; R4  =   X(I1) + R2
    STF      R4,*AR1               ; X(I2)
                                   ;
    MPYF3    *++AR3,R6,R1          ;
||  STF      R4,*AR0               ; X(I1)
    ADDF3    R0,R1,R2
    MPYF3    *AR5,*-AR4(IR0),R0
||  SUBF3    R0,R1,R3
    SUBF3    *++AR1,R3,R4
    ADDF3    *AR1,R3,R4
||  STF      R4,*AR2
    SUBF3    R2,*--AR0,R4
||  STF
    STF      R4,*AR1

    MPYF3    *--AR2,R7,R4
||  STF      R4,*AR0
    MPYF3    *++AR3,R7,R1
||  MPYF3    *AR5,*AR3,R0
    ADDF3   R0,R1,R2
    MPYF3    *AR7,*++AR4(IR1),R0
||  SUBF3    R4,R0,R3
    SUBF3    *++AR1,R3,R4
    ADDF3    *AR1,R3,R4
    STF      R4,*AR2++(IR1)
    SUBF3    R2,*--AR0,R4
||  STF      R4,*AR3++(IR1)
    ADDF3    *AR0,R2,R4
||  STF      R4,*AR1++(IR1)

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

```
                ||  STF     R4,*AR2--
                    SUBF3   R2,*++AR0(IR0),R4
                ||  STF     R4,*AR3
                    ADDF3   *AR0,R2,R4
                ||  STF     R4,*AR1


                    MPYF3   *++AR3,R6,R1
                ||  STF     R4,*AR0
                    ADDF3   R0,R1,R2
                    MPYF3   *AR5,*-AR4(IR0),R0
                ||  SUBF3   R0,R1,R3
                    SUBF3   *++AR1,R3,R4
                    ADDF3   *AR1,R3,R4
                ||  STF     R4,*AR2
                    SUBF3   R2,*--AR0,R4
                ||  STF     R4,*AR3
                    ADDF3   *AR0,R2,R4
                ||  STF     R4,*AR1


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


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

```
    MPYF3    *++AR3,R6,R1
||  STF      R4,*AR0
    ADDF3    R0,R1,R2
    MPYF3    *AR5,*-AR4(IR0),R0
||  SUBF3    R0,R1,R3
    SUBF3    *++AR1,R3,R4
    ADDF3    *AR1,R3,R4
||  STF      R4,*AR2
    SUBF3    R2,*--AR0,R4
||  STF      R4,*AR3
    ADDF3    *AR0,R2,R4
    STF      R4,*AR1

    MPYF3    *--AR2,R7,R4
||  STF      R4,*AR0
    MPYF3    *++AR3,R7,R1
    MPYF3    *AR5,*AR3,R0
||  ADDF3    R0,R1,R2
    SUBF3    R4,R0,R3
    SUBF3    *++AR1,R3,R4
    ADDF3    *AR1,R3,R4
||  STF      R4,*AR2
    SUBF3    R2,*--AR0,R4
||  STF      R4,*AR3
    ADDF3    *AR0,R2,R4
    STF      R4,*AR1

    STF      R4,*AR0
```

```
; Perform remaining FFT loops (loop 4 onwards).
```




```
                LDI     @FFT_SIZE,IR0
                LSH     –2,IR0
                STI     IR0,@SEPARATION
                LSH     –2,IR0
                LDI     5,R5
                LDI     3,R7
                LDI     16,R6
                LDI     @DEST_ADDR,AR5
                LDI     @DEST_ADDR,AR1
                LSH     –1,IR0
                LSH     1,R7
LOOP:           ADDI    1,R7
                LSH     1,R6
                LDI     AR1,AR4
```

```
                ADDI    R7,AR1              ; AR1 points at A.
                LDI     AR1,AR2
                ADDI    2,AR2               ; AR2 points at B.
                ADDI    R6,AR4
                SUBI    R7,AR4              ; AR4 points at D.
                LDI     AR4,AR3
                SUBI    2,AR3               ; AR3 points at C.

                LDI     @SINE_TABLE,AR0     ; AR0 points at SIN/COS table.
                LDI     R7,IR1
                LDI     R7,RC

INLOP:          ADDF3   *--AR1(IR1),*++AR2(IR1),R0; R0  =  X'(I1) + X'(I3)
                SUBF3   *--AR3(IR1),*AR1++,R1     ; R1  =  X'(I1) - X'(I3)
                NEGF    *--AR4,R2                 ; R2  =  -X'(I4)
              ||STF     R0,*-AR1                  ; X'(I1)
                STF     R1,*AR2--                 ; X'(I3)
              ||STF     R2,*AR4++(IR1)            ; X'(I4)

                LDI     @SEPARATION,IR1     ; IR1=SEPARATION
                                            BETWEEN SIN/COS TBLS
                SUBI    3,RC

                MPYF3   *++AR0(IR0),*AR4,R4 ; R4 =  X(I4)*SIN
                MPYF3   *AR0,*++AR3,R1      ; R1 =  X(I3)*SIN
                MPYF3   *++AR0(IR1),*AR4,R0 ; R0 =  X(I4)*COS
                MPYF3   *AR0,*AR3,R0        ; R0 =  X(I3)*COS
              ||SUBF3   R1,R0,R3            ; R3 = -[X(I3)*SIN - X(I4)*COS]
                MPYF3   *++AR0(IR0),*-AR4,R0
              ||ADDF3   R0,R4,R2            ; R2 = X(I3)*COS + X(I4)*SIN
                SUBF3   *AR2,R3,R4          ; R4 = R3 - X(I2)
                ADDF3   *AR2,R3,R4          ; R4 = R3 + X(I2)
              ||STF     R4,*AR3++           ; X(I3)
                SUBF3   R2,*AR1,R4          ; R4 = X(I1) - R2
              ||STF     R4,*AR4--           ; X(I4)
                ADDF3   *AR1,R2,R4          ; R4 = X(I1) + R2
              ||STF     R4,*AR2--           ; X(I2)
                                            ;
                RPTB    IN_BLK              ;
                LDF     *-AR0(IR1),R3       ;
                MPYF3   *AR4,R3,R4          ;
              ||STF     R4,*AR1++           ; X(I1)
                MPYF3   *AR3,R3,R1
                MPYF3   *AR0,*AR3,R0
              ||SUBF3   R1,R0,R3
                MPYF3   *++AR0(IR0),*-AR4,R0
              ||ADDF3   R0,R4,R2
                SUBF3   *AR2,R3,R4
                ADDF3   *AR2,R3,R4
              ||STF     R4,*AR3++
                SUBF3   R2,*AR1,R4
              ||STF     R4,*AR4--
IN_BLK:         ADDF3   *AR1,R2,R4
              ||STF     R4,*AR2--
```

```
LDF     *–AR0(IR1),R3
MPYF3   *AR4,R3,R4
   ||   STF          R4,*AR1++
MPYF3   *AR3,R3,R1
MPYF3   *AR0,*AR3,R0
   ||   SUBF3        R1,R0,R3
LDI     R6,IR1
ADDF3   R0,R4,R2
SUBF3   *AR2,R3,R4
ADDF3   *AR2,R3,R4
   ||   STF          R4,*AR3++(IR1)
SUBF3   R2,*AR1,R4
   ||   STF          R4,*AR4++(IR1)
ADDF3   *AR1,R2,R4
   ||   STF          R4,*AR2++(IR1)

STF     R4,*AR1++(IR1)

SUBI3   AR5,AR1,R0
CMPI    @FFT_SIZE,R0
BLTD    INLOP         ; LOOP BACK TO THE
                        INNER LOOP
LDI     @SINE_TABLE,AR0     ; AR0 POINTS TO
                        SIN/COS TABLE
LDI     R7,IR1
LDI     R7,RC

ADDI    1,R5
CMPI    @LOG_SIZE,R5
BLED    LOOP
LDI     @DEST_ADDR,AR1
LSH     –1,IR0
LSH     1,R7
```

```
                                ;
                                ; Return to C environment.
                                ;

            POP     DP          ; Restore C environment
                                ;    variables.
            POP     AR7
            POP     AR6
            POP     AR5
            POP     AR4
            POPF    R7
            POP     R7
            POPF    R6
            POP     R6
            POP     R5
            POP     R4
            POP     FP
            RETS

            .end

*
* No more.
*
*******************************************************************************
```

Example 11–39. Real Inverse, Radix-2 FFT

```
*  Real Inverse FFT
*******************************************************************************
*
* FILENAME    : ifft_rl.asm
*
* WRITTEN BY  : Daniel Mazzocco
*               Texas Instruments, Houston
*
* DATE        : 18th Feb 1992
*
* VERSION     : 1.0
*

*******************************************************************************
* VER      DATE                                  COMMENTS
* ---    ------------      ------------------------------------------------
* 1.0    18th Feb 92       Original release. Started from forward real FFT
*                          routine written by Alex Tessarolo, rev 2.0 .
*

*******************************************************************************
*
* SYNOPSIS:       int      ifft_rl( FFT_SIZE, LOG_SIZE, SOURCE_ADDR,
                                        DEST_ADDR, SINE_TABLE, BIT_REVERSE );
*
*                 int      FFT_SIZE     ; 64, 128, 256, 512, 1024, ...
*                 int      LOG_SIZE     ;  6,   7,   8,   9,   10, ...
*                 float    *SOURCE_ADDR ; Points to where data is originated
*                                       ; and operated on.
*                 float    *DEST_ADDR   ; Points to where data will be stored.
*                 float    *SINE_TABLE  ; Points to the SIN/COS table.
*                 int      BIT_REVERSE  ; = 0, bit reversing is disabled.
*                                       ; <> 0, bit reversing is enabled.
*
*                 NOTE:    1) If SOURCE_ADDR = DEST_ADDR, then in place bit
*                             reversing is performed, if enabled (more
*                             processor intensive).
*                          2) FFT_SIZE must be >= 64 (this is not checked).
*
```

```
* DESCRIPTION:    Generic function to do an inverse radix–2 FFT computation
*                 on the C30.
*                 The data array is FFT_SIZE long with real and imaginary
*                 points R and I as follows:
*
*                 SOURCE_ADDR[0]          →  R(0)
*                                            R(1)
*                                            R(2)
*                                            R(3)
*                                            .
*                                            .
*                                            R(FFT_SIZE/2)
*                                            I(FFT_SIZE/2 – 1)
*                                            .
*                                            .
*                                            I(2)
*                 SOURCE_ADDR[FFT_SIZE–1]  →  I(1)
*
*                 The output data array will contain only real values.
*                 Bit reversal is optionally implemented at the end
*                 of the function.
*
*                 The sine/cosine table for the twiddle factors is expected
*                 to be supplied in the following format:
*
*                 SINE_TABLE[0]           →  sin(0*2*pi/FFT_SIZE)
*                                            sin(1*2*pi/FFT_SIZE)
*                                                 .
*                                                 .
*                                            sin((FFT_SIZE/2–2)*2*pi/FFT_SIZE)
*                 SINE_TABLE[FFT_SIZE/2–1] →  sin((FFT_SIZE/2–1)*2*pi/FFT_SIZE)
*
*                 NOTE: The table is the first half period of a sine wave.
*
*                 Stack structure upon call:
*
*
*                          –FP(7)    BIT_REVERSE
*                          –FP(6)    SINE_TABLE
*                          –FP(5)    DEST_ADDR
*                          –FP(4)    SOURCE_ADDR
*                          –FP(3)    LOG_SIZE
*                          –FP(2)    FFT_SIZE
*                          –FP(1)    returne
*                          –FP(0)    addr
*                                    old FP
*
*******************************************************************************
```

```
*                NOTE:    Calling C program can be compiled using either large
*                         or small model.
*
*                WARNING: DP initialized only once in the program. Be wary
*                         with interrupt service routines. Make sure interrupt
*                         service routines save the DP pointer.
*
*                WARNING: The SOURCE_ADDR must be aligned such that the first
*                         LOG_SIZE bits are zero (this is not checked by the
*                         program).
*
*******************************************************************************
*
* REGISTERS USED: R0, R1, R2, R3, R4, R5, R6, R7
*                 AR0, AR1, AR2, AR3, AR4, AR5, AR6, AR7
*                 IR0, IR1
*                 RC, RS, RE
*                 DP
*
* MEMORY REQUIREMENTS:    Program  = 322 words (approximately)
*                         Data     =   7 words
*                         Stack    =  12 words
*
*******************************************************************************

*
* BENCHMARKS:     Assumptions  – Program in RAM0
*                              – Reserved data in RAM0
*                              – Stack on primary/expansion bus RAM
*                              – Sine/cosine tables in RAM0
*                              – Processing and data destination in RAM1
*                              – Primary/expansion bus RAM, 0 wait state
*
*                FFT Size     Bit Reversing   Data Source     Cycles(C30)
*                --------     -------------   -----------     -----------
*                1024         OFF             RAM1            25892 approx.
*                Note:    This number does not include the C callable overheads.
*                         Add 57 cycles for these overheads.
*******************************************************************************

FP               .set     AR3

                 .global  _ifft_rl             ; Entry execution point.

FFT_SIZE:        .usect   ".ifftdata",1        ; Reserve memory for arguments.
LOG_SIZE:        .usect   ".ifftdata",1
SOURCE_ADDR:     .usect   ".ifftdata",1
DEST_ADDR:       .usect   ".ifftdata",1
SINE_TABLE:      .usect   ".ifftdata",1
BIT_REVERSE:     .usect   ".ifftdata",1
SEPARATION:      .usect   ".ifftdata",1
```

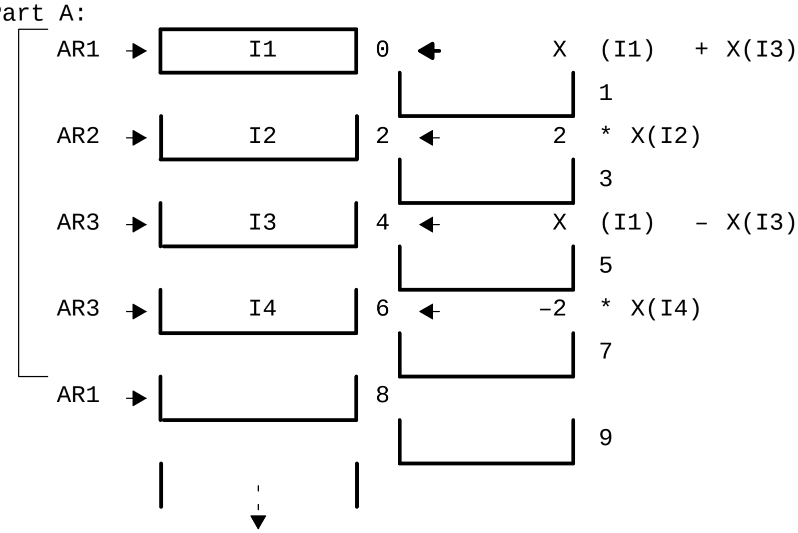
```
                                        ;
                                        ; Initialize C Function.
                                        ;
                .sect   ".iffttext"

_ifft_rl:       PUSH    FP              ; Preserve C environment.
                LDI     SP,FP
                PUSH    R4
                PUSH    R5
                PUSH    R6
                PUSHF   R6
                PUSH    R7
                PUSHF   R7
                PUSH    AR4
                PUSH    AR5
                PUSH    AR6
                PUSH    AR7
                PUSH    DP

                LDP     FFT_SIZE        ; Initialize DP pointer.

                LDI     *-FP(2),R0      ; Move arguments from stack.
                STI     R0,@FFT_SIZE
                LDI     *-FP(3),R0
                STI     R0,@LOG_SIZE
                LDI     *-FP(4),R0
                STI     R0,@SOURCE_ADDR
                LDI     *-FP(5),R0
                STI     R0,@DEST_ADDR
                LDI     *-FP(6),R0
                STI     R0,@SINE_TABLE
                LDI     *-FP(7),R0
                STI     R0,@BIT_REVERSE
```

```
;
; Perform last FFT loops first (loop 2 onwards).
;
;                                LOOP
;                               1st  2nd
;                                ▼    ▼
;                    X'(I1)      0    0   ◄  X'(I1)+ X'(I3)
;   AR1 ►            X(I1) (1st) 1    1   ◄  X(I1) + [X(I2)
;                    X(I1) (2nd) 2    2            .
;                    X(I1) (3rd) 3    3            .
;                        .
;                        .
;   A   ►
;                    X'(I2)      8   16   ◄  X'(12)* 2
;   B   ►                .
;                        .
;
;                    X(I2) (3rd) 13  29            .
;                    X(I2) (2nd) 14  30            .
;   AR2 ►            X(I2) (1st) 15  31   ◄  X[I4] – [X(I3)
;                    X'(I3)      16  32   ◄  X'(I1)– X'(I3)
;   AR3 ►            X(I3) (1st) 17  33   ◄  [X(I1)–X(I2)]*COS–[X(I3)+X(I4)]*SIN
;                    X(I3) (2nd) 18  34            .
;                    X(I3) (3rd) 19  35            .
;                        .
;                        .
;   C   ►
;                    X'(I4)      24  48   ◄  –X'(I4)*2
;   D   ►                .
;                        .
;
;                    X(I4) (3rd) 29  61            .
;                    X(I4) (2nd) 30  62            .
;   AR4 ►            X(I4) (1st) 31  63   ◄  [X(I2)–X(I2)]*SIN+[X(I3)+X(I4)]*COS
;                                32  64
;   AR1 ►                        33  65
;                        .
;                        ▼
;
                LDI     1,IR0           ; Step between two consecutive sines
                LDI     4,R5            ; Stage number from 4 to M.
                LDI     @FFT_SIZE,R7
                LSH     –2,R7           ; R7 is FFT_SIZE/4–1 (ie 15 for 64 pts)
                SUBI    1,R7            ; and will be used to point at A & D.
                LDI     @FFT_SIZE,R6    ; R6 will be used to point at D.
                LSH     1,R6
                LDI     @SOURCE_ADDR,AR5
                LDI     @SOURCE_ADDR,AR1

LOOP:           LSH     –1,R6             ; R6 is FFT_SIZE at the 1st loop.
                LDI     AR1,AR4
                ADDI    R7,AR1            ; AR1 points at A.
```

```
                LDI     AR1,AR2
                ADDI    2,AR2           ; AR2 points at B.
                ADDI    R6,AR4
                SUBI    R7,AR4          ; AR4 points at D.
                LDI     AR4,AR3
                SUBI    2,AR3           ; AR3 points at C.

                LDI     R7,IR1
                LDI     R7,RC

INLOP:          ADDF3   *--AR1(IR1),*
                        --AR3(IR1),R0   ; R0 = X'(I1) + X'(I3)
                SUBF3   *AR3,*AR1,R1    ; R1 = X'(I1) - X'(I3)
                LDF     *--AR4,R2
            ||  STF     R0,*AR1++       ; X'(I1)
                MPYF    -2.0,R2         ; R2 = -2*X'(I4)
                LDF     *--AR2,R3
            ||  STF     R1,*AR3++       ; X'(I3)
                MPYF    2.0,R3          ; R3 = 2*X'(I2)
                STF     R3,*AR2++(IR1)  ; X'(I2)
            ||  STF     R2,*AR4++(IR1)  ; X'(I4)

                LDI     @FFT_SIZE,IR1   ; IR1=separation between SIN/
                                        ; COS tbls
                LDI     @SINE_TABLE,AR0; AR0 points at SIN/COS table.
                LSH     -2,IR1
                SUBI    3,RC

                SUBF3   *AR2,*AR1,R3    ; R3 = X(I1)-X(I2)
                ADDF3   *AR1,*AR2,R2    ; R2 = X(I1)+X(I2)
                MPYF3   R3,*++AR0(IR0),R1; R1 = R3*SIN
                LDF     *AR4,R4         ; R4 = X(I4)
                MPYF3   R3,*++AR0(IR1),R0; R0 = R3*COS
            ||  SUBF3   *AR3,R4,R3      ; R3 = X(I4)-X(I3)
                ADDF3   R4,*AR3,R2      ; R2 = X(I3)+X(I4)
            ||  STF     R2,*AR1++       ; X(I1)
                MPYF3   R2,*AR0--(IR1),R4; R4 = R2*COS
            ||  STF     R3,*AR2--       ; X(I2)
                ADDF3   R4,R1,R3        ; R3 = R3*SIN + R2*COS
                MPYF3   R2,*AR0,R1      ; R1 = R2*SIN
            ||  STF     R3,*AR4--       ; X(I4)
                SUBF3   R1,R0,R4        ; R4 = R3*COS - R2*SIN

                RPTB    IN_BLK
```

```
                SUBF3   *AR2,*AR1,R3    ; R3 = X(I1)–X(I2)
                ADDF3   *AR1,*AR2,R2    ; R2 = X(I1)+X(I2)
                MPYF3   R3,*++AR0(IR0),R1; R1 = R3*SIN
            ||  STF     R4,*AR3++       ; X(I3)
                LDF     *AR4,R4         ; R4 = X(I4)
                MPYF3   R3,*++AR0(IR1),R0; R0 = R3*COS
            ||  SUBF3   *AR3,R4,R3      ; R3 = X(I4)–X(I3)
                ADDF3   R4,*AR3,R2      ; R2 = X(I3)+X(I4)
            ||  STF     R2,*AR1++       ; X(I1)
                MPYF3   R2,*AR0––(IR1),R4; R4 = R2*COS
            ||  STF     R3,*AR2––       ; X(I2)
                ADDF3   R4,R1,R3        ; R3 = R3*SIN + R2*COS
                MPYF3   R2,*AR0,R1      ; R1 = R2*SIN
            ||  STF     R3,*AR4––       ; X(I4)
IN_BLK:         SUBF3   R1,R0,R4        ; R4 = R3*COS – R2*SIN

                SUBF3   *AR2,*AR1,R3    ; R3 = X(I1)–X(I2)
                ADDF3   *AR1,*AR2,R2    ; R2 = X(I1)+X(I2)
                MPYF3   R3,*++AR0(IR0),R1; R1 = R3*SIN
            ||  STF     R4,*AR3++       ; X(I3)
                LDF     *AR4,R4         ; R4 = X(I4)
                MPYF3   R3,*++AR0(IR1),R0; R0 = R3*COS
            ||  SUBF3   *AR3,R4,R3      ; R3 = X(I4)–X(I3)
                ADDF3   R4,*AR3,R2      ; R2 = X(I3)+X(I4)
            ||  STF     R2,*AR1         ; X(I1)
                MPYF3   R2,*AR0––(IR1),R4; R4 = R2*COS
            ||  STF     R3,*AR2         ; X(I2)
                LDI     R6,IR1          ; Get prepared for the next
                ADDF3   R4,R1,R3        ; R3 = R3*SIN + R2*COS
                MPYF3   R2,*AR0,R1      ; R1 = R2*SIN
            ||  STF     R3,*AR4++(IR1)  ; X(I4)
                SUBF3   R1,R0,R4        ; R4 = R3*COS – R2*SIN
                NEGF    *AR1++(IR1),R2  ; Dummy
            ||  STF     R4,*AR3++(IR1)  ; X(I3)

                SUBI3   AR5,AR1,R0
                CMPI    @FFT_SIZE,R0
                BLTD    INLOP           ; Loop back to the inner loop
                NOP     *AR2++(IR1)     ; Dummy
                LDI     R7,IR1
                LDI     R7,RC

                ADDI    1,R5
                CMPI    @LOG_SIZE,R5    ; Next stage if any left
                BLED    LOOP
                LDI     @SOURCE_ADDR,AR1
                LSH     1,IR0           ; Double step in sinus table
                LSH     –1,R7
```

```
;
; Perform third FFT loop.

;Part A:
```




```
          LDI    @SOURCE_ADDR,AR1
          LDI    AR1,AR2
          LDI    AR1,AR3
          LDI    AR1,AR4
          ADDI   2,AR2
          ADDI   4,AR3
          ADDI   6,AR4
          LDI    8,IR0
          LDI    @FFT_SIZE,RC
          LSH    –3,RC
          SUBI   1,RC
          LDI    @SINE_TABLE,AR0      ; AR0 points at SIN/COS table.

          RPTB   LOOP3_A
          LDF    *AR3,R3
          ADDF3  R3,*AR1,R0      ; R0 = X’(I1) + X’(I3)
          SUBF3  R3,*AR1,R1      ; R1 = X’(I1) – X’(I3)
          LDF    *AR4,R2         ;
      ||  STF    R0,*AR1++(IR0)  ; X’(I1)
          MPYF   –2.0,R2         ; R2 = –2*X’(I4)
          LDF    *AR2,R3         ;
      ||  STF    R1,*AR3++(IR0)  ; X’(I3)
          MPYF   2.0,R3          ; R3 = 2*X’(I2)
LOOP3_A:  STF    R3,*AR2++(IR0)  ; X’(I2)
      ||  STF    R2,*AR4++(IR0)  ; X’(I4)
```

```
;
; Part B:
;
;                               0
;    AR1  →   |  I1  |          1   ←  X(I1) + X(I2)
;                               2
;    AR2  →   |  I2  |          3   ←  X(I1) – X(I3)
;                               4
;    AR3  →   |  I3  |          5   ←  [X(I1)– X(I2)]*COS– [X(I3)+ X(I4)]*SIN
;                               6
;    AR4  →   |  I4  |          7   ←  [X(I1)– X(I2)]*SIN+ [X(I3)+ X(I4)]*COS]
;                               8
;    AR1  →   |      |          9      NOTE: COS(2*pi/8) = SIN(2*pi/8)
;                 ↓
;
;

            LDI     @SOURCE_ADDR,AR1
            LDI     AR1,AR2
            LDI     AR1,AR3
            LDI     AR1,AR4
            ADDI    1,AR1
            ADDI    3,AR2
            ADDI    5,AR3
            ADDI    7,AR4
            LDI     @SINE_TABLE,AR7     ; AR7 points at SIN/COS table.
            LDI     @FFT_SIZE,RC
            LSH     –3,RC
            LDI     RC,IR1
            SUBI    2,RC
```

```
                LDF     *AR2,R6             ; R6 = X(I2)
                LDF     *AR3,R0             ; R0 = X(I3)
                ADDF3   R6,*AR1,R5          ; R5 = X(I1)+X(I2)
                SUBF3   R6,*AR1,R4          ; R4 = X(I1)-X(I2)
                SUBF3   R0,R4,R3            ; R3 = X(I1)-X(I2)-X(I3)
                ADDF3   R0,R4,R2            ; R2 = X(I1)-X(I2)+X(I3)
                SUBF3   R0,*AR4,R1          ; R1 = X(I4)-X(I3)
          ||    STF     R5,*AR1++(IR0)      ; X(I1)
                ADDF3   R2,*AR4,R5          ; R5 = X(I1)-X(I2)+X(I3)+X(I4)
          ||    STF     R1,*AR2++(IR0)      ; X(I2)
                MPYF3   R5,*++AR7(IR1),R1   ; R1 = R5*SIN
          ||    SUBF3   *AR4,R3,R2          ; R2 = X(I1)-X(I2)-X(I3)-X(I4)
                MPYF3   R2,*AR7,R0          ; R0 = R2*SIN
          ||    STF     R1,*AR4++(IR0)      ; X(I4)
                                            ;
                RPTB    LOOP3_B             ;
                                            ;
                LDF     *AR2,R6             ; R6 = X(I2)
          ||    STF     R0,*AR3++(IR0)      ; X(I3)
                ADDF3   R6,*AR1,R5          ; R5 = X(I1)+X(I2)
                LDF     *AR3,R0             ; R0 = X(I3)
                SUBF3   R6,*AR1,R4          ; R4 = X(I1)-X(I2)
                SUBF3   R0,R4,R3            ; R3 = X(I1)-X(I2)-X(I3)
                ADDF3   R0,R4,R2            ; R2 = X(I1)-X(I2)+X(I3)
                SUBF3   R0,*AR4,R1          ; R1 = X(I4)-X(I3)
          ||    STF     R5,*AR1++(IR0)      ; X(I1)
                ADDF3   R2,*AR4,R5          ; R5 = X(I1)-X(I2)+X(I3)+X(I4)
          ||    STF     R1,*AR2++(IR0)      ; X(I2)
                MPYF3   R5,*AR7,R1          ; R1 = R5*SIN
          ||    SUBF3   *AR4,R3,R2          ; R2 = X(I1)-X(I2)-X(I3)-X(I4)
LOOP3_B:        MPYF3   R2,*AR7,R0          ; R0 = R2*SIN
          ||    STF     R1,*AR4++(IR0)      ; X(I4)

                STF     R0,*AR3             ; X(I3)
```

```
;
; Perform first and second FFT loops.
;
;   AR1  →  | I1 |  0  ←  X(I1) + X(I3) +  2*X(I2)
;   AR2  →  | I2 |  1  ←  X(I1) + X(I3) –  2*X(I2)
;   AR3  →  | I3 |  2  ←  X(I1) – X(I3) –  2*X(I4)
;   AR4  →  | I4 |  3  ←  X(I1) – X(I3) +  2*X(I4)
;   AR1  →  |    |  4
;           |    |
;             ↓
;

                  LDI     @SOURCE_ADDR,AR1
                  LDI      AR1,AR2
                  LDI      AR1,AR3
                  LDI      AR1,AR4
                  ADDI     1,AR2
                  ADDI     2,AR3
                  ADDI     3,AR4
                  LDI      4,IR0
                  LDI      @FFT_SIZE,RC
                  LSH      –2,RC
                  SUBI     2,RC
```

```
                LDF     *AR4,R6             ; R6 = X(I4)
                LDF     *AR2,R7             ; R7 = X(I2)
             ||  LDF    *AR1,R1             ; R1 = X(I1)
                MPYF    2.0,R6              ; R6 = 2 * X(I4)
                MPYF    2.0,R7              ; R7 = 2 * X(I2)
                SUBF3   R6,*AR3,R5          ; R5 = X(I3) – 2*X(I4)
                SUBF3   R5,R1,R4            ; R4 = X(I1)–X(I3)+2X(I4)
                SUBF3   R7,*AR3,R5          ; R5 = X(I3) – 2*X(I2)
              ||  STF   R4,*AR4++(IR0)      ; X(I4)
                ADDF3   R5,R1,R3            ; R3 = X(I1)+X(I3)–2X(I2)
                ADDF3   R6,*AR3,R4          ; R4 = X(I3) + 2*X(I4)
              ||  STF   R3,*AR2++(IR0)      ; X(I2)
                SUBF3   R4,R1,R4            ; R4 = X(I1)–X(I3)–2X(I4)
                ADDF3   R7,*AR3,R0          ; R0 = X(I3) + 2*X(I2)
              ||  STF   R4,*AR3++(IR0)      ; X(I3)
                ADDF3   R0,R1,R0            ; R0 = X(I1)+X(I3)+2X(I2)
                                            ;
                RPTB    LOOP1_2             ;
                LDF     *AR4,R6             ; R6 = X(I4)
              ||  STF   R0,*AR1++(IR0)      ; X(I1)
                MPYF    2.0,R6              ; R6 = 2 * X(I4)
                LDF     *AR2,R7             ; R7 = X(I2)
              ||  LDF   *AR1,R1             ; R1 = X(I1)
                MPYF    2.0,R7              ; R7 = 2 * X(I2)
                SUBF3   R6,*AR3,R5          ; R5 = X(I3) – 2*X(I4)
                SUBF3   R5,R1,R4            ; R4 = X(I1)–X(I3)+2X(I4)
                SUBF3   R7,*AR3,R5          ; R5 = X(I3) – 2*X(I2)
              ||  STF   R4,*AR4++(IR0)      ; X(I4)
                ADDF3   R5,R1,R3            ; R3 = X(I1)+X(I3)–2X(I2)
                ADDF3   R6,*AR3,R4          ; R4 = X(I3) + 2*X(I4)
              ||  STF   R3,*AR2++(IR0)      ; X(I2)
                SUBF3   R4,R1,R4            ; R4 = X(I1)–X(I3)–2X(I4)
                ADDF3   R7,*AR3,R0          ; R0 = X(I3) + 2*X(I2)
              ||  STF   R4,*AR3++(IR0)      ; X(I3)
LOOP1_2:        ADDF3   R0,R1,R0            ; R0 = X(I1)+X(I3)+2X(I2)
                                            ;
                STF     R0,*AR1             ; LAST X(I1)
```

```
;
; Check bit reversing mode (on or off).
;
; BIT_REVERSING =  0, then OFF (no bit reversing).
; BIT_REVERSING <> 0, then ON.
;
                LDI     @BIT_REVERSE,R0
                CMPI    0,R0
                BZ      MOVE_DATA

;
; Check bit reversing type.
;
; If SourceAddr =  DestAddr, then in place bit reversing.
; If SourceAddr <> DestAddr, then standard bit reversing.
;

                LDI     @SOURCE_ADDR,R0
                CMPI    @DEST_ADDR,R0
                BEQ     IN_PLACE

;
; Bit reversing type 1 (from source to destination).
;
; NOTE: abs(SOURCE_ADDR – DEST_ADDR) must be > FFT_SIZE, this is not checked.
;

                LDI     @FFT_SIZE,R0
                SUBI    2,R0
                LDI     @FFT_SIZE,IR0
                LSH     –1,IR0              ; IRO = half FFT size.
                LDI     @SOURCE_ADDR,AR0
                LDI     @DEST_ADDR,AR1

                LDF     *AR0++,R1

                RPTS    R0
                LDF     *AR0++,R1
          ||    STF     R1,*AR1++(IR0)B

                STF     R1,*AR1++(IR0)B

                BR      DIVISION
```

```
                                ;
                                ; In-place bit reversing.
                                ;

                                ; Bit reversing on even locations, 1st half
                                ;    only.
IN_PLACE:  LDI     @FFT_SIZE,IR0
           LSH     -2,IR0       ; IRO = quarter FFT size.
           LDI     2,IR1

           LDI     @FFT_SIZE,RC
           LSH     -2,RC
           SUBI    3,RC
           LDI     @DEST_ADDR,AR0
           LDI     AR0,AR1
           LDI     AR0,AR2

           NOP     *AR1++(IR0)B
           NOP     *AR2++(IR0)B
           LDF     *++AR0(IR1),R0
           LDF     *AR1,R1
           CMPI    AR1,AR0      ; Xchange locations only if AR0<AR1.
           LDFGT   R0,R1
           LDFGT   *AR1++(IR0)B,R1

           RPTB    BITRV1
           LDF     *++AR0(IR1),R0
||         STF     R0,*AR0
           LDF     *AR1,R1
||         STF     R1,*AR2++(IR0)B
           CMPI    AR1,AR0
           LDFGT   R0,R1
BITRV1:    LDFGT   *AR1++(IR0)B,R0

           STF     R0,*AR0
           STF     R1,*AR2

                                ; Perform bit reversing on odd locations,
                                ;    2nd half only.

           LDI     @FFT_SIZE,RC
           LSH     -1,RC
           LDI     @DEST_ADDR,AR0
           ADDI    RC,AR0
           ADDI    1,AR0
           LDI     AR0,AR1
           LDI     AR0,AR2
           LSH     -1,RC
           SUBI    3,RC

           NOP     *AR1++(IR0)B
           NOP     *AR2++(IR0)B
           LDF     *++AR0(IR1),R0
```

```
            LDF     *AR1,R1
            CMPI    AR1,AR0         ; Xchange locations only if AR0<AR1.
            LDFGT   R0,R1
            LDFGT   *AR1++(IR0)B,R1

            RPTB    BITRV2
            LDF     *++AR0(IR1),R0
||          STF     R0,*AR0
            LDF     *AR1,R1
||          STF     R1,*AR2++(IR0)B
            CMPI    AR1,AR0
            LDFGT   R0,R1
BITRV2:     LDFGT   *AR1++(IR0)B,R0

            STF     R0,*AR0
            STF     R1,*AR2

                                    ; Perform bit reversing on odd
                                    ;     locations, 1st half only.

            LDI     @FFT_SIZE,RC
            LSH     -1,RC
            LDI     RC,IR0
            LDI     @DEST_ADDR,AR0
            LDI     AR0,AR1
            ADDI    1,AR0
            ADDI    IR0,AR1
            LSH     -1,RC
            LDI     RC,IR0
            SUBI    2,RC

            LDF     *AR0,R0
            LDF     *AR1,R1

            RPTB    BITRV3
            LDF     *++AR0(IR1),R0
||          STF     R0,*AR1++(IR0)B
BITRV3:     LDF     *AR1,R1
||          STF     R1,*-AR0(IR1)

            STF     R0,*AR1
            STF     R1,*AR0

            BR      DIVISION
```

```
                                                ;
                                                ; Check data source locations.
                                                ;
                                                ; If SourceAddr =
                                                ;     DestAddr, then do nothing.
                                                ; If SourceAddr <>
                                                ;     DestAddr, then move data.
                          ;
MOVE_DATA:        LDI     @SOURCE_ADDR,R0
                  CMPI    @DEST_ADDR,R0
                  BEQ     DIVISION

                  LDI     @FFT_SIZE,R0
                  SUBI    2,R0
                  LDI     @SOURCE_ADDR,AR0
                  LDI     @DEST_ADDR,AR1

                  LDF     *AR0++,R1

                  RPTS    R0
                  LDF     *AR0++,R1
            ||    STF     R1,*AR1++

                  STF     R1,*AR1

DIVISION:         LDI     2,IR0
                  LDI     @FFT_SIZE,R0
                  FLOAT   R0                ; exp = LOG_SIZE
                  PUSHF   R0                ; 32 MSB'S saved
                  POP     R0
                  NEGI    R0                ; Neg exponent
                  PUSH    R0
                  POPF    R0                ; R0 = 1/FFT_SIZE
                  LDI     @DEST_ADDR,AR1
                  LDI     @DEST_ADDR,AR2
                  NOP     *AR2++
                  LDI     @FFT_SIZE,RC
                  LSH     -1,RC
                  SUBI    2,RC
                  MPYF3   R0,*AR1,R1        ; 1st location
                  RPTB    LAST_LOOP
                  MPYF3   R0,*AR2,R2        ; 2nd,4th,6th,... location
            ||    STF     R1,*AR1++(IR0)
LAST_LOOP:        MPYF3   R0,*AR1,R1        ; 3rd,5th,7th,... location
            ||    STF     R2,*AR2++(IR0)

                  MPYF3   R0,*AR2,R2        ; Last location
            ||    STF     R1,*AR1
                  STF     R2,*AR2
```

```
                                ; Return to C environment.
                                ;
            POP     DP          ; Restore C environment variables.
            POP     AR7
            POP     AR6
            POP     AR5
            POP     AR4
            POPF    R7
            POP     R7
            POPF    R6
            POP     R6
            POP     R5
            POP     R4
            POP     FP
            RETS

            .end

*
* No more.
*
*******************************************************************************
*
```

The TMS320C3x quickly executes FFT lengths up to 1024 points (complex) or 2048 (real), covering most applications, because it can do so almost entirely in on-chip memory. Table 11–1 and Table 11–2 summarize the number of CPU clock cycles and the execution time required for FFT lengths between 64 and 1024 points for the four algorithms.

Table 11–1. TMS320C3x FFT Timing Benchmarks (Cycles)

	FFT Timing in Cycles			
Number of Points	**RADIX-2 (Complex)**	**RADIX-4 (Complex)**	**RADIX-2 (Real)**	**RADIX-2 (Real Inverse)**
64	2770	2050	810	1070
128	6170	—	1760	2370
256	13600	10400	3940	5290
512	29740	—	8860	11740
1024	64570	50670	19820	25900
1024†	39500			

† This benchmark is based on the Meyer and Schwarz program found in *Digital Signal Processing Applications With the TMS320 Family, Volume 3*.

Table 11–2. TMS320C3x FFT Timing Benchmarks (Milliseconds)

	FFT Timing in Milliseconds			
Number of Points	**RADIX-2 (Complex)**	**RADIX-4 (Complex)**	**RADIX-2 (Real)**	**RADIX-2 (Real Inverse)**
64	0.139	0.103	0.041	0.054
128	0.309	—	0.088	0.119
256	0.680	0.520	0.197	0.265
512	1.487	—	0.443	0.587
1024	3.229	2.534	0.991	1.295
1024†	1.975			

† This benchmark is based on the Meyer and Schwarz program found in *Digital Signal Processing Applications With the TMS320 Family, Volume 3*.

11.4.5 Lattice Filters

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

If H(z) is the transfer function of a digital filter that has only poles, A(z) = 1/H(z) will be a filter having only 0s, and it will be called the inverse filter. The inverse lattice filter is shown in Figure 11–5. These equations describe the filter in mathematical terms:

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

Initial conditions:

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

Final conditions:

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

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

Figure 11–5. Structure of the Inverse Lattice Filter

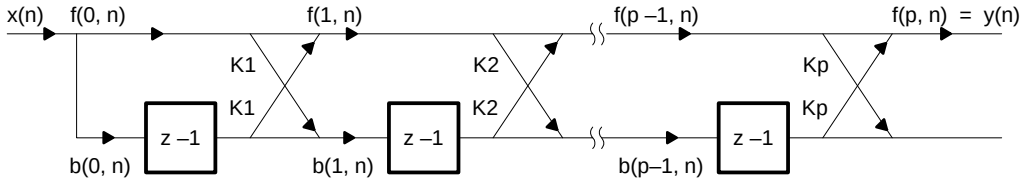
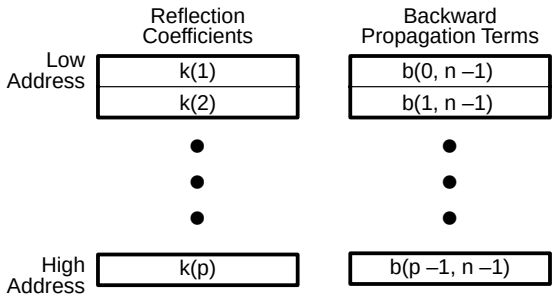


Figure 11–6 shows the data memory organization of the inverse lattice-filter on the TMS320C3x.

Figure 11–6. Data Memory Organization for Lattice Filters



Example 11–40 shows the implementation of an inverse lattice filter.

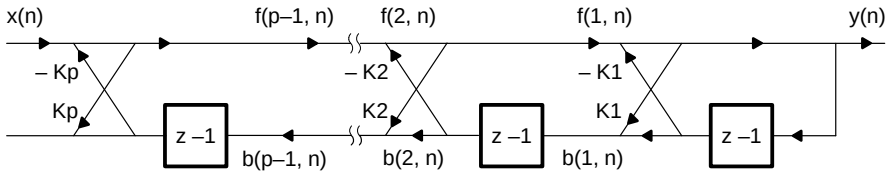
Example 11–40. Inverse Lattice Filter

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


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


*  ARGUMENT  ASSIGNMENTS:


*  ARGUMENT  |  FUNCTION
*  ----------+-------------------------------------
*  R2        |  f(0,n) = x(n)
*  AR0       |  ADDRESS OF FILTER COEFFICIENTS  (k(1))
*  AR1       |  ADDRESS OF BACKWARD PROPAGATION
*            |      VALUES (b(0,n±1))
*  RC        |  RC = p ± 2
*


*  REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*  REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*  REGISTER CONTAINING RESULT: R2 (f(p,n))
*
*
*  PROGRAM SIZE: 10 WORDS
*
*  EXECUTION CYCLES: 13 + 3 * (p±1)
*
*


      .global LATINV
*
*  i = 1
*
LATINV MPYF3 *AR0, *AR1, R0
```

```
*                                       ;  k(1) * b(0,n±1) ±> R0
*                                       ;  Assume f(0,n) ±> R2.
       LDF    R2,R3                     ;  Put b(0,n) = f(0,n) ±> R3.
       MPYF3  *AR0++(1),R2,R1
*                                       ;  k(1) * f(0,n) ±> R1
*

*   2 <= i <= p

*
       RPTB   LOOP
       MPYF3  *AR0,*++AR1(1),R0         ;  k(i) * b(i±1,n±1) ±> R0
||     ADDF3  R2,R0,R2                  ;  f(i±1±1,n)+k(i±1)
*                                       ;  *b(i±1±1,n±1)
*                                       ;  = f(i±1,n) ±> R2
*
*                                       ;  b(i±1±1,b±1)+k(i±1)*f(i±1±1,n)
       ADDF3  *±AR1(1), R1, R3          ;  = b(i±1,n) ±> R3
||     STF R3, *±AR1(1)                 ;  b(i±1±1,n) ±> b(i±1±1,n±1)
*
LOOP   MPYF3  *AR0++(1),R2,R1
*                                       ;  k(i) * f(i±1,n) ±> R1
*
*      I = P+1 (CLEANUP)


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

*   RETURN SEQUENCE
*
       RETS                             ;  RETURN
     *
     *   end
     *
        .end
```

The forward lattice filter is similar in structure to the inverse filter, as shown in Figure 11–7.

Figure 11–7. Structure of the (Forward) Lattice Filter



These corresponding equations describe the lattice filter:

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

Initial conditions:

$f(p,n) = x(n),\ b(i,n-1) = 0 \quad$ for $i = 1,...,p$

Final conditions:

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

The data memory organization is identical to that of the inverse filter, as shown in Figure 11–6 on page 11-126. Example 11–41 shows the implementation of the lattice filter on the TMS320C3x.

Example 11–41. Lattice Filter

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

*     LOAD   AR0
*     LOAD   AR1
*     LOAD   RC
*     CALL   LATICE
*
*
*  ARGUMENT  ASSIGNMENTS:
*  ARGUMENT  |   FUNCTION
*  ----------+-----------------------------------
*  R2        |   F(P,N) = E(N) = EXCITATION
*  AR0       |   ADDRESS OF FILTER COEFFICIENTS (K(P))
*  AR1       |   ADDRESS OF BACKWARD PROPAGATION VALUES (B(P±1,N±1))
   IR0       |   3
*  RC        |   RC = P ± 3
*
*  REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*  REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*  REGISTER CONTAINING RESULT: R2 (f(0,n))
*
*  STACK USAGE: NONE
*
*  PROGRAM SIZE: 12 WORDS
*
*  EXECUTION CYCLES: 15 + 3 * (P±2)
*
```

```
       .global  LATICE
*
*
LATICE MPYF3  *AR0,*AR1,R0
*                              ;  K(P) * B(P±1,N±1) ±> R0
                               ;  Assume F(P,N) ±> R2
       SUBF3  R0,R2,R2         ;  F(P,N)±K(P)*B(P±1,N±1)
                               ;     = F(P±1,N) ±> R2
||     MPYF3  *−−AR0(1),*−−AR1(1),R0
                               ;  K(P−1) * B(P±2,N±1) ±> R0
       SUBF3  R0,R2,R2         ;  F(P−1,N)±K(P−1)*B(P±2,N±1)
                               ;     = F(P±2,N) ±> R2
||     MPYF3  *−−AR0(1),*−−AR1(1),R0
                               ;  K(P−2) * B(P−3,N−1) ±> R0
       MPYF3  R2,*+AR0(1),R1   ;  F(P−2,N) * K(P−1) ±> R1
       ADDF3  R1,*+AR1(1),R3   ;  F(P±2,N) * K(P−1) + B(P±2,N−1)
                               ;     = B(P−1,N) ±> R3

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

11.5 Programming Tips

Programming style reflects personal preference. The purpose of this section is not to impose any particular style; rather, it is to highlight features of the TMS320C3x that can help to produce faster and/or shorter programs. The tips cover the C compiler, assembly language programming, and low-power-mode wakeup.

11.5.1 C-Callable Routines

The TMS320C3x was designed with a large register file, software stack, and large memory space to implement a high-level language (HLL) compiler easily. The first such implementation supplied is a C compiler. Use of the C compiler increases the transportability of applications that have been tested on large, general-purpose computers, and it decreases their porting time.

For best use of the compiler, complete the following steps:

1) Write the application in the high-level language.

2) Debug the program.

3) Determine whether it runs in real-time.

4) If it doesn't, identify the places where most of the execution time is spent.

5) Optimize these areas by writing assembly language routines that implement the functions.

6) Call the routines from the C program as C functions.

When writing a C program, you can increase the execution speed by maximizing the use of register variables. For more information, refer to the *TMS320C3x C Compiler Reference Guide*.

You must observe certain conventions when writing a C-callable routine. These conventions are outlined in the *Runtime Environment* chapter of the *TMS320C3x C Compiler Reference Guide*. Certain registers are saved by the calling function, and others need to be saved by the called function. The C compiler manual helps achieve a clean interface. The end result is the readability and natural flow of a high-level language combined with the efficiency and special-feature use of assembly language.

11.5.2 Hints for Assembly Coding

Each program has particular requirements. Not all possible optimizations will make sense in every case. You can use the suggestions presented in this section as a checklist of available software tools.

- **Use delayed branches**. Delayed branches execute in a single cycle; regular branches execute in four cycles. The following three instructions are also executed whether the branch is taken or not. If fewer than three instructions can be used, use the delayed branch and append NOPs. Machine cycles (time) are still being saved.
- **Apply the repeat single/block construct**. In this way, loops are achieved with no overhead. Nesting such constructs will not normally increase efficiency, so try to use the feature on the most often performed loop. Note that RPTS is not interruptible, and the executed instruction is not refetched for execution. This frees the buses for operands.
- **Use parallel instructions**. It is possible to have a multiply in parallel with an add (or subtract) and to have stores in parallel with any multiply or ALU operation. This increases the number of operations executed in a single cycle. For maximum efficiency, observe the addressing modes used in parallel instructions and arrange the data appropriately.
- **Maximize the use of registers**. The registers are an efficient way to access scratch-pad memory. Extensive use of the register file facilitates the use of parallel instructions and helps avoid pipeline conflicts when you use the registers in addressing modes.
- **Use the cache**. This is especially important in conjunction with external slow memory. The cache is transparent to the user, so make sure that it is enabled.
- **Use internal memory instead of external memory.** The internal memory (2K x 32 bits RAM and 4K x 32 bits ROM) is considerably faster to access. In a single cycle, two operands can be brought from internal memory. You can maximize performance if you use the DMA in parallel with the CPU to transfer data to internal memory before you operate on it.
- **Avoid pipeline conflicts**. If there is no problem with program speed, ignore this suggestion. For time-critical operations, make sure you do not miss any cycles because of conflicts. To identify conflicts, run the trace function on the development tools (simulator, emulators) with the program tracing option enabled. The tracing immediately identifies the pipeline conflicts. Consult the appropriate section of this user's guide for an explanation of the reason for the conflict. You can then take steps to correct the problem.

The above checklist is not exhaustive, and it does not address the more detailed features outlined in other sections of this manual. To learn how to exploit the full power of the TMS320C3x, study the architecture, hardware configuration, and instruction set of the device. These subjects are described in earlier chapters.

11.5.3 Low-Power-Mode Wakeup Example

There are two instructions by which the TMS320C31 is placed in the low power consumption mode:

- IDLE2
- LOPOWER

The LOPOWER instruction will slow down the H1/H3 clock by a factor of 16 during the read phase of the instruction. The MAXSPEED instruction will wake the device from the low-power mode and return it to full frequency during MAXSPEED's read cycle. However, the H1/H3 clock may resume with the phase opposite from before the clocks were shut down.

The IDLE2 instruction has the same functions that the IDLE instruction has, except that the clock is stopped during the execute phase of the IDLE2 instruction. The clock pin will stop with H1 high and H3 low. The status of all of the signals will remain the same as in the execute phase of the IDLE2 instruction. In emulation mode, however, the clocks will continue to run, and IDLE2 will operate identically to IDLE. The external interrupts INT(0–3) are the only signals that start the processor up from the mode the device was in. Therefore, you must enable the external interrupt before going to IDLE2 power-down mode. (See Example 11–42.) If the proper external interrupt is not set up before executing IDLE2 to power down, the only way to wake up the processor is with a device RESET.

Example 11–42. Setup of IDLE2 Power-Down-Mode Wakeup

```
*
*  TITLE IDLE2 POWER-DOWN MODE WAKEUP ROUTINE SETUP
*
*  THIS EXAMPLE SETS UP THE EXTERNAL INTERRUPT 0, INT0, BEFORE
*  EXECUTING THE IDLE2 INSTRUCTION. WHEN THE INT0 SIGNAL IS RECEIVED
*  LATER, THE PROCESSOR WILL RESUME FROM ITS PREVIOUS
*  STATE. NOTE: THE "INTRPT" SECTION IS MAPPED FROM THE
*  ADDRESS 0 FROM THE RESET AND INTERRUPT VECTORS.
*


   .  sect   "INTRPT"
RESET  .word  START     ;  Reset vector
INT0   .word  INT0_ISR  ;  INT0 interrupt vector
INT1   .word  INT1_ISR  ;  INT1 interrupt vector
INT2   .word  INT2_ISR  ;  INT2 interrupt vector
INT3   .word  INT3_ISR  ;  INT3 interrupt vector
       :  :
       :  :
       .text
       :  :
```

```
        :   :
      LDP    @SP_ADR
      LDI    @SP_ADR,SP    ;  Set up stack pointer
      OR     01h, IE       ;  Enable INT0
      IDLE2                ;  Set GIE = 1 and stop clock
        :   :
        :   :
        :   :
        :   :
INT0_ISR  RETI             ;  Return to instruction after IDLE2
```

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

Chapter 12

Hardware Applications

The TMS320C3x's advanced interface design can implement many system configurations. Its two external buses and DMA capability provide a parallel 32-bit interface to external devices, while the interrupt interface, dual serial ports, and general-purpose digital I/O provide communication with many peripherals.

This chapter describes how to use the TMS320C3x's interfaces to connect to various external devices. Specific discussions include implementation of parallel interface to devices with and without wait states, use of general-purpose I/O, and system control functions. All interfaces shown in this chapter have been built and tested to verify proper operation and apply to the TMS320C30. Comparable designs for the other TMS320C3x devices can be implemented with appropriate logic.

Major topics discussed in this chapter are as follows:

Topic **Page**

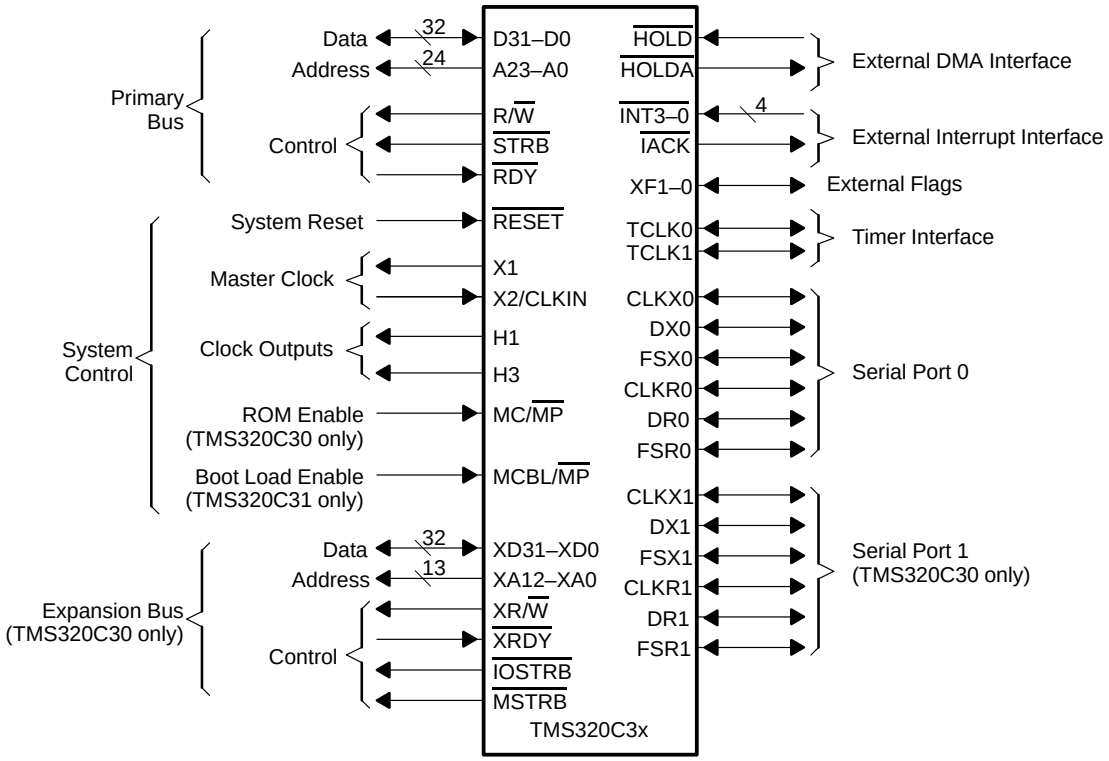
12.1 System Configuration Options Overview

The various TMS320C3x interfaces connect to many different device types. Each of these interfaces is tailored to a particular family of devices.

12.1.1 Categories of Interfaces on the TMS320C3x

The TMS320C3x interface types fall into several categories, depending on the devices to which they are intended to be connected. Each interface comprises one or more signal lines that transfer information and control its operation. Figure 12–1 shows the signal line groupings for each of these various interfaces.

Figure 12–1. External Interfaces on the TMS320C3x



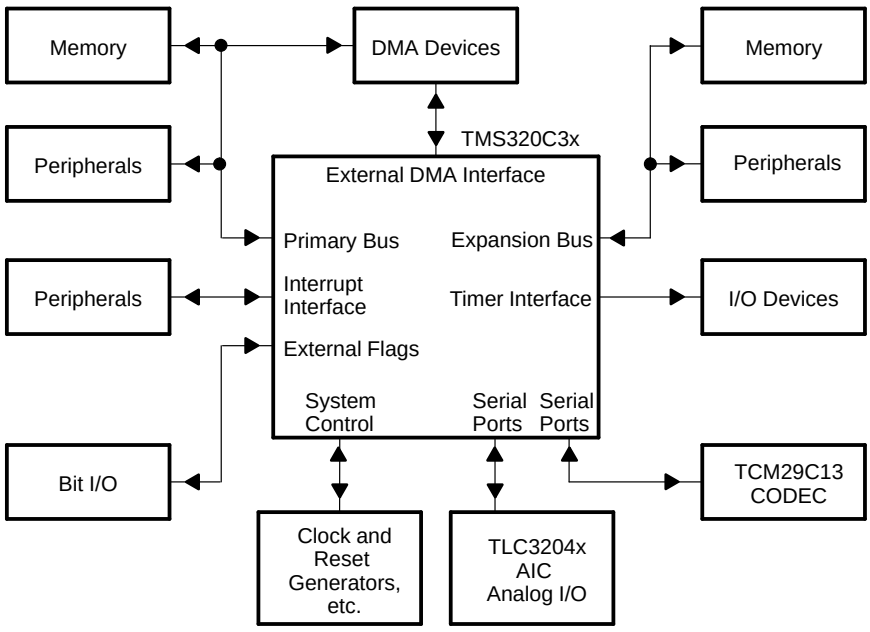
All of the interfaces are independent of one another, and you can perform different operations simultaneously on each interface.

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

12.1.2 Typical System Block Diagram

The devices that can be interfaced to the TMS320C3x include memory, DMA devices, and numerous parallel and serial peripherals and I/O devices. Figure 12–2 illustrates a typical configuration of a TMS320C3x system with different types of external devices and the interfaces to which they are connected.

Figure 12–2. Possible System Configurations



This block diagram constitutes essentially a fully expanded system. In an actual design, you can use any subset of the illustrated configuration as appropriate.

12.2 Primary Bus Interface

The TMS320C3x uses the primary bus to access the majority of its memory-mapped locations. Therefore, typically, when a large amount of external memory is required in a system, it is interfaced to the primary bus. The expansion bus (discussed in Section 12.3 on page 12-19) actually comprises two mutually exclusive interfaces, controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals, respectively. Cycles on the expansion bus controlled by the $\overline{\text{MSTRB}}$ signal are essentially equivalent to cycles on the primary bus, except that bank switching is not implemented on the expansion bus. Accordingly, the discussion of primary bus cycles in this section applies equally to $\overline{\text{MSTRB}}$ cycles on the expansion bus.

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

12.2.1 Zero-Wait-State Interface to Static RAMs

Zero-wait-state read access time for the TMS320C3x is determined by the difference between the cycle time (specification 10 in Table 13–12 on page 13-30) and the sum of the times for H1 low to address valid (specification 14.1 in Table 13–13 on page 13-33) and data setup before next H1 low (specification 15.1 in Table 13–13 on page 13-33):

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

For example, for full-speed, zero-wait-state interface to any device, the 60-ns TMS320C3x requires a read access time of 30 ns from address stable to data valid. Because for most memories access time from chip select is the same as access time from address, it is theoretically possible to use 30-ns memories at full speed with the TMS320C3x-33. This requires that there be no delays between the processor and the memories. However, because of interconnection delays and because some gating is normally required for chip-select generation, this is usually not the case. Therefore, slightly faster memories are required in most systems.

Among currently available RAMs, there are two distinct categories of devices with different interface characteristics:

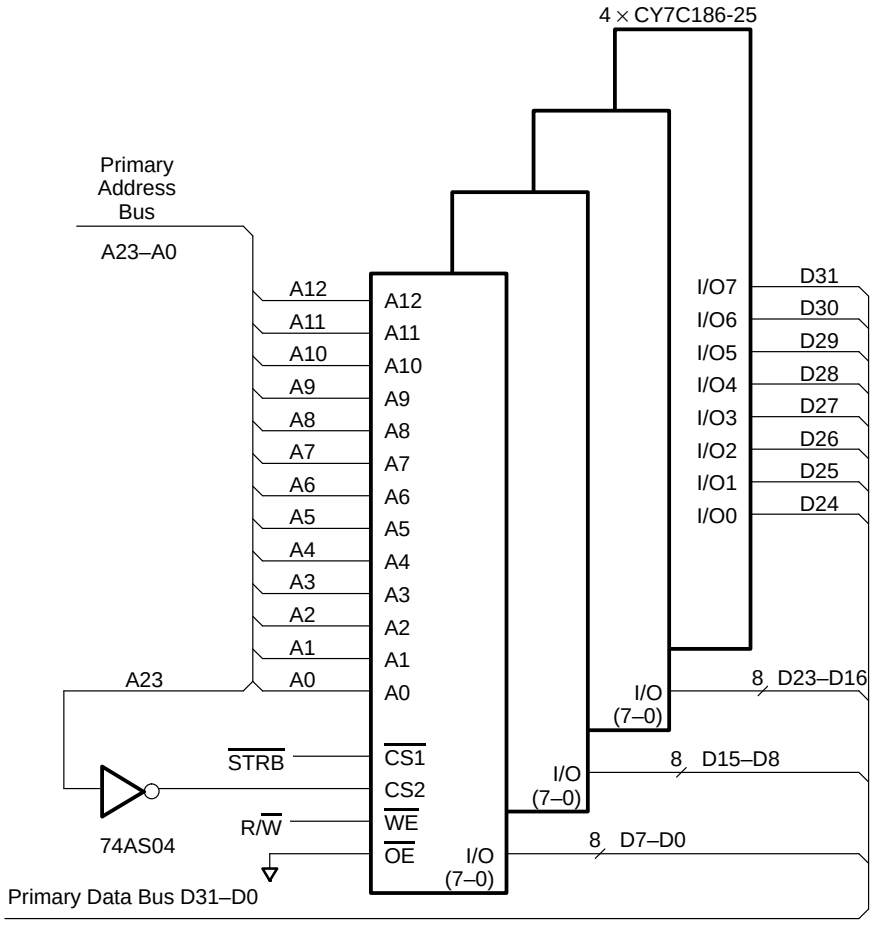
- RAMs without output enable control lines ($\overline{\text{OE}}$), which include the one-bit-wide organized RAMs and most of the four-bit wide RAMs
- RAMs with $\overline{\text{OE}}$ controls, which include the byte-wide RAMs and a few of the four-bit wide RAMs

Many of the fastest RAMs do not provide $\overline{OE}$ control; they use chip-select ($\overline{CS}$) controlled write cycles to ensure that data outputs do not turn on for write operations. In $\overline{CS}$-controlled write cycles, the write control line ($\overline{WE}$) goes low before $\overline{CS}$ goes low, and internal logic holds the outputs disabled until the cycle is completed. Using $\overline{CS}$-controlled write cycles is an efficient way to interface fast RAMs without $\overline{OE}$ controls to the TMS320C30 at full speed.

In the case of RAMs with $\overline{OE}$ controls, using this signal can add flexibility to many systems. Additionally, many of these devices can be interfaced by using $\overline{CS}$-controlled write cycles with $\overline{OE}$ tied low in the same manner as with RAMs without $\overline{OE}$ controls. There are, however, two requirements for interfacing to $\overline{OE}$ RAMs in this manner. First, the RAM's $\overline{OE}$ input must be gated with chip select and $\overline{WE}$ internally so that the device's outputs do not turn on unless a read is being performed. Second, the RAM must allow its address inputs to change while $\overline{WE}$ is low; some RAMs specifically prohibit this.

Figure 12–3 shows the TMS320C3x interfaced to Cypress Semiconductor's CY7C186 25-ns 8K x 8-bit CMOS static RAM with the $\overline{OE}$ control input tied low and using a $\overline{CS}$-controlled write cycle.

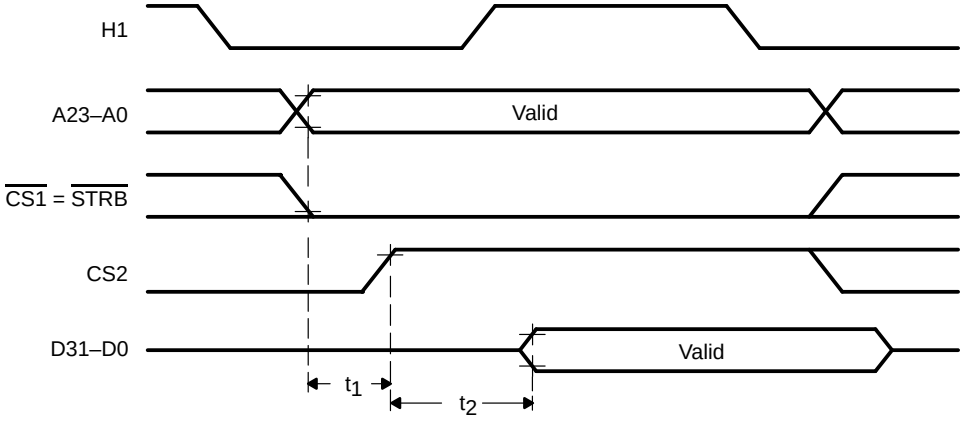
Figure 12–3. TMS320C3x Interface to Cypress Semiconductor CY7C186 CMOS SRAM



In this circuit, the two chip selects on the RAM are driven by $\overline{\text{STRB}}$ and $\overline{\text{A23}}$, which are ANDed together internally. $\overline{\text{A23}}$ locates the RAM at addresses 00000h through 03FFFh in external memory, and $\overline{\text{STRB}}$ establishes the $\overline{\text{CS}}$-controlled write cycle. The $\overline{\text{WE}}$ control input is then driven by the TMS320C3x R/$\overline{\text{W}}$ signal, and the $\overline{\text{OE}}$ input is not used and is therefore connected to ground.

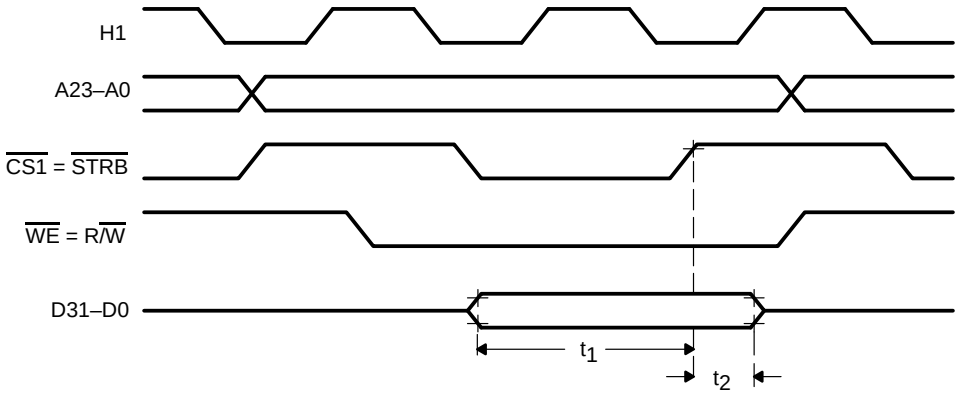
The timing of read operations, shown in Figure 12–4, is very straightforward because the two chip-select inputs are driven directly. The read access time of the circuit is therefore the inverter propagation delay added to the RAM's chip-select access time, or $t_1 + t_2 = 5 + 25 = 30$ ns. This access time therefore meets the TMS320C3x-33's specified 30-ns read access time requirement.

Figure 12–4. Read Operations Timing



During write operations, as shown in Figure 12–5, the RAM's outputs do not turn on at all, because of the use of the chip-select controlled write cycles. The chip-select controlled write cycles are generated because R/$\overline{W}$ goes active (low) before the $\overline{STRB}$ term of the chip-select input. Because the RAM's output drivers are disabled whenever the $\overline{WE}$ input is low (regardless of the state of the $\overline{OE}$ input), bus conflicts with the TMS320C3x are automatically avoided with this interface. The circuit's data setup and hold times (t_1 and t_2 in the timing diagram) of approximately 50 and 20 ns, respectively, also easily meet the RAM's timing requirements of 10 and 0 ns.

Figure 12–5. Write Operations Timing



If you require more complex chip-select decode than can be accomplished in time to meet zero-wait-state timing, you should use wait states (see subsection 12.2.2) or bank-switching techniques (see subsection 12.2.3).

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

You can easily interface RAMs without $\overline{OE}$ controls to the TMS320C3x by using an approach similar to that used with RAMs with $\overline{OE}$ controls. If only one bank of memory is implemented and no other devices are present on the bus, the memories' $\overline{CS}$ input can usually be connected to $\overline{STRB}$ directly. If several devices must be selected, however, a gate is generally required to AND the device select and $\overline{STRB}$ to drive the $\overline{CS}$ input to generate the chip-select controlled write cycles. In either case, the $\overline{WE}$ input is driven by the TMS320C3x R/$\overline{W}$ signal. Provided sufficiently fast gating is used, 25-ns RAMs can still be used.

As with the case of RAMs with $\overline{OE}$ control lines, this approach works well if only a few banks of memory are implemented where the chip-select decode can be accomplished with only one level of gating. If many banks are required to implement very large memory spaces, bank switching can be used to provide for multiple bank select generation while still maintaining full-speed accesses within each bank. Bank switching is discussed in detail in subsection 12.2.3.

12.2.2 Ready Generation

The use of wait states can greatly increase system flexibility and reduce hardware requirements over systems without wait-state capability. The TMS320C3x has the capability of generating wait states on either the primary bus or the expansion bus; both buses have independent sets of ready control logic.This subsection discusses ready generation from the perspective of the primary bus interface; however, wait-state operation on the expansion bus is similar to that on the primary bus. Therefore, these discussions also pertain to expansion bus operation. Accordingly, ready generation is not included in the specific discussions of the expansion bus interface.

Wait states are generated on the basis of:

- the internal wait-state generator,
- the external ready input ($\overline{\text{RDY}}$), or
- the logical AND or OR of the two.

When enabled, internally generated wait states affect all external cycles, regardless of the address accessed. If different numbers of wait states are required for various external devices, the external $\overline{\text{RDY}}$ input may be used to tailor wait-state generation to specific system requirements.

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

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

ORing of the Ready Signals

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

You can use the OR of the two ready signals if conditions occur that require termination of bus cycles prior to the number of wait states implemented with external logic. In this case, a shorter wait count is specified internally than the number of wait states implemented with the external ready logic, and the bus cycle is terminated after the wait count. This feature can also be a safeguard against inadvertent accesses to nonexistent memory that would never respond with ready and would therefore lock up the TMS320C3x.

If the OR of the two ready signals is used, however, and the internal wait-state count is less than the number of wait states implemented externally, the external ready generation logic must have the ability to reset its sequencing to allow a new cycle to begin immediately following the end of the internal wait count. This requires that, under these conditions, consecutive cycles be from independently decoded areas of memory and that the external ready generation logic be capable of restarting its sequence as soon as a new cycle begins. Otherwise, the external ready generation logic might lose synchronization with bus cycles and therefore generate improperly timed wait states.

ANDing of the Ready Signals

The AND of the two ready signals can be used to implement wait states for devices that are equipped to provide a ready signal but cannot respond quickly enough to meet the TMS320C3x's timing requirements. In particular, if these devices normally indicate a ready condition and, when accessed, respond with a wait until they become ready, the logical AND of the two ready signals can be used to save hardware in the system. In this case, the internal wait counter can provide wait states initially and become ready after the external device has had time to send a not ready indication. The internal wait counter then remains ready until the external device also becomes ready, which terminates the cycle.

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

External Ready Generation

In the implementation of external ready generation hardware, the particular technique employed depends heavily on the specific characteristics of the system. The optimum approach to ready generation varies, depending on the relative number of wait-state and non-wait-state devices in the system and on the maximum number of wait states required for any one device. The approaches discussed here are intended to be general enough for most applications and are easily modifiable to comprehend many different system configurations.

In general, ready generation involves the following three functions:

- Segmentating the address space in some fashion to distinguish fast and slow devices
- Generating properly timed ready indications
- Logically ORing all of the separate ready timing signals together to connect to the physical ready input

Segmentation of the address space is required to obtain a unique indication of each particular area within the address space that requires wait states. This segmentation is commonly implemented in a system in the form of chip-select generation. In many cases, you can use chip-select signals to initiate wait states; however chip-select decoding considerations might occasionally provide signals that will not allow ready input timing requirements to be met. In this case, you could make coarse address space segmentation on the basis of a small number of address lines, where simpler gating allows signals to be generated more quickly. In either case, the signal indicating that a particular area of memory is being addressed is normally used to initiate a ready or wait-state indication.

Once the region of address space being accessed has been established, a timing circuit of some sort is normally used to provide a ready indication to the processor at the appropriate point in the cycle to satisfy each device's unique requirements.

Finally, since indications of ready status from multiple devices are typically present, the signals are logically ORed by using a single gate to drive the $\overline{\text{RDY}}$ input.

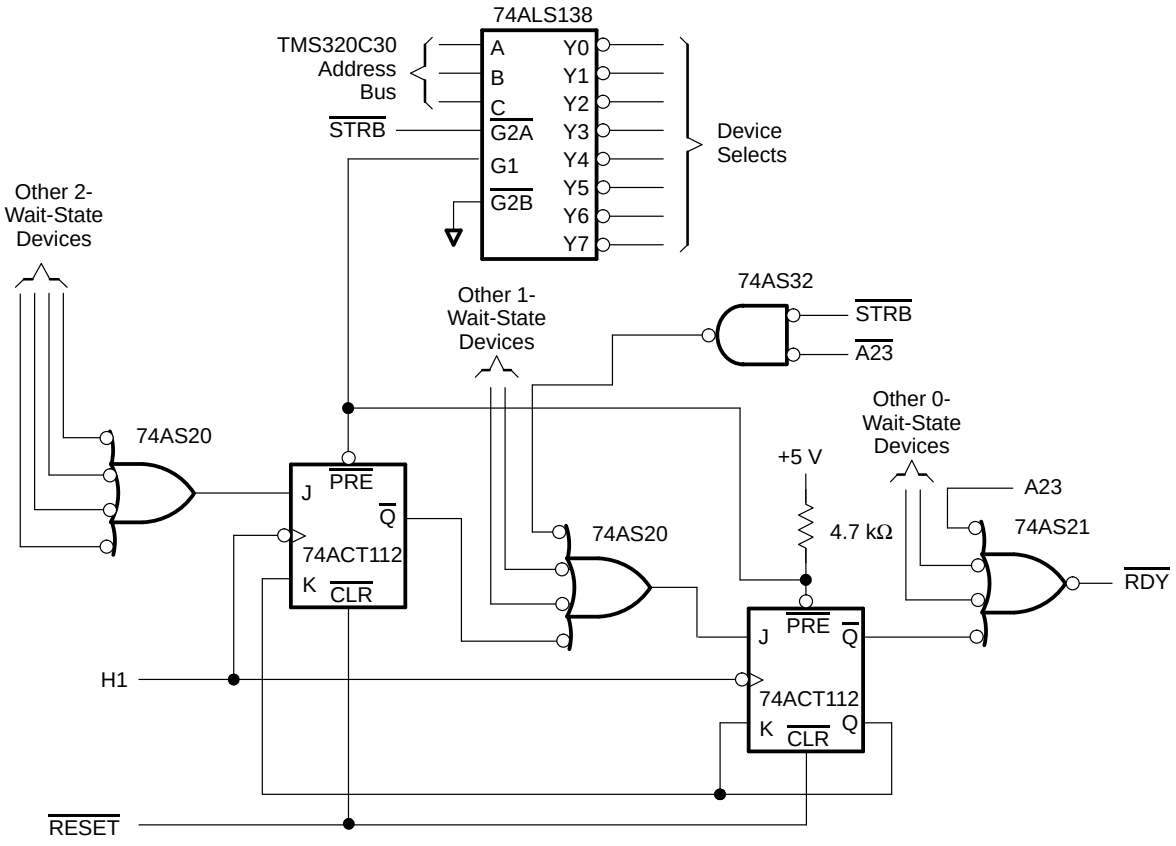
Ready Control Logic

You can take one of two basic approaches in the implementation of ready control logic, depending on the state of the ready input between accesses. If $\overline{\text{RDY}}$ is low between accesses, the processor is always ready unless a wait state is required; if $\overline{\text{RDY}}$ is high between accesses, the processor will always enter a wait state unless a ready indication is generated.

If $\overline{\text{RDY}}$ is low between accesses, control of full-speed devices is straightforward; no action is necessary because ready is always active unless otherwise required. Devices requiring wait states, however, must drive ready high fast enough to meet the input timing requirements. Then, after an appropriate delay, a ready indication must be generated. This can be quite difficult in many circumstances because wait-state devices are inherently slow and often require complex select decoding.

If $\overline{RDY}$ is high between accesses, zero-wait-state devices, which tend to be inherently fast, can usually respond immediately with a ready indication. Wait-state devices might delay their select signals appropriately to generate a ready. Typically, this approach results in the most efficient implementation of ready control logic. Figure 12–6 shows a circuit of this type, which can be used to generate zero, one, or two wait states for multiple devices in a system.

Figure 12–6. Circuit for Generation of Zero, One, or Two Wait States for Multiple Devices



Example Circuit

In this circuit, full-speed devices drive ready directly through the ’74AS21, and the two flip-flops delay wait-state devices’ select signals one or two H1 cycles to provide one or two wait states.

Considering the TMS320C3x-33’s ready delay time of eight ns following address, zero-wait-state devices must use ungated address lines directly to drive the input of the ’74AS21, since this gate contributes a maximum propagation delay of six ns to the $\overline{\text{RDY}}$ signal. Thus, zero-wait-state devices should be grouped together within a coarse segmentation of address space if other devices in the system require wait states.

With this circuit, devices requiring wait states might take up to 36 ns from a valid address on the TMS320C3x to provide inputs to the ’74AS20’s inputs. This usually allows sufficient time for any decoding required in generating select signals for slower devices in the system. For example, the 74ALS138, driven by address and $\overline{\text{STRB}}$, can generate select decodes in 22 ns, which easily meets the TMS320C3x-33’s timing requirements.

With this circuit, unused inputs to either the 74AS20s or the 74AS21 should be tied to a logic high level to prevent noise from generating spurious wait states.

If more than two wait states are required by devices within a system, other approaches can be employed for ready generation. If between three and seven wait states are required, additional flip-flops can be included in the same manner shown in Figure 12–6, or internally generated wait states can be used in conjunction with external hardware. If more than seven wait states are required, an external circuit using a counter may be used to supplement the capabilities of the internal wait-state generators.

12.2.3 Bank Switching Techniques

The TMS320C3x’s programmable bank switching feature can greatly ease system design when large amounts of memory are required. Because, in general, devices take longer to release the bus than they take to drive the bus, bank switching is used to provide a period of time for disabling all device selects that would not be present otherwise (refer to Section 7.4 on page 7-30 for further information regarding bank switching). During this interval, slow devices are allowed time to turn off before other devices have the opportunity to drive the data bus, thus avoiding bus contention.

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

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

When you use bank switching to implement large multiple-bank memory systems, an important consideration is address line fanout. Besides parametric specifications for which account must be made, AC characteristics are also crucial in memory system design. With large memory arrays, which commonly require large numbers of address line inputs to be driven in parallel, capacitive loading of address outputs is often quite large. Because all TMS320C3x timing specifications are guaranteed up to a capacitive load of 80 pF, driving greater loads will invalidate guaranteed AC characteristics. Therefore, it is often necessary to provide buffering for address lines when driving large memory arrays. AC timings for buffer performance can then be derated according to manufacturer specifications to accommodate a wide variety of memory array sizes.

The circuit shown in Figure 12–7 illustrates the use of bank switching with Cypress Semiconductor's CY7C185 25-ns 8K $\times$ 8 CMOS static RAM. This circuit implements 32K 32-bit words of memory with one-wait-state accesses within each bank.

A wait state is required with this implementation of bank memory because of the added propagation delay presented by the address bus buffers used in the circuit. The wait state is not a function of the memory organization of multiple banks or the use of bank switching. When bank switching is used, memory access speeds are the same as without bank switching, once bank boundaries are crossed. Therefore, no speed penalty is paid when bank switching is used, except for the occasional extra cycle inserted when bank boundaries are crossed. Note, however, that if the extra cycle inserted when bank boundaries are crossed does impact software performance significantly, you can often restructure code to minimize bank boundary crossings, thereby reducing the effect of these boundary crossings on software performance.

The wait state for this bank memory is generated by using the wait-state generator circuit presented in the previous section. Because A23 is the signal that enables the entire bank memory system, the inverted version of this signal is ANDed with $\overline{\text{STRB}}$ to derive a one-wait-state device select. This signal is then connected in the circuit along with the other one-wait-state device selects. Thus, any time a bank memory access is made, one wait state is generated.

Each of the four banks in this circuit is selected by using a decode of A15–A13 generated by the 74AS138 (see Figure 12–8). With the BNKCMPR register set to 0Bh, the banks will be selected on even 8K-word boundaries starting at location 080A000h in external memory space.

Figure 12–7. Bank Switching for Cypress Semiconductor's CY7C185

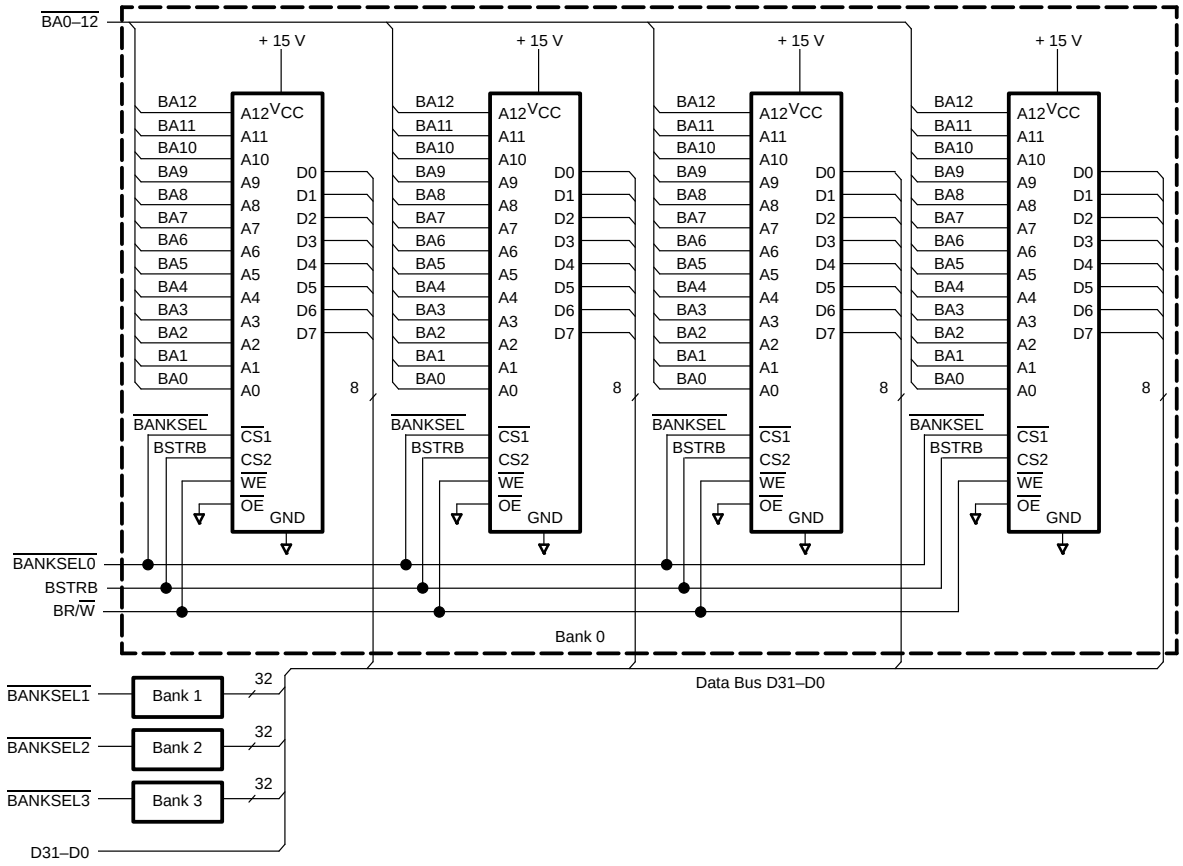
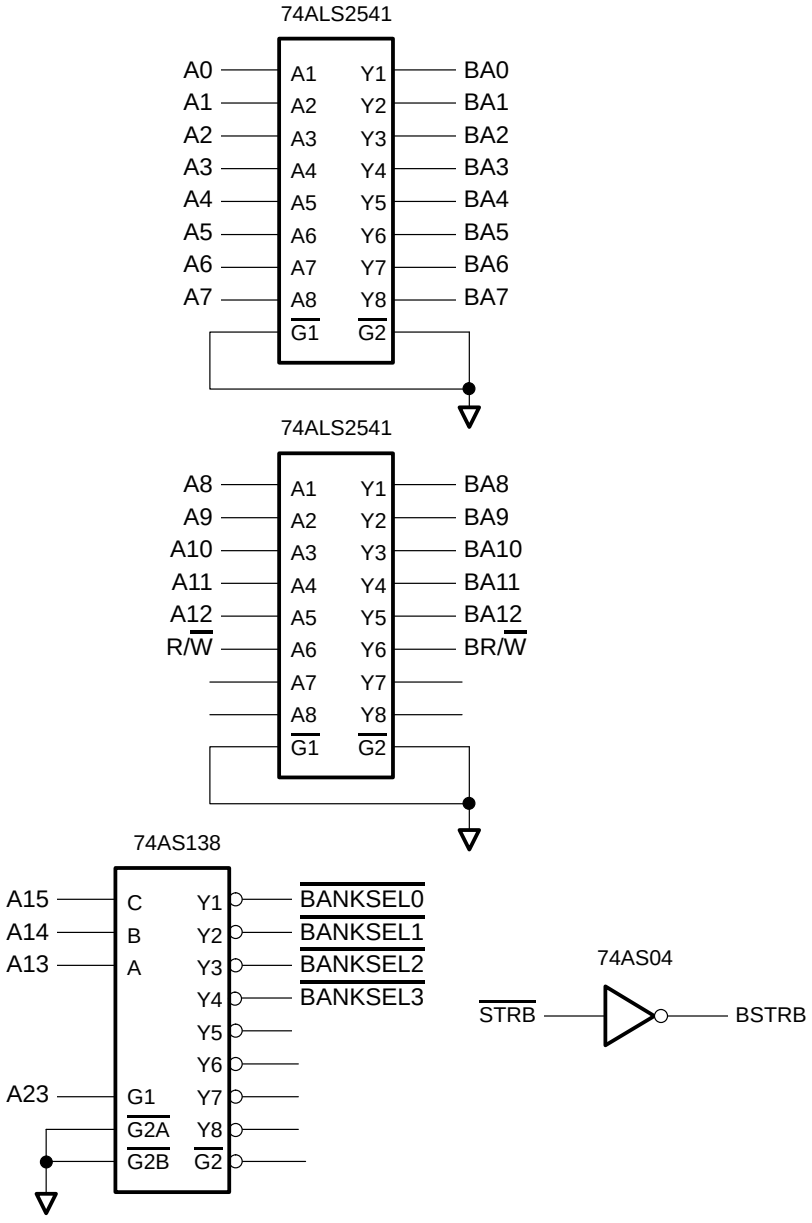


Figure 12–8. Bank Memory Control Logic



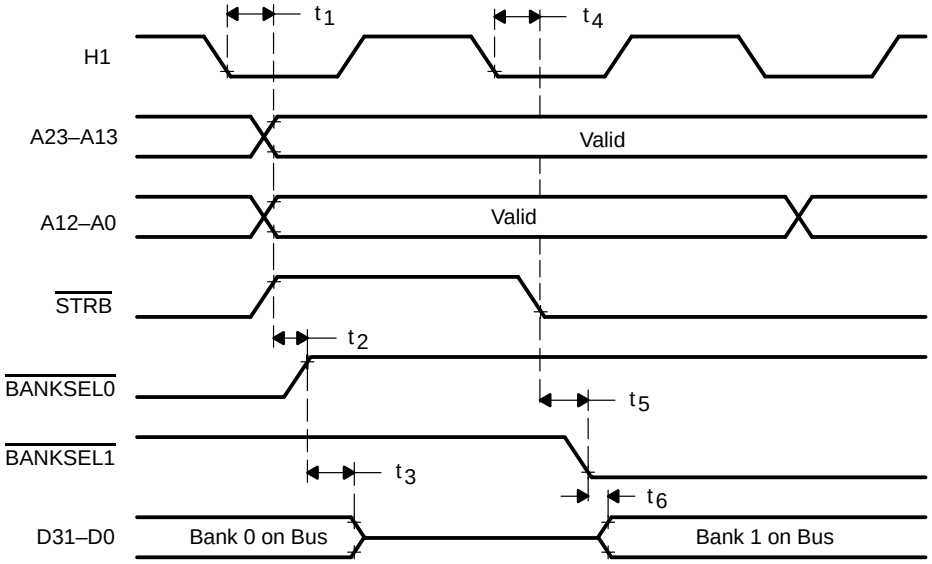
The 74ALS2541 buffers used on the address lines are necessary in this design because the total capacitive load presented to each address line is a maximum of 16 × 10 pF or 160 pF (bank memory plus zero-wait-state static RAM), which exceeds the TMS320C3x rated capacitive loading of 80 pF. Using the manufacturer's derating curves for these devices at a load of 80 pF (the load presented by the bank memory) predicts propagation delays at the output of the buffers of a maximum of 16 ns. The access time of a read cycle within a bank of the memory is therefore the sum of the memory access time and the maximum buffer propagation delay, or 25 + 16 = 41 ns, which, since it falls between 30 and 90 ns, requires one wait state on the TMS320C3x-33.

The 74ALS2541 buffers offer one additional system-performance enhancement in that they include 25-ohm resistors in series with each individual buffer output. These resistors greatly improve the transient response characteristics of the buffers, especially when driving CMOS loads such as the memories used here. The effect of these resistors is to reduce overshoot and ringing, which is common when driving predominantly capacitive loads such as CMOS. The result is reduced noise and increased immunity to latch-up in the circuit, which in turn results in a more reliable memory system. Having these resistors included in the buffers eliminates the need to put discrete resistors in the system, which is often required in high-speed memory systems.

This circuit cannot be implemented without bank switching because data output's turn-on and turn-off delays cause bus conflicts. Here, the propagation delay of the 74AS138 is involved only during bank switches, when there is sufficient time between cycles to allow new chip selects to be decoded.

The timing of this circuit for read operations using bank switching is shown in Figure 12–9. With the BNKCMPR register set to 0Bh, when a bank switch occurs, the bank address on address lines A23–A13 is updated during the extra H1 cycle while $\overline{\text{STRB}}$ is high. Then, after chip-select decodes have stabilized and the previously selected bank has disabled its outputs, $\overline{\text{STRB}}$ goes low for the next read cycle. Further accesses occur at normal bus timings with one wait state, as long as another bank switch is not necessary. Write cycles do not require bank switching due to the inherent address setup provided in their timings.

Figure 12–9. Timing for Read Operations Using Bank Switching



This timing is summarized in Table 12–1.

Table 12–1. Bank Switching Interface Timing

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

† Timing for the TMS320C3x-33

12.3 Expansion Bus Interface

The TMS320C30's expansion bus interface provides a second complete parallel bus, which can be used to implement data transfers concurrently with (and independently of) operations on the primary bus. The expansion bus comprises two mutually exclusive interfaces controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals, respectively. This subsection discusses interface to the expansion bus using $\overline{\text{IOSTRB}}$ cycles; $\overline{\text{MSTRB}}$ cycles are essentially equivalent in timing to primary bus cycles and are discussed in Section 12.2, beginning on page 12-4. This section applies to TMS320C30 devices.

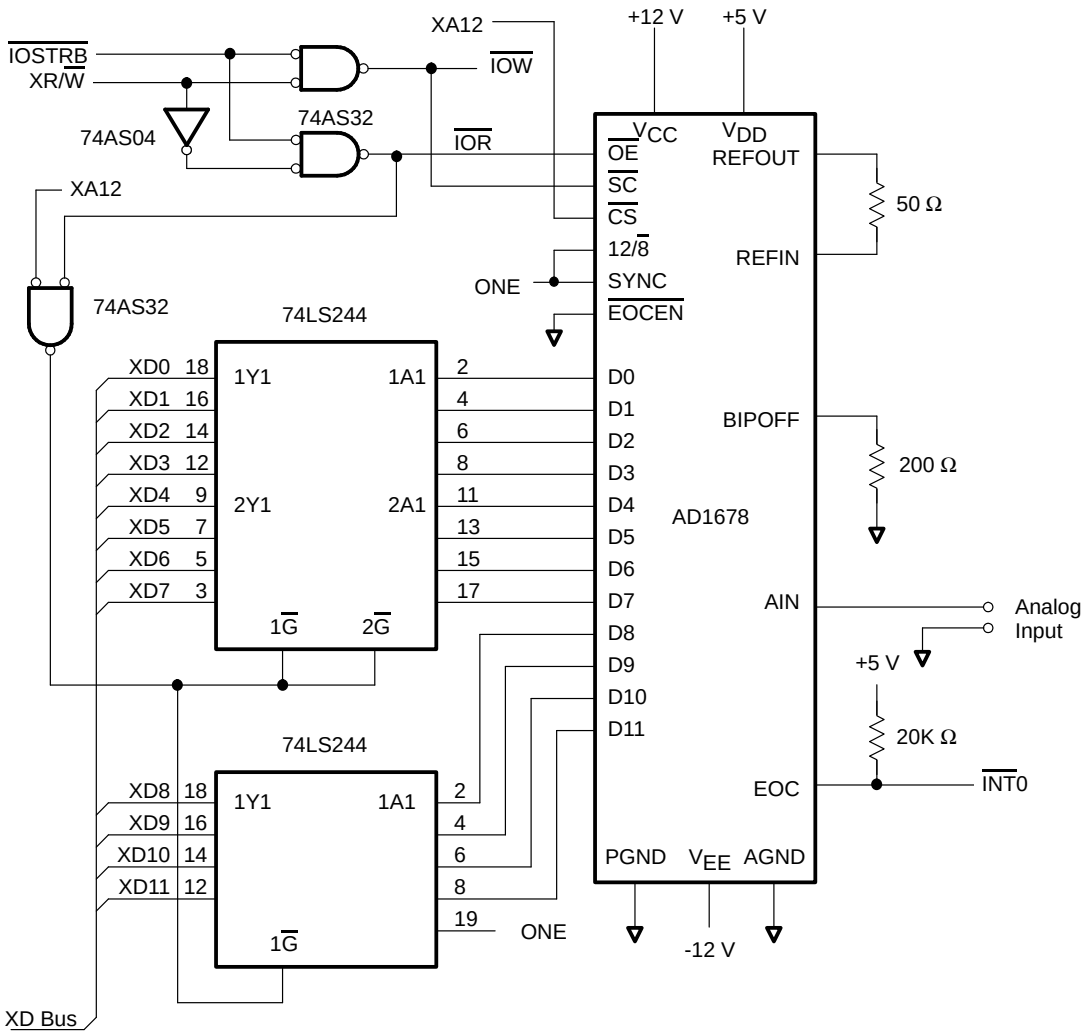
Unlike the primary bus, both read and write cycles on the I/O portion of the expansion bus are two H1 cycles in duration and exhibit the same timing. The XR/$\overline{\text{W}}$ signal is high for reads and low for writes. Since I/O accesses take two cycles, many peripherals that require wait states if interfaced either to the primary bus or by using $\overline{\text{MSTRB}}$ can be used in a system without the need for wait states. Specifically, in cases where there is only one device on the expansion bus, devices with address access times greater than the 30 ns required by the primary bus, but less than 59 ns, can be interfaced to the I/O bus of the TMS320C30-33 without wait states.

12.3.1 A/D Converter Interface

A/D and D/A converters are commonly required in DSP systems and interface efficiently to the I/O expansion bus. These devices are available in many speed ranges and with a variety of features. While some might require one or more wait states on the I/O bus, others can be used at full speed.

Figure 12–10 illustrates a TMS320C30 interface to an Analog Devices AD1678 analog-to-digital converter. The AD1678 is a 12-bit, 5-µs converter that allows sample rates up to 200 kHz and has an input voltage range of 10 volts, bipolar or unipolar. The converter is connected according to manufacturer's specifications to provide 0- to +10-volt operation. This interface illustrates a common approach to connecting devices such as this to the TMS320C30. Note that the interface requires only a minimum amount of control logic.

Figure 12–10. Interface to AD1678 A/D Converter



The AD1678 is a very flexible converter and is configurable in a number of different operating modes. These operating modes include byte or word data format, continuous or noncontinuous conversions, enabled or disabled chip-select function, and programmable end-of-conversion indication. This interface utilizes 12-bit word data format, rather than byte format, to be compatible with the TMS320C3x. Noncontinuous conversions are selected so that variable sample rates can be used; continuous conversions occur only at a rate of 200 kHz. With noncontinuous conversions, the host processor determines the conversion rate by initiating conversions through write operations to the converter.

The chip-select function is enabled, so the chip-select input is required to be active when accessing the device. Enabling the chip select function is necessary to allow a mechanism for the AD1678 to be isolated from other peripheral devices connected to the expansion bus. To establish the desired operating modes, the SYNC and 12/$\overline{8}$ inputs to the converter are pulled high and $\overline{\text{EOCEN}}$ is grounded, as specified in the AD1678 data sheet.

In this application, the converter's chip select is driven by XA12, which maps this device at 804000h in I/O address space. Conversions are initiated by writing any data value to the device, and the conversion results are obtained by reading from the device after the conversion is completed. To generate the device's start conversion (SC) and output enable ($\overline{\text{OE}}$) inputs, $\overline{\text{IOSTRB}}$ is ANDed with XR/$\overline{\text{W}}$. Therefore, the converter is selected whenever XA12 is low; $\overline{\text{OE}}$ is driven when reads are performed, while SC is driven when writes are performed.

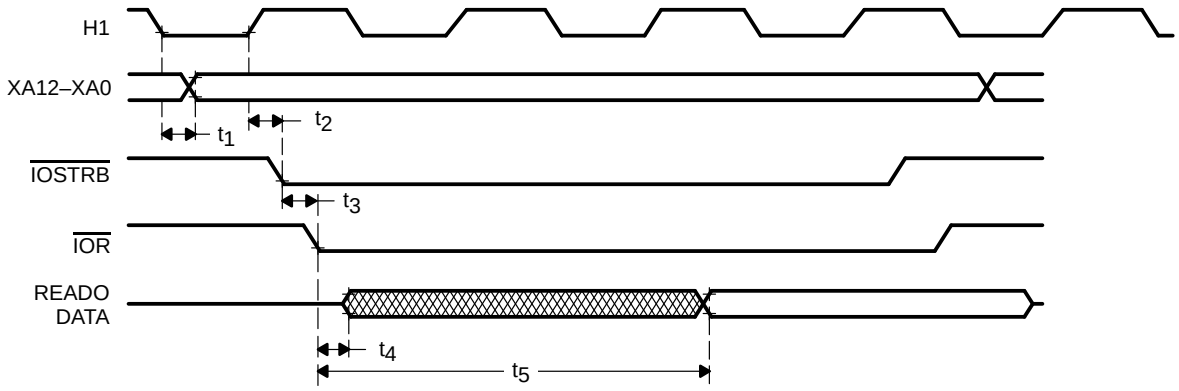
As with many A/D converters, at the end of a read cycle the AD1678 data output lines enter a high-impedance state. This occurs after the output enable ($\overline{\text{OE}}$) or read control line goes inactive. Also common with these types of devices is that the data output buffers often require a substantial amount of time to actually attain a full high-impedance state. When used with the TMS320C30-33, devices must have their outputs fully disabled no later than 65 ns following the rising edge of $\overline{\text{IOSTRB}}$ because the TMS320C30 will begin driving the data bus at this point if the next cycle is a write. If this timing is not met, bus conflicts between the TMS320C30 and the AD1678 might occur, potentially causing degraded system performance and even failure due to damaged data bus drivers. The actual disable time for the AD1678 can be as long as 80 ns; therefore, buffers are required to isolate the converter outputs from the TMS320C30. The buffers used here are 74LS244s that are enabled when the AD1678 is read and turned off 30.8 ns following $\overline{\text{IOSTRB}}$ going high. Therefore, the TMS320C30-33 requirement of 65 ns is met.

When data is read following a conversion, the AD1678 takes 100 ns after its $\overline{\text{OE}}$ control line is asserted to provide valid data at its outputs. Thus, including the propagation delay of the 74LS244 buffers, the total access time for reading the converter is 118 ns. This requires two wait states on the TMS320C30-33 expansion I/O bus.

The two wait states required in this case are implemented using software wait states; however, depending on the overall system configuration, it might be necessary to implement a separate wait-state generator for the expansion bus (refer to subsection 12.2.2 on page 12-9). This would be the case if multiple devices that required different numbers of wait states were connected to the expansion bus.

Figure 12–11 shows the timing for read operations between the TMS320C30-33 and the AD1678. At the beginning of the cycle, the address and XR/$\overline{W}$ lines become valid t_1 = 10 ns following the falling edge of H_1. Then, after t_2 = 10 ns from the next rising edge of H_1, $\overline{IOSTRB}$ goes low, beginning the active portion of the read cycle. After t_3 = 5.8 ns (the control logic propagation delay), the $\overline{IOR}$ signal goes low, asserting the $\overline{OE}$ input to the AD1678. The '74LS244 buffers take t_4 = 30 ns to enable their outputs, and then, following the converters access delay and the buffer propagation delay (t_5 = 100 + 18 = 118 ns), data is provided to the TMS320C30. This provides approximately 46 ns of data setup before the rising edge of $\overline{IOSTRB}$. Therefore, this design easily satisfies the TMS320C30-33's requirement of 15 ns of data setup time for reads.

Figure 12–11. Read Operations Timing Between the TMS320C30 and AD1678



Unlike the primary bus, read and write cycles on the I/O expansion bus are timed the same with the exception that XR/$\overline{W}$ is high for reads and low for writes and that the data bus is driven by the TMS320C30 during writes. When writing to the AD1678, the '74LS244 buffers do not turn on and no data is transferred. The purpose of writing to the converter is only to generate a pulse on the converter's $\overline{SC}$ input, which initiates a conversion cycle. When a conversion cycle is completed, the AD1678's EOC output is used to generate an interrupt on the TMS320C30 to indicate that the converted data can be read.

It should be noted that for different applications, use of TLC1225 or TLC1550 A/D converters from Texas Instruments can be beneficial. The TLC1225 is a self-calibrating 12-bit-plus-sign bipolar or unipolar converter, which features 10-µs conversion times. The TLC1550 is a 10-bit, 6-µs converter with a high-speed DSP interface. Both converters are parallel-interface devices.

12.3.2 D/A Converter Interface

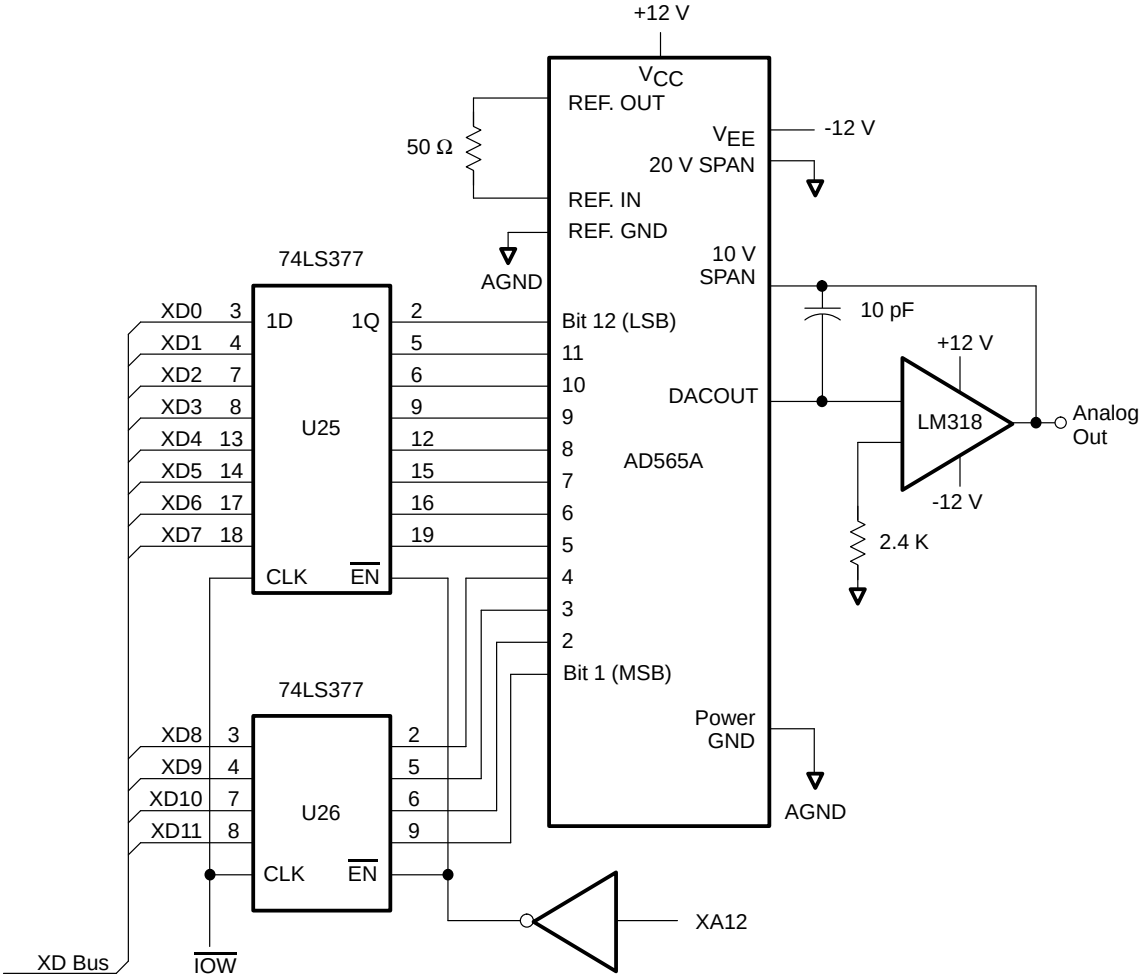
In many DSP systems, the requirement for generating an analog output signal is a natural consequence of sampling an analog waveform with an A/D converter and then processing the signal digitally internally. Interfacing D/A converters to the TMS320C30 on the expansion I/O bus is also quite straightforward.

As with A/D converters, D/A converters are also available in a number of varieties. One of the major distinctions between various types of D/A converters is whether or not the converter includes both latches to store the digital value to be converted to an analog quantity, and the interface to control those latches. With latches and control logic included with the converter, interface design is often simplified; however, internal latches are often included only in slower D/A converters.

Because slower converters limit signal bandwidths, the converter used in this design was selected to allow a reasonably wide range of signal frequencies to be processed, and to illustrate the technique of interfacing to a converter that uses external data latches.

Figure 12–12 shows an interface to an Analog Devices AD565A digital-to-analog converter. This device is a 12-bit, 250-ns current output DAC with an on-chip 10-volt reference. Using an offchip current-to-voltage conversion circuit connected according to manufacturers specifications, the converter exhibits output signal ranges of 0 to +10 volts, which is compatible with the conversion range of the A/D converter discussed in the previous section.

Figure 12–12. Interface Between the TMS320C30 and the AD565A

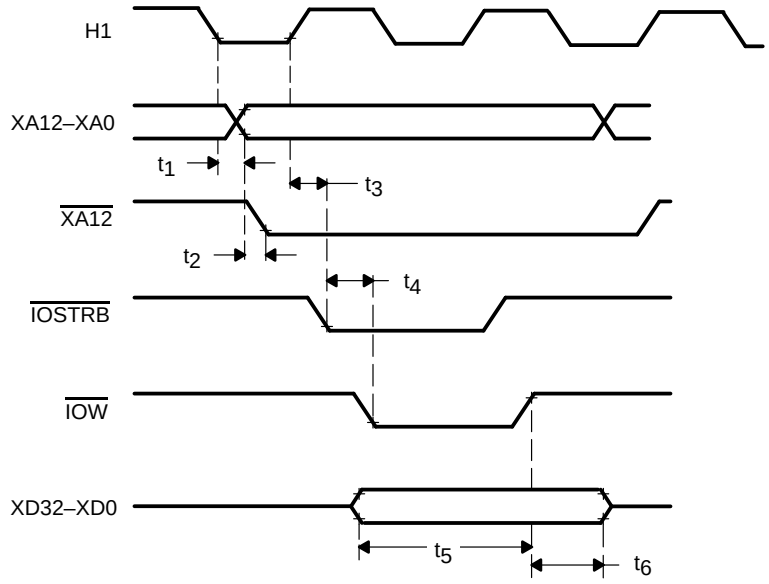


Because this DAC essentially performs continuous conversions based on the digital value provided at its inputs, periodic sampling is maintained by periodically updating the value stored in the external latches. Therefore, between sample updates, the digital value is stored and maintained at the latch outputs that provide the input to the DAC. This results in the analog output remaining stable until the next sample update is performed.

The external data latches used in this interface are '74LS377 devices that have both clock and enable inputs. These latches serve as a convenient interface with the TMS320C30; the enable inputs provide a device select function, and the clock inputs latch the data. Therefore, with the enable input driven by inverted XA12 and the clock input by $\overline{IOW}$, which is the AND of $\overline{IOSTRB}$ and XR/$\overline{W}$, data will be stored in the latches when a write is performed to I/O address 805000h. Reading this address has no effect on the circuit.

Figure 12–13 shows a timing diagram of a write operation to the D/A converter latches.

Figure 12–13. Write Operation to the D/A Converter Timing Diagram



Because the write is actually being performed to the latches, the key timings for this operation are the timing requirements for these devices. For proper operation, these latches require simply a minimal setup and hold time of data and control signals with respect to the rising edge of the clock input. Specifically, the latches require a data setup time of 20 ns, enable setup of 25 ns, disable setup of 10 ns, and data and enable hold times of 5 ns. This design provides approximately 60 ns of enable setup, 30 ns of data setup, and 7.2 ns of data hold time. Therefore, the setup and hold times provided by this design are well in excess of those required by the latches. The key timing parameters for this interface are summarized in Table 12–2.

Table 12–2. Key Timing Parameter for D/A Converter Write Operation

Time Interval	Event	Time Period†
t_1	H1 falling to address valid	10 ns
t_2	XA12 to $\overline{\text{XA12}}$ delay	5 ns
t_3	H1 rising to $\overline{\text{IOSTRB}}$ falling	10 ns
t_4	$\overline{\text{IOSTRB}}$ to $\overline{\text{IOW}}$ delay	5.8 ns
t_5	Data setup to $\overline{\text{IOW}}$	30 ns
t_6	Data hold from $\overline{\text{IOW}}$	7.2 ns

† Timing for the TMS320C30-33

12.4 System Control Functions

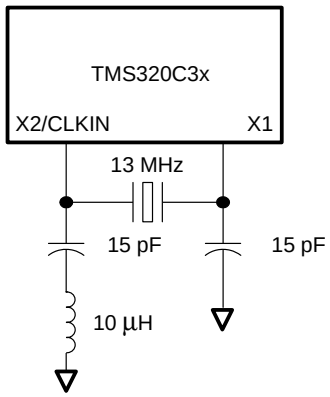
Several aspects of TMS320C3x system hardware design are critical to overall system operation. These include such functions as clock and reset signal generation and interrupt control.

12.4.1 Clock Oscillator Circuitry

You can provide an input clock to the TMS320C3x either from an external clock input or by using the onboard oscillator. Unless special clock requirements exist, the onboard oscillator is generally a convenient method for clock generation. This method requires few external components and can provide stable, reliable clock generation for the device.

Figure 12–14 shows the external clock generator circuit designed to operate the TMS320C3x at 33.33 MHz. Since crystals with fundamental oscillation frequencies of 30 MHz and above are not readily available, a parallel-resonant third-overtone crystal is used with crystal frequency of 13 MHz.

Figure 12–14. Crystal Oscillator Circuit



In a third-overtone oscillator, the crystal fundamental frequency must be attenuated so that oscillation is at the third harmonic. This is achieved with an LC circuit that filters out the fundamental, thus allowing oscillation at the third harmonic. The impedance of the LC circuit must be inductive at the crystal fundamental and capacitive at the third harmonic. The impedance of the LC circuit is represented by

$$z(\omega) = j\omega L + \frac{1}{j\omega C} \quad (3)$$

Therefore, the LC circuit has a 0 at

$$\omega_P = \frac{1}{\sqrt{LC}} \quad (4)$$

At frequencies significantly lower than ω_P, the $1/(\omega C)$ term in (3) becomes the dominating term, while ωL can be neglected. This is expressed as

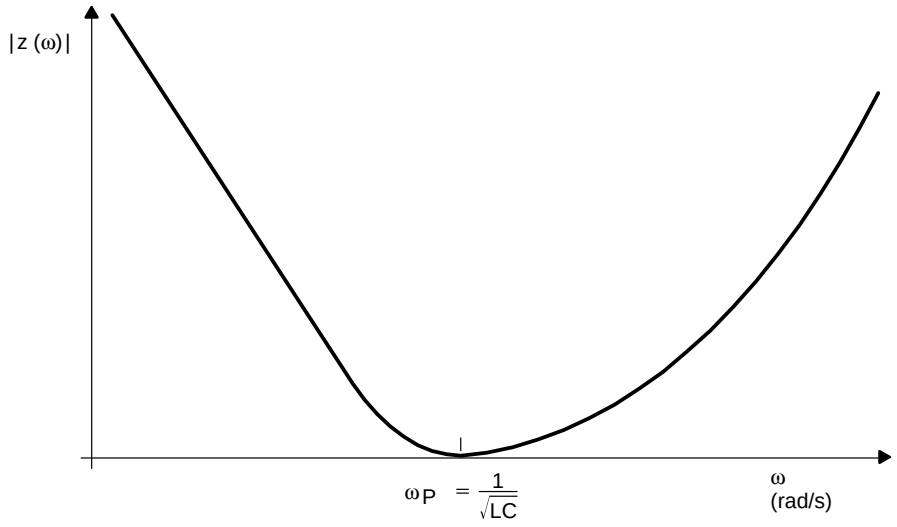
$$z(\omega) = \frac{1}{j\omega C} \qquad \text{for } \omega < \omega_P \tag{3}$$

In (5), the LC circuit appears conductive at frequencies lower than ω_P. On the other hand, at frequencies much higher than ω_P, the ωL term is the dominant term in (3), and $1/(\omega C)$ can be neglected. This is expressed as

$$z(\omega) = j\omega L \qquad \text{for } \omega < \omega_P \tag{3}$$

The LC circuit in (6) appears increasingly inductive as the frequency increases above ω_P. This is shown in Figure 12–15, which is a plot of the magnitude of the impedance of the LC circuit of Figure 12–14 versus frequency.

Figure 12–15. Magnitude of the Impedance of the Oscillator LC Network



Based on the discussion above, the design of the LC circuit proceeds as follows:

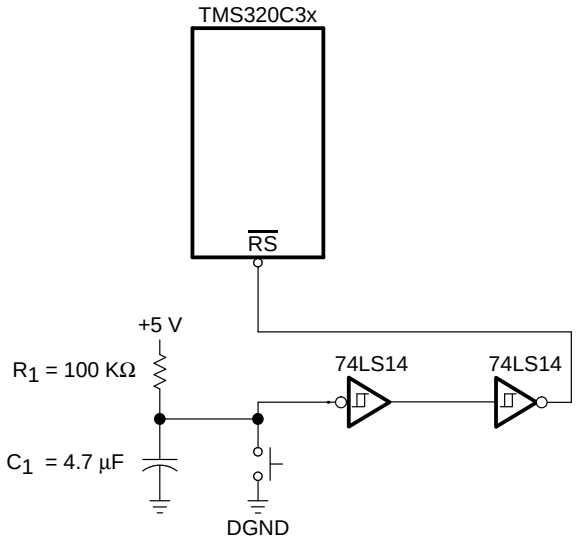
1) Choose the pole frequency ω_P slightly above the crystal fundamental.
2) The circuit now appears inductive at the fundamental frequency and capacitive at the third harmonic.

In the oscillator of Figure 12–14 on page 12-27, choose f_P = 13 MHz, which is slightly above the fundamental frequency of the crystal. Choose C = 15 pF. Then, using equation (4), L = 10 μH.

12.4.2 Reset Signal Generation

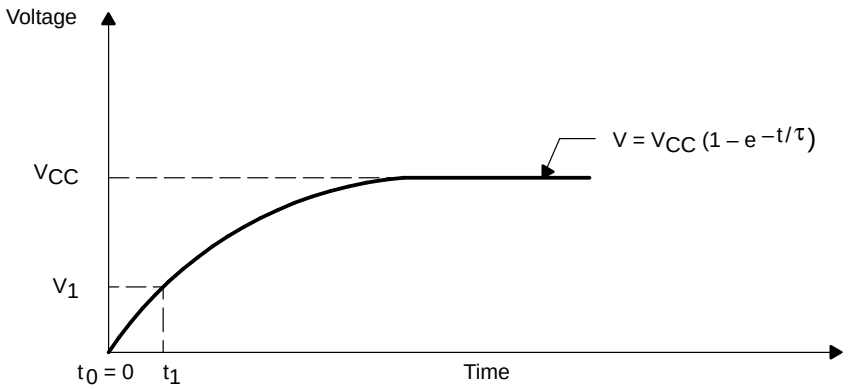
The reset input controls initialization of internal TMS320C3x logic and also causes execution of the system initialization software. For proper system initialization, the reset signal must be applied for at least ten H1 cycles, i.e., 600 ns for a TMS320C3x operating at 33.33 MHz. Upon power-up, however, it can take 20 ms or more before the system oscillator reaches a stable operating state. Therefore, the power-up reset circuit should generate a low pulse on the reset line for 100 to 200 ms. Once a proper reset pulse has been applied, the processor fetches the reset vector from location 0, which contains the address of the system initialization routine. Figure 12–16 shows a circuit that will generate an appropriate power-up reset circuit.

Figure 12–16. Reset Circuit



The voltage on the reset pin ($\overline{\text{RESET}}$) is controlled by the R_1C_1 network. After a reset, this voltage rises exponentially according to the time constant R_1C_1, as shown in Figure 12–17.

Figure 12–17. Voltage on the TMS320C30 Reset Pin



The duration of the low pulse on the reset pin is approximately t_1, which is the time it takes for the capacitor C_1 to be charged to 1.5 V. This is approximately the voltage at which the reset input switches from a logic 0 to a logic 1. The capacitor voltage is expressed as

$$V = V_{CC}\left[1 - e^{-\frac{t}{\tau}}\right] \tag{7}$$

where $\tau = R_1C_1$ is the reset circuit time constant. Solving equation (7) for t results in

$$t = -R_1C_1 \ln\left[1 - \frac{V}{V_{CC}}\right] \tag{8}$$

Setting the following:

$R_1 = 100\ K\Omega$

$C_1 = 4.7\ \mu F$

$V_{CC} = 5\ V$

$V = V_1 = 1.5\ V$

results in t = 167 ms. Therefore, the reset circuit of Figure 12–16 provides a low pulse of long enough duration to ensure the stabilization of the system oscillator.

Note that if synchronization of multiple TMS320C3xs is required, all processors should be provided with the same input clock and the same reset signal. After power-up, when the clock has stabilized, all processors can be synchronized by generating a falling edge on the common reset signal. Because it is the falling edge of reset that establishes synchronization, reset must be high for at least ten H1 cycles initially. Following the falling edge, reset should remain low for at least ten H1 cycles and then be driven high. This sequencing of reset can be accomplished using additional circuitry based on either RC time delays or counters.

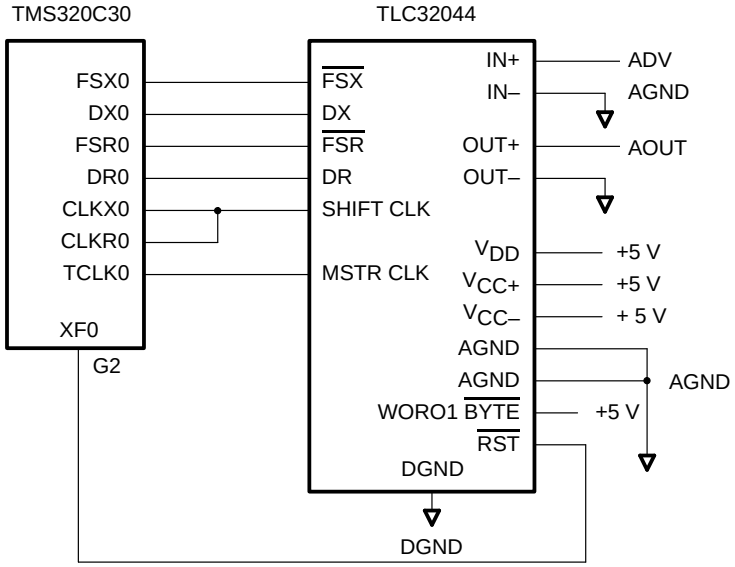
12.5 Serial-Port Interface

For applications such as modems, speech, control, instrumentation, and analog interface for DSPs, a complete analog-to-digital (A/D) and digital-to-analog (D/A) input/output system on a single chip might be appropriate. The TLC32044 analog interface circuit (AIC) integrates a bandpass, switched-capacitor, antialiasing input filter, 14-bit resolution A/D and D/A converters, and a low-pass, switched-capacitor, output-reconstruction filter, all on a single monolithic CMOS chip. The TLC32044 offers numerous combinations of master clock input frequencies and conversion/sampling rates, which can be changed via digital signal processor control.

Four serial port modes on the TLC32044 allow direct interface to TMS320C3x processors. When the transmit and receive sections of the AIC are operating synchronously, it can interface to two SN54299 or SN74299 serial-to-parallel shift registers. These shift registers can then interface in parallel to the TMS320C30, to other TMS320 digital processors, or to external FIFO circuitry. Output data pulses inform the processor that data transmission is complete or allow the DSP to differentiate between two transmitted bytes. A flexible control scheme is provided so that the functions of the AIC can be selected and adjusted coincidentally with signal processing via software control. Refer to the TLC32044 data sheet for detailed information.

When you interface the AIC to the TMS320C3x via one of the serial ports, no additional logic is required. This interface is shown in Figure 12–18. The serial data, control, and clock signals connect directly between the two devices, and the AIC's master clock input is driven from TCLK0, one of the TMS320C3x's internal timer outputs. The AIC's WORD/$\overline{\text{BYTE}}$ input is pulled high, selecting 16-bit serial port transfers to optimize serial port data transfer rate. The TMS320C3x's XF0 pin, configured as an output, is connected to the AIC's reset ($\overline{\text{RST}}$) input to allow the AIC to be reset by the TMS320C3x under program control. This allows the TMS320C3x timer and serial port to be initialized before beginning conversions on the AIC.

Figure 12–18. AIC to TMS320C30 Interface



To provide the master clock input for the AIC, the TCLK0 timer is configured to generate a clock signal with a 50% duty cycle at a frequency of f(H1)/4 or 4.167 MHz. To accomplish this, the global control register for timer 0 is set to the value 3C1h, which establishes the desired operating modes. The period register for timer 0 is set to 1, which sets the required division ratio for the H1 clock.

To properly communicate with the AIC, the TMS320C30 serial port must be configured appropriately by initializing several TMS320C30 registers and memory locations. First, reset the serial port by setting the serial port global control register to 2170300h. (The AIC should also be reset at this time. See description below of resetting the AIC via XF0.) This resets the serial port logic, configures the serial port operating modes, including data transfer lengths, and enables the serial port interrupts. This also configures another important aspect of serial port operation: polarity of serial port signals. Because active polarity of all serial port signals is programmable, it is critical to set appropriately the bits in the serial port global control register that control the polarity. In this application, all polarities are set to positive except FSX and FSR, which are driven by the AIC and are true low.

The serial port transmit and receive control registers must also be initialized for proper serial port operation. In this application, both of these registers are set to 111h, which configures all of the serial port pins in the serial port mode, rather than the general-purpose digital I/O mode.

When the operations described above are completed, interrupts are enabled, and, provided that the serial port interrupt vector(s) are properly loaded, serial port transfers can begin after the serial port is taken out of reset. You can do this by loading E170300h into the serial port global control register.

To begin conversion operations on the AIC and subsequent transfers of data on the serial port, first reset the AIC by setting XF0 to 0 at the beginning of the TMS320C3x initialization routine. Set XF0 to 0 by setting the TMS320C3x IOF register to 2. This sets the AIC to a default configuration and halts serial port transfers and conversion operations until reset is set high. Once the TMS320C3x serial port and timer have been initialized as described above, set XF0 high by setting the IOF register to 6. This allows the AIC to begin operating in its default configuration, which in this application is the desired mode. In this mode, all internal filtering is enabled, sample rate is set at approximately 6.4 kHz, and the transmit and receive sections of the device are configured to operate synchronously. This mode of operation is appropriate for a variety of applications; if a 5.184-MHz master clock input is used, the default configuration results in an 8-kHz sample rate, which makes this device ideal for speech and telecommunications applications.

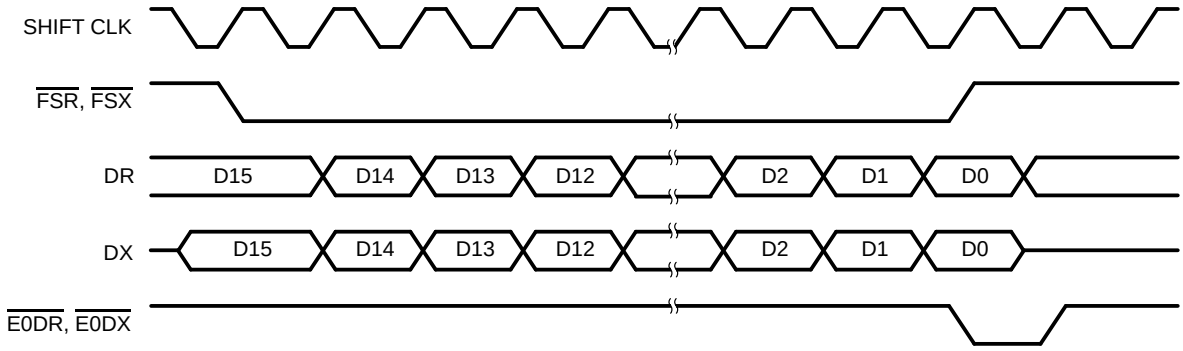
In addition to the benefit of a convenient default operating configuration, the AIC can also be programmed for a wide variety of other operating configurations. Sample rates and filter characteristics can be varied, and numerous connections in the device can be configured to establish different internal architectures by enabling or disabling various functional blocks.

To configure the AIC in a fashion different from the default state, you must first send the device a serial data word with the two LSBs set to 1. The two LSBs of a transmitted data word are not part of the transferred data information and are not set to 1 during normal operation. This condition indicates that the next serial transmission will contain secondary control information, not data. This information is then used to load various internal registers and specify internal configuration options. Four different types of secondary control words are distinguished by the state of the two LSBs of the transferred control information. Note that each transferred secondary control word must be preceded by a data word with the two LSBs set to 1.

The TMS320C3x can communicate with the AIC either synchronously or asynchronously, depending on the information in the control register. The operating sequence for synchronous communication with the TMS320C30 shown in Figure 12–19 is as follows:

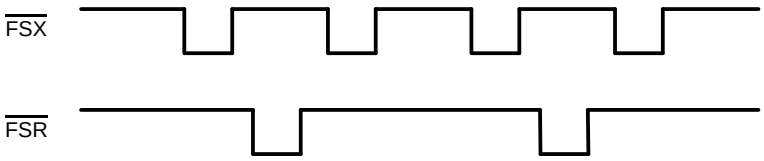
1) The $\overline{\text{FSX}}$ or $\overline{\text{FSR}}$ pin is brought low.
2) One 16-bit word is transmitted, or one 16-bit word is received.
3) The $\overline{\text{FSX}}$ or $\overline{\text{FSR}}$ pin is brought high.
4) The $\overline{\text{E0DX}}$ or $\overline{\text{E0DR}}$ pin emits a low-going pulse.

Figure 12–19. Synchronous Timing of TLC32044 to TMS320C3x



For asynchronous communication, the operating sequence is similar, but $\overline{\text{FSX}}$ and $\overline{\text{FSR}}$ do not occur at the same time (see Figure 12–20). After each receive and transmit operation, the TMS320C30 asserts an internal receive (RINT) and transmit (XINT) interrupt, which can be used to control program execution.

Figure 12–20. Asynchronous Timing of TLC32044 to TMS320C30



12.6 Low-Power-Mode Interrupt Interface

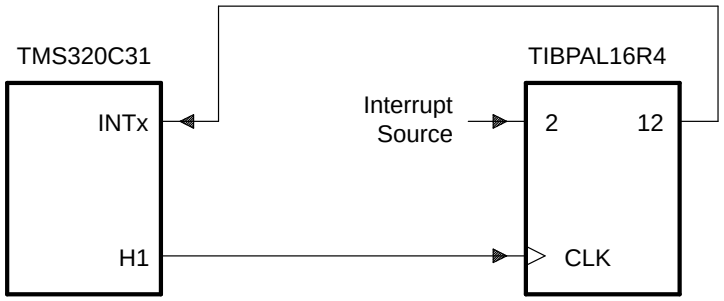
This section explains how to generate interrupts when the IDLE2 power-down mode is used.

The execution of the IDLE2 instruction causes the H1 and H3 processor clocks to be held at a constant level until the occurrence of an external interrupt. To use the TMS320C31 IDEL2 power management feature effectively, interrupts must be generated with or without the presence of the H1 clock. For normal (non-IDLE2) operation, however, the interrupt inputs must be synchronized with the falling edge of the H1 clock. An interrupt must satisfy the following conditions:

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

For an interrupt to be recognized during IDLE2 operation and turn the clocks back on, it must first be held low for one H1 cycle. The logic in Figure 12–21 can be used to generate an interrupt signal to the TMS320C31 with the correct timing during non-IDLE2 and IDLE2 operation. Figure 12–21 shows the interrupt circuit, which uses a 16R4 PLD to generate the appropriate interrupt signal.

Figure 12–21. Interrupt Generation Circuit for Use With IDLE2 Operation



Example 12–1 shows the PLD equations for the 16R4 using the ABEL™ language. This implementation makes the following assumptions regarding the interrupt source:

- The interrupt source is at least one H1 cycle in duration. One H1 cycle is required to turn the H1 clock on again.
- The interrupt source is a low-going pulse or a falling edge. If the interrupt source stays active for more than one H1 cycle, it is regarded as the same interrupt request and not a new one.

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

Example 12–1. State Machine and Equations for the Interrupt Generation 16R4 PLD

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

TMS320C31'

c3xu5    device       'P16R4';

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

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

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

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

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



"Name Substitutions for Test Vectors and Equations


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

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

"state bits
outstate = [samesrc,sync];

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

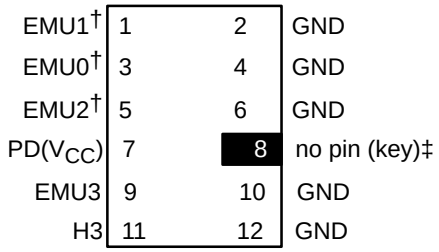
```
state_diagram outstate

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


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


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



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

@page

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

12.7 XDS Target Design Considerations

12.7.1 Designing Your MPSD Emulator Connector (12-Pin Header)

The ’C3x uses a modular port scan device (MPSD) technology to allow complete emulation via a serial scan path of the ’C3x. To communicate with the emulator, *your target system must have a 12-pin header* (2 rows of 6 pins) with the connections that are shown in Figure 12–22.To use the target cable, supply the signals shown in Table 12–3 to a 12-pin header with pin 8 cut out to provide keying. For the latest information, refer to the *JTAG/MPSD Emulation Technical Reference* (literature number SPDU079).

Figure 12–22. 12-Pin Header Signals and Header Dimensions



Header Dimensions:
Pin-to-pin spacing, 0.100 in. (X,Y)
Pin width: 0.025-in. square post
Pin length: 0.235-in. nominal

† These signals should always be pulled up with separate 20-kΩ resistors to V_{CC}.
‡ While the corresponding female position on the cable connector is plugged to prevent improper connection, the cable lead for pin 8 is present in the cable and is grounded as shown in the schematics and wiring diagrams in this document.

Table 12–3.12-Pin Header Signal Descriptions and Pin Numbers

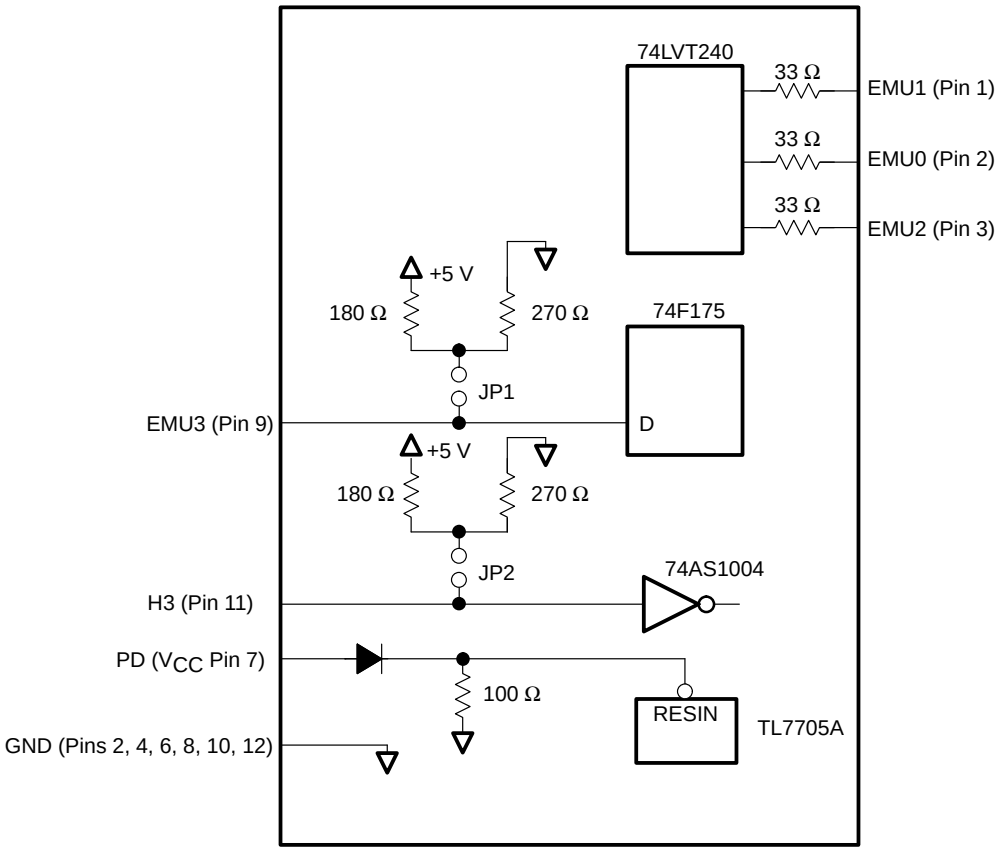
XDS510 Signal	Description	’C30 Pin Number	’C31 Pin Number
EMU0	Emulation pin 0	F14	124
EMU1	Emulation pin 1	E15	125
EMU2	Emulation pin 2	F13	126
EMU3	Emulation pin 3	E14	123
H3	’C3x H3	A1	82
PD	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to V_{CC} in the target system.		

Although you can use other headers, recommended parts include:

straight header, unshrouded — DuPont Connector Systems
part numbers: 65610–112
65611–112
37996–112
67997–112

Figure 12–23 shows a portion of logic in the emulator pod. Note that 33-Ω resistors have been added to the EMU0, EMU1, and EMU2 lines; this minimizes cable reflections.

Figure 12–23. Emulator Cable Pod Interface



12.7.2 MPSD Emulator Cable Signal Timing

Figure 12–24 shows the signal timings for the emulator pod. Table 12–4 defines the timing parameters. The timing parameters are calculated from values specified in the standard data sheets for the emulator and cable pod and are for reference only. Texas Instruments does not test or guarantee these timings.

Figure 12–24. Emulator Cable Pod Timings

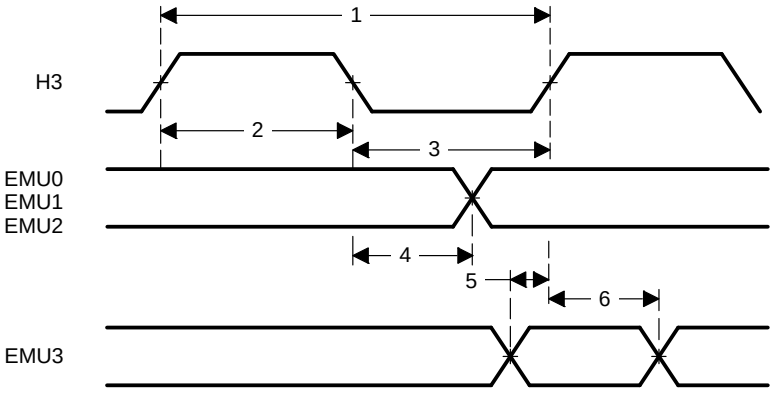


Table 12–4. Emulator Cable Pod Timing Parameters

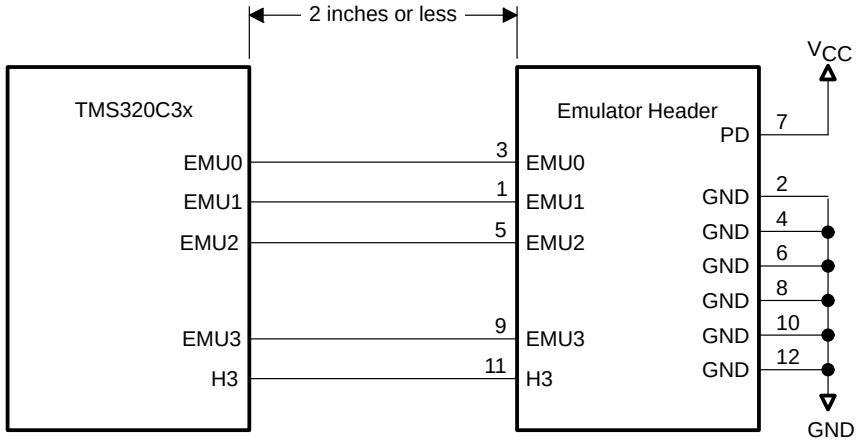
No.	Reference	Description	Min	Max	Unit
1	$t_{H3\ min}$ $t_{H3\ max}$	H3 period	35	200	ns
2	$t_{H3\ high\ min}$	H3 high pulse duration	15		ns
3	$t_{H3\ low\ min}$	H3 low pulse duration	15		ns
4	t_d (EMU0, 1, 2)	EMU0, 1, 2 valid from H3 low	7	23	ns
5	t_{su} (EMU3)	EMU3 setup time to H3 high	3		ns
6	t_{hd} (EMU3)	EMU3 hold time from H3 high	11		ns

12.7.3 Connections Between the Emulator and the Target System

It is extremely important to provide high-quality signals between the emulator and the ’C3x on the target system. In many cases, the signal must be buffered to produce high quality. The need for signal buffering can be divided into three categories, depending on the placement of the emulation header:

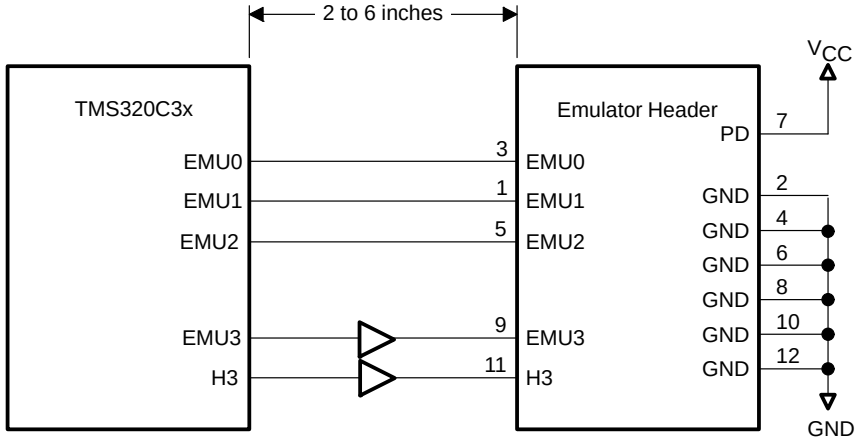
- **No signals buffered.** In this situation, the distance between the emulation header and the ’C3x should be no more than two inches. (See Figure 12–25.)

Figure 12–25. Signals Between the Emulator and the ’C3x With No Signals Buffered



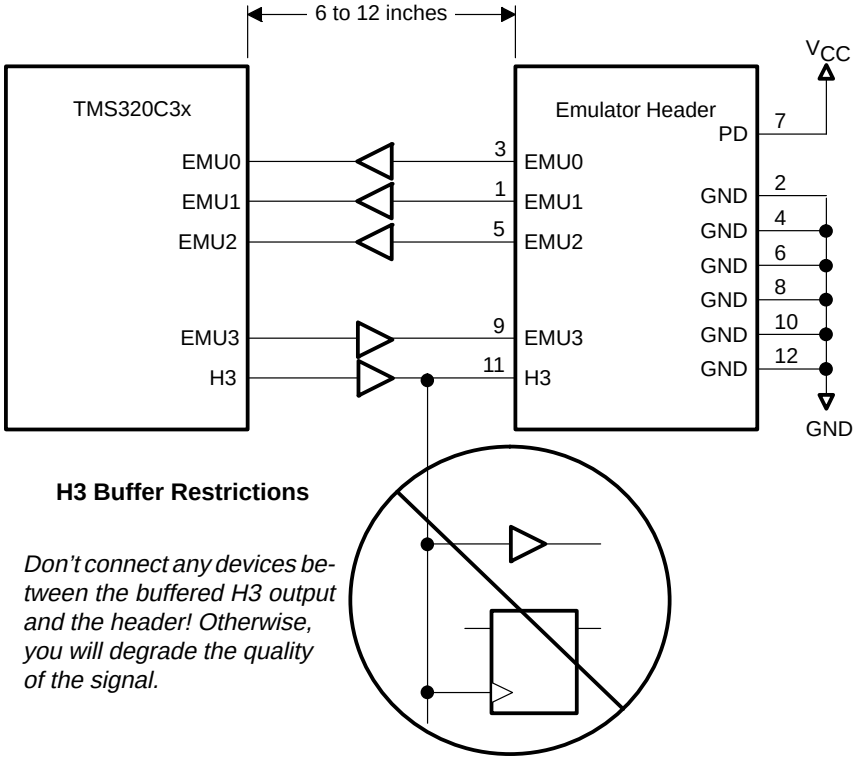
- **Transmission signals buffered.** In this situation, the distance between the emulation header and the ’C3x is greater than two inches but less than six inches. The transmission signals, H3 and EMU3, are buffered through the same package. (See Figure 12–26.)

Figure 12–26. Signals Between the Emulator and the ’C3x With Transmission Signals Buffered



- **All signals buffered.** The distance between the emulation header and the ’C3x is greater than 6 inches but less than 12 inches. All ’C3x emulation signals, EMU0, EMU1, EMU2, EMU3, and H3, are buffered through the same package. (See Figure 12–27.)

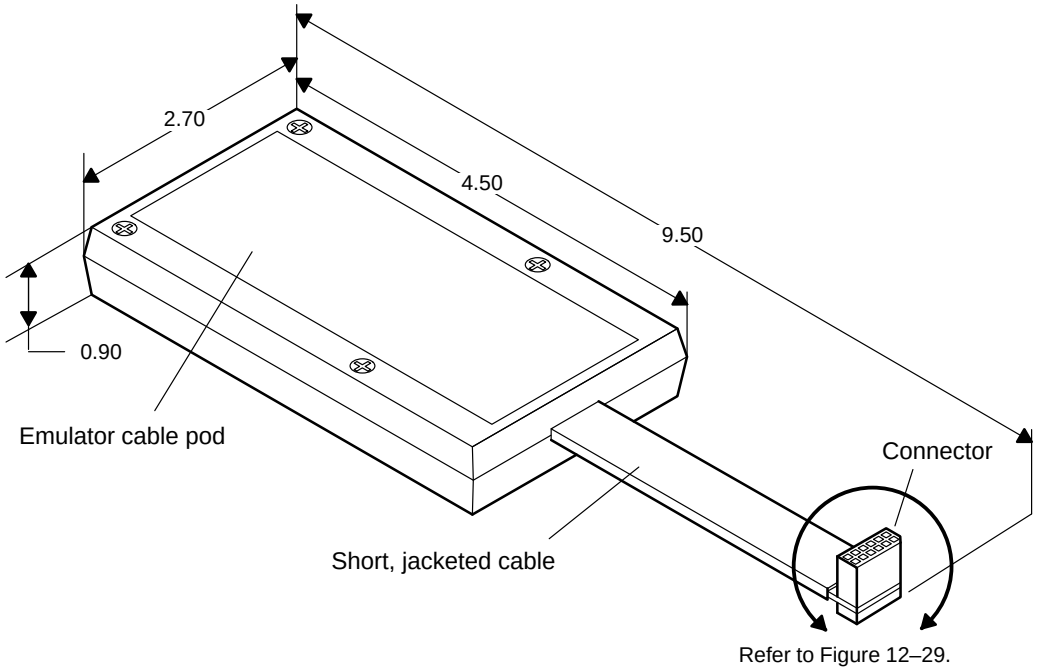
Figure 12–27. All Signals Buffered



12.7.4 Mechanical Dimensions for the 12-Pin Emulator Connector

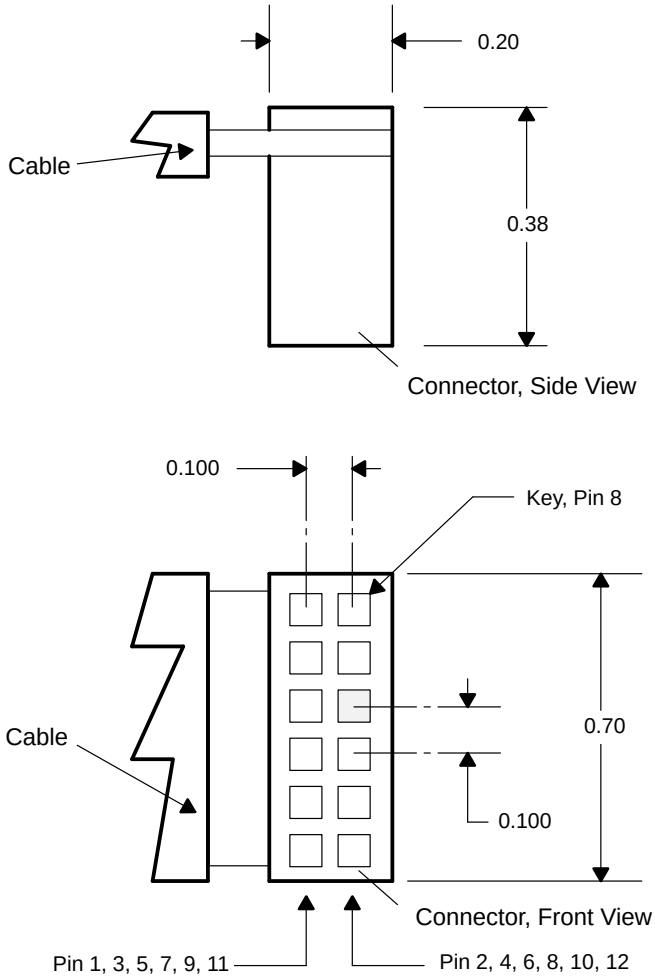
The ’C3x emulator target cable consists of a three-foot section of jacketed cable, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately three feet, ten inches. Figure 12–28 and Figure 12–29 show the mechanical dimensions for the target cable pod and short cable. Note that the pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes. The cable pod box is nonconductive plastic with four recessed metal screws.

Figure 12–28. Pod/Connector Dimensions



Note: All dimensions are in inches and are nominal unless otherwise specified.

Figure 12–29. 12-Pin Connector Dimensions

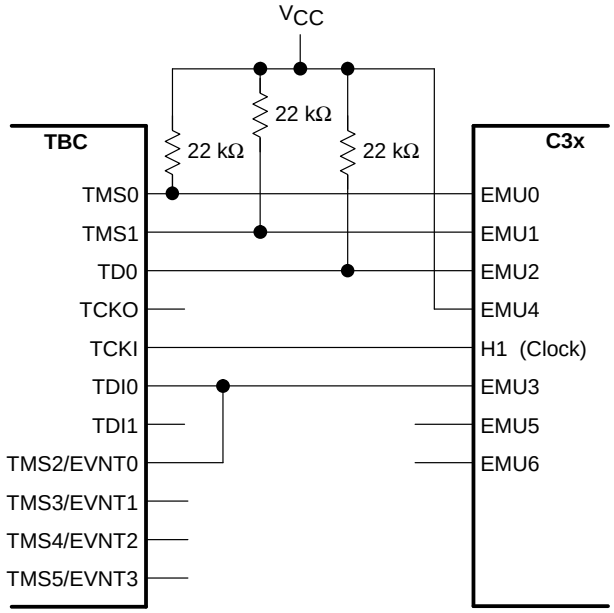


Note: All dimensions are in inches and are nominal unless otherwise specified.

12.7.5 Diagnostic Applications

For system diagnostics applications, or to embed emulation compatibility on your target system, you can connect a ’C3x device directly to a TI ACT8990 test bus controller (TBC) as shown in Figure 12–30. The TBC is described in the Texas Instruments *Advanced Logic and Bus Interface Logic Data Book* (literature number SCYD001). A TBC can connect to only one ’C3x device.

Figure 12–30. TBC Emulation Connections for ’C3x Scan Paths



Notes: 1) In a ’C3x design, the TBC can connect to only one ’C3x device.

2) The ’C3x device’s H1 clock drives TCKI on the TBC. This is different from the emulation header connections where H3 is used.

Chapter 13

TMS320C3x Signal Descriptions and Electrical Characteristics

This chapter covers the TMS320C3x pinouts, signal descriptions, and electrical characteristics.

Major topics discussed in this chapter are as follows:

Topic **Page**

13.1 Pinout and Pin Assignments

13.1.1 TMS320C30 Pinouts and Pin Assignments

The TMS320C30 digital signal processor is available in a 181-pin grid array (PGA) package. Figure 13–1 and Figure 13–2 show the pinout for this package. Figure 13–3 shows the mechanical layout. Table 13–1 shows the associated pin assignments alphabetically; Table 13–2 shows the pin assignments numerically.

Figure 13–1. TMS320C30 Pinout (Top View)

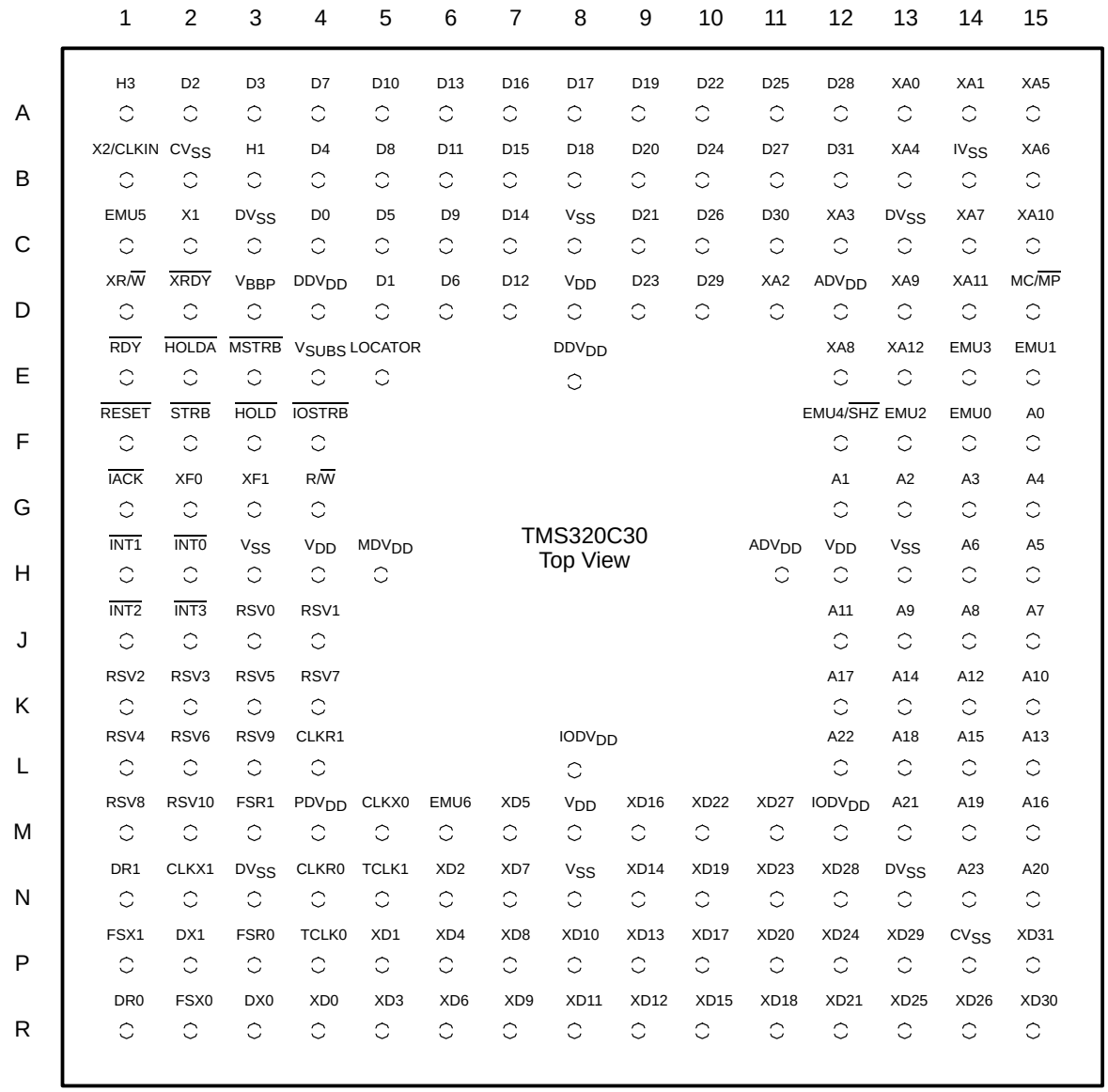


Figure 13–2. TMS320C30 Pinout (Bottom View)

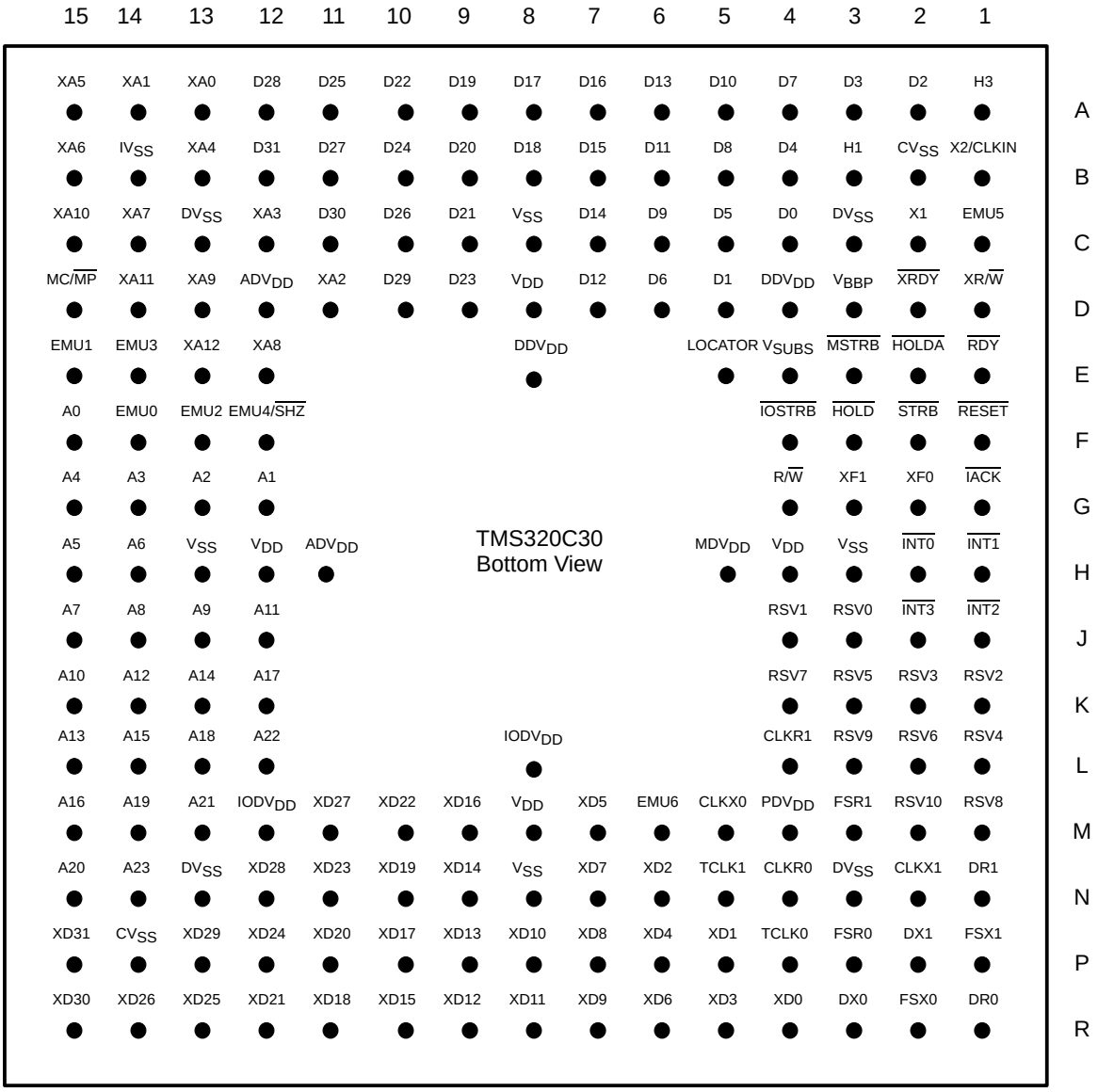
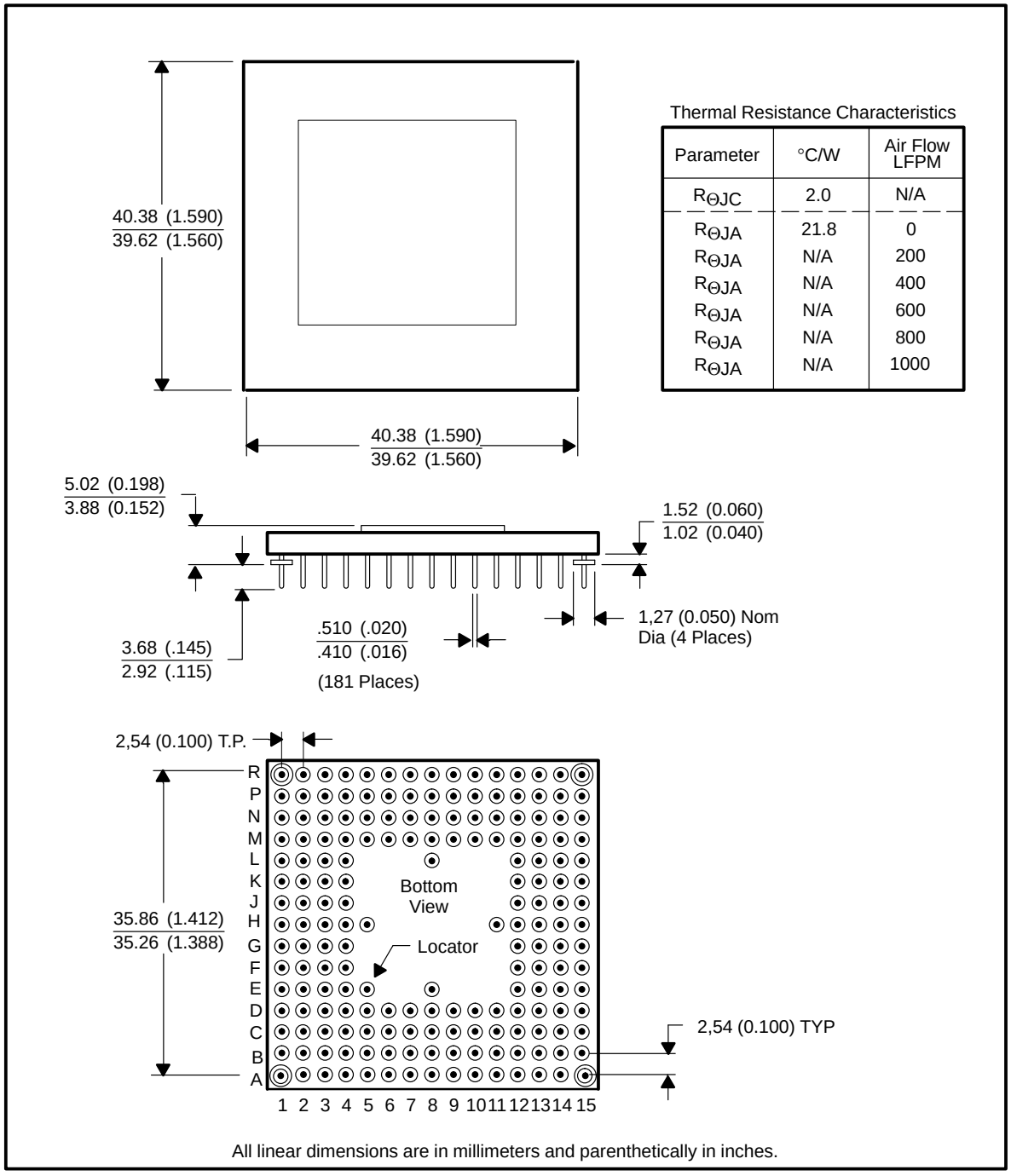


Figure 13–3. TMS320C30 181-Pin PGA Dimensions—GEL Package



Thermal Resistance Characteristics

Parameter	°C/W	Air Flow LFPM
$R_{\Theta JC}$	2.0	N/A
$R_{\Theta JA}$	21.8	0
$R_{\Theta JA}$	N/A	200
$R_{\Theta JA}$	N/A	400
$R_{\Theta JA}$	N/A	600
$R_{\Theta JA}$	N/A	800
$R_{\Theta JA}$	N/A	1000

All linear dimensions are in millimeters and parenthetically in inches.

Table 13–1. TMS320C30–PGA Pin Assignments (Alphabetical)†

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
A0	F15	D8	B5	EMU8	F14	V_{BBP}	D3	XD15	R10
A1	G12	D9	C6	FSR0	P3	V_{DD}	D8	XD16	M9
A2	G13	D10	A5	FSR1	M3	V_{DD}	H4	XD17	P10
A3	G14	D11	B6	FSX0	R2	V_{DD}	H12	XD18	R11
A4	G15	D12	D7	FSX1	P1	V_{DD}	M8	XD19	N10
A5	H15	D13	A6	H1	B3	V_{SS}	C8	XD20	P11
A6	H14	D14	C7	H3	A1	V_{SS}	H3	XD21	R12
A7	J15	D15	B7	$\overline{\text{HOLD}}$	F3	V_{SS}	H13	XD22	M10
A8	J14	D16	A7	$\overline{\text{HOLDA}}$	E2	V_{SS}	N8	XD23	N11
A9	J13	D17	A8	$\overline{\text{IACK}}$	G1	V_{SUBS}	E4	XD24	P12
A10	K15	D18	B8	$\overline{\text{INT0}}$	H2	X1	C2	XD25	R13
A11	J12	D19	A9	$\overline{\text{INT1}}$	H1	X2/CLKIN	B1	XD26	R14
A12	K14	D20	B9	$\overline{\text{INT2}}$	J1	XA0	A13	XD27	M11
A13	L15	D21	C9	$\overline{\text{INT3}}$	J2	XA1	A14	XD28	N12
A14	K13	D22	A10	$IODV_{DD}$	L8	XA2	D11	XD29	P13
A15	L14	D23	D9	$\overline{IODV_{DD}}$	M12	XA3	C12	XD30	R15
A16	M15	D24	B10	$\overline{\text{IOSTRB}}$	F4	XA4	B13	XD31	P15
A17	K12	D25	A11	IV_{SS}	B14	XA5	A15	XF0	G2
A18	L13	D26	C10	LOCATOR	E5	XA6	B15	XF1	G3
A19	M14	D27	B11	MC/$\overline{\text{MP}}$	D15	XA7	C14	$\overline{\text{XRDY}}$	D2
A20	M13	D28	A12	MDV_{DD}	H5	XA8	E12	XR/$\overline{\text{W}}$	D1
A21	N15	D29	D10	MSTRB	E3	XA9	D13		
A22	L12	D30	C11	PDV_{DD}	M4	XA10	C15		
A23	N14	D31	B12	$\overline{\text{RDY}}$	E1	XA11	D14		
ADV_{DD}	D12	DDV_{DD}	D4	$\overline{\text{RESET}}$	F1	XA12	E13		
ADV_{DD}	H11	DDV_{DD}	E8	RSV0	J3	XD0	R4		
CLKR0	N4	DR0	R1	RSV1	J4	XD1	P5		
CLKR1	L4	DR1	N1	RSV2	K1	XD2	N6		
CLKX0	M5	DV_{SS}	C3	RSV3	K2	XD3	R5		
CLKX1	N2	DV_{SS}	C13	RSV4	L1	XD4	P6		
CV_{SS}	B2	DV_{SS}	N3	RSV5	K3	XD5	M7		
CV_{SS}	P14	DV_{SS}	N13	RSV6	L2	XD6	R6		
D0	C4	DX0	R3	RSV7	K4	XD7	N7		
D1	D5	DX1	P2	RSV8	M1	XD8	P7		
D2	A2	EMU1	E15	RSV9	L3	XD9	R7		
D3	A3	EMU2	F13	RSV10	M2	XD10	P8		
D4	B4	EMU3	E14	R/$\overline{\text{W}}$	G4	XD11	R8		
D5	C5	EMU4/$\overline{\text{SHZ}}$	F12	$\overline{\text{STRB}}$	F2	XD12	R9		
D6	D6	EMU5	C1	TCLK0	P4	XD13	P9		
D7	A4	EMU6	M6	TCLK1	N5	XD14	N9		

† ADV_{DD}, CV_{SS}, DDV_{DD}, DV_{SS}, $IODV_{DD}$, IV_{SS}, MDV_{DD}, PDV_{DD}, V_{DD}, and V_{SS} pins are on a common plane internal to the device.

Table 13–2. TMS320C30–PGA Pin Assignments (Numerical)†

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
H3	A1	D30	C11	XF1	G3	A13	L15	XD17	P10
D2	A2	XA3	C12	R/$\overline{W}$	G4	RSV8	M1	XD20	P11
D3	A3	DV_{SS}	C13	A1	G12	RSV10	M2	XD24	P12
D7	A4	XA7	C14	A2	G13	FSR1	M3	XD29	P13
D10	A5	XA10	C15	A3	G14	PDV_{DD}	M4	CV_{SS}	P14
D13	A6	XR/$\overline{W}$	D1	A4	G15	CLKX0	M5	XD31	P15
D16	A7	$\overline{XRDY}$	D2	$\overline{INT1}$	H1	EMU6	M6	DR0	R1
D17	A8	V_{BBP}	D3	$\overline{INT0}$	H2	XD5	M7	FSX0	R2
D19	A9	DDV_{DD}	D4	V_{SS}	H3	V_{DD}	M8	DX0	R3
D22	A10	D1	D5	V_{DD}	H4	XD16	M9	XD0	R4
D25	A11	D6	D6	MDV_{DD}	H5	XD22	M10	XD3	R5
D28	A12	D12	D7	ADV_{DD}	H11	XD27	M11	XD6	R6
XA0	A13	V_{DD}	D8	V_{DD}	H12	$IODV_{DD}$	M12	XD9	R7
XA1	A14	D23	D9	V_{SS}	H13	A20	M13	XD11	R8
XA5	A15	D29	D10	A6	H14	A19	M14	XD12	R9
X2/CLKIN	B1	XA2	D11	A5	H15	A16	M15	XD15	R10
CV_{SS}	B2	ADV_{DD}	D12	$\overline{INT2}$	J1	DR1	N1	XD18	R11
H1	B3	XA9	D13	$\overline{INT3}$	J2	CLKX1	N2	XD21	R12
D4	B4	XA11	D14	RSV0	J3	DV_{SS}	N3	XD25	R13
D8	B5	MC/$\overline{MP}$	D15	RSV1	J4	CLKR0	N4	XD26	R14
D11	B6	$\overline{RDY}$	E1	A11	J12	TCLK1	N5	XD30	R15
D15	B7	$\overline{HOLDA}$	E2	A9	J13	XD2	N6		
D18	B8	$\overline{MSTRB}$	E3	A8	J14	XD7	N7		
D20	B9	V_{SUBS}	E4	A7	J15	V_{SS}	N8		
D24	B10	LOCATOR	E5	RSV2	K1	XD14	N9		
D27	B11	DDV_{DD}	E8	RSV3	K2	XD19	N10		
D31	B12	XA8	E12	RSV5	K3	XD23	N11		
XA4	B13	XA12	E13	RSV7	K4	XD28	N12		
IV_{SS}	B14	EMU3	E14	A17	K12	DV_{SS}	N13		
XA6	B15	EMU1	E15	A14	K13	A23	N14		
EMU5	C1	$\overline{RESET}$	F1	A12	K14	A21	N15		
X1	C2	$\overline{STRB}$	F2	A10	K15	FSX1	P1		
DV_{SS}	C3	$\overline{HOLD}$	F3	RSV4	L1	DX1	P2		
D0	C4	$\overline{IOSTRB}$	F4	RSV6	L2	FSR0	P3		
D5	C5	EMU4/$\overline{SHZ}$	F12	RSV9	L3	TCLK0	P4		
D9	C6	EMU2	F13	CLKR1	L4	XD1	P5		
D14	C7	EMU8	F14	$IODV_{DD}$	L8	XD4	P6		
V_{SS}	C8	A0	F15	A22	L12	XD8	P7		
D21	C9	$\overline{IACK}$	G1	A18	L13	XD10	P8		
D26	C10	XF0	G2	A15	L14	XD13	P9		

† ADV_{DD}, CV_{SS}, DDV_{DD}, DV_{SS}, $IODV_{DD}$, IV_{SS}, MDV_{DD}, PDV_{DD}, V_{DD}, and V_{SS} pins are on a common plane internal to the device.

13.1.2 TMS320C30 PPM Pinouts and Pin Assignments

The TMS320C30 PPM device is packaged in a 208-pin plastic quad flat pack (PQFP) JDEC standard package. Figure 13–4 shows the pinouts for this package, and Figure 13–5 shows the mechanical layout. Table 13–3 shows the associated pin assignments alphabetically; Table 13–4 shows the assignments numerically.

Figure 13–4. TMS320C30 PPM Pinout (Top View)

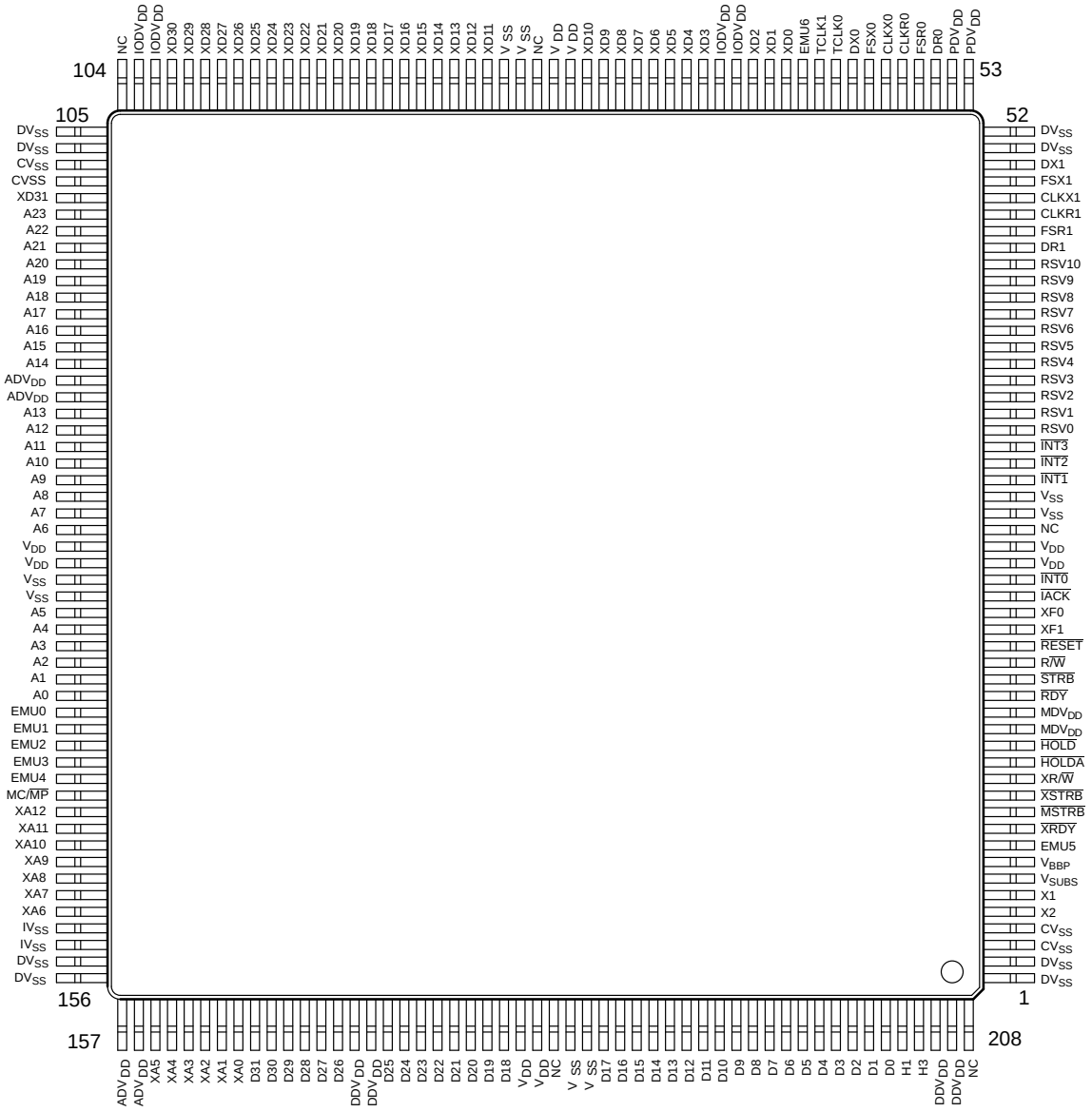
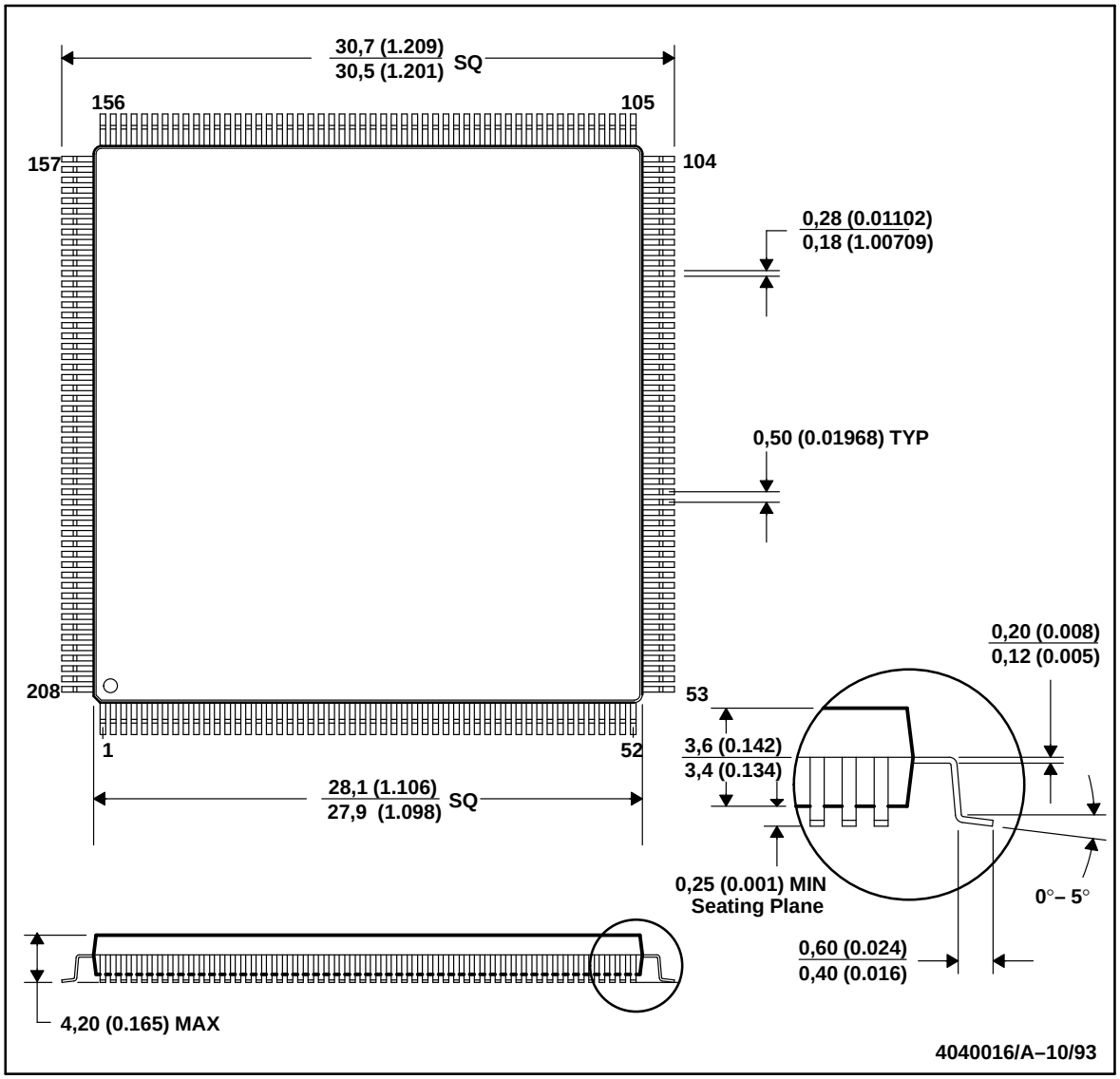


Figure 13–5. TMS320C30 PPM 208-Pin Plastic Quad Flat Pack—PQL Package



Notes: 1) All linear dimensions are in millimeters and parenthetically in inches.

2) This drawing is subject to change without notice.

3) Contact a field sales office to determine if a tighter coplanarity requirement is available for this package.

Table 13–3. TMS320C30–PPM Pin Assignments (Alphabetical)†

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
A0	139	D6	197	EMU0	140	RSV3	37	XA10	148
A1	138	D7	196	EMU1	141	RSV4	38	XA11	147
A2	137	D8	195	EMU2	142	RSV5	39	XA12	146
A3	136	D9	194	EMU3	143	RSV6	40	XD0	64
A4	135	D10	193	EMU4/$\overline{SHZ}$	144	RSV7	41	XD1	65
A5	134	D11	192	EMU5	9	RSV8	42	XD2	66
A6	129	D12	191	EMU6	63	RSV9	43	XD3	69
A7	128	D13	190	FSR0	56	RSV10	44	XD4	70
A8	127	D14	189	FSR1	46	R/$\overline{W}$	20	XD5	71
A9	126	D15	188	FSX0	59	$\overline{STRB}$	19	XD6	72
A10	125	D16	187	FSX1	49	TCLK0	61	XD7	73
A11	124	D17	186	H1	204	TCLK1	62	XD8	74
A12	123	D18	180	H3	205	V_{BBP}	8	XD9	75
A13	122	D19	179	$\overline{HOLD}$	15	V_{DD}	26	XD10	76
A14	119	D20	178	$\overline{HOLDA}$	14	V_{DD}	27	XD11	82
A15	118	D21	177	$\overline{IACK}$	24	V_{DD}	77	XD12	83
A16	117	D22	176	$\overline{INT0}$	25	V_{DD}	78	XD13	84
A17	116	D23	175	$\overline{INT1}$	31	V_{DD}	130	XD14	85
A18	115	D24	174	$\overline{INT2}$	32	V_{DD}	131	XD15	86
A19	114	D25	173	$\overline{INT3}$	33	V_{DD}	181	XD16	87
A20	113	D26	170	$IODV_{DD}$	67	V_{DD}	182	XD17	88
A21	112	D27	169	$IODV_{DD}$	68	V_{SS}	29	XD18	89
A22	111	D28	168	$IODV_{DD}$	102	V_{SS}	30	XD19	90
A23	110	D29	167	$IODV_{DD}$	103	V_{SS}	80	XD20	91
ADV_{DD}	120	D30	166	IV_{SS}	153	V_{SS}	81	XD21	92
ADV_{DD}	121	D31	165	IV_{SS}	154	V_{SS}	132	XD22	93
ADV_{DD}	157	DDV_{DD}	171	MC/$\overline{MP}$	145	V_{SS}	133	XD23	94
ADV_{DD}	158	DDV_{DD}	172	MDV_{DD}	16	V_{SS}	184	XD24	95
CLKR0	57	DDV_{DD}	206	$\overline{MDV_{DD}}$	17	V_{SS}	185	XD25	96
CLKR1	47	DDV_{DD}	207	$\overline{MSTRB}$	11	V_{SUBS}	7	XD26	97
CLKX0	58	DR0	55	NC	28	X1	6	XD27	98
CLKX1	48	DR1	45	NC	79	X2/CLKIN	5	XD28	99
CV_{SS}	3	DV_{SS}	1	NC	104	XA0	164	XD29	100
CV_{SS}	4	DV_{SS}	2	NC	183	XA1	163	XD30	101
CV_{SS}	107	DV_{SS}	51	NC	208	XA2	162	XD31	109
CV_{SS}	108	DV_{SS}	52	PDV_{DD}	53	XA3	161	XF0	23
D0	203	DV_{SS}	105	$\overline{PDV_{DD}}$	54	XA4	160	$\overline{XF1}$	22
D1	202	DV_{SS}	106	$\overline{RDY}$	18	XA5	159	$\overline{XRDY}$	10
D2	201	DV_{SS}	155	$\overline{RESET}$	21	XA6	152	XR/$\overline{W}$	13
D3	200	DV_{SS}	156	RSV0	34	XA7	151	$\overline{XSTRB}$	12
D4	199	DX0	60	RSV1	35	XA8	150		
D5	198	DX1	50	RSV2	36	XA9	149		

† ADV_{DD}, CV_{SS}, DDV_{DD}, DV_{SS}, $IODV_{DD}$, IV_{SS}, MDV_{DD}, PDV_{DD}, V_{DD}, and V_{SS} pins are on a common plane internal to the device.

Table 13–4. TMS320C30–PPM Pin Assignments (Numerical)†

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	DV_{SS}	43	RSV9	85	XD14	127	A8	169	D27
2	DV_{SS}	44	RSV10	86	XD15	128	A7	170	D26
3	CV_{SS}	45	DR1	87	XD16	129	A6	171	DDV_{DD}
4	CV_{SS}	46	FSR1	88	XD17	130	V_{DD}	172	DDV_{DD}
5	X2	47	CLKR1	89	XD18	131	V_{DD}	173	D25
6	X1	48	CLKX1	90	XD19	132	V_{SS}	174	D24
7	V_{SUBS}	49	FSX1	91	XD20	133	V_{SS}	175	D23
8	V_{BBP}	50	DX1	92	XD21	134	A5	176	D22
9	$\overline{EMU5}$	51	DV_{SS}	93	XD22	135	A4	177	D21
10	$\overline{XRDY}$	52	DV_{SS}	94	XD23	136	A3	178	D20
11	$\overline{MSTRB}$	53	PDV_{DD}	95	XD24	137	A2	179	D19
12	$\overline{XSTRB}$	54	PDV_{DD}	96	XD25	138	A1	180	D18
13	XR/$\overline{W}$	55	DR0	97	XD26	139	A0	181	V_{DD}
14	$\overline{HOLDA}$	56	FSR0	98	XD27	140	EMU0	182	V_{DD}
15	$\overline{HOLD}$	57	CLKR0	99	XD28	141	EMU1	183	NC
16	MDV_{DD}	58	CLKX0	100	XD29	142	EMU2	184	V_{SS}
17	MDV_{DD}	59	FSX0	101	XD30	143	EMU3	185	V_{SS}
18	$\overline{RDY}$	60	DX0	102	$IODV_{DD}$	144	EMU4/$\overline{SHZ}$	186	D17
19	$\overline{STRB}$	61	TCLK0	103	$IODV_{DD}$	145	MC/$\overline{MP}$	187	D16
20	R/$\overline{W}$	62	TCLK1	104	NC	146	XA12	188	D15
21	$\overline{RESET}$	63	EMU6	105	DV_{SS}	147	XA11	189	D14
22	XF1	64	XD0	106	DV_{SS}	148	XA10	190	D13
23	$\overline{XF0}$	65	XD1	107	CV_{SS}	149	XA9	191	D12
24	$\overline{IACK}$	66	XD2	108	CV_{SS}	150	XA8	192	D11
25	$\overline{INT0}$	67	$IODV_{DD}$	109	XD31	151	XA7	193	D10
26	V_{DD}	68	$IODV_{DD}$	110	A23	152	XA6	194	D9
27	V_{DD}	69	XD3	111	A22	153	IV_{SS}	195	D8
28	NC	70	XD4	112	A21	154	IV_{SS}	196	D7
29	V_{SS}	71	XD5	113	A20	155	DV_{SS}	197	D6
30	V_{SS}	72	XD6	114	A19	156	DV_{SS}	198	D5
31	$\overline{INT1}$	73	XD7	115	A18	157	ADV_{DD}	199	D4
32	$\overline{INT2}$	74	XD8	116	A17	158	ADV_{DD}	200	D3
33	$\overline{INT3}$	75	XD9	117	A16	159	XA5	201	D2
34	RSV0	76	XD10	118	A15	160	XA4	202	D1
35	RSV1	77	V_{DD}	119	A14	161	XA3	203	D0
36	RSV2	78	V_{DD}	120	ADV_{DD}	162	XA2	204	H1
37	RSV3	79	NC	121	ADV_{DD}	163	XA1	205	H3
38	RSV4	80	V_{SS}	122	A13	164	XA0	206	DDV_{DD}
39	RSV5	81	V_{SS}	123	A12	165	D31	207	DDV_{DD}
40	RSV6	82	XD11	124	A11	166	D30	208	NC
41	RSV7	83	XD12	125	A10	167	D29		
42	RSV8	84	XD13	126	A9	168	D28		

† ADV_{DD}, CV_{SS}, DDV_{DD}, DV_{SS}, $IODV_{DD}$, IV_{SS}, MDV_{DD}, PDV_{DD}, V_{DD}, and V_{SS} pins are on a common plane internal to the device.

13.1.3 TMS320C31 Pinouts and Pin Assignments

The TMS320C31 device is packaged in a 132-pin plastic quad flat pack (PQFP) JDEC standard package. Figure 13–6 shows the pinouts for this package, and Figure 13–7 shows the mechanical layout. Table 13–5 shows the associated pin assignments alphabetically; Table 13–6 shows the pin assignments numerically.

Figure 13–6. TMS320C31 Pinout (Top View)

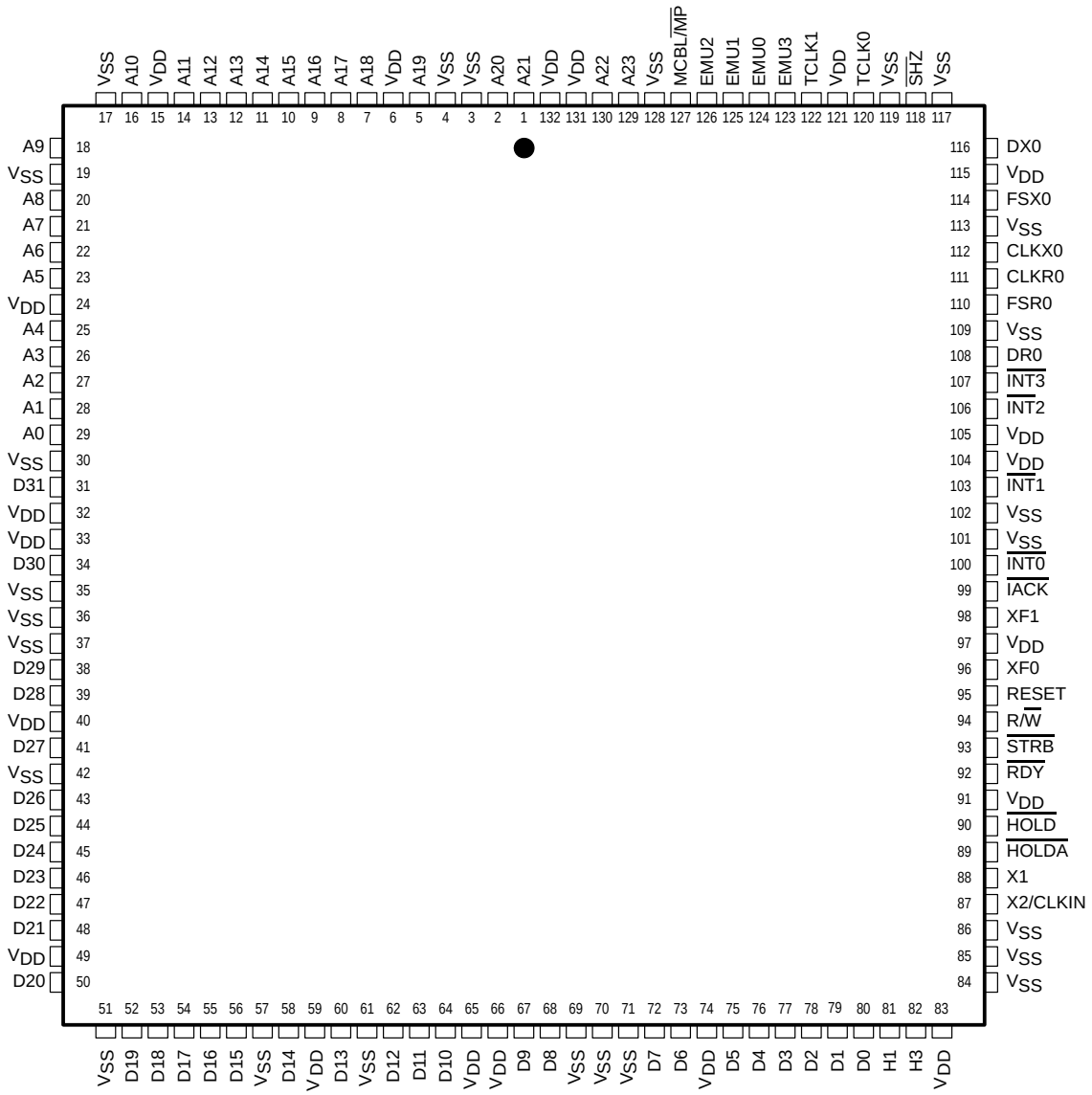
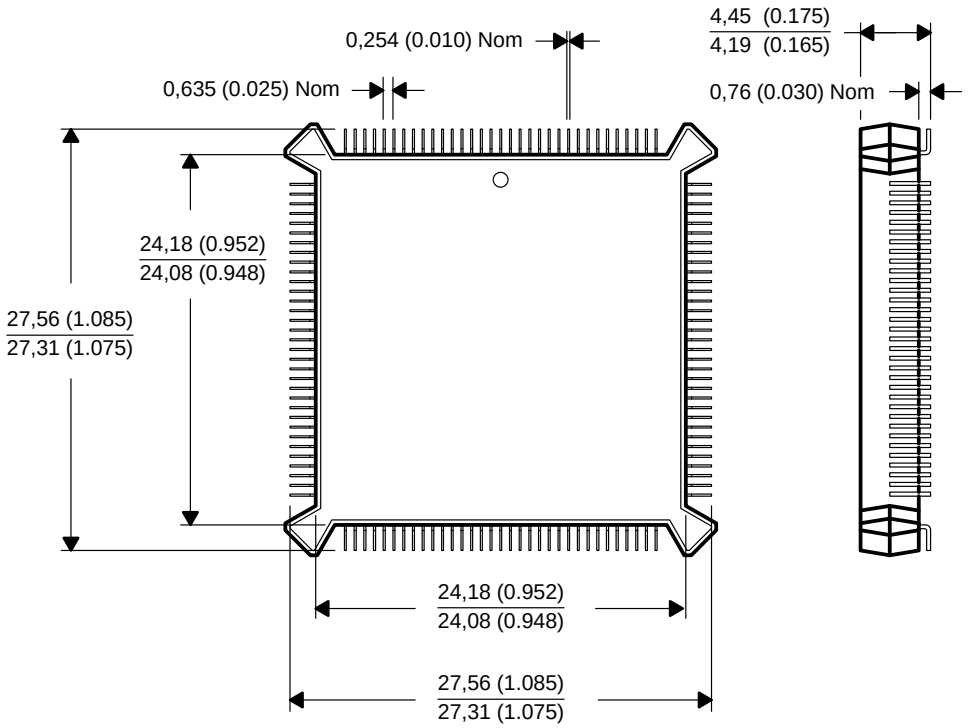


Figure 13–7. TMS320C31 132-Pin Plastic Quad Flat Pack—PQL Package



Thermal Resistance Characteristics

Parameter	°C/W	Air Flow LFPM
$R_{\Theta JC}$	11.0	N/A
$R_{\Theta JA}$	49.0	0
$R_{\Theta JA}$	35.5	200
$R_{\Theta JA}$	28.0	400
$R_{\Theta JA}$	23.5	600
$R_{\Theta JA}$	21.6	800
$R_{\Theta JA}$	20.0	1000

All linear dimensions are in millimeters and parenthetically in inches.

Table 13–5. TMS320C31 Pin Assignments (Alphabetical)†

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
A0	29	D4	76	EMU0	124	V_{DD}	40	V_{SS}	84
A1	28	D5	75	EMU1	125	V_{DD}	49	V_{SS}	85
A2	27	D6	74	EMU2	126	V_{DD}	59	V_{SS}	86
A3	26	D7	73	EMU3	123	V_{DD}	65	V_{SS}	101
A4	25	D8	68	FSR0	110	V_{DD}	66	V_{SS}	102
A5	24	D9	67	FSX0	114	V_{DD}	74	V_{SS}	109
A6	23	D10	64	H1	81	V_{DD}	83	V_{SS}	113
A7	22	D11	63	H3	82	V_{DD}	91	V_{SS}	117
A8	21	D12	62	$\overline{\text{HOLD}}$	90	V_{DD}	97	V_{SS}	119
A9	20	D13	60	$\overline{\text{HOLDA}}$	89	V_{DD}	104	V_{SS}	128
A10	19	D14	58	$\overline{\text{IACK}}$	99	V_{DD}	105	X1	88
A11	18	D15	56	$\overline{\text{INT0}}$	100	V_{DD}	115	X2/CLKIN	87
A12	17	D16	55	$\overline{\text{INT1}}$	103	V_{DD}	121	XF0	96
A13	16	D17	54	$\overline{\text{INT2}}$	106	V_{DD}	131	XF1	98
A14	15	D18	53	$\overline{\text{INT3}}$	107	V_{DD}	132		
A15	14	D19	52	MCBL/$\overline{\text{MP}}$	127	V_{SS}	3		
A16	13	D20	50	$\overline{\text{RDY}}$	92	V_{SS}	4		
A17	12	D21	48	$\overline{\text{RESET}}$	95	V_{SS}	17		
A18	11	D22	47	R/$\overline{\text{W}}$	94	V_{SS}	19		
A19	10	D23	46	$\overline{\text{SHZ}}$	118	V_{SS}	30		
A20	9	D24	45	$\overline{\text{STRB}}$	93	V_{SS}	35		
A21	8	D25	44	TCLK0	120	V_{SS}	36		
A22	7	D26	43	TCLK1	122	V_{SS}	37		
A23	6	D27	41			V_{SS}	42		
CLKR0	5	D28	39			V_{SS}	51		
CLKX0	4	D29	38	V_{DD}	6	V_{SS}	57		
D0	3	D30	34	V_{DD}	15	V_{SS}	61		
D1	2	D31	31	V_{DD}	24	V_{SS}	69		
D2	1	DR0	108	V_{DD}	32	V_{SS}	70		
D3	130	DX0	116	V_{DD}	33	V_{SS}	71		

† V_{DD} and V_{SS} pins are on a common plane internal to the device.

Table 13–6. TMS320C31 Pin Assignments (Numerical)†

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	A21	31	D31	61	V_{SS}	91	V_{DD}	121	V_{DD}
2	A20	32	V_{DD}	62	D12	92	$\overline{RDY}$	122	TCLK1
3	V_{SS}	33	V_{DD}	63	D11	93	$\overline{STRB}$	123	EMU3
4	V_{SS}	34	D30	64	D10	94	R/$\overline{W}$	124	EMU0
5	A19	35	V_{SS}	65	V_{DD}	95	$\overline{RESET}$	125	EMU1
6	V_{DD}	36	V_{SS}	66	V_{DD}	96	XF0	126	EMU2
7	A18	37	V_{SS}	67	D9	97	V_{DD}	127	MCBL/$\overline{MP}$
8	A17	38	D29	68	D8	98	XF1	128	V_{SS}
9	A16	39	D28	69	V_{SS}	99	$\overline{IACK}$	129	A23
10	A15	40	V_{DD}	70	V_{SS}	100	$\overline{INT0}$	130	A22
11	A14	41	D27	71	V_{SS}	101	V_{SS}	131	V_{DD}
12	A13	42	V_{SS}	72	D7	102	V_{SS}	132	V_{DD}
13	A12	43	D26	73	D6	103	$\overline{INT1}$		
14	A11	44	D25	74	V_{DD}	104	V_{DD}		
15	V_{DD}	45	D24	75	D5	105	V_{DD}		
16	A10	46	D23	76	D4	106	$\overline{INT2}$		
17	V_{SS}	47	D22	77	D3	107	$\overline{INT3}$		
18	A9	48	D21	78	D2	108	DR0		
19	V_{SS}	49	V_{DD}	79	D1	109	V_{SS}		
20	A8	50	D20	80	D0	110	FSR0		
21	A7	51	V_{SS}	81	H1	111	CLKR0		
22	A6	52	D19	82	H3	112	CLKX0		
23	A5	53	D18	83	V_{DD}	113	V_{SS}		
24	V_{DD}	54	D17	84	V_{SS}	114	FSX0		
25	A4	55	D16	85	V_{SS}	115	V_{DD}		
26	A3	56	D15	86	V_{SS}	116	DX0		
27	A2	57	V_{SS}	87	X2/CLKIN	117	V_{SS}		
28	A1	58	D14	88	X1	118	SHZ		
29	A0	59	V_{DD}	89	$\overline{HOLDA}$	119	V_{SS}		
30	V_{SS}	60	D13	90	$\overline{HOLD}$	120	TCLK0		

† V_{DD} and V_{SS} pins are on a common plane internal to the device.

13.2 Signal Descriptions

13.2.1 TMS320C30 Signal Descriptions

Table 13–7 describes the signals that the TMS320C30 device uses in the microprocessor mode. It lists the signal/port/bit name; the number of pins allocated; the input (I), output (O), or high-impedance state (Z) operating modes; a brief description of the signal's function; and the condition that places an output pin in high impedance. A line over a signal name (for example, $\overline{\text{RESET}}$) indicates that the signal is active (low) (true at a logic 0 level). Pins labeled NC are not to be connected by the user. The signals are grouped according to function.

Table 13–7. TMS320C30 Signal Descriptions

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡		
			Primary Bus Interface (61 Pins)			
D31–D0	32	I/O/Z	32-bit data port of the primary bus interface	S	H	R
A23–A0	24	O/Z	24-bit address port of the primary bus interface	S	H	R
R/$\overline{W}$	1	O/Z	Read/write signal for primary bus interface. This pin is high when a read is performed and low when a write is performed over the parallel interface.	S	H	R
$\overline{STRB}$	1	O/Z	External access strobe for the primary bus interface	S	H	
$\overline{RDY}$	1	I	Ready signal. This pin indicates that the external device is prepared for a primary bus interface transaction to complete.	S		
$\overline{HOLD}$	1	I	Hold signal for primary bus interface. When $\overline{HOLD}$ is a logic low, any ongoing transaction is completed. The A23–A0, D31–D0, $\overline{STRB}$, and R/$\overline{W}$ signals are placed in a high-impedance state, and all transactions over the primary bus interface are held until $\overline{HOLD}$ becomes a logic high or the NOHOLD bit of the primary bus control register is set.			
$\overline{HOLDA}$	1	O/Z	Hold acknowledge signal for primary bus interface. This signal is generated in response to a logic low on $\overline{HOLD}$. It signals that A23–A0, D31–D0, $\overline{STRB}$, and R/$\overline{W}$ are placed in a high-impedance state and that all transactions over the bus will be held. $\overline{HOLDA}$ will be high in response to a logic high of $\overline{HOLD}$ or when the NOHOLD bit of the primary bus control register is set.	S		
			Expansion Bus Interface (49 Pins)			
XD31–XD0	32	I/O/Z	32-bit data port of the expansion bus interface	S		R
XA12–XA0	13	O/Z	13-bit address port of the expansion bus interface	S		R
XR/$\overline{W}$	1	O/Z	Read/write signal for expansion bus interface. When a read is performed, this pin is held high; when a write is performed, this pin is low.	S		R
$\overline{MSTRB}$	1	O/Z	External memory access strobe for the expansion bus interface	S		

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{HOLD}$ active, R = $\overline{RESET}$ active

Table 13–7. TMS320C30 Signal Descriptions (Continued)

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡	
Expansion Bus Interface (49 Pins) (Continued)					
$\overline{\text{IOSTRB}}$	1	O/Z	External I/O access strobe for expansion bus interface	S	
$\overline{\text{XRDY}}$	1	I	Ready signal. This pin indicates that the external device is prepared for an expansion bus interface transaction to complete.		
Control Signals (9 Pins)					
$\overline{\text{RESET}}$	1	I	Reset. When this pin is a logic low, the device is placed in the reset condition. After reset becomes a logic high, execution begins from the location specified by the reset vector.		
$\overline{\text{INT3}}$–$\overline{\text{INT0}}$	4	I	External interrupts		
$\overline{\text{IACK}}$	1	O/Z	Interrupt acknowledge signal. $\overline{\text{IACK}}$ is set to 1 (logic high) by the IACK instruction. This can be used to indicate the beginning or end of an interrupt service routine.	S	
MC/$\overline{\text{MP}}$	1	I	Microcomputer/microprocessor mode pin		
XF1, XF0	2	I/O/Z	External flag pins. They are used as general-purpose I/O pins or to support interlocked processor instructions.	S	R
Serial Port 0 Signals (6 Pins)					
CLKX0	1	I/O/Z	Serial port 0 transmit clock. Serves as the serial shift clock for the serial port 0 transmitter.	S	R
DX0	1	I/O/Z	Data transmit output. Serial port 0 transmits serial data on this pin.	S	R
FSX0	1	I/O/Z	Frame synchronization pulse for transmit. The FSX0 pulse initiates the transmit data process over pin DX0.	S	R
CLKR0	1	I/O/Z	Serial port 0 receive clock. Serves as the serial shift clock for the serial port 0 receiver.	S	R
DR0	1	I/O/Z	Data receive. Serial port 0 receives serial data via the DR0 pin.	S	R
FSR0	1	I/O/Z	Frame synchronization pulse for receive. The FSR0 pulse initiates the receive data process over DR0.	S	R

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active

Table 13–7. TMS320C30 Signal Descriptions (Continued)

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡	
			Serial Port 1 Signals (6 Pins)		
CLKX1	1	I/O/Z	Serial port 1 transmit clock. Serves as the serial shift clock for the serial port 1 transmitter.	S	R
DX1	1	I/O/Z	Data transmit output. Serial port 1 transmits serial data on this pin.	S	R
FSX1	1	I/O/Z	Frame synchronization pulse for transmit. The FSX1 pulse initiates the transmit data process over pin DX1.	S	R
CLKR1	1	I/O/Z	Serial port 1 receive clock. Serves as serial shift clock for the serial port 1 receiver.	S	R
DR1	1	I/O/Z	Data receive. Serial port 1 receives serial data via the DR1 pin.	S	R
FSR1	1	I/O/Z	Frame synchronization pulse for receive. The FSR1 pulse initiates the receive data process over DR1.	S	R
			Timer 0 Signals (1 Pin)		
TCLK0	1	I/O/Z	Timer clock. As input, TCLK0 is used by timer 0 to count external pulses. As output pin, TCLK0 outputs pulses generated by timer 0.	S	R
			Timer 1 Signals (1 Pin)		
TCLK1	1	I/O/Z	Timer clock. As input, TCLK1 is used by timer 1 to count external pulses. As output pin, TCLK1 outputs pulses generated by timer 1.	S	R
			Supply and Oscillator Signals (29 Pins)		
V_{DD3}–V_{DD0}	4	I	Four +5-V supply pins §		
$IODV_{DD1}$, $IODV_{DD0}$	2	I	Two +5-V supply pins §		
ADV_{DD1}, ADV_{DD0}	2	I	Two +5-V supply pins §		
PDV_{DD}	1	I	One +5-V supply pin §		

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active
§ The recommended decoupling capacitor is 0.1 μF.

Table 13–7. TMS320C30 Signal Descriptions (Continued)

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡
Supply and Oscillator Signals (29 Pins) (Continued)				
DDV_{DD1}, DDV_{DD0}	2	I	Two +5-V supply pins §	
MDV_{DD}	1	I	One +5-V supply pin §	
V_{SS3}–V_{SS0}	4	I	Four ground pins	
DV_{SS3}–DV_{SS0}	4	I	Four ground pins	
CV_{SS1}, CV_{SS0}	2	I	Two ground pins	
IV_{SS}	1	I	One ground pin	
V_{BBP}	1	NC	V_{BB} pump oscillator output	
V_{SUBS}	1	I	Substrate pin. Tie to ground.	
X1	1	O	Output pin from internal oscillator for the crystal. If crystal not used, pin should be left unconnected.	
X2/CLKIN	1	I	Input pin to internal oscillator from a crystal or a clock	
H1	1	O/Z	External H1 clock—has a period equal to twice CLKIN.	S
H3	1	O/Z	External H3 clock—has a period equal to twice CLKIN.	S

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active
§ Follow the connections specified for the reserved pins. 18- to 22-kΩ pull-up resistors are recommended. All +5-volt supply pins must be connected to a common supply plane, and all ground pins must be connected to a common ground plane.

Table 13–7.TMS320C30 Signal Descriptions (Continued)

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡
			Reserved (18 Pins) §	
EMU2–EMU0	3	I	Reserved. Use pull-ups to +5 volts. See Section 12.7 on page 12-39.	
EMU3	1	O	Reserved. See Section 12.7 on page 12-39.	
EMU4/$\overline{\text{SHZ}}$	1	I	Shutdown high impedance. An active low shuts down the TMS320C30 and places all pins in a high-impedance state. This signal is used for board-level testing to ensure that no dual drive conditions occur. **CAUTION:** An active low on the $\overline{\text{SHZ}}$ pin corrupts TMS320C30 memory and register contents. Reset the device with an $\overline{\text{SHZ}}$=1 to restore it to a known operating condition.	
EMU6, EMU5	2	NC	Reserved.	
RSV10–RSV5	6	I/O	Reserved. Use pull-ups on each pin to +5 volts.	
RSV4–RSV0	5	I	Reserved. Tie pins directly to +5 volts.	
			Locator (1 Pin)	
Locator	1	NC	Reserved. See Figure 13–1 on page 13-3 and Table 13–1 on page 13-6.	

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active
§ Follow the connections specified for the reserved pins. 18- to 22-kΩ pull-up resistors are recommended. All +5-volt supply pins must be connected to a common supply plane, and all ground pins must be connected to a common ground plane.

13.2.2 TMS320C31 Signal Descriptions

Table 13–8 describes the signals that the TMS320C31 device uses in the microprocessor mode. They are listed according to the signal name; the number of pins allocated; the input (I), output (O), or high-impedance state (Z) operating modes; a brief description of the signal's function; and the condition that places an output pin in high impedance. A line over a signal name (for example, $\overline{\text{RESET}}$) indicates that the signal is active (low) (true at a logic 0 level).

Table 13–8. TMS320C31 Signal Descriptions

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡		
			Primary Bus Interface (61 Pins)			
D31–D0	32	I/O/Z	32-bit data port	S	H	R
A23–A0	24	O/Z	24-bit address port	S	H	R
$\overline{\text{HOLD}}$	1	I	Hold signal. When $\overline{\text{HOLD}}$ is a logic low, any ongoing transaction is completed. The A23–A0, D31–D0, $\overline{\text{STRB}}$, and R/$\overline{\text{W}}$ signals are placed in a high-impedance state, and all transactions over the primary bus interface are held until $\overline{\text{HOLD}}$ becomes a logic high or until the NOHOLD bit of the primary bus control register is set.			
$\overline{\text{HOLDA}}$	1	O/Z	Hold acknowledge signal. This signal is generated in response to a logic low on $\overline{\text{HOLD}}$. It signals that A23–A0, D31–D0, $\overline{\text{STRB}}$, and R/$\overline{\text{W}}$ are placed in a high-impedance state and that all transactions over the bus will be held. $\overline{\text{HOLDA}}$ will be high in response to a logic high of $\overline{\text{HOLD}}$ or until the NOHOLD bit of the primary bus control register is set.	S		
R/$\overline{\text{W}}$	1	O/Z	Read/write signal. This pin is high when a read is performed and low when a write is performed over the parallel interface.	S	H	R
$\overline{\text{RDY}}$	1	I	Ready signal. This pin indicates that the external device is prepared for a transaction completion.			
$\overline{\text{STRB}}$	1	O/Z	External access strobe	S	H	

† Input (I), output (O), high-impedance (Z) state
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active

Table 13–8. TMS320C31 Signal Descriptions (Continued)

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡	
			Control Signals (10 Pins)		
$\overline{\text{INT3}}$–$\overline{\text{INT0}}$	4	I	External interrupts		
$\overline{\text{IACK}}$	1	O/Z	Interrupt acknowledge signal. $\overline{\text{IACK}}$ is set to 1 by the IACK instruction. This can be used to indicate the beginning or end of an interrupt service routine.	S	
MCBL/$\overline{\text{MP}}$	1	I	Microcomputer boot loader/microprocessor mode pin		
$\overline{\text{RESET}}$	1	I	Reset. When this pin is a logic low, the device is placed in the reset condition. When reset becomes a logic 1, execution begins from the location specified by the reset vector.		
$\overline{\text{SHZ}}$	1	I	Shut down high Z. An active (low) shuts down the TMS320C31 and places all pins in a high-impedance state. This signal is used for board-level testing to ensure that no dual drive conditions occur. **CAUTION:** An active (low) on the $\overline{\text{SHZ}}$ pin corrupts TMS320C31 memory and register contents. Reset the device with an $\overline{\text{SHZ}}$ = 1 to restore it to a known operating condition.		
XF1, XF0	2	I/O/Z	External flag pins. These are used as general-purpose I/O pins or to support interlocked processor instructions.	S	R
			Serial Port 0 Signals (6 Pins)		
CLKR0	1	I/O/Z	Serial port 0 receive clock. This pin serves as the serial shift clock for the serial port 0 receiver.	S	R
CLKX0	1	I/O/Z	Serial port 0 transmit clock. Serves as the serial shift clock for the serial port 0 transmitter.	S	R
DR0	1	I/O/Z	Data receive. Serial port 0 receives serial data via the DR0 pin.	S	R
DX0	1	I/O/Z	Data transmit output. Serial port 0 transmits serial data on this pin.	S	R
FSR0	1	I/O/Z	Frame synchronization pulse for receive. The FSR0 pulse initiates the receive data process over DR0.	S	R

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active

Table 13–8. TMS320C31 Signal Descriptions (Continued)

Signal/Port	# Pins	I/O/Z†	Description	Condition When Signal Is in High Z‡	
Serial Port 0 Signals (6 Pins) (Continued)					
FSX0	1	I/O/Z	Frame synchronization pulse for transmit. The FSX0 pulse initiates the transmit data process over pin DX0.	S	R
Timer Signals (2 Pins)					
TCLK0	1	I/O/Z	Timer clock 0. As an input, TCLK0 is used by timer 0 to count external pulses. As an output pin, TCLK0 outputs pulses generated by timer 0.	S	
TCLK1	1	I/O/Z	Timer clock 1. As an input, TCLK0 is used by timer 1 to count external pulses. As an output pin, TCLK1 outputs pulses generated by timer 1.	S	
Supply and Oscillator Signals (49 Pins)					
H1	1	O/Z	External H1 clock. This clock has a period equal to twice CLKIN.	S	
H3	1	O/Z	External H3 clock. This clock has a period equal to twice CLKIN.	S	
V_{DD}	20	I	+5-V_{DC} supply pins. All pins must be connected to a common supply plane. §		
V_{SS}	25	I	Ground pins. All ground pins must be connected to a common ground plane.		
X1	1	O/Z	Output pin from the internal crystal oscillator. If a crystal is not used, this pin should be left unconnected.		
X2/CLKIN	1	I	The internal oscillator input pin from a crystal or a clock.		
Reserved (4 Pins) ¶					
EMU2–EMU0	3	I	Reserved. Use 20-kΩ pull-up resistors to +5 volts.		
EMU3	1	O	Reserved.		

† Input (I), output (O), high-impedance state (Z)
‡ S = SHZ active, H = $\overline{\text{HOLD}}$ active, R = $\overline{\text{RESET}}$ active
§ The recommended decoupling capacitor value is 0.1 μF.
¶ Follow the connections specified for the reserved pins. 18- to 22-kΩ pull-up resistors are recommended. All +5-volt supply pins must be connected to a common supply plane, and all ground pins must be connected to a common ground plane.

13.3 Electrical Specifications

Table 13–9, Table 13–10, Table 13–11, and Figure 13–8 show the electrical specifications for the TMS320C3x.

Table 13–9. Absolute Maximum Ratings Over Specified Temperature Range

Condition/Characteristic	’C30/’C31 Range	’LC31 Range
Supply voltage range, V_{DD}	–0.3 V to 7 V	–0.3 V to 5 V
Input voltage range	–0.3 V to 7 V	–0.3 V to 5 V
Output voltage range	–0.3 V to 7 V	–0.3 V to 5 V
Continuous power dissipation (worst case)	3.15 W for TMS320C30–33 1.7 W for TMS320C31–33 (See Note 3)	1.1 W (See Note 3)
Operating case temperature range	TMS320C30GEL 0°C to 85°C TMS320C31PQL 0°C to 85°C TMS320C31PQA –40°C to +125°C	0°C to 85°C
Storage temperature range	–55°C to 150°C	–55°C to 150°C

Notes: 1) All voltage values are with respect to V_{SS}.

2) Stresses beyond those listed above may cause permanent damage to the device. This is a stress rating only; functional operation of the device at these or any other conditions beyond those indicated in Table 13–10 is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

3) Actual operating power will be less than stated. These values were obtained under specially produced worst-case test conditions, which are not sustained during normal device operation. These conditions consist of continuous parallel writes of a checkerboard pattern to both primary and expansion buses at the maximum rate possible. See nominal (I_{DD}) current specification in Table 13–11.

Table 13–10. Recommended Operating Conditions

		’C30/’C31			’LC31–33			
	Parameter	**Min**	**Nom**	**Max**	**Min**	**Nom**	**Max**	**Unit**
V_{DD}	Supply voltages (DDV_{DD}, etc.)	4.75	5	5.25	3.13	3.3	3.47	V
V_{SS}	Supply voltages (CV_{SS}, etc.)		0			0		V
V_{IH}	High-level input voltage	2		V_{DD} + 0.3†	1.8		V_{DD} + 0.3†	V
V_{IL}	Low-level input voltage	–0.3		0.8	–0.3†		0.6	V
I_{OH}	High-level output current			–300			–300	µA
I_{OL}	Low-level output current			2			2	mA
T	Operating case temperature range	0		85	0		85	°C
V_{TH}	CLKIN high-level input voltage for CLKIN	2.6	V_{DD} + 0.3†		2.5	V_{DD} + 0.3†		V

† Guaranteed from characterization but not tested

Note: All voltage values are with respect to V_{SS}. All input and output voltages except those for CLKIN are TTL compatible. CLKIN can be driven by a CMOS clock.

Table 13–11. Electrical Characteristics Over Specified Free-Air Temperature Range†

	Electrical Characteristic		’C30/’C31			’LC31-33			Unit
			Min	Nom‡	Max	Min	Nom‡	Max	
V_{OH}	High-level output voltage (V_{DD} = Min, I_{OH} = Max)		2.4	3		2.0			V
V_{OL}§	Low-level output voltage (V_{DD} = Min, I_{OL} = Max)			0.3	0.6			0.4	V
I_Z	Three-state current (V_{DD} = Max)		–20		20	–20		20	µA
I_I	Input current (V_I = V_{SS} to V_{DD})		–10		10	–10		10	µA
I_{IP}	Input current (Inputs with internal pull-ups) ¶		–400		20	–400		10	µA
I_{CC}	Supply current (T_A = 25° C, V_{DD} = Max, f_x = Max) #‖	’C30-33 ’C30-27 ’C30-40 ’C31-27 ’C31-33 ’C31-33 (ext. temp) ’C31-40 ’C31-50 ’C30 PPM		200 175 170 120 150 150 160 200 170	600 500 600 260 325 325 390 425 600		120	300	mA
I_{DD}	Supply current, standby; IDLE2, clocks shut off			50			21		mA
C_i	Input capacitance	All inputs except CLKIN			15☆			15☆	pF
		CLKIN			25			25	
C_o	Output capacitance				20☆			20☆	pF

† All input and output voltage levels are TTL compatible.

‡ All nominal values are at V_{DD} = 5 V, T_A = 25°C.

§ For ’C30 PPM: V_{OL}(max)=0.6 V, except for the following:
V_{OL}(max)=1 V for A(0–31)
V_{OL}(max)=0.9 V for XA(0–12), D(0–31)
V_{OL}(max)=0.7 V for $\overline{\text{STRB}}$, $\overline{\text{XSTRB}}$, $\overline{\text{MSTRB}}$, FSX0/I, CLKX0/1, CLKR0/1, DX0/1 R/$\overline{\text{W}}$, XR/$\overline{\text{W}}$

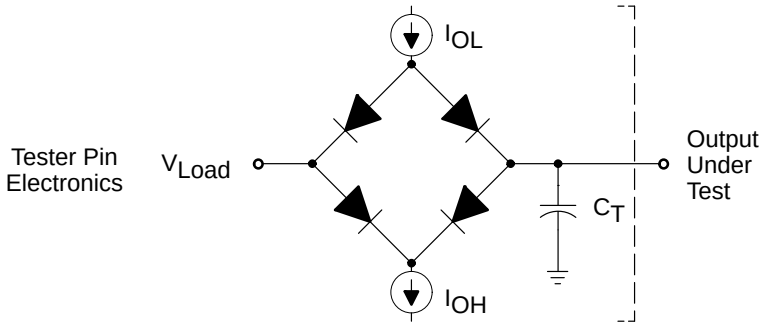
¶ Pins with internal pull-up devices: $\overline{\text{INT3}}$–$\overline{\text{INT0}}$, MC/$\overline{\text{MP}}$, RSV10–RSV0. Although RSV10–RSV0 have internal pullup devices, external pullups should be used on each pin as described in Table 13–7 beginning on page 13-17.

Actual operating current will be less than this maximum value. This value was obtained under specially produced worst-case test conditions, which are not sustained during normal device operation. These conditions consist of continuous parallel writes of a checkerboard pattern to both primary and expansion buses at the maximum rate possible. See *Calculation of TMS320C30 Power Dissipation,* Appendix D.

‖ f_x is the input clock frequency. The maximum value is 40 MHz.

☆ Guaranteed by design but not tested

Figure 13–8. Test Load Circuit



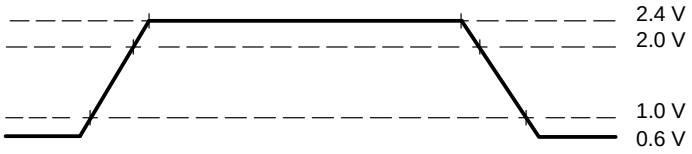
Where: I_{OL} = 2.0 mA (all outputs)
I_{OH} = 300 µA (all outputs)
V_{Load} = 2.15 V
C_T = 80 pF typical load circuit capacitance

13.4 Signal Transition Levels

13.4.1 TTL-Level Outputs

TTL-compatible output levels are driven to a minimum logic-high level of 2.4 volts and to a maximum logic-low level of 0.6 volt. Figure 13–9 shows the TTL-level outputs.

Figure 13–9. TTL-Level Outputs



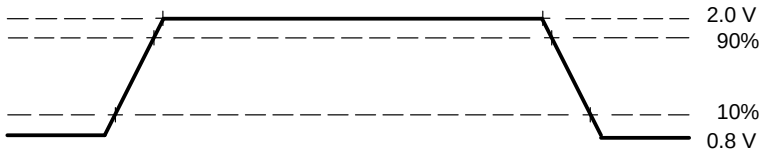
TTL-output transition times are specified as follows:

- For a *high-to-low transition,* the level at which the output is said to be no longer high is 2.0 volts, and the level at which the output is said to be low is 1.0 volt.
- For a *low-to-high transition*, the level at which the output is said to be no longer low is 1.0 volt, and the level at which the output is said to be high is 2.0 volts.

13.4.2 TTL-Level Inputs

Figure 13–10 shows the TTL-level inputs.

Figure 13–10. TTL-Level Inputs



TTL-compatible input transition times are specified as follows:

- For a *high-to-low transition* on an input signal, the level at which the input is said to be no longer high is 2.0 volts, and the level at which the input is said to be low is 0.8 volt.
- For a *low-to-high transition* on an input signal, the level at which the input is said to be no longer low is 0.8 volt, and the level at which the input is said to be high is 2.0 volts.

13.5 Timing

Timing specifications apply to the TMS320C30 and TMS320C31.

13.5.1 X2/CLKIN, H1, and H3 Timing

Table 13–12 defines the timing parameters for the X2/CLKIN, H1, and H3 interface signals. The numbers shown in parentheses in Figure 13–11 and Figure 13–12 correspond with those in the No. column of Table 13–12. Refer to the $\overline{\text{RESET}}$ timing in Figure 13–23 on page 13-48 for CLKIN to H1/H3 delay specification.

Table 13–12. Timing Parameters for X2/CLKIN, H1, and H3§

			’C30-27/ ’C31-27		’C30-33/ ’C31-33/ ’LC31		’C30-40/ ’C31-40		’C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{f(CI)}$	CLKIN fall time		6‡		5‡		5‡		5‡	ns
(2)	$t_{w(CIL)}$	CLKIN low pulse duration $t_{c(CI)}$ = min	14		10		9		7		ns
(3)	$t_{w(CIH)}$	CLKIN high pulse duration $t_{c(CI)}$ = min	14		10		9		7		ns
(4)	$t_{r(CI)}$	CLKIN rise time		6‡		5‡		5‡		5‡	ns
(5)	$t_{c(CI)}$	CLKIN cycle time	37	303	30	303	25	303	20	303	ns
(6)	$t_{f(H)}$	H1/H3 fall time		4		3		3		3	ns
(7)	$t_{w(HL)}$	H1/H3 low pulse duration	P–6		P–6		P–5		P–5		ns
(8)	$t_{w(HH)}$	H1/H3 high pulse duration	P–7		P–7		P–6		P–6		ns
(9)	$t_{r(H)}$	H1/H3 rise time		5		4		3		3	ns
(9.1)	$t_{d(HL–HH)}$	Delay from H1(H3) low to H3(H1) high	0†	6	0†	5	0†	4	0†	4	ns
(10)	$t_{c(H)}$	H1/H3 cycle time	74	606	60	606	50	606	40	606	ns

† Guaranteed from characterization but not tested
‡ Guaranteed by design but not tested
§ P = $t_{c(CI)}$

Figure 13–11. Timing for X2/CLKIN

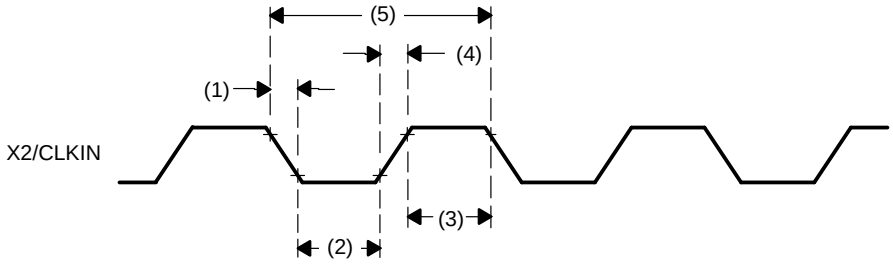
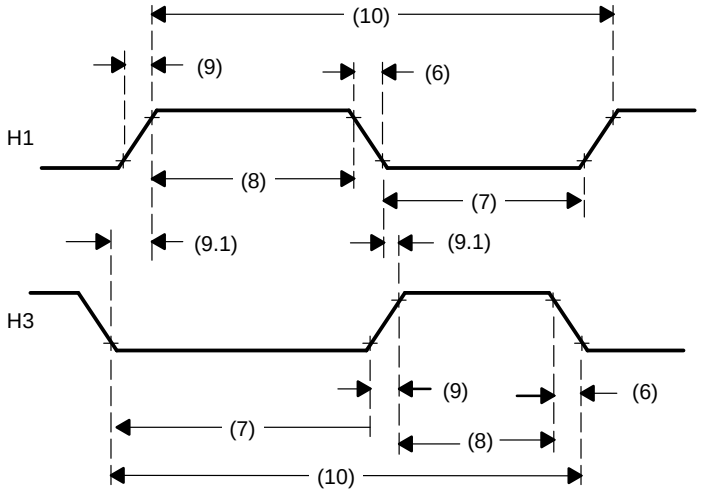


Figure 13–12. Timing for H1/H3



13.5.2 Memory Read/Write Timing

Table 13–13 defines memory read/write timing parameters for $\overline{\text{(M)STRB}}$. The numbers shown in parentheses in Figure 13–13 and Figure 13–14 correspond with those in the No. column of Table 13–13.

Table 13–13. Timing Parameters for a Memory ($\overline{(M)STRB}$) = 0) Read/Write

No.	Name	Description	’C30-27 ’C31-27		’C30-33 ’C31-33 ’LC31		’C30-40 ’C31-40		’C31-50		Unit
			Min	Max	Min	Max	Min	Max	Min	Max	
(11)	$t_{d(H1L–(M)SL)}$	H1 low to $\overline{(M)STRB}$ low delay	0‡	13	0‡	10	0‡	6§	0‡	4	ns
(12)	$t_{d(H1L–(M)SH)}$	H1 low to $\overline{(M)STRB}$ high delay	0‡	13	0‡	10	0‡	6	0‡	4	ns
(13.1)	$t_{d(H1H–RWL)}$	H1 high to R/$\overline{W}$ low delay	0‡	13	0‡	10	0‡	9	0‡	7	ns
(13.2)	$t_{d(H1H–XRWL)}$	H1 high to XR/$\overline{W}$ low delay	0‡	19	0‡	15	0‡	13			ns
(14.1)	$t_{d(H1L–A)}$	H1 low to A valid delay	0‡	16	0‡	14	0‡	11	0‡	9	ns
(14.2)	$t_{d(H1L–XA)}$	H1 low to XA valid delay	0‡	12	0‡	10	0‡	9			ns
(15.1)	$t_{su(D)R}$	D setup before H1 low (read)	18		16		14		10		ns
(15.2)	$t_{su(XD)R}$	XD setup before H1 low (read)	21		18		16				ns
(16)	$t_{h((X)D)R}$	(X)D hold time after H1 low (read)	0		0		0		0		ns
(17.1)	$t_{su(RDY)}$	$\overline{RDY}$ setup before H1 high	10		8		8		6		ns
(17.2)	$t_{su(XRDY)}$	$\overline{XRDY}$ setup before H1 high	11		9		9				ns
(18)	$t_{h((X)RDY)}$	$\overline{(X)RDY}$ hold time after H1 high	0		0		0		0		ns
(19)	$t_{d(H1H–(X)RWH)}$	H1 high to (X)R/$\overline{W}$ high (write) delay		13		10		9		7	ns
(20)	$t_{v((X)D)W}$	(X)D valid after H1 low (write)		25		20		17		14	ns
(21)	$t_{h((X)D)W}$	(X)D hold time after H1 high (write)	0‡		0‡		0‡		0‡		ns

‡ Guaranteed by design but not tested

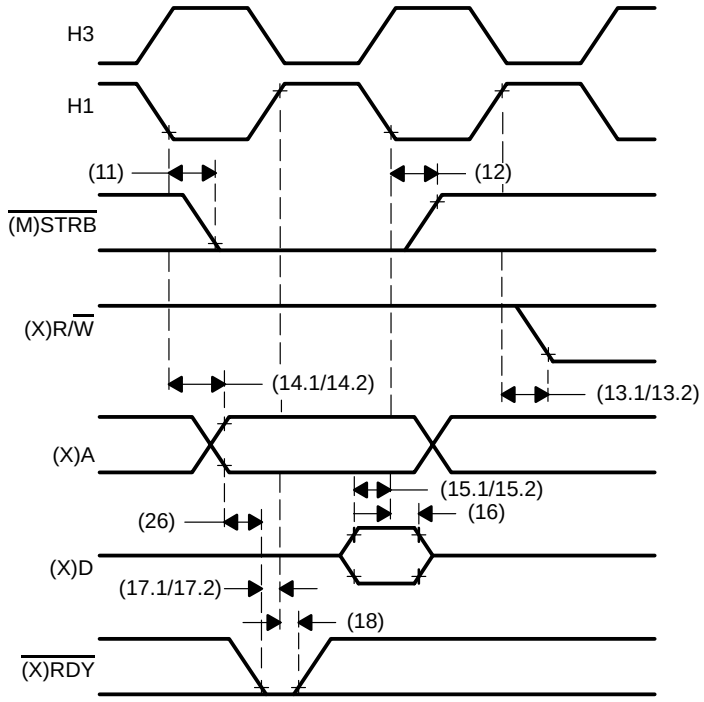
§ For ’C30 PPM, $t_{d(H1L–(M)SL)}$ (max)=7ns

Table 13–13. Timing Parameters for a Memory ($\overline{(M)STRB}$) = 0) Read/Write (Continued)

No.	Name	Description	'30-27 '31-27 Min	'30-27 '31-27 Max	'30-33 '31-33 'LC31 Min	'30-33 '31-33 'LC31 Max	'30-40 '31-40 Min	'30-40 '31-40 Max	'31-50 Min	'31-50 Max	Unit
(22.1)	$t_{d(H1H–A)}$	H1 high to A valid on back-to-back write cycles (write) delay		23		18		15		12	ns
(22.2)	$t_{d(H1H–XA)}$	H1 high to XA valid on back-to-back write cycles (write) delay		32		25		21			ns
(26)	$t_{d(A–(X)RDY)}$	$\overline{(X)RDY}$ delay from A valid delay		10†		8†		7†		6	ns

† Guaranteed from characterization but not tested
‡ Guaranteed by design but not tested
§ For 'C30 PPM, $t_{d(H1L–(M)SL)}$ (max)=7ns

Figure 13–13. Timing for Memory ($\overline{(M)STRB}$ = 0) Read



Note: $\overline{(M)STRB}$ will remain low during back-to-back read operations.

Figure 13–14. Timing for Memory ($\overline{(M)STRB}$ = 0) Write

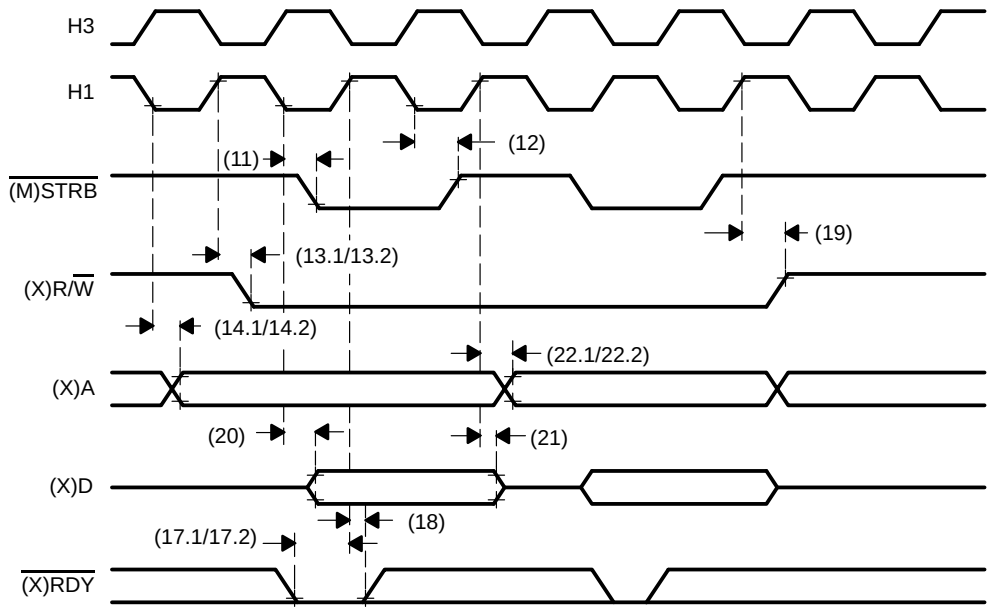


Table 13–14 defines memory read timing parameters for $\overline{IOSTRB}$. The numbers shown in parentheses in Figure 13–15 and Figure 13–16 correspond with those in the No. column of Table 13–14 and Table 13–15.

Table 13–14. Timing Parameters for a Memory ($\overline{IOSTRB}$ = 0) Read

			’C30-27		’C30-33		’C30-40		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(11.1)	$t_{d(H1H–IOSL)}$	H1 high to $\overline{IOSTRB}$ low delay	0†	13	0†	10	0†	9	ns
(12.1)	$t_{d(H1H–IOSH)}$	H1 high to $\overline{IOSTRB}$ high delay	0†	13	0†	10	0†	9	ns
(13.1)	$t_{d(H1L–XRWH)}$	H1 low to XR/$\overline{W}$ high delay	0†	13	0†	10	0†	9	ns
(14.3)	$t_{d(H1L–XA)}$	H1 low to XA valid delay	0†	13	0†	10	0‡	9	ns
(15.3)	$t_{su(XD)R}$	XD setup before H1 high	19		15		13		ns
(16.1)	$t_{h(XD)R}$	XD hold time after H1 high	0		0		0		ns
(17.3)	$t_{su(XRDY)}$	$\overline{XRDY}$ setup before H1 high	11		9		9		ns
(18.1)	$t_{h(XRDY)}$	$\overline{XRDY}$ hold time after H1 high	0		0		0		ns

† Guaranteed by design but not tested

Figure 13–15. Timing for Memory ($\overline{IOSTRB}$ = 0) Read

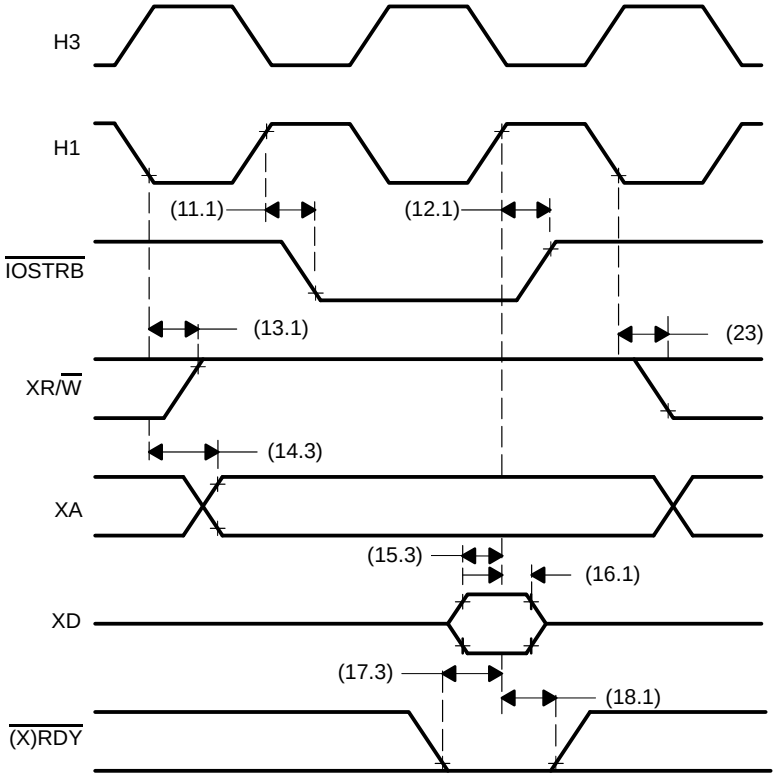


Figure 13–16. Timing for Memory ($\overline{IOSTRB}$ = 0) Write

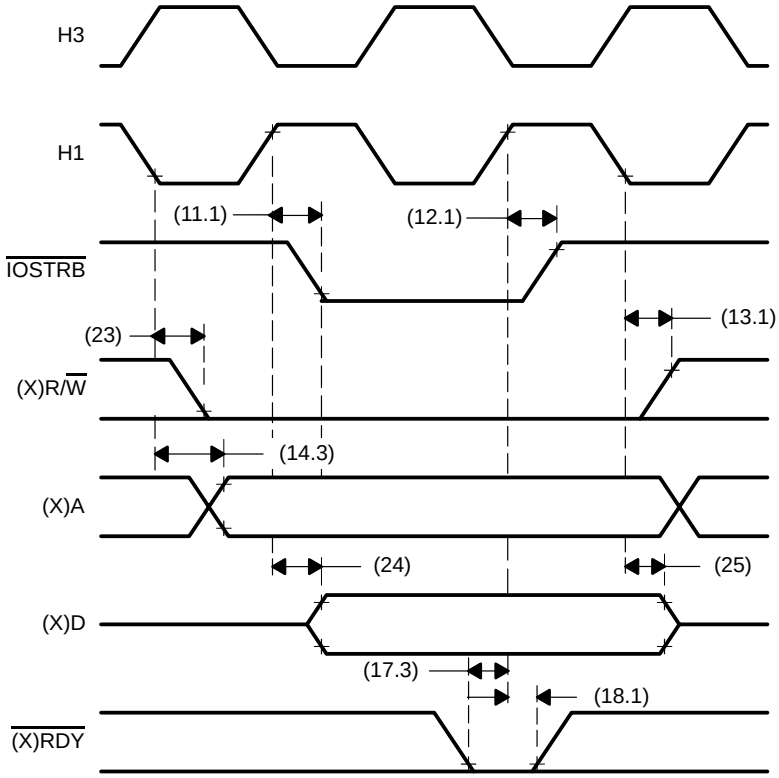


Table 13–15 defines memory write timing parameters for $\overline{IOSTRB}$. The numbers shown in parentheses in Figure 13–15 and Figure 13–16 correspond with those in the No. column of Table 13–14 and Table 13–15.

Table 13–15. Timing Parameters for a Memory ($\overline{IOSTRB}$ = 0) Write

			'C30-27		'C30-33		'C30-40		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(23)	$t_{d(H1L–XRWL)}$	H1 low to XR/$\overline{W}$ low delay	0†	19	0†	15	0†	13	ns
(24)	$t_{v(XD)W}$	XD valid after H1 high		38		30		25	ns
(25)	$t_{h(XD)W}$	XD hold time after H1 low	0		0		0		ns

† Guaranteed by design but not tested

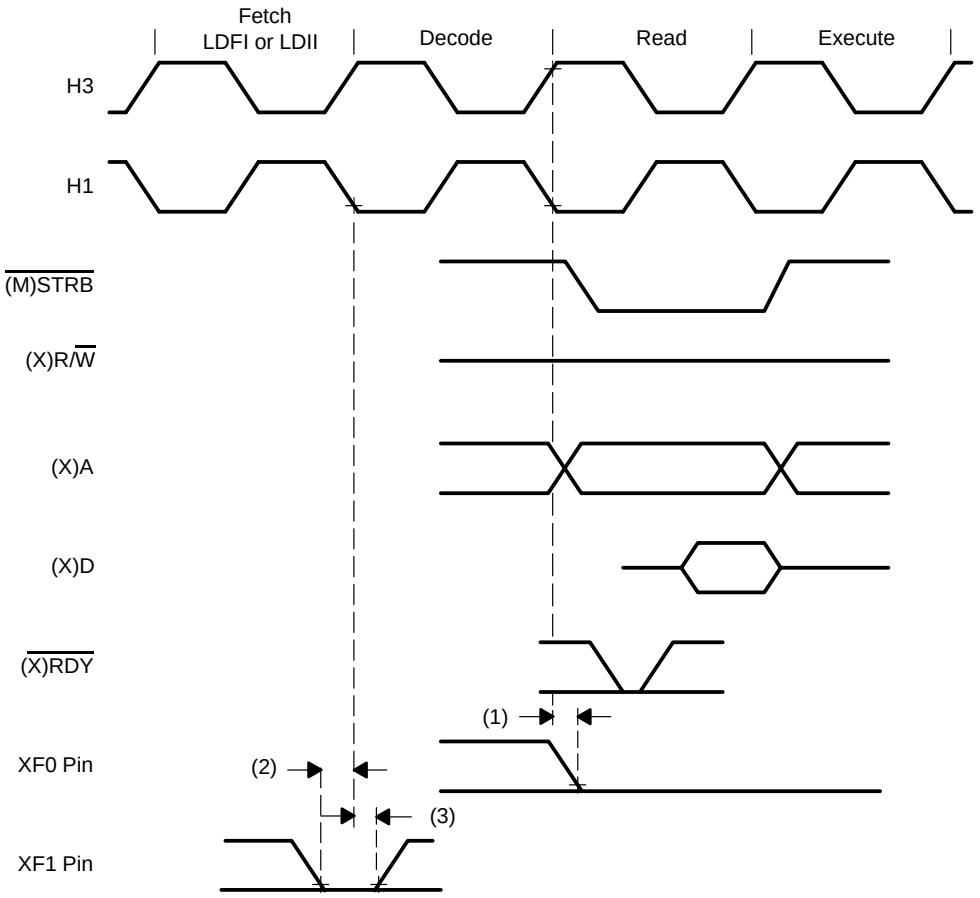
13.5.3 XF0 and XF1 Timing When Executing LDFI or LDII

Table 13–16 defines the timing parameters for XF0 and XF1 during execution of LDFI or LDII. The numbers shown in parentheses in Figure 13–17 correspond with those in the No. column of Table 13–16.

Table 13–16. Timing Parameters for XF0 and XF1 When Executing LDFI or LDII

No.	Name	Description	’C30-27 ’C31-27		’C30-33 ’C31-33 ’LC31		’C30-40 ’C31-40		’C31-50		Unit
			Min	Max	Min	Max	Min	Max	Min	Max	
(1)	$t_{d(H3H–XF0L)}$	H3 high to XF0 low delay		19		15		13		12	ns
(2)	$t_{su(XF1)}$	XF1 setup before H1 low	13		10		9		9		ns
(3)	$t_{h(XF1)}$	XF1 hold time after H1 low	0		0		0		0		ns

Figure 13–17. Timing for XF0 and XF1 When Executing LDFI or LDII



13.5.4 XF0 Timing When Executing STFI and STII

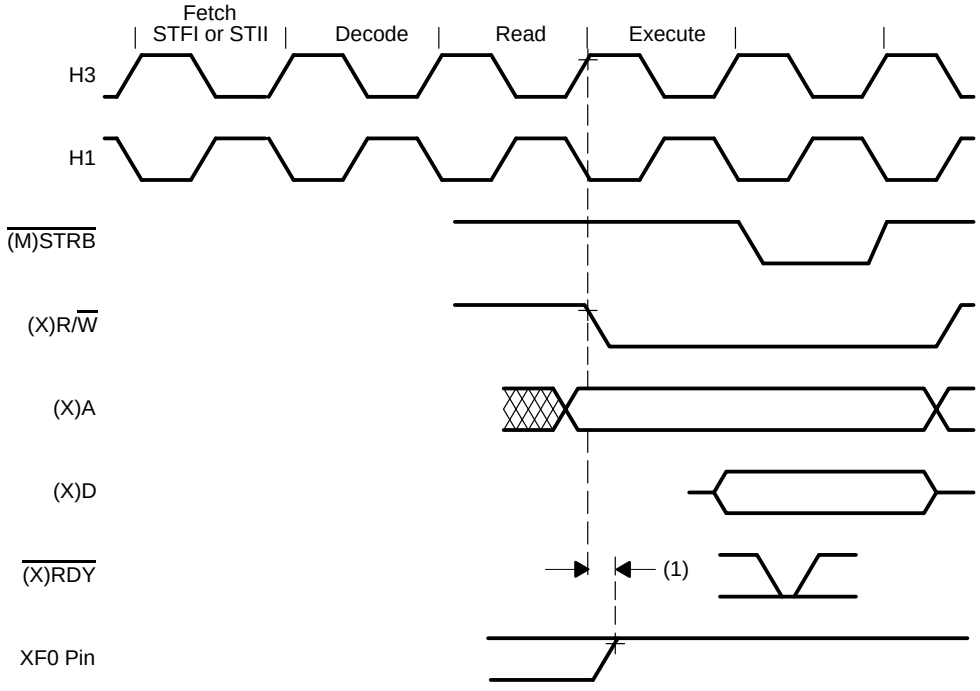
Table 13–17 defines the timing parameters for the XF0 and XF1 pins during execution of STFI or STII. The number shown in parentheses in Figure 13–18 corresponds with the number in the No. column of Table 13–17.

Table 13–17. Timing Parameters for XF0 When Executing STFI or STII

			'C30-27 'C31-27		'C30-33 'C31-33 'LC31		'C30-40 'C31-40		'C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{d(H3H–XF0H)}$	H3 high to XF0 high delay		19		15		13		12	ns

XF0 is always set high at the beginning of the execute phase of the interlock store instruction. When no pipeline conflicts occur, the address of the store is also driven at the beginning of the execute phase of the interlock store instruction. However, if a pipeline conflict prevents the store from executing, the address of the store will not be driven until the store can execute.

Figure 13–18. Timing for XF0 When Executing an STFI or STII



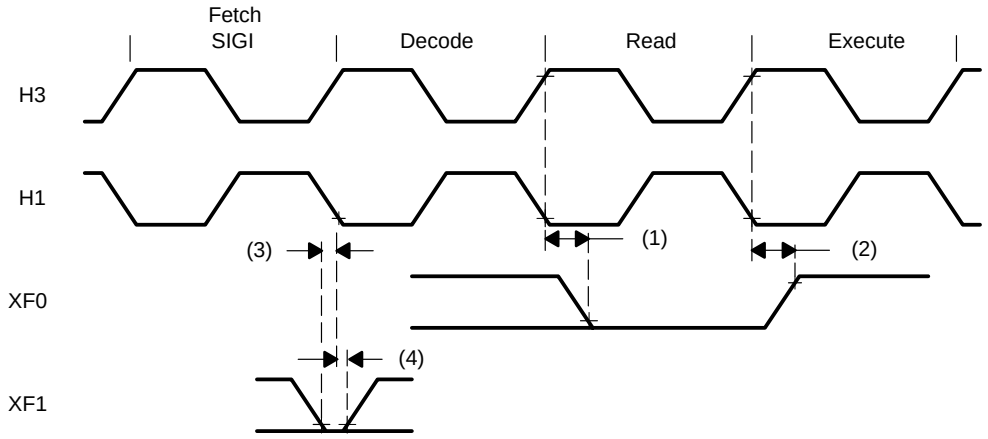
13.5.5 XF0 and XF1 Timing When Executing SIGI

Table 13–18 defines the timing parameters for the XF0 and XF1 pins during execution of SIGI. The numbers shown in parentheses in Figure 13–19 correspond with those in the No. column of Table 13–18.

Table 13–18. Timing Parameters for XF0 and XF1 When Executing SIGI

			’C30-27 ’C31-27		’C30-33 ’C31-33 ’LC31		’C30-40 ’C31-40		’C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{d(H3H–XF0L)}$	H3 high to XF0 low delay		19		15		13		12	ns
(2)	$t_{d(H3H–XF0H)}$	H3 high to XF0 high delay		19		15		13		12	ns
(3)	$t_{su(XF1)}$	XF1 setup before H1 low	13		10		9		9		ns
(4)	$t_{h(XF1)}$	XF1 hold time after H1 low	0		0		0		0		ns

Figure 13–19. Timing for XF0 and XF1 When Executing SIGI



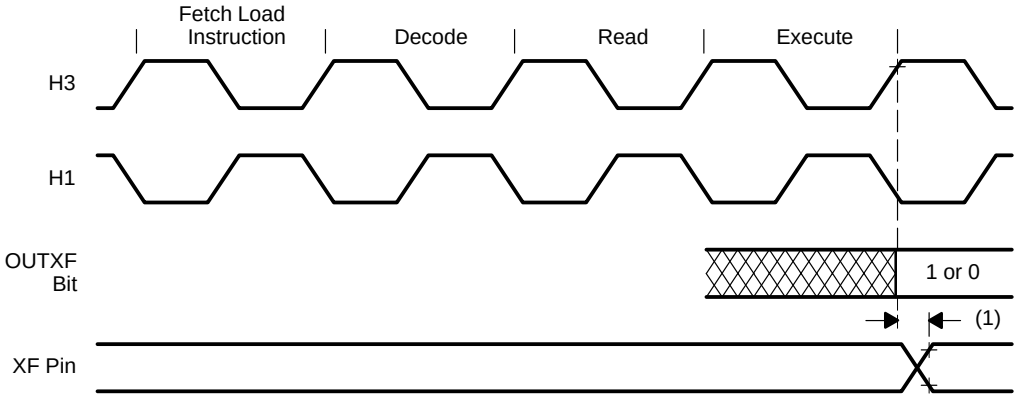
13.5.6 Loading When the XF Pin Is Configured as an Output

Table 13–19 defines the timing parameter for loading the XF register when the XF pin is configured as an output. The number shown in parentheses in Figure 13–20 corresponds with the number in the No. column of Table 13–19.

Table 13–19. Timing Parameters for Loading the XF Register When Configured as an Output Pin

			'C30-27 'C31-27		'C30-33 'C31-33 'LC31		'C30-40 'C31-40		'C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{v(H3H–XF)}$	H3 high to XF valid		19		15		13		12	ns

Figure 13–20. Timing for Loading XF Register When Configured as an Output Pin



13.5.7 Changing the XF Pin From an Output to an Input

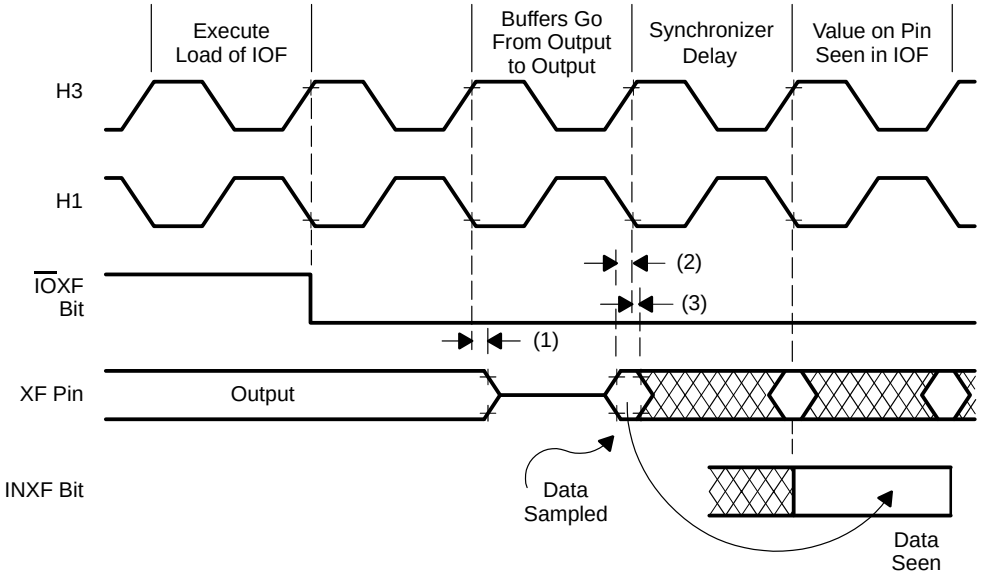
Table 13–20 defines the timing parameters for changing the XF pin from an output pin to an input pin. The numbers shown in parentheses in Figure 13–21 correspond with those in the No. column of Table 13–20.

Table 13–20. Timing Parameters of XF Changing From Output to Input Mode

			'C30-27 'C31-27		'C30-33 'C31-33 'LC31		'C30-40 'C31-40		'C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{h(H3H–XF01)}$	XF hold after H3 high		19		15		13†		12	ns
(2)	$t_{su(XF)}$	XF setup before H1 low	13		10		9		9		ns
(3)	$t_{h(XF)}$	XF hold after H1 low	0		0		0		0		ns

† For 'C30 PPM, $t_{n(H3H–XF01)}$ (max)=14ns

Figure 13–21. Timing for Change of XF From Output to Input Mode



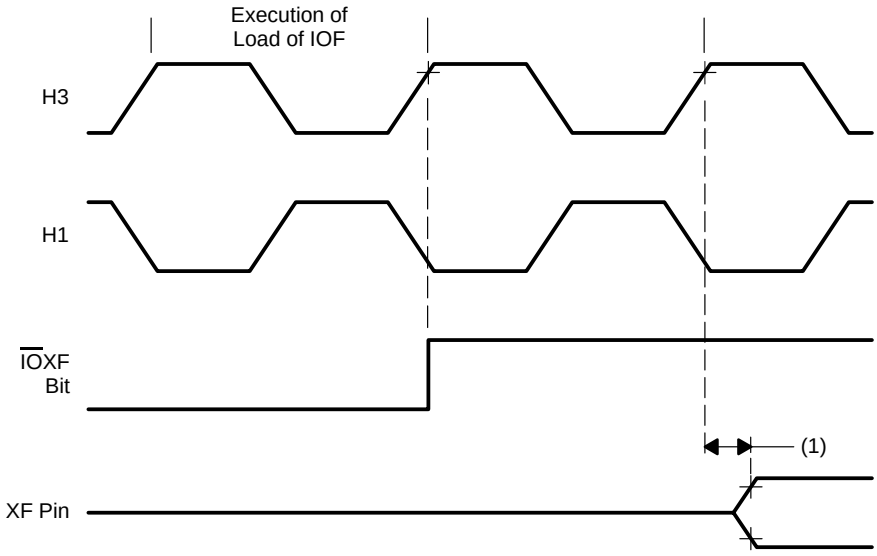
13.5.8 Changing the XF Pin From an Input to an Output

Table 13–21 defines the timing parameter for changing the XF pin from an input pin to an output pin. The number shown in parentheses in Figure 13–22 corresponds with the number in the No. column of Table 13–21.

Table 13–21. Timing Parameters of XF Changing From Input to Output Mode

			’C30-27 ’C31-27		’C30-33 ’C31-33 ’LC31		’C30-40 ’C31-40		’C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{d(H3H–XFIO)}$	H3 high to XF switching from input to output delay		25		20		17		17	ns

Figure 13–22. Timing for Change of XF From Input to Output Mode



13.5.9 Reset Timing

$\overline{\text{RESET}}$ is an asynchronous input that can be asserted at any time during a clock cycle. If the specified timings are met, the exact sequence shown in Figure 13–23 on page 13-48 will occur; otherwise, an additional delay of one clock cycle is possible.

The asynchronous reset signals include XF0/1, CLKX0/1, DX0/1, FSX0/1, CLKR0/1, DR0/1, FSR0/1, and TCLK0/1.

Table 13–22 (’C30) and Table 13–23 (’C31) define the timing parameters for the $\overline{\text{RESET}}$ signal. The numbers shown in parentheses in Figure 13–23 correspond with those in the No. column of Table 13–22 or Table 13–23.

Resetting the device initializes the primary and expansion bus control registers to seven software wait states and therefore results in slow external accesses until these registers are initialized.

Note also that $\overline{\text{HOLD}}$ is an asynchronous input and can be asserted during reset.

Table 13–22. Timing Parameters for $\overline{RESET}$ for the TMS320C30

No.	Name	Description	’C30-27		’C30-33		’C30-40		Unit
			Min	Max	Min	Max	Min	Max	
(1)	$t_{su(RESET)}$	Setup for $\overline{RESET}$ before CLKIN low	28	P†§	10	P†	10	P†§	ns
(2.1)	$t_{d(CLKINH–H1H)}$	CLKIN high to H1 high delay‡	6	20	4	14	2	12	ns
(2.2)	$t_{d(CLKINH–H1L)}$	CLKIN high to H1 low delay‡	6	20	4	14	2	12	ns
(3)	$t_{su(RESETH–H1L)}$	Setup for $\overline{RESET}$ high before H1 low and after 10 H1 clock cycles	13		10		9		ns
(5.1)	$t_{d(CLKINH–H3L)}$	CLKIN high to H3 low delay‡	6	20	4	14	2	12	ns
(5.2)	$t_{d(CLKINH–H3H)}$	CLKIN high to H3 high delay‡	6	20	4	14	2	12	ns
(8)	$t_{dis(H1H–(X)D)}$	H1 high to (X)D disabled (high impedance)		19†		15†		13†	ns
(9)	$t_{dis(H3H–(X)A)}$	H3 high to (X)A disabled (high impedance)		13†		10†		9†	ns
(10)	$t_{d(H3H–CONTROLH)}$	H3 high to control signals high delay		13†		10†		9†	ns
(11)	$t_{d(H1H–RWH)}$	H1 high to R/$\overline{W}$ high delay		13†		10†		9†	ns
(13)	$t_{d(H1H–IACKH)}$	H1 high to $\overline{IACK}$ high delay		13†		10†		9†	ns
(14)	$t_{dis(RESETL–ASYNCH)}$	$\overline{RESET}$ low to asynchronously reset signals disabled (high impedance)		31†		25†		21†	ns

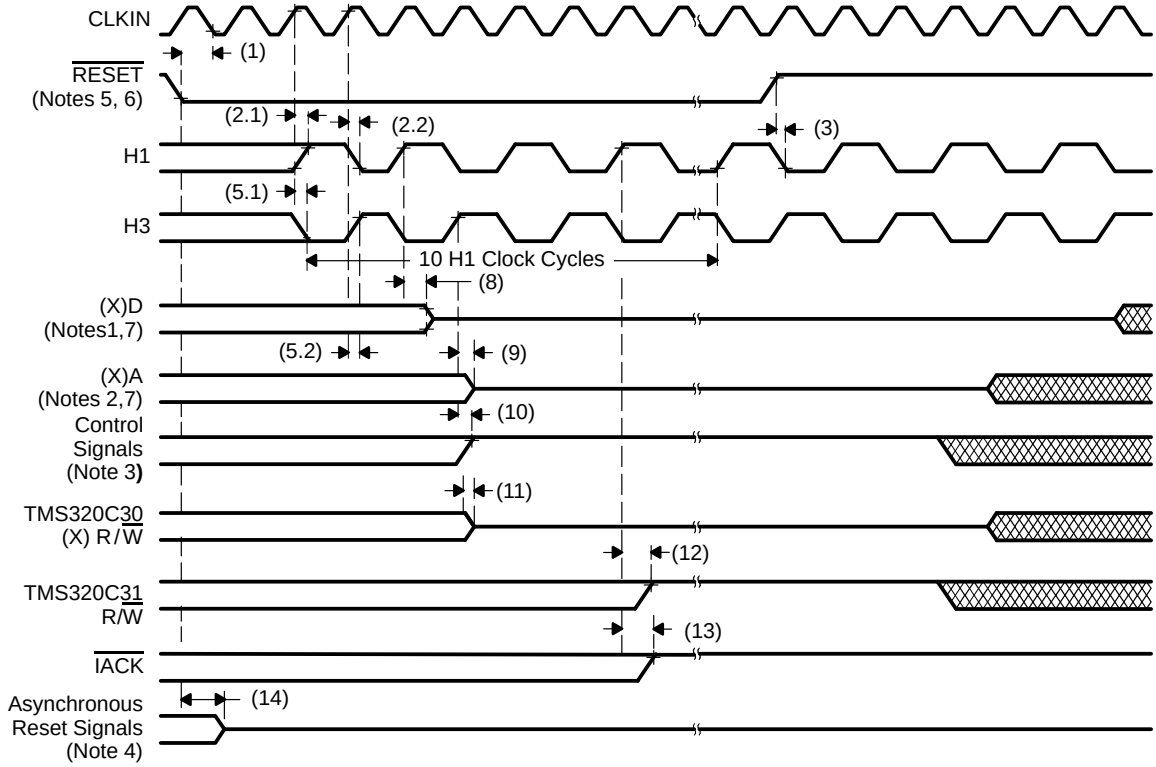
† Characterized but not tested
‡ See Figure 13–24 for temperature dependence for the 33-MHz TMS320C30. See Figure 13–25 for temperature dependence for the 40-MHz TMS320C30.
§ P = $t_{c(CI)}$

Table 13–23. Timing Parameters for $\overline{\text{RESET}}$ for the TMS320C31

No.	Name	Description	'C31-27		'C31-33 'LC31		'C31-40		'C31-50		Unit
			Min	Max	Min	Max	Min	Max	Min	Max	
(1)	$t_{su(RESET)}$	Setup for $\overline{\text{RESET}}$ before CLKIN low	28	P†¶	10	P†¶	10	P†¶	10	P†¶	ns
(2.1)	$t_{d(CLKINH–H1H)}$	CLKIN high to H1 high delay §#	2	12	2	12‡	2	12	2	10	ns
(2.2)	$t_{d(CLKINH–H1L)}$	CLKIN high to H1 low delay §#	2	12	2	12‡	2	12	2	10	ns
(3)	$t_{su(RESETH–H1L)}$	Setup for $\overline{\text{RESET}}$ high before H1 low and after 10 H1 clock cycles	13		10		9		7		ns
(5.1)	$t_{d(CLKINH–H3L)}$	CLKIN high to H3 low delay §#	2	12	2	12‡	2	12	2	10	ns
(5.2)	$t_{d(CLKINH–H3H)}$	CLKIN high to H3 high delay §#	2	12	2	12‡	2	12	2	10	ns
(8)	$t_{dis(H1H–(X)D)}$	H1 high to D disabled (high impedance)		19†		15†		13†		12†	ns
(9)	$t_{dis(H3H–(X)A)}$	H3 high to A disabled (high impedance)		13†		10†		9†		8†	ns
(10)	$t_{d(H3H–CONTROLH)}$	H3 high to control signals high delay		13†		10†		9†		8†	ns
(12)	$t_{d(H1H–RWH)}$	H1 high to R/$\overline{\text{W}}$ high delay		13†		10†		9†		8†	ns
(13)	$t_{d(H1H–IACKH)}$	H1 high to $\overline{\text{IACK}}$ high delay		13†		10†		9†		8†	ns
(14)	$t_{dis(RESETL–ASYNCH)}$	$\overline{\text{RESET}}$ low to asynchronously reset signals disabled (high impedance)		31†		25†		21†		17†	ns

† Characterized but not tested
‡ 14 ns for the extended temperature 'C31-33
§ See Figure 13–25 for temperature dependence for the TMS320C31-27, TMS320C31-33, and the extended-temperature TMS320C31-33.
¶ P = $t_{c(CI)}$
# See Figure 13–26 for temperature dependence for the TMS320C31-50.

Figure 13–23. Timing for $\overline{RESET}$



Notes: 1) (X)D includes D31–D0 and XD31–XD0.

2) (X)A includes A23–A0 and XA12–XA0.

3) Control signals include $\overline{STRB}$, $\overline{MSTRB}$, and $\overline{IOSTRB}$.

4) Asynchronously reset signals include XF0/1, CLKX0/1, DX0/1, FSX0/1, CLKR0/1, DR0/1, FSR0/1, and TCLK0/1.

5) $\overline{RESET}$ is an asynchronous input and can be asserted at any point during a clock cycle. If the specified timings are met, the exact sequence shown will occur; otherwise, an additional delay of one clock cycle is possible.

6) Note that the R/$\overline{W}$ and XR/$\overline{W}$ outputs are placed in a high-impedance state during reset and can be provided with a resistive pull-up, nominally 18–22 kΩ, if undesirable spurious writes could be caused when these outputs go low.

7) In microprocessor mode, the reset vector is fetched twice, with seven software wait states each time. In microcomputer mode, the reset vector is fetched twice, with no software wait states.

Figure 13–24. CLKIN to H1/H3 as a Function of Temperature

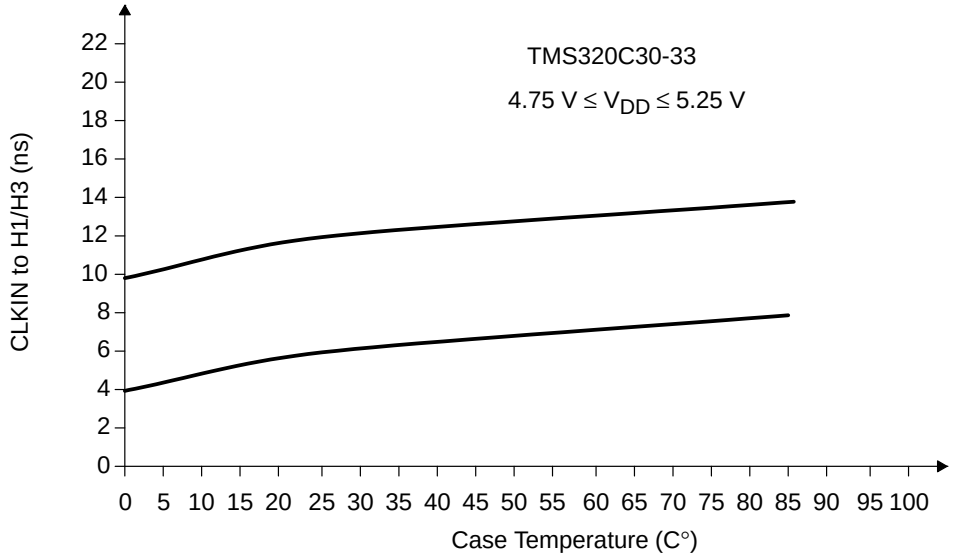


Figure 13–25. CLKIN to H1/H3 as a Function of Temperature

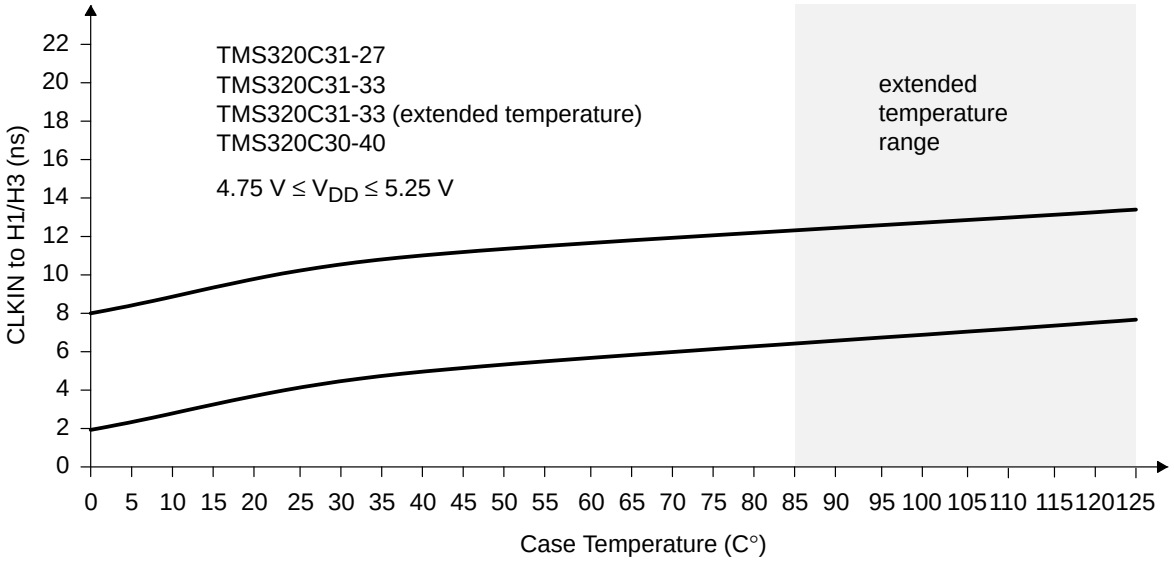
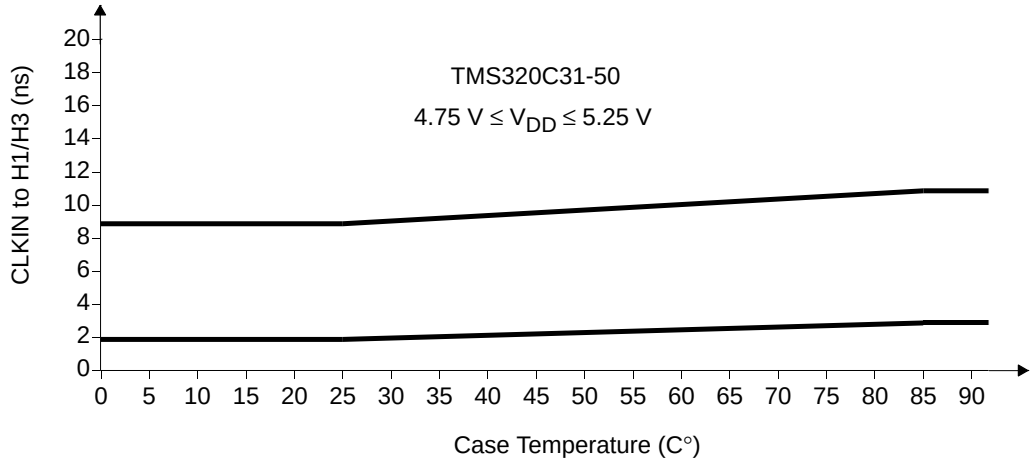


Figure 13–26. CLKIN to H1/H3 as a Function of Temperature



13.5.10 $\overline{\text{SHZ}}$ Pin Timing

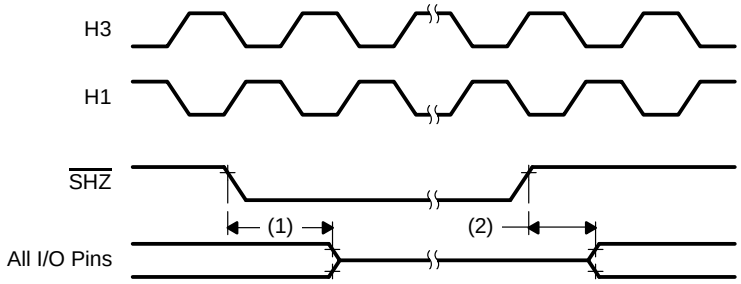
Table 13–24 defines the timing parameters for the $\overline{\text{SHZ}}$ pin. The numbers shown in parentheses in Figure 13–27 correspond with those in the No. column of Table 13–24.

Table 13–24. Timing Parameters for the $\overline{\text{SHZ}}$ Pin

			'C30 'C31 'LC31		
No.	**Name**	**Description**	**Min**	**Max**	**Unit**
(1)	$t_{dis(SHZ)}$	$\overline{\text{SHZ}}$ low to all O, I/O pins disabled (high impedance)	0†	2P†‡	ns
(2)	$t_{en(SHZ)}$	$\overline{\text{SHZ}}$ high to all O, I/O pins enabled (active)	0†	2P†‡	ns

† Characterized but not tested
‡ P = $t_{c(CI)}$

Figure 13–27. Timing for $\overline{\text{SHZ}}$ Pin



Note: Enabling $\overline{\text{SHZ}}$ destroys TMS320C3x register and memory contents. Assert $\overline{\text{SHZ}}$ = 1 and reset the TMS320C3x to restore it to a known condition.

13.5.11 Interrupt Response Timing

Table 13–25 defines the timing parameters for the $\overline{\text{INT}}$ signals. The numbers shown in parentheses in Figure 13–28 correspond with those in the No. column of Table 13–25.

Table 13–25. Timing Parameters for $\overline{INT3}$–$\overline{INT0}$

No.	Name	Description	'C30-27 'C31-27 Min	'C30-27 'C31-27 Max	'C30-33 'C31-33 'LC31 Min	'C30-33 'C31-33 'LC31 Max	'C30-40 'C31-40 Min	'C30-40 'C31-40 Max	'C31-50 Min	'C31-50 Max	Unit
(1)	$t_{su(INT)}$	$\overline{\text{INT3}}$–$\overline{\text{INT0}}$ setup before H1 low	19		15		13		10		ns
(2)	$t_{w(INT)}$	Interrupt pulse duration to guarantee only one interrupt	P	2P†‡	P	2P†‡	P	2P†‡	P	2P†‡	ns

† Characterized but not tested
‡ P = $t_{c(H)}$

The interrupt ($\overline{\text{INT}}$) pins are asynchronous inputs that can be asserted at any time during a clock cycle. The TMS320C3x interrupts are level-sensitive, not edge-sensitive. Interrupts are detected on the falling edge of H1. Therefore, interrupts must be set up and held to the falling edge of H1 for proper detection. The CPU and DMA respond to detected interrupts on instruction fetch boundaries only.

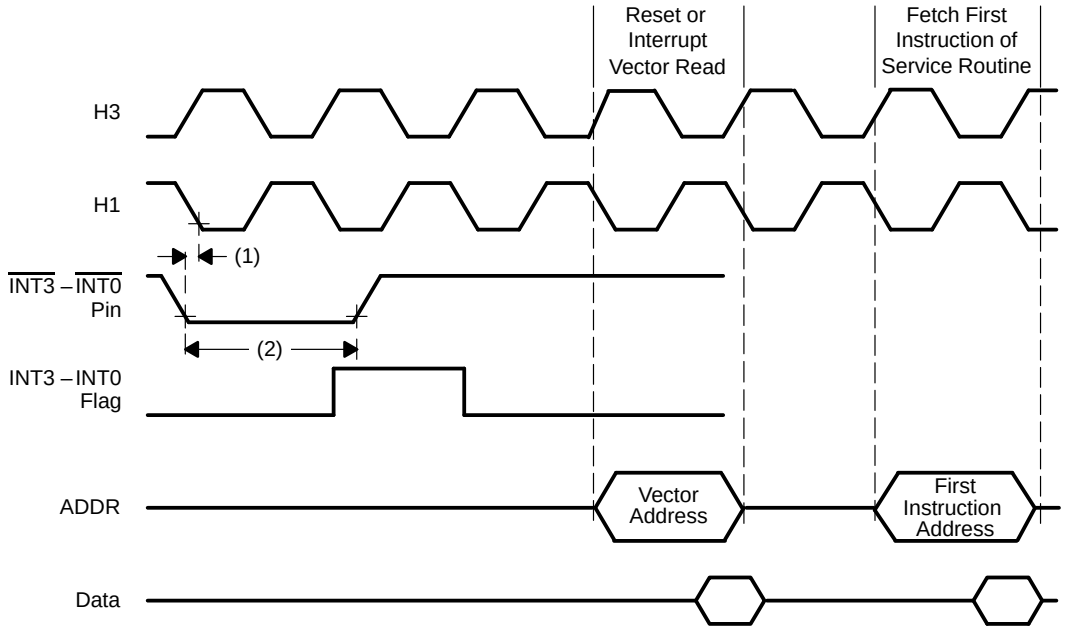
For the processor to recognize only one interrupt on a given input, an interrupt pulse must be set up and held to:

- A minimum of one H1 falling edge, and
- No more than two H1 falling edges.

The TMS320C3x can accept an interrupt from the same source every two H1 clock cycles.

If the specified timings are met, the exact sequence shown in Figure 13–28 will occur; otherwise, an additional delay of one clock cycle is possible.

Figure 13–28. Timing for $\overline{INT3}$–$\overline{INT0}$ Response



13.5.12 Interrupt Acknowledge Timing

The $\overline{\text{IACK}}$ output goes active on the first half-cycle (HI rising) of the decode phase of the IACK instruction and goes inactive at the first half-cycle (HI rising) of the read phase of the IACK instruction.

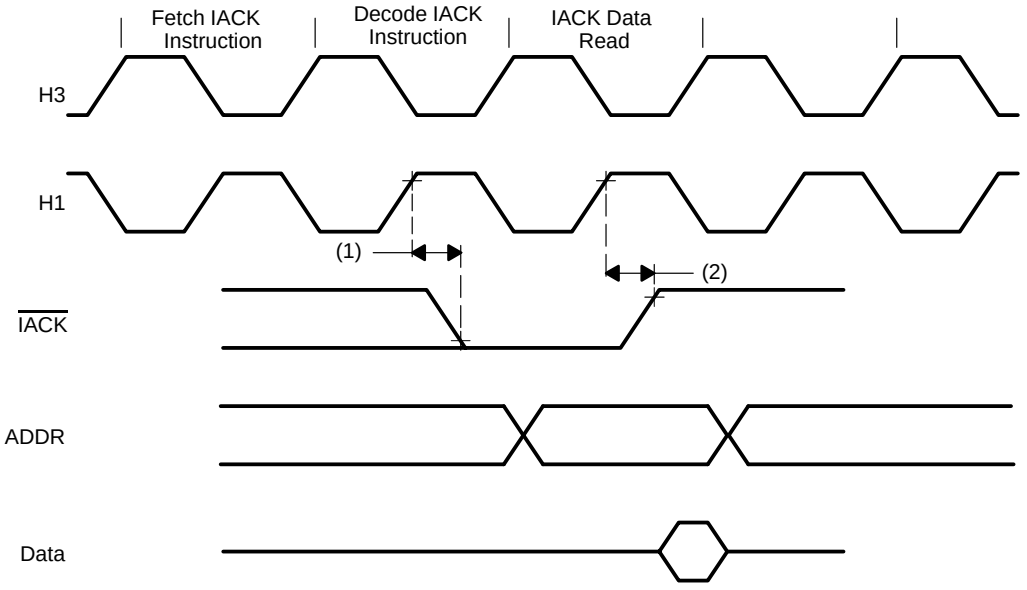
Table 13–26 defines the timing parameters for the $\overline{\text{IACK}}$ signal. The numbers shown in parentheses in Figure 13–29 correspond with those in the No. column of Table 13–26.

Table 13–26. Timing Parameters for $\overline{\text{IACK}}$

			'C30-27 'C31-27		'C30-33 'C31-33 'LC31		'C30-40 'C31-40		'C31-50		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{d(H1H–IACKL)}$	H1 high to $\overline{\text{IACK}}$ low delay		13		10		9		7	ns
(2)	$t_{d(H1H–IACKH)}$	H1 high to $\overline{\text{IACK}}$ high delay		13		10		9		7	ns

Note: The $\overline{\text{IACK}}$ output is active for the entire duration of the bus cycle and is therefore extended if the bus cycle utilizes wait states.

Figure 13–29. Timing for $\overline{\text{IACK}}$

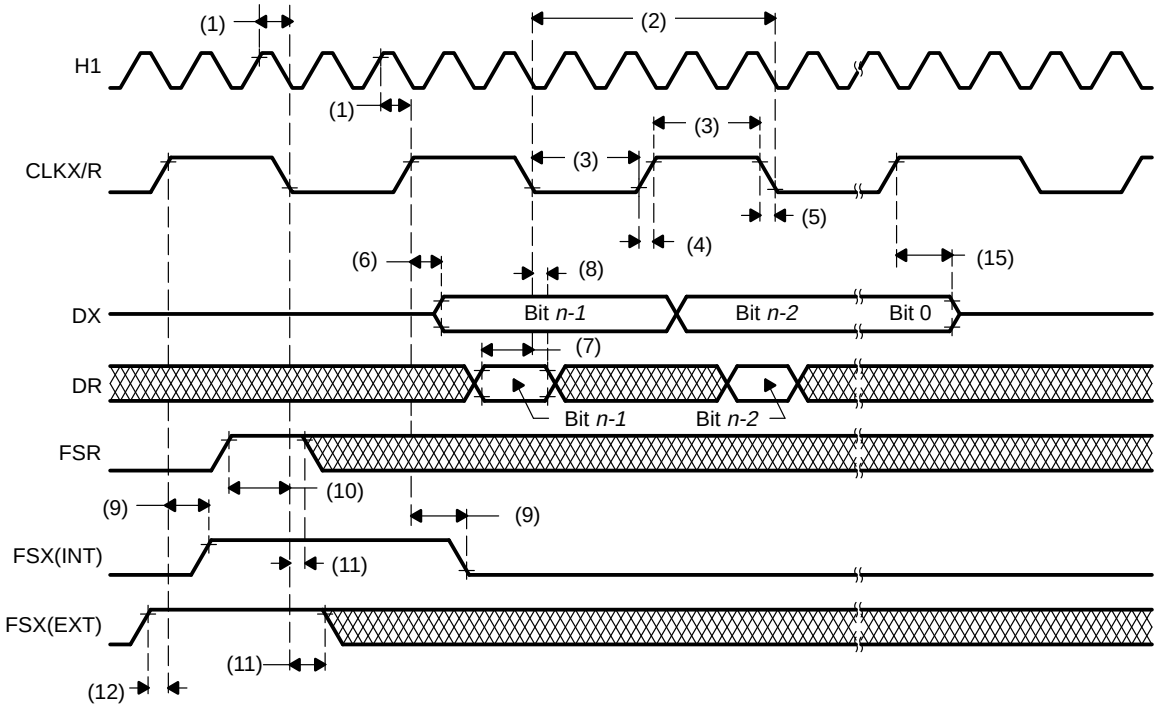


13.5.13 Data Rate Timing Modes

Unless otherwise indicated, the data rate timings shown in Figure 13–30 and Figure 13–31 are valid for all serial port modes, including handshake. For a functional description of serial port operation, refer to subsection 8.2.12 on page 8-30.

Table 13–27 defines the serial port timing parameters for eight ’C3x devices. The numbers shown in parentheses in Figure 13–30 and Figure 13–31 correspond with those in the No. column of Table 13–27.

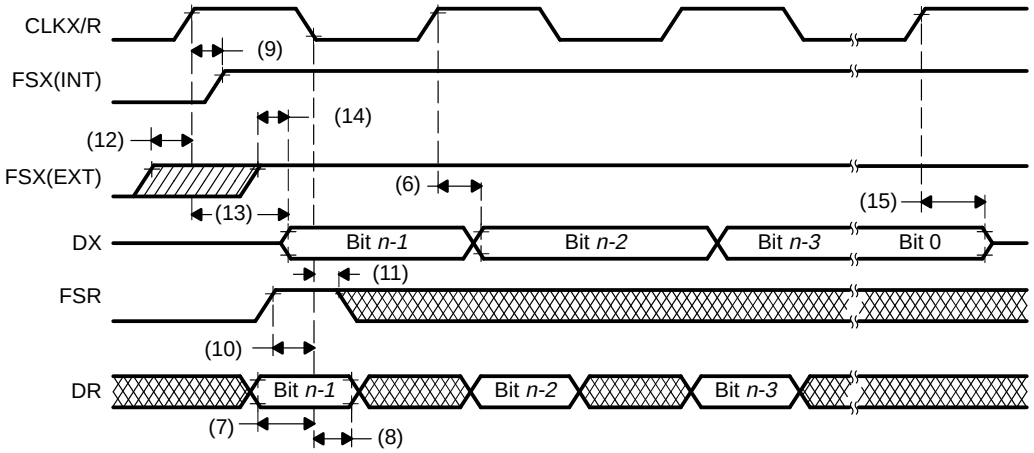
Figure 13–30. Timing for Fixed Data Rate Mode



Notes: 1) Timing diagrams show operations with CLKXP = CLKRP = FSXP = FSRP = 0.

2) Timing diagrams depend on the length of the serial port word, where n = 8, 16, 24, or 32 bits, respectively.

Figure 13–31. Timing for Variable Data Rate Mode



Notes: 1) Timing diagrams show operation with CLKXP = CLKRP = FSXP = FSRP = 0.

2) Timing diagrams depend on the length of the serial port word, where n = 8, 16, 24, or 32 bits, respectively.

3) The timings that are not specified expressly for the variable data rate mode are the same as those that are specified for the fixed data rate mode.

Table 13–27. Serial-Port Timing Parameters

No.	Name	Description		TMS320C30-27/TMS320C31-27 Min	TMS320C30-27/TMS320C31-27 Max	Unit
(1)	$t_{d(H1–SCK)}$	H1 high to internal CLKX/R delay			19	ns
(2)	$t_{c(SCK)}$	CLKX/R cycle time	CLKX/R ext	$t_{c(H)}$x2.6†		ns
			CLKX/R int	$t_{c(H)}$x2	$t_{c(H)}$x2^{32}‡	
(3)	$t_{w(SCK)}$	CLKX/R high/low pulse duration	CLKX/R ext	$t_{c(H)}$+12†		ns
			CLKX/R int	[$t_{c(SCK)}$/2]–15	[$t_{c(SCK)}$/2]+5	
(4)	$t_{r(SCK)}$	CLKX/R rise time			10†	ns
(5)	$t_{f(SCK)}$	CLKX/R fall time			10†	ns
(6)	$t_{d(DX)}$	CLKX to DX valid delay	CLKX ext		44	ns
			CLKX int		25	
(7)	$t_{su(DR)}$	DR setup before	CLKR ext	13		ns
		CLKR low	CLKR int	31		
(8)	$t_{h(DR)}$	DR hold from	CLKR ext	13		ns
		CLKR low	CLKR int	0		
(9)	$t_{d(FSX)}$	CLKX to internal	CLKX ext		40	ns
		FSX high/low delay	CLKX int		21	
(10)	$t_{su(FSR)}$	FSR setup before CLKR low	CLKR ext	13		ns
			CLKR int	13		
(11)	$t_{h(FS)}$	FSX/R input hold from CLKX/R low	CLKX/R ext	13		ns
			CLKX/R int	0		
(12)	$t_{su(FSX)}$	External FSX setup before CLKX	CLKX ext	–[$t_{c(H)}$–8]	[$t_{c(SCK)}$/2]–10‡	ns
			CLKX int	–[$t_{c(H)}$–21]	$t_{c(SCK)}$/2‡	
(13)	$t_{d(CH–DX)V}$	CLKX to first DX bit, FSX precedes CLKX high delay	CLKX ext		45	ns
			CLKX int		26	
(14)	$t_{d(FSX–DX)V}$	FSX to first DX bit, CLKX precedes FSX delay			45	ns
(15)	$t_{d(DXZ)}$	CLKX high to DX high impedance following last data bit delay			25†	ns

† Guaranteed by design but not tested
‡ Not tested

Table 13–27. Serial-Port Timing Parameters (Continued)

No.	Name	Description		TMS320C30-33/TMS320C31-33/ TMS320LC31 Min	Max	Unit
(1)	$t_{d(H1–SCK)}$	H1 high to internal CLKX/R delay			15	ns
(2)	$t_{c(SCK)}$	CLKX/R cycle time	CLKX/R ext	$t_{c(H)}$x2.6†		ns
			CLKX/R int	$t_{c(H)}$x2	$t_{c(H)}$x2^{32}‡	
(3)	$t_{w(SCK)}$	CLKX/R high/low pulse duration	CLKX/R ext	$t_{c(H)}$+12†		ns
			CLKX/R int	[$t_{c(SCK)}$/2]–15	[$t_{c(SCK)}$/2]+5	
(4)	$t_{r(SCK)}$	CLKX/R rise time			8†	ns
(5)	$t_{f(SCK)}$	CLKX/R fall time			8†	ns
(6)	$t_{d(DX)}$	CLKX to DX valid delay	CLKX ext		35	ns
			CLKX int		20	
(7)	$t_{su(DR)}$	DR setup before CLKR low	CLKR ext CLKR int	10 25		ns
(8)	$t_{h(DR)}$	DR hold from CLKR low	CLKR ext CLKR int	10 0		ns
(9)	$t_{d(FSX)}$	CLKX to internal FSX high/low delay	CLKX ext CLKX int		32 17	ns
(10)	$t_{su(FSR)}$	FSR setup before CLKR low	CLKR ext CLKR int	10 10		ns
(11)	$t_{h(FS)}$	FSX/R input hold from CLKX/R low	CLKX/R ext CLKX/R int	10 0		ns
(12)	$t_{su(FSX)}$	External FSX setup before CLKX	CLKX ext CLKX int	–[$t_{c(H)}$–8] [$t_{c(H)}$–21]	[$t_{c(SCK)}$/2]–10‡ $t_{c(SCK)}$/2‡	ns
(13)	$t_{d(CH–DX)V}$	CLKX to first DX bit, FSX precedes CLKX high delay	CLKX ext CLKX int		36 21	ns
(14)	$t_{d(FSX–DX)V}$	FSX to first DX bit, CLKX precedes FSX delay			36	ns
(15)	$t_{d(DXZ)}$	CLKX high to DX high impedance following last data bit delay			20†	ns

† Guaranteed by design but not tested
‡ Not tested

Table 13–27. Serial-Port Timing Parameters (Continued)

No.	Name	Description		TMS320C30-40/TMS320C31-40 Min	Max	Unit
(1)	$t_{d(H1–SCK)}$	H1 high to internal CLKX/R delay			13	ns
(2)	$t_{c(SCK)}$	CLKX/R cycle time	CLKX/R ext	$t_{c(H)}$x2.6†		ns
			CLKX/R int	$t_{c(H)}$x2	$t_{c(H)}$x2^{32}‡	
(3)	$t_{w(SCK)}$	CLKX/R high/low pulse duration	CLKX/R ext CLKX/R int	$t_{c(H)}$+10† [$t_{c(SCK)}$/2]–5	[$t_{c(SCK)}$/2]+5	ns
(4)	$t_{r(SCK)}$	CLKX/R rise time			7†	ns
(5)	$t_{f(SCK)}$	CLKX/R fall time			7†	ns
(6)	$t_{d(DX)}$	CLKX to DX valid delay	CLKX ext CLKX int		30 17	ns
(7)	$t_{su(DR)}$	DR setup before	CLKR ext	9		ns
		CLKR low	CLKR int	21		
(8)	$t_{h(DR)}$	DR hold from	CLKR ext	9		ns
		CLKR low	CLKR int	0		
(9)	$t_{d(FSX)}$	CLKX to internal FSX high/low delay	CLKX ext CLKX int		27 15	ns
(10)	$t_{su(FSR)}$	FSR setup before CLKR low	CLKR ext CLKR int	9 9		ns
(11)	$t_{h(FS)}$	FSX/R input hold from CLKX/R low	CLKX/R ext CLKX/R int	9 0		ns
(12)	$t_{su(FSX)}$	External FSX setup before CLKX	CLKX ext CLKX int	–[$t_{c(H)}$–8] –[$t_{c(H)}$–21]	[$t_{c(SCK)}$/2]–10‡ $t_{c(SCK)}$/2‡	ns
(13)	$t_{d(CH–DX)V}$	CLKX to first DX bit, FSX precedes CLKX high delay	CLKX ext CLKX int		30 18	ns
(14)	$t_{d(FSX–DX)V}$	FSX to first DX bit, CLKX precedes FSX delay			30	ns
(15)	$t_{d(DXZ)}$	CLKX high to DX high impedance following last data bit delay			17†	ns

† Guaranteed by design but not tested
‡ Not tested

Table 13–27. Serial-Port Timing Parameters (Continued)

No.	Name	Description		TMS320C31-50 Min	TMS320C31-50 Max	Unit
(1)	$t_{d(H1\text{-}SCK)}$	H1 high to internal CLKX/R delay			10	ns
(2)	$t_{c(SCK)}$	CLKX/R cycle time	CLKX/R ext CLKX/R int	$t_{c(H)} \times 2.6$† $t_{c(H)} \times 2$	$t_{c(H)} \times 2^{32}$‡	ns
(3)	$t_{w(SCK)}$	CLKX/R high/low pulse duration	CLKX/R ext CLKX/R int	$t_{c(H)}+10$† $[t_{c(SCK)}/2]-5$	$[t_{c(SCK)}/2]+5$	ns
(4)	$t_{r(SCK)}$	CLKX/R rise time			6†	ns
(5)	$t_{f(SCK)}$	CLKX/R fall time			6†	ns
(6)	$t_{d(DX)}$	CLKX to DX valid delay	CLKX ext CLKX int		24 16	ns
(7)	$t_{su(DR)}$	DR setup before CLKR low	CLKR ext CLKR int	9 17		ns
(8)	$t_{h(DR)}$	DR hold from CLKR low	CLKR ext CLKR int	7 0		ns
(9)	$t_{d(FSX)}$	CLKX to internal FSX high/low delay	CLKX ext CLKX int		22 15	ns
(10)	$t_{su(FSR)}$	FSR setup before CLKR low	CLKR ext CLKR int	7 7		ns
(11)	$t_{h(FS)}$	FSX/R input hold from CLKX/R low	CLKX/R ext CLKX/R int	7 0		ns
(12)	$t_{su(FSX)}$	External FSX setup before CLKX	CLKX ext CLKX int	$-[t_{c(H)}-8]$ $-[t_{c(H)}-21]$	$[t_{c(SCK)}/2]-10$‡ $t_{c(SCK)}/2$‡	ns
(13)	$t_{d(CH\text{-}DX)V}$	CLKX to first DX bit, FSX precedes CLKX high delay	CLKX ext CLKX int		24 14	ns
(14)	$t_{d(FSX\text{-}DX)V}$	FSX to first DX bit, CLKX precedes FSX delay			24	ns
(15)	$t_{d(DXZ)}$	CLKX high to DX high impedance following last data bit delay			14†	ns

† Assured by design but not tested
‡ Not tested

13.5.14 $\overline{\text{HOLD}}$ Timing

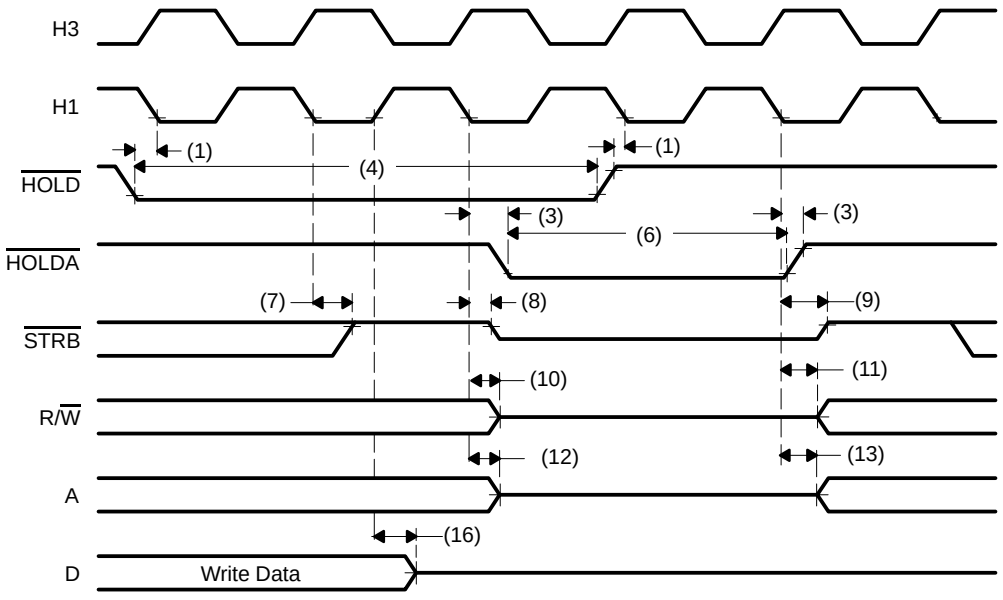
$\overline{\text{HOLD}}$ is an asynchronous input that can be asserted at any time during a clock cycle. If the specified timings are met, the exact sequence shown in Figure 13–32 will occur; otherwise, an additional delay of one clock cycle is possible.

Table 13–28 defines the timing parameters for the $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ signals. The numbers shown in parentheses in Figure 13–32 correspond with those in the No. column of Table 13–28.

The NOHOLD bit of the primary bus control register (see subsection 7.1.1 on page 7-3) overrides the $\overline{\text{HOLD}}$ signal. When this bit is set, the device comes out of hold and prevents future hold cycles.

Asserting $\overline{\text{HOLD}}$ prevents the processor from accessing the primary bus. Program execution continues until a read from or a write to the primary bus is requested. In certain circumstances, the first write will be pending, thus allowing the processor to continue until a second write is encountered.

Figure 13–32. Timing for $\overline{\text{HOLD}}$/$\overline{\text{HOLDA}}$



Note: $\overline{\text{HOLDA}}$ will go low in response to $\overline{\text{HOLD}}$ going low and will continue to remain low until one H1 cycle after $\overline{\text{HOLD}}$ goes back high, as shown in Figure 13–32.

Table 13–28. Timing Parameters for $\overline{HOLD}$/$\overline{HOLDA}$

No.	Name	Description	'30-27 '31-27 Min	'30-27 '31-27 Max	'30-33 '31-33 'LC31 Min	'30-33 '31-33 'LC31 Max	'30-40 '31-40 Min	'30-40 '31-40 Max	'31-50 Min	'31-50 Max	Unit
(1)	$t_{su(HOLD)}$	$\overline{HOLD}$ setup before H1 low	19		15		13		10		ns
(3)	$t_{v(HOLDA)}$	$\overline{HOLDA}$ valid after H1 low	0‡	14	0‡	10	0‡	9	0‡	7	ns
(4)	$t_{w(HOLD}$§$)$	$\overline{HOLD}$ low duration	$2t_{c(H)}$		$2t_{c(H)}$		$2t_{c(H)}$		$2t_{c(H)}$		ns
(6)	$t_{w(HOLDA)}$	$\overline{HOLDA}$ low duration	t_{cH}-5†		t_{cH}-5†		t_{cH}-5†		t_{cH}–5†		ns
(7)	$t_{d(H1L–SH)H)}$	H1 low to $\overline{STRB}$ high for a $\overline{HOLD}$ delay	0‡	13	0‡	10	0‡	9	0‡	7	ns
(8)	$t_{dis(H1L–S)}$	H1 low to $\overline{STRB}$ disabled (high-impedance state)	0‡	13†	0‡	10†	0‡	9†	0‡	8†	ns
(9)	$t_{en(H1L–S)}$	H1 low to $\overline{STRB}$ enabled (active)	0‡	13	0‡	10	0‡	9	0‡	7	ns
(10)	$t_{dis(H1L–RW)}$	H1 low to R/$\overline{W}$ disabled (high-impedance state)	0‡	13†	0‡	10†	0‡	9†	0‡	8†	ns
(11)	$t_{en(H1L–RW)}$	H1 low to R/$\overline{W}$ enabled (active)	0‡	13	0‡	10	0‡	9	0‡	7	ns
(12)	$t_{dis(H1L–A)}$	H1 low to address disabled (high-impedance state)	0‡	13†	0‡	10†	0‡		0‡	8†	ns
(13)	$t_{en(H1L–A)}$	H1 low to address enabled (valid)	0‡	19	0‡	15	0‡	13	0‡	12	ns
(16)	$t_{dis(H1H–D)}$	H1 high to data disabled (high-impedance state)	0‡	13†	0‡	10†	0‡	9†	0‡	8†	ns

† Characterized but not tested
‡ Not tested
§ $\overline{HOLD}$ is an asynchronous input and can be asserted at any point during a clock cycle. If the specified timings are met, the exact sequence shown will occur; otherwise, an additional delay of one clock cycle is possible.

13.5.15 General-Purpose I/O Timing

Peripheral pins include CLKX0/1, CLKR0/1, DX0/1, DR0/1, FSX0/1, FSR0/1, and TCLK0/1. The contents of the internal control registers associated with each peripheral define the modes for these pins.

13.5.15.1 Peripheral Pin I/O Timing

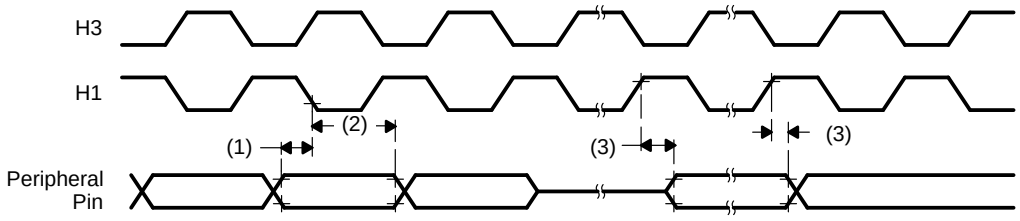
Table 13–29 defines peripheral pin general-purpose I/O timing parameters. The numbers shown in parentheses in Figure 13–33 correspond with those in the No. column of Table 13–29.

Table 13–29. Timing Parameters for Peripheral Pin General-Purpose I/O

			'C30-27 'C31-27		**'C30-33 'C31-33 'LC31**		**'C30-40 'C31-40**		**'C31-50**		
No.	**Name**	**Description**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{su(GPIOH1L)}$	General-purpose input setup before H1 low	15		12		10		9		ns
(2)	$t_{h(GPIOH1L)}$	General-purpose input hold time after H1 low	0		0		0		0		ns
(3)	$t_{d(GPIOH1H)}$	General-purpose output delay after H1 high		19		15		13		10	ns

Note: Peripheral pins include CLKX0/1, CLKR0/1, DX0/1, DR0/1, FSX0/1, FSR0/1, and TCLK0/1. The modes of these pins are defined by the contents of internal control registers associated with each peripheral.

Figure 13–33. Timing for Peripheral Pin General-Purpose I/O



13.5.15.2 Changing the Peripheral Pin I/O Modes

Table 13–30 and Table 13–31 show the timing parameters for changing the peripheral pin from a general-purpose output pin to a general-purpose input pin and vice versa. The numbers shown in parentheses in Figure 13–34 and Figure 13–35 correspond to those shown in the No. column of Table 13–30 and Table 13–31, respectively.

Table 13–30. Timing Parameters for Peripheral Pin Changing From General-Purpose Output to Input Mode

No.	Name	Description	'30-27 '31-27 Min	'30-27 '31-27 Max	'30-33 '31-33 'LC31 Min	'30-33 '31-33 'LC31 Max	'30-40 '31-40 Min	'30-40 '31-40 Max	'31-50 Min	'31-50 Max	Unit
(1)	$t_{h(H3H)}$	Hold after H1 high		19		15		13		10	ns
(2)	$t_{su(GPIOH1L}$	Peripheral pin setup before H1 low	13		10		9		9		ns
(3)	$t_{h(GPIOH1L}$	Peripheral pin hold after H1 low	0		0		0		0		ns

Table 13–31. Timing Parameters for Peripheral Pin Changing From General-Purpose Input to Output Mode

No.	Name	Description	'30-27 '31-27 Min	'30-27 '31-27 Max	'30-33 '31-33 'LC31 Min	'30-33 '31-33 'LC31 Max	'30-40 '31-40 Min	'30-40 '31-40 Max	'31-50 Min	'31-50 Max	Unit
(1)	$t_{d(GPIOH1H)}$	H1 high to peripheral pin switching from input to output delay		19		15		13		10	ns

Figure 13–34. Timing for Change of Peripheral Pin From General-Purpose Output to Input Mode

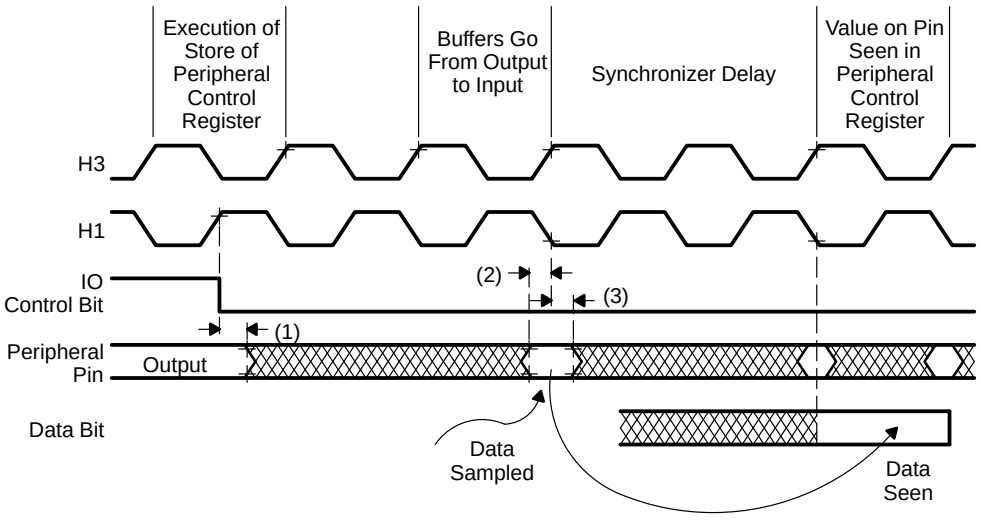
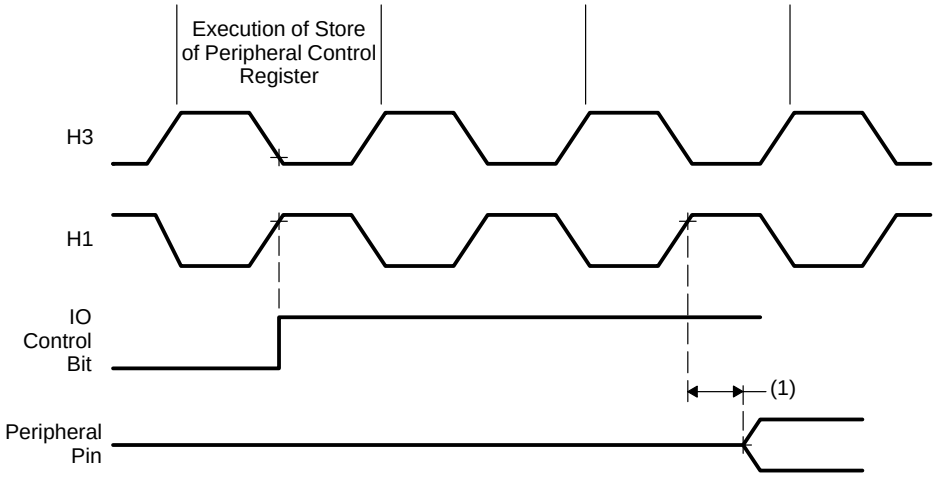


Figure 13–35. Timing for Change of Peripheral Pin From General-Purpose Input to Output Mode



13.5.16 Timer Pin Timing

Valid logic-level periods and polarity are specified by the contents of the internal control registers.

Table 13–32 and Table 13–33 define the timing parameters for the timer pin. The numbers shown in parentheses in Figure 13–36 correspond with those in the No. column of Table 13–32 and Table 13–33.

Table 13–32. Timing Parameters for Timer Pin

				’C30-27/’C31-27		’C30-33/’C31-33		
No.	**Name**	**Description‡**		**Min**	**Max**	**Min**	**Max**	**Unit**
(1)	$t_{su(TCLKH1L}$	TCLK ext setup before H1 low	TCLK ext	15		12		ns
(2)	$t_{h(TCLKH1L}$	TCLK ext hold after H1 low	TCLK ext	0		0		ns
(3)	$t_{d(TCLKH1H)}$	H1 high to TCLK int valid delay	TCLK int		13		10	ns
(4)	$t_{c(TCLK)}$	TCLK cycle time	TCLK ext	$t_{c(H)}$×2.6†		$t_{c(H)}$×2.6†		ns
			TCLK int	$t_{c(H)}$×2	$t_{c(H)}$×2^{32}†	$t_{c(H)}$×2	$t_{c(H)}$×2^{32}†	ns
(5)	$t_{w(TCLK)}$	TCLK high/ low pulse duration	TCLK ext	$t_{c(H)}$+12†		$t_{c(H)}$+12†		ns
			TCLK int	[$t_{c(TCLK)}$/2]–15	[$t_{c(TCLK)}$/2]+5	[$t_{c(TCLK)}$/2]–15	[$t_{c(TCLK)}$/2]+5	ns

† Guaranteed by design but not tested

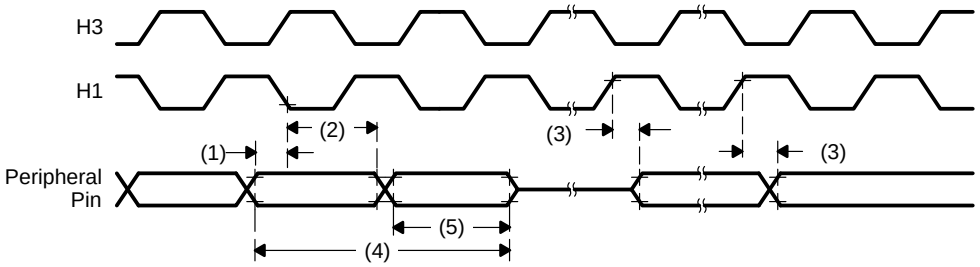
‡ Timing parameters 1 and 2 are applicable for a synchronous input clock. Timing parameters 4 and 5 are applicable for an asynchronous input clock.

Table 13–33. Timing Parameters for Timer Pin

No.	Name	Description‡		’C30-40/’C31-40 Min	’C30-40/’C31-40 Max	’C31-50 Min	’C31-50 Max	Unit
(1)	$t_{su(TCLKH1L)}$	TCLK ext set-up before H1 low	TCLK ext	10		8		ns
(2)	$t_{h(TCLKH1L)}$	TCLK ext hold after H1 low	TCLK ext	0		0		ns
(3)	$t_{d(TCLKH1H)}$	H1 high to TCLK int valid delay	TCLK int		9		9	ns
(4)	$t_{c(TCLK)}$	TCLK cycle time	TCLK ext	$t_{c(H)}\times2.6$†		$t_{c(H)}\times2.6$†		ns
			TCLK int	$t_{c(H)}\times2$	$t_{c(H)}\times2^{32}$†	$t_{c(H)}\times2$	$t_{c(H)}\times2^{32}$†	ns
(5)	$t_{w(TCLK)}$	TCLK high/ low pulse duration	TCLK ext	$t_{c(H)}+10$†		$t_{c(H)}+10$†		ns
			TCLK int	$[t_{c(TCLK)}/2]-5$	$[t_{c(TCLK)}/2]+5$	$[t_{c(TCLK)}/2]-5$	$[t_{c(TCLK)}/2]+5$	ns

† Guaranteed by design but not tested
‡ Timing parameters 1 and 2 are applicable for a synchronous input clock. Timing parameters 4 and 5 are applicable for an asynchronous input clock.

Figure 13–36. Timing for Timer Pin



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 10). 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.

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
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

Table A–1. TMS320C3x Instruction Opcodes (Continued)

INSTRUCTION	31	30	29	28	27	26	25	24	23
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
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

Table A–1. TMS320C3x Instruction Opcodes (Continued)

INSTRUCTION	31	30	29	28	27	26	25	24	23
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	–	–
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	–

† Opcode same for standard and delayed instructions.

Table A–1. TMS320C3x Instruction Opcodes (Concluded)

INSTRUCTION	31	30	29	28	27	26	25	24	23
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	–	–
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

Appendix B

Development Support/Part Ordering Information

This appendix provides development support information, device part numbers, and support tool ordering information for the TMS320C3x generation.

Each TMS320C3x support product is described in the *TMS320 Family Development Support Reference Guide* (literature number SPRU011).In addition, more than 100 third-party developers offer products that support the TI TMS320 family. For more information, refer to the *TMS320 Third-Party Reference Guide* (literature number SPRU052).

For information on pricing and availability, contact the nearest TI field sales office or authorized distributor.

This appendix discusses the following major topics:

Topic **Page**

B.1 TMS320C3x Development Support Tools

Texas Instruments offers an extensive line of development tools for the TMS320C3x generation of DSPs, including tools to evaluate the performance of the processors, generate code, develop algorithm implementations, and fully integrate and debug software and hardware modules.

The following products support development of ’C3x applications:

Code Generation Tools

- *Optimizing ANSI C compiler*. Translates ANSI C language directly into highly optimized assembly code. You can then assemble and link this code with the TI assembler/linker, which is shipped with the compiler. It supports both ’C3x and ’C4x assembly code. This product is currently available for PC (DOS, DOS extended memory, and OS/2), VAX/VMS, and SPARC workstations. Refer to the *TMS320 Floating-Point DSP Optimizing C Compiler User’s Guide* (SPRU034) for detailed information.

- *Assembler/linker*. Converts source mnemonics to executable object code. It supports both ’C3x and ’C4x assembly code. This product is currently available for PC (DOS, DOS extended memory, and OS/2). The ’C3x/’C4x assembler for the VAX/VMS and SPARC workstations is only available as part of the optimizing ’C3x/’C4x compiler. Refer to the *TMS320 Floating-Point DSP Assembly Language Tools User’s Guide* (SPRU035) for detailed information.

System Integration and Debug Tools

- *Simulator*. Simulates via software the operation of the ’C3x and can be used in C and assembly software development. This product is currently available for PC (DOS and Windows) and SPARC workstations. Refer to the *TMS320C3x C Source Debugger User’s Guide* (SPRU054) for detailed information.

- *XDS510 emulator*. Performs full-speed in-circuit emulation with the ’C3x, providing access to all registers as well as to internal and external memory. It can be used in C and assembly software development and has the capability of debugging multiple processors. This product is currently available for PC (DOS, Windows, and OS/2) and SPARC workstations. This product includes the emulator board (emulator box, power supply, and SCSI connector cables in the SPARC version), the ’C3x C source debugger software, and the JTAG cable.

Because ’C3x and ’C5x XDS510 emulators also come with the same emulator board (or box), you can buy the ’C3x C source debugger software as a separate product called ’C3x C Source Debugger Conversion Software. This enables you to debug ’C3x/’C4x/’C5x applications with the same emulator board. The emulator cable that comes with the ’C5x XDS510 emulator is not compatible with the ’C3x. You need a JTAG emulation conversion cable. Refer to the *TMS320C3x C Source Debugger User’s Guide* (SPRU053) for detailed information on the ’C3x emulator.

- *Evaluation module (EVM)*. Each EVM comes complete with a PC halfcard and software package. The EVM board contains the following:
  - A TMS320C30 and a 33-MFLOPS, 32-bit floating-point DSP
  - A 16K-word, zero-state SRAM, allowing coding of most algorithms directly on the board
  - A speaker/microphone-ready analog interface for multimedia, speech, and audio applications development
  - A multiprocessor serial port interface for connecting to multiple EVMs
  - A host port for PC communications

  The system also comes with all the software required to begin applications development on a PC host. Equipped with a C and assembly language source level debugger for the DSP, the EVM has a window-oriented, mouse-driven interface that enables the downloading, executing, and debugging of assembly code or C code.

  The TMS320C3x assembler/linker is also included with the EVM. For users who prefer programming in a high-level language, an optimizing ANSI C compiler and Ada compiler are offered separately.

- *Emulation porting kit (EPK)*. Enables you to integrate emulation technology directly into your system without the need of an XDS510 board. This product is intended to be used by third parties and high-volume board manufacturers and requires a licensing agreement with Texas Instruments. The kit contains host (or PC) source and object code, which lets you tailor 'C30 EVM-like capabilities to your TMS320C3x system via the SM74ACT8990 test bus controller (TBC). The EPK can be used in such applications as program download for system self-test and initialization or system emulation and debug to feature resident emulation support. EPK software includes the TI high-level language (HLL) debugger in object as well as source code for the TBC communication interface. The HLL code is the windowed debugger found with many TI DSP simulators, evaluation modules (EVMs), and emulators. With the EPK, the HLL user interface can be ported directly to the system board. The source code for the TBC communication interface consists of such commands as read/write, memory run, stop, and reset that communicate with the TMS320C3x device. Using the EPK reduces system and development cost and speeds time to market. For more information on the kit, call the DSP hotline at (713) 274-2320.

B.1.1 TMS320 Third Parties

The TMS320 family is supported by product and service offerings from more than 100 independent vendors and consultants, known as third parties. These support products take various forms (both software and hardware) from cross-assemblers, simulators, and DSP utility packages to logic analyzers and emulators. Additionally, TI third parties offer more than 150 algorithms that are available for license through the TMS320 software cooperative. These algorithms can greatly reduce development time and decrease time to market. The expertise of those involved in support services ranges from speech encoding and vector quantization to software/hardware design and system analysis.

For a more detailed description of services and products offered by third parties, refer to the *TMS320 Third Party Support Reference Guide* (literature number SPRU052) and the *TMS320 Software Cooperative Data Sheet Packet* (literature number SPRT111). Call the Literature Response Center at (800) 477–8924 to request a copy.

B.1.2 TMS320 Literature

Extensive DSP documentation is available; this includes data sheets, user's guides, and application reports. In addition, DSP textbooks that aid research and education have been published by Prentice-Hall, John Wiley and Sons, and Computer Science Press. To order literature or to subscribe to the DSP newsletter *Details on Signal Processing* (for up-to-date information on new products and services), call the Literature Response Center at (800) 477–8924.

B.1.3 DSP Hotline

For answers to TMS320 technical questions on device problems, development tools, documentation, upgrades, and new products, you can contact the DSP hotline via:

- Phone at (713)274–2320 Monday through Friday from 8:30 a.m. to 5:00 p.m. central time
- Fax at (713)274–2324
- Electronic mail at 4389750@mcimail.com.
- European fax at 33–1–3070–1032
- Semiconductor Product Information Center (PIC) at (214) 644–5580

To ask about third-party applications and algorithm development packages, contact the third party directly. Refer to the *TMS320 Third-Party Support Reference Guide* (literature number SPRU052) for addresses and phone numbers.

Extensive DSP documentation is available; this includes data sheets, user's guides, and application reports. Call the hotline at (800) 477–8924 for information on literature that you can request from the Literature Response Center.

The DSP hotline does not provide pricing information. Contact the nearest TI field sales office or the TI PIC for prices and availability of TMS320 devices and support tools.

B.1.4 Bulletin Board Service (BBS)

The TMS320 DSP Bulletin Board Service (BBS) is a telephone-line computer service that provides information on TMS320 devices, specification updates for current or new devices and development tools, silicon and development tool revisions and enhancements, new DSP application software as it becomes available, and source code for programs from any TMS320 user's guide.

You can access the BBS via the following:

- Modem: (300-, 1200-, or 2400-bps) dial (713)274–2323. Set your modem to 8 data bits,1 stop bit, no parity.
- Internet: Use anonymous *ftp* to *ti.com* (Internet port address 192.94.94.1). The BBS content is located in the subdirectory called *mirrors*.

To find out more about the BBS, refer to the *TMS320 Family Development Support Reference Guide* (literature number SPRU011).

B.1.5 Technical Training Organization (TTO) TMS320 Workshop

The TMS320C3x DSP design workshop is tailored for hardware and software design engineers and decision-makers who will be designing and utilizing the TMS320C3x generation of DSP devices. Hands-on exercises throughout the course give participants a rapid start in utilizing TMS320C3x design skills. Microprocessor/assembly language experience is required. Experience with digital design techniques and C language programming experience is desirable. The following topics are covered in the TMS320C3x workshop:

- TMS320C3x architecture/instruction set
- Use of the PC-based TMS320C3x software simulator and EVM
- Floating-point and parallel operations
- Use of the TMS320C3x assembler/linker
- C programming environment
- System architecture considerations
- Memory and I/O interfacing
- TMS320C3x development support

For registration, pricing, or enrollment information on this and other TTO TMS320 workshops, call (800) 336–5236, ext. 3904.

B.2 TMS320C3x Part Ordering Information

This section provides the device and support tool part numbers. Table B–1 lists the part numbers for the TMS320C30 and TMS320C31; Table B–2 gives ordering information for TMS320C3x hardware and software support tools. An explanation of the TMS320 family device and development support tool prefix and suffix designators follows the two tables to assist in understanding the TMS320 product numbering system.

Table B–1. TMS320C3x Digital Signal Processor Part Numbers

Device	Technology	Operating Frequency	Package Type	Typical Power Dissipation
TMS320C30GEL	0.8-µm CMOS	33 MHz	Ceramic 181-pin PGA	1.00 W
TMS320C30GEL27	0.8-µm CMOS	27 MHz	Ceramic 181-pin PGA	0.875 W
TMS320C30GEL40	0.8-µm CMOS	40 MHz	Ceramic 181-pin PGA	1.25 W
TMS320C30PPM40	0.8-µm CMOS	40 MHz	Plastic 208-pin QFP	0.85 W
TMS320C31PQL/PQA	0.8-µm CMOS	33 MHz	Plastic 132-pin QFP	0.75 W
TMS320C31PQL27	0.8-µm CMOS	27 MHz	Plastic 132-pin QFP	0.60 W
TMS320C31PQL40	0.8-µm CMOS	40 MHz	Plastic 132-pin QFP	0.90 W
TMS320LC31PQL	0.8-µm CMOS	33 MHz	Plastic 132-pin QFP	0.50 W
TMS320C31PQL50	0.8-µm CMOS	50 MHz	Plastic 132-pin QFP	1.00 W
SMJ320C316FA27	0.8-µm CMOS	28 MHz	Ceramic 141-pin PGA	0.60 W
SMJ320C31HF627			Ceramic 132-pin QFP	0.60 W
SMJ320C316FA33			Ceramic 141-pin PGA	0.75 W
SMJ320C316HF633			Ceramic 132-pin PGA	0.75 W
SMJ320C306BM33	0.8-µm CMOS	33 MHz	Ceramic 181-pin PGA	1.10 W
SMJ320C30HF633			Ceramic 196-pin QFP	
SMJ320C30GBM28	0.8-µm CMOS	28 MHz	Ceramic 181-pin PGA	1.00 W
SMJ320C30HF628			Ceramic 196-pin QFP	1.00 W
SMJ320C30HTM28				
SMJ320C30GBM25	0.8-µm CMOS	25 MHz	Ceramic 181-pin PGA	1.00 W
SMJ320C30HF625			Ceramic 196-pin QFP	1.00 W
SMJ320C30HTM25				

Table B–2.TMS320C3x Support Tool Part Numbers

Tool Description	Operating System	Part Number
(a) Software		
C Compiler & Macro Assembler/ Linker	VAX/VMS PC-DOS/MS-DOS SPARC (Sun OS)†	TMDS3243255-08 TMDS3243855-02 TMDS3243555-08
Assembler/Linker	PC-DOS/MS-DOS; OS/2	TMDS3243850-02
Simulator	VAX VMS PC-DOS/MS-DOS SPARC (SUN OS) †	TMDS3243251-08 TMDS3243851-02 TMDS3243551-09
Tartan Floating-Point Library	PC-DOS SPARC (Sun OS)	320 FLO-PC C30 320 FLO-Sun-C30
Digital Filter Design Package	PC-DOS	DFDP
Tartan C++ Compiler/Debugger	PC-DOS; OS/2, Wiredown SPARC (Sun OS)	TAR-CCM-PC-C3x TAR-CCM-SP-C3x
Tartan C++ Compiler	PC-DOS; OS/2, Wiredown SPARC (Sun OS)	TAR-SIM-PC-C3x TAR-SIM-SP-C3x
TMS320C3x Emulation Porting Kit		TMSX3240030
(b) Hardware		
XDS510 Emulator	PC/MS-DOS	TMDS3260130
Evaluation Module (EVM)	PC-DOS/MS-DOS	TMDS3260030

† Note that SUN UNIX supports TMS320C3x software tools on the 68000 family-based SUN-3 series workstations and on the SUN-4 series machines that use the SPARC processor, but not on the SUN-386i series of workstations.

B.2.1 Device and Development Support Tool Prefix Designators

Prefixes to TI part numbers designate phases in the product's development stage for both devices and support tools, as shown in the following definitions:

Device Development Evolutionary Flow

- **TMX:** Experimental device that is not necessarily representative of the final device's electrical specifications
- **TMP:** Final silicon die that conforms to the device's electrical specifications but has not completed quality and reliability verification
- **TMS:** Fully qualified production device

Support Tool Development Evolutionary Flow

- **TMDX:** Development support product that has not yet completed TI's internal qualification testing for development systems
- **TMDS:** Fully qualified development support product

TMX and TMP devices and TMDX development support tools are shipped with the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

Note: Prototype Devices

TI recommends that prototype devices (TMX or TMP) not be used in production systems because their expected end-use failure rate is undefined but predicted to be greater than standard qualified production devices.

TMS devices and TMDS development support tools have been fully characterized, and their quality and reliability have been fully demonstrated. TI's standard warranty applies to TMS devices and TMDS development support tools.

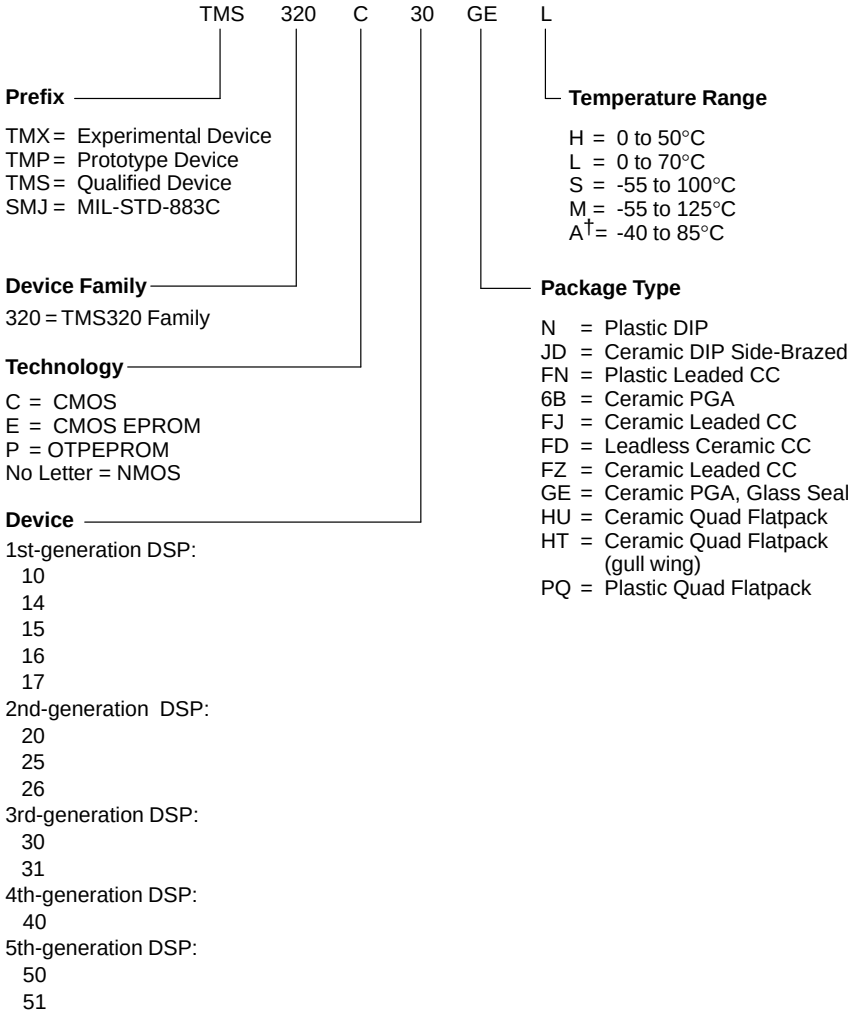
TMDX development support products are intended for internal evaluation purposes only. They are covered by TI's Warranty and Update Policy for Microprocessor Development Systems products; however, they should be used by customers only with the understanding that they are developmental in nature.

B.2.2 Device Suffixes

The suffix indicates the package type (for example, N, FN, or GE) and temperature range (for example, L).

Figure B–1 presents a legend for reading the complete device name for any TMS320 family member.

Figure B–1. TMS320 Device Nomenclature



† See electrical specifications for TMS320C31 PQA case temperature ratings.

Appendix C

Quality and Reliability

The quality and reliability of Texas Instruments (TI) microprocessor and microcontroller products, which include TMS320 digital signal processors, relies on feedback from the following:

- Our customers,
- Our total manufacturing operation from front-end wafer fabrication to final shipping inspection, and
- Product quality and reliability monitoring.

Our customer's perception of quality is the governing criterion for judging performance. This concept is the basis for TI Corporate Quality Policy, which is as follows:

"For every product or service we offer, we shall define the requirements that solve the customer's problems, and we shall conform to those requirements without exception."

Texas Instruments has developed a leadership reliability qualification system, based on years of experience with leading-edge memory technology and on years of research into customer requirements. To achieve constant improvement, programs that support that system respond to customer input and internal information.

This appendix presents the following major topics:

Topic **Page**

C.1 Reliability Stress Tests

Accelerated stress tests are performed on new semiconductor products and process changes to qualify them and ensure excellence in product reliability. The following test environments are typical:

- High-temperature operating life
- Storage life
- Temperature cycling
- Biased humidity
- Autoclave
- Electrostatic discharge
- Package integrity
- Electromigration
- Channel-hot electrons (performed on geometries less than 2.0 μm)

Typical events or changes that require internal requalification of a product include the following:

- New die design, shrink, or layout
- Wafer process (baseline/control systems, flow, mask, chemicals, gases, dopants, passivation, or metal systems)
- Packaging assembly (baseline control systems or critical assembly equipment)
- Piece parts (such as lead frame, mold compound, mount material, bond wire, or lead finish)
- Manufacturing site

TI reliability control systems extend beyond qualification. Total reliability controls and management include product reliability monitoring as well as final product release controls. MOS memories, utilizing high-density active elements, serve as the leading indicator in wafer-process integrity at TI MOS fabrication sites, enhancing all MOS logic device yields and reliability. TI places more than several thousand MOS devices per month on reliability tests to ensure and sustain built-in product excellence.

Table C–1 lists the microprocessor and microcontroller reliability tests, the duration of the test, and sample size. Table C–2 contains definitions and descriptions of terms used in those tests.

Table C–1. Microprocessor and Microcontroller Tests

Test	Duration	Sample Size Plastic	Ceramic
Operating life, 125° C, 5.0 V	1000 hrs	129	129
Storage life, 150° C	1000 hrs	45†	45
Biased humidity, 85° C/85 percent RH, 5.0 V	1000 hrs	77	–
Autoclave, 121° C, 1 ATM	240 hrs	45	–
Temperature cycle, –65 to 150° C	1000 cyc‡	77	77
Temperature cycle, 0 to 125° C	3000 cyc	77	77
Thermal shock,–65 to 150° C	200 cyc	77	77
Electrostatic discharge, ±2 kV		15	15
Latch-up (CMOS devices only)		5	5
Mechanical sequence		–	22
Thermal sequence		–	22
Thermal/mechanical sequence		–	22
PIND		–	45
Internal water vapor		–	3
Solderability		22	22
Solder heat		22	22
Resistance to solvents		15	15
Lead integrity		15	15
Lead pull		22	–
Lead finish adhesion		15	15
Salt atmosphere		15	15
Flammability (UL94-V0)		3	–
Thermal impedance		5	5

† If junction temperature does not exceed plasticity of package
‡ For severe environments; reduced cycles for office environments

Table C–2. Definitions of Microprocessor Testing Terms

Term	Definition/Description	References
Average Outgoing Quality (AOQ)	Amount of defective product in a population, usually expressed in terms of parts per million (PPM).	
Failure in Time (FIT)	Estimated field failure rate in number of failures per billion power-on device hours; 1000 FITS equal 0.1 percent failure per 1000 device hours.	
Operating Life	Device dynamically exercised at a high ambient temperature (usually 125° C) to simulate field usage that would expose the device to a much lower ambient temperature (such as 55° C). Using a derived high temperature, a 55°C ambient failure rate can be calculated.	
Storage Life	Device exposed to 150° C unbiased condition. Bond integrity is stressed in this environment.	
Biased Humidity	Moisture and bias used to accelerate corrosion-type failures in plastic packages. Conditions include 85° C ambient temperature with 85% relative humidity (RH). Typical bias voltage is +5V and is grounded on alternating pins.	
Autoclave (Pressure Cooker)	Plastic-packaged devices exposed to moisture at 121° C using a pressure of one atmosphere above normal pressure. The pressure forces moisture permeation of the package and accelerates corrosion mechanisms (if present) on the device. External package contaminants can also be activated and caused to generate inter-pin current leakage paths.	
Temperature Cycle	Device exposed to severe temperature extremes in an alternating fashion (–65° C for 15 minutes and 150° C for 15 minutes per cycle) for at least 1000 cycles. Package strength, bond quality, and consistency of assembly process are tested in this environment.	
Electrostatic Discharge	Device exposed to electrostatic discharge pulses. Calibration is according to MIL STD 883C, method 3015.6. Devices are stressed to determine failure threshold of the design.	

Table C–2. Definitions of Microprocessor Testing Terms (Continued)

Term	Definition/Description	References
Thermal Shock	Test similar to the temperature cycle test, but involving a liquid-to-liquid transfer.	MIL-STD-883C, Method 1011
Particle Impact Noise Detection (PIND)	A nondestructive test to detect loose particles inside a device cavity.	
Mechanical Sequence	Fine and gross leak	MIL-STD-883C, Method 1014
	Mechanical shock	MIL-STD-883C, Method 2002, 1500 g, 0.5 ms, Condition B
	PIND (optional)	MIL-STD-883C, Method 2020
	Vibration, variable frequency	MIL-STD-883C, Method 2007, 20g, Condition A
	Constant acceleration	MIL-STD-883C, Method 2001
	Fine and gross leak	MIL-STD-883C, Method 1014
	Electrical test	To data sheet limits
Thermal Sequence	Fine and gross leak	MIL-STD-883C, Method 1014
	Solder heat (optional)	MIL-STD-750C, Method 1014
	Temperature cycle (10 cycles minimum)	MIL-STD-883C, Method 1010, –65 to + 150 °C, Condition C
	Thermal shock (10 cycles minimum)	MIL-STD-883C, Method 1011, –55 to +125 °C, Condition B
	Moisture resistance	MIL-STD-883C, Method 1004
	Fine and gross leak	MIL-STD-883C, Method 1014
	Electrical test	To data sheet limits
Thermal/Mechanical Sequence	Fine and gross leak	MIL-STD-883C, Method 1014
	Temperature cycle (10 cycles minimum)	MIL-STD-883C, Method 1010, –65 to +150 °C, Condition C
	Constant acceleration	MIL-STD-883C, Method 2001, 30 kg, Y1 Plane
	Fine and gross leak	MIL-STD-883C, Method 1014
	Electrical test	To data sheet limits
	Electrostatic discharge	MIL-STD-883C, Method 3015
	Solderability	MIL-STD-883C, Method 2033
	Solder heat	MIL-STD-750C, Method 2031, 10 sec
	Salt atmosphere	MIL-STD-883C, Method 1009, Condition A, 24 hrs min
	Lead pull	MIL-STD-883C, Method 2004, Condition A
	Lead integrity	MIL-STD-883C, Method 2004, Condition B1
	Electromigration	Accelerated stress testing of conductor patterns to ensure acceptable lifetime of power-on operation
	Resistance to solvents	MIL-STD-883C, Method 2015

Table C–3 lists the TMS320C3x devices, the approximate number of transistors, and the equivalent gates. The numbers have been determined from design verification runs.

Table C–3. TMS320C3x Transistors

Device	# Transistors	# Gates
CMOS: TMS320C30	600K–700K	200K
CMOS: TMS320C31	500K–600K	100K

Note: MOS Semiconductors

Texas Instruments reserves the right to make changes in MOS semiconductor test limits, procedures, or processing without notice. Unless prior arrangements for notification have been made, TI advises all customers to reverify current test and manufacturing conditions prior to relying on published data.

C.2 TMS320C31 PQFP Reflow Soldering Precautions

Recent tests have identified an industry-wide problem experienced by surface-mounted devices exposed to reflow soldering temperatures. This problem involves a package-cracking phenomenon sometimes experienced by large (for example, 132-pin) plastic quad flat pack (PQFP) packages during surface-mount manufacturing. This phenomenon occurs if the TMS320C31 PQA or PQL is exposed to uncontrolled levels of humidity prior to reflow solder. This moisture can flash to steam during solder reflow and cause sufficient stress to crack the package and compromise device integrity. Once the device is soldered or socketed into the board, no special handling precautions are required.

To minimize moisture absorption, TI ships the TMS320C31 PQA or PQL in dry pack shipping bags with a relative humidity (RH) indicator card and moisture-absorbing desiccant. These moisture-barrier shipping bags will adequately block moisture transmission to allow shelf storage for 12 months from date of seal when stored at less than 60% RH and less than 30° C. Devices may be stored outside the sealed bags indefinitely if stored at less than 25% RH and less than 30° C.

Once the bag seal is broken, the devices should, within two days of removal, be reflow soldered and stored at less than 60% RH and less than 30° C. If these conditions are not met, TI recommends baking the devices in a clean oven at 125° C and 10% maximum RH for 25 hours. This procedure restores the devices to their dry-packed moisture level.

Note: ESD Precautions

Shipping tubes will not withstand the 125° C baking process. Before baking, transfer the devices to a metal tray or tube. Follow standard ESD precautions.

TI recommends that the reflow process not exceed two solder cycles and that the temperature not exceed 220° C.

If you have questions or concerns, please contact your local TI representative.

Appendix D

Calculation of TMS320C30 Power Dissipation

The TMS320C30 is a state-of-the-art, high-performance, 32-bit floating-point digital signal processing (DSP) microprocessor fabricated in CMOS technology. This device is the first member of the third generation of TMS320 family single-chip DSP microprocessors. Since 1982, when the first-generation TMS32010 was introduced, the TMS320 family has established itself as the industry standard for DSP. The TMS320C30's innovative architecture and specialized instruction set provide high-speed and increased flexibility for DSP applications. This combination makes it possible to execute up to 40 million floating point operations per second (MFLOPS).

As device sophistication and levels of integration increase with evolving semiconductor technologies, actual levels of power dissipation vary widely and depend heavily on the particular application in which the device is used and the nature of the program being executed. In addition, due to the inherent characteristics of CMOS technology, power requirements vary according to clock rates and data values being processed.

This appendix presents the information necessary to determine TMS320C30 power supply current requirements under different operating conditions. With this information, you can determine the device's power dissipation, which, in turn, you can use to calculate thermal management requirements.

This appendix discusses the following major topics:

Topic **Page**

D.1 Fundamental Power Dissipation Characteristics

Typically, an IC's (integrated circuit) power specification is expressed as a function of operating frequency, supply voltage, operating temperature, and output load. As devices become more complex, the specification must also be based on device functionality. CMOS devices inherently draw current only during switching through the linear region. Therefore, the power supply current is related to the rate of switching. Furthermore, since the output drivers of the TMS320C30 are specified to drive direct current (DC) loads, the power supply current resulting from external writes depends not only on switching rate but also on the value of data written.

D.1.1 Components of Power Supply Current Requirements

There are four basic components of the power supply current:

- Quiescent,
- Internal Operations,
- Internal Bus Operations, and
- External Bus Operations

D.1.2 Dependencies

The power supply current consumption depends on many factors. Four are system-related:

- Operating frequency,
- Supply voltage,
- Operating temperature, and
- Output load

Several others are also related to TMS320C30 operation, including:

- Duty cycle of operations,
- Number of buses used,
- Wait states,
- Cache usage, and
- Data value

The total power supply current for the device is described in this equation, which applies the four basic power supply current components and the dependencies described above:

$$I = \left(I_q + I_{iops} + I_{ibus} + I_{xbus}\right) \times FV \times T$$

where

I_q is the quiescent current component,

I_{iops} is the current component due to internal operations,

I_{ibus} is the current component due to internal bus usage, including data value and cycle time dependencies,

I_{xbus} is the current component due to external bus usage, including data value, wait state, cycle time, and capacitive load dependencies,

FV is a scale factor for frequency and supply voltage, and

T is a scale factor for operating temperature.

Application of this equation and determination of all of the dependencies are described in detail in this appendix.

This appendix explains, in detail, how to determine the power supply current requirement for the TMS320C30. If a less detailed analysis is sufficient, the minimum, typical, and maximum values can be used to determine a rough estimate of the power supply current requirements. The minimum power supply current requirement is 110 mA. The typical and average current consumption is 200 mA, as described in the TMS320C30 data sheet, and will be associated with most algorithms running on the device unless data output is excessive.

Maximum Current Requirement
The maximum current requirement is 600 mA and occurs only under *worst case* conditions: namely, writing alternating data (AAAAAAAAh to 55555555h) out of both external buses simultaneously, every cycle, with 80 pF loads and running at 33 MHz.

If an extremely conservative approach is desired, the maximum value can be used.

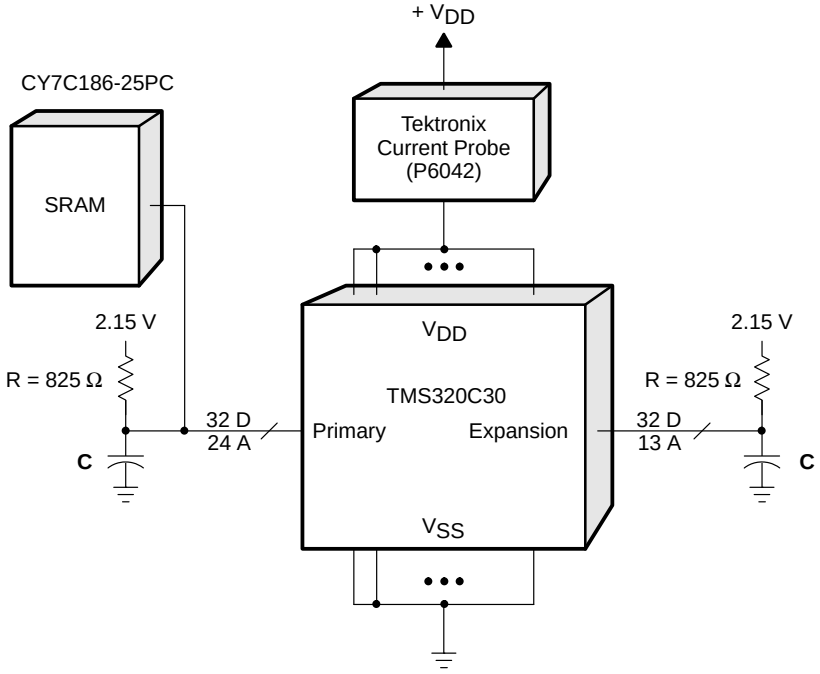
D.1.3 Determining Algorithm Partitioning

Each part of an algorithm behaves differently, depending on its internal and external bus usage. To analyze the power supply current requirement, you must partition an algorithm into segments with distinct concentrations of internal or external bus usage. The analysis that follows is applied to each distinct program segment to determine the power supply current requirement for that section. The average power supply current requirement can then be calculated from the requirements of each segment of the algorithm.

D.1.4 Test Setup Description

All TMS320C30 supply current measurements were performed on the test setup shown in Figure D–1. The test setup consists of a TMS320C30, 8K words of zero-wait-state Cypress Semiconductor SRAMs (CY7C186–25PC), and RC loads on all data and address lines. A Tektronix Current Probe (P6042) measures the power supply current in all V_{DD} lines of the device. The supply voltage on the output load is 2.15 V. Unless otherwise specified, all measurements are made at a supply voltage of 5.0 V, an input clock frequency of 33 MHz, a capacitive load of 80 pF, and an operating temperature of 25°C.

Figure D–1.Current Measurement Test Setup



D.2 Current Requirement for Internal Circuitry

The power supply current requirement for internal circuitry consists of three components: quiescent, internal operations, and internal bus operations. Quiescent and internal operations are constants, but the internal bus operations component varies with the rate of internal bus usage and the data values being transferred.

D.2.1 Quiescent

Quiescent refers to the baseline supply current drawn by the TMS320C30 during minimal internal activity, such as executing the IDLE instruction or branching to self. It includes the current required to fetch an instruction from on- or off-chip memory. The quiescent requirement for the TMS320C30 is 110 mA. Examples of quiescent current include:

- Maintaining timers and serial ports
- Executing the IDLE instruction
- TMS320C30 in HOLD mode pending external bus access
- TMS320C30 in reset
- Branching to self

D.2.2 Internal Operations

Internal operations are those that require more current than quiescent activity but do not include external bus usage or significant internal bus usage. Internal operations include register-to-register multiplication, ALU operations, and branches. They add a constant 55 mA above the quiescent so that the total contribution of quiescent and internal operations is 165 mA. Note, however, that internal and/or external bus operations executed via an RPTS instruction do not contribute an internal operations power supply current component and hence do not add 55 mA to quiescent current. During an instruction in RPTS, activity other than the instruction being repeated is suspended; therefore, power supply current is related only to the operation performed by the instruction being executed. The next contributing factor to the power supply current requirement is internal bus operations.

D.2.3 Internal Bus Operations

The internal bus operations include all operations that utilize the internal buses extensively, such as accessing internal RAM every cycle. No distinction is made between internal reads (such as instruction or operand fetches from internal ROM or internal RAM banks) and internal writes (such as operand stores to internal RAM banks), because internally they are equal. Significant use of internal buses adds a term to the power supply current requirement that is data-dependent. Since switching requires more current, moving changing data at high rates requires higher power supply current.

Pipeline conflicts, use of cache, fetches from external wait-state memory, and writes to external wait-state memory all affect the internal and external bus cycles of an algorithm executing on the TMS320C30. Therefore, the internal bus usage of the algorithm must be determined to accurately calculate power supply current requirements. The TMS320C30 software simulator and XDS emulator both provide benchmarking and timing capabilities that allow bus usage to be determined.

The current resulting from internal bus usage varies roughly exponentially with transfer rates. Figure D–2 shows internal bus current requirements for transferring alternating data (AAAAAAAAh to 55555555h) at several transfer rates (expressed as the transfer cycle time). A transfer rate less than 1 implies multiple accesses per single H1 cycle (that is, using direct memory access (DMA), etc.). Transfer cycle times greater than 1 refer to single-cycle transfers with one or more cycles between them. The minimum transfer cycle time is one-third, which corresponds to three accesses in a single H1 cycle.

The data set AAAAAAAAh to 55555555h exhibits the maximum current for these types of operations. Less current is required for transferring other data patterns, and current values can be derated accordingly as described later in this subsection.

As the transfer rate decreases (that is, transfer cycle time increases), the incremental I_{DD} approaches 0 mA. Transfer rates corresponding to more than seven H1 cycles do not add any current and are considered insignificant. This figure represents the incremental I_{DD} due to internal bus operations and is added to quiescent and internal operations current values.

For example, the maximum transfer rate corresponds to three accesses every cycle or one-third H1 transfer cycle time. At this rate, 85 mA is added to the quiescent (110 mA) and internal operation (55 mA) current values for a total of 250 mA.

IncrementalFigure D–2 shows the internal bus current requirement when transferring As, followed by 5s, for various transfer rates. Figure D–3 shows the data dependence of the internal bus current requirement when the data is other than As followed by 5s. The trapezoidal region bounds all possible data values transferred. The lower line represents the scale factor for transferring the same data. The upper line represents the scale factor for transferring alternating data (all 0s to all Fs or all As to all 5s, etc.).

Figure D–2. Internal Bus Current Versus Transfer Rate

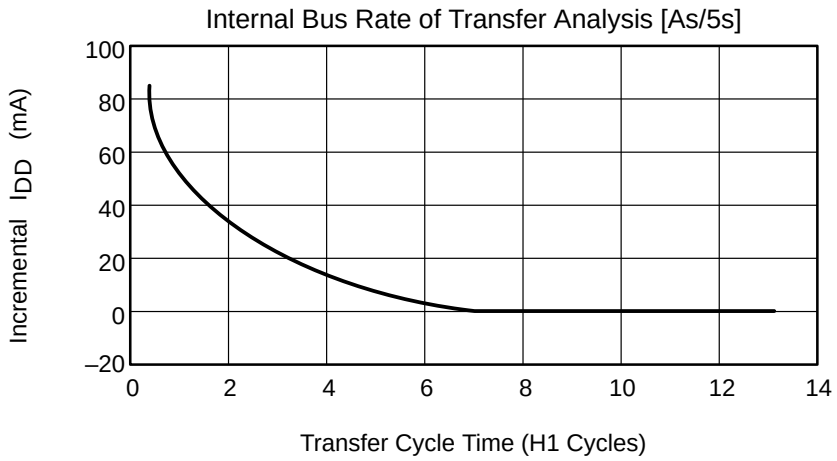
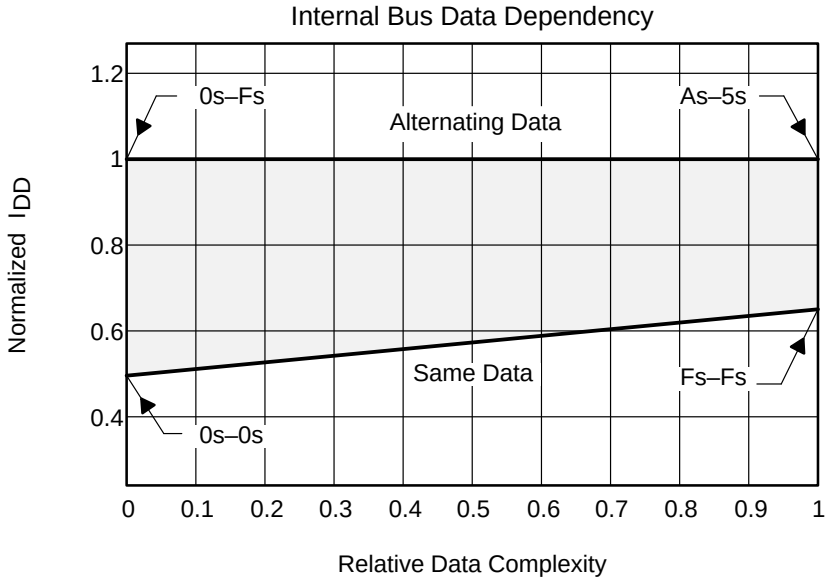


Figure D–3. Internal Bus Current Versus Data Complexity Derating Curve



Since the possible permutations of data values is quite large, the extent to which data varies is referred to as relative data complexity. This term represents a relative measure of the extent to which data values are changing and the extent to which the number of bits are changing state. Therefore, relative data complexity ranges from 0, signifying minimal variation of data, to a normalized value of 1, signifying greatest data variation.

If a statistical knowledge of the data exists, Figure D–3 can be used to determine the exact power supply requirement according to internal bus usage. For example, Figure D–3 indicates a 63% scale factor when all Fs are moved internally every cycle with two accesses per cycle. This scale factor is multiplied by 55 mA (from Figure D–2, at one-half H1 cycle transfer time), yielding 34.65 mA because of internal bus usage. Therefore, an algorithm running under these conditions requires about 200 mA of power supply current (110 + 55 + 34.65).

Since a statistical knowledge of the data might not be readily available, a nominal scale factor will suffice. The median between the minimum and maximum values at 50% relative data complexity yields a value of 0.80. This value will serve as an estimate of a nominal scale factor. Therefore, you can use this nominal data scale factor of 80% for internal bus data dependency, adding 44 mA to 110 mA (quiescent) and 55 mA (internal operations) to yield 210 mA. As an upper bound, assume worst case conditions of three accesses of alternating data every cycle, adding 85 mA to 110 mA (quiescent) and 55 mA (internal operations) to yield 250 mA.

D.3 Current Requirement for Output Driver Circuitry

The output driver circuits on the TMS320C30 are required to drive significantly higher DC and capacitive loads than internal device logic. Therefore, they are designed to drive larger currents than internal devices. Because of this, output drivers impose higher supply current requirements than other sections of circuitry on the device.

Accordingly, the highest values of supply current are exhibited when external writes are being performed at high speed. During reads, or when the external buses are not being used, the TMS320C30 is not driving the data bus; this eliminates the most significant component of output buffer current. Furthermore, in typical cases, only a few address lines are changing, or the whole address bus is static. Under these conditions, an insignificant amount of supply current is consumed. Therefore, when no external writes are being performed or when writes are performed infrequently, current due to output buffer circuitry can be ignored.

When external writes are being performed, the current required to supply the output buffers depends on several considerations. As with internal bus operations, current required for output drivers depends on the data being transferred and the rate at which transfers are being made. Additionally, output driver current requirements depend on the number of wait states implemented, because wait states affect rates at which bus signals switch. Finally, current values are also dependent upon external bus DC and capacitive loading.

External operations involve writes external to the device and constitute the major power supply current component. The power supply current for the external buses is made up of three components and is summarized in the following equation:

$$I_{base} + I_{prim} + I_{exp}$$

where

I_{base} is the 60-mA baseline current component

I_{prim} is the primary bus current component

I_{exp} is the expansion bus current component

The remainder of this section describes in detail the calculation of external bus current components.

D.3.1 Primary Bus

The current due to primary bus writes varies roughly exponentially with both wait states and write cycle time. Also, current components due to output driver circuitry are represented as offsets from the baseline value. Since the baseline value is related to internal current components, negative values for current offset are obtained under some circumstances. Note, however, that actual negative current does not occur.

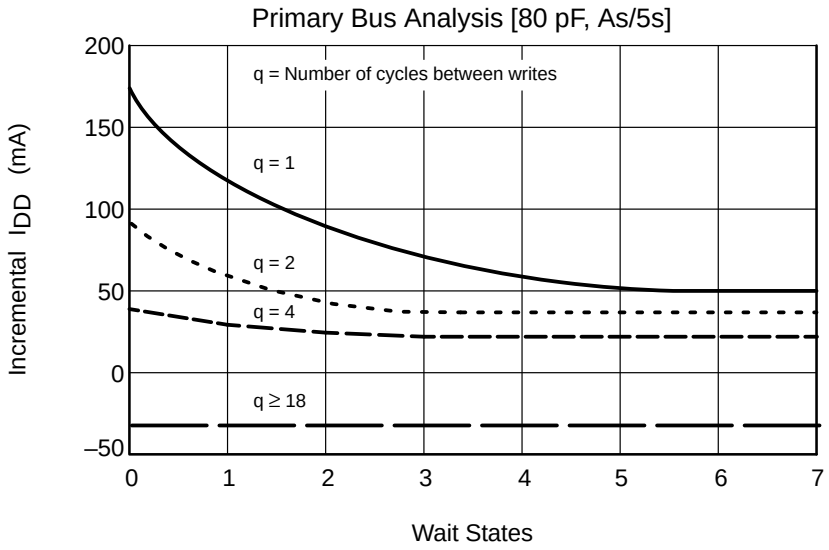
As previously mentioned, to obtain accurate current values, you must first establish timing of write cycles on the buses. To determine the rate and timings at which write cycles to the external buses occur, you must analyze program activity, including any pipeline conflicts that may exist. Information from this manual and the TMS320C30 emulator or simulator is useful in making these determinations. Note that effects from the use of cache must also be accounted for in these analyses because use of cache can affect whether instructions are fetched from external memory.

When evaluating external write activity in a given program segment, you must consider whether a particular level of external write activity constitutes significant activity. If writes are being performed at a slow enough rate, they do not significantly impact supply current requirements; therefore, current due to external writes can be ignored. This is the case, however, only if writes are being performed at very slow rates on both the primary and the expansion buses. If writes are being performed at high speed on only one of the two external buses, you should still use the approach described in this section to calculate current requirements.

Note that, although you obtain negative incremental current values under some circumstances, the total contribution for external buses, including baseline current, must always be positive. The reason is that, when external buses are used minimally, total current requirements always approach the current contribution due to internal components, which is solely a function of internal activity. This places a lower limit on current contributions resulting from the primary and expansion buses, because the total current due to external buses is the sum of the 60-mA baseline value and the primary and expansion bus components. This effect is discussed in further detail in the rest of this subsection.

When you have established bus-write cycle timing, you can use Figure D–4 to determine the contribution to supply current due to this bus activity. Figure D–4 shows values of current contribution from the primary bus for various numbers of wait states and H1 cycles between writes. These characteristics are exhibited when writes of alternating 55555555h and AAAAAAAAh are being performed at a capacitive load of 80 pF per output signal line. The conditions exhibit the highest current values on the device. The values presented in the figure represent incremental or additional current contributed by the primary bus output driver circuitry under the given conditions. Current values obtained from this graph are later scaled and added to several other current terms to calculate the total current for the device. As indicated in the figure, the lower curve represents the current contribution for 18 or more cycles between writes.

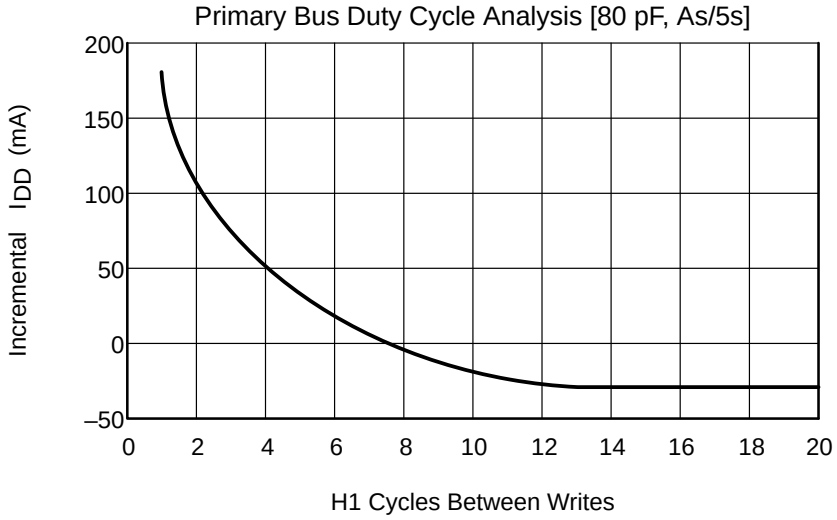
Figure D–4. Primary Bus Current Versus Transfer Rate and Wait States



Note that *number of cycles between writes* refers to the number of H1 cycles between the active portion of the write cycles as defined in Chapter 13—that is, between H1 cycles when $\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$, or $\overline{\text{IOSTRB}}$ and R/$\overline{\text{W}}$ (or XR/$\overline{\text{W}}$, as the case may be) are low. As shown in Figure D–4, the minimum number of cycles between writes is 1 because with back-to-back writes there is one H1 cycle between active portions of the writes.

To further illustrate the relationship of current and write cycle time, Figure D–5 shows the characteristics of current for various numbers of cycles between writes for zero wait states. The information on this curve can be used to obtain more precise values of current if zero wait states are being used and the number of cycles between writes does not fall on one of the curves in Figure D–4.

Figure D–5. Primary Bus Current Versus Transfer Rate at Zero Wait States



Note that, although these graphs contain negative current values, negative current has not necessarily actually occurred. The negative values exist because the graphs represent a current offset from a common baseline current value, which is not necessarily the lowest current exhibited. Using this approach to depict current contributions due to different components simplifies current calculations because it allows calculations to be made independently. Independent calculations are possible because information about relationships between different sections of the device are included implicitly in the information for each section.

Figure D–4 and Figure D–5 show that the contribution of writes for external bus activities becomes insignificant if writes are being performed at intervals of more than 18 cycles. Under these conditions, you should use the incremental value of –30-mA current contribution due to the primary bus. Note, however, that you should use a value of –30 mA only if the expansion bus is being used extensively. This is because the total contribution for external buses, including baseline current, must always be *positive*. If the expansion bus is not being used and the primary bus is being used minimally, the current contribution due to the primary bus must always be greater than or equal to 20 mA. This ensures that the correct total current value is obtained when summing external bus components. Once a current value has been obtained from Figure D–4 or Figure D–5, this value can, if necessary, be scaled by a data dependency factor, as described at the end of this section. This scaled value is then summed along with several other current terms to determine the total supply current. Calculation of total supply current is described in detail in Section D.4 on page D-18.

D.3.2 Expansion Bus

Currents due to the primary and expansion buses are similar in characteristics but differ slightly because of several factors, including the fact that the expansion bus has 11 fewer address outputs than the primary bus (13 rather than 24). This difference is exhibited in an overall current contribution that is slightly lower from the expansion bus than from the primary bus.

Accordingly, determination of expansion bus current follows the same basic premises as determination of the primary bus current. Figure D–6 and Figure D–7 show the same current relationships for the expansion bus as Figure D–4 and Figure D–5 show for the primary bus. Also, since the total external buses' current contributions must be positive, if the primary bus is not being used and the expansion bus is being used minimally, then the minimum current contribution due to the expansion bus is –30 mA. Finally, as with the primary bus, current values obtained from these figures may require scaling by a data dependency factor, as described in subsection D.3.3 on page D-14.

Figure D–6. Expansion Bus Current Versus Transfer Rate and Wait States

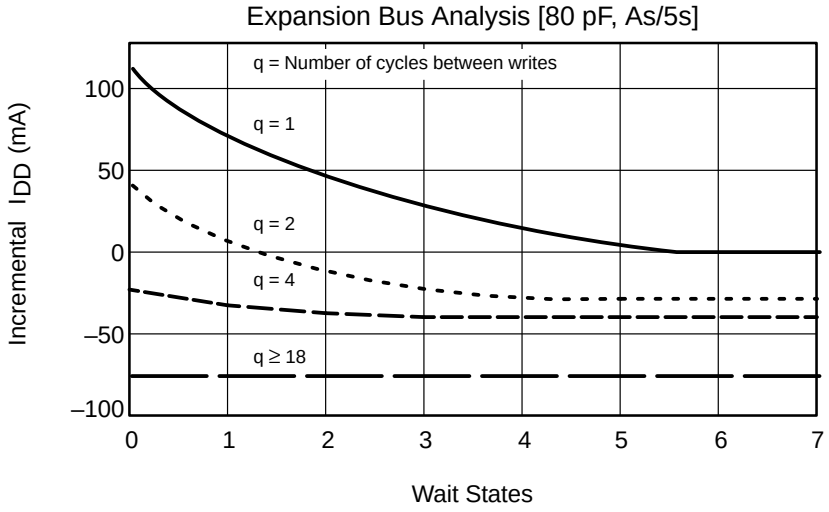
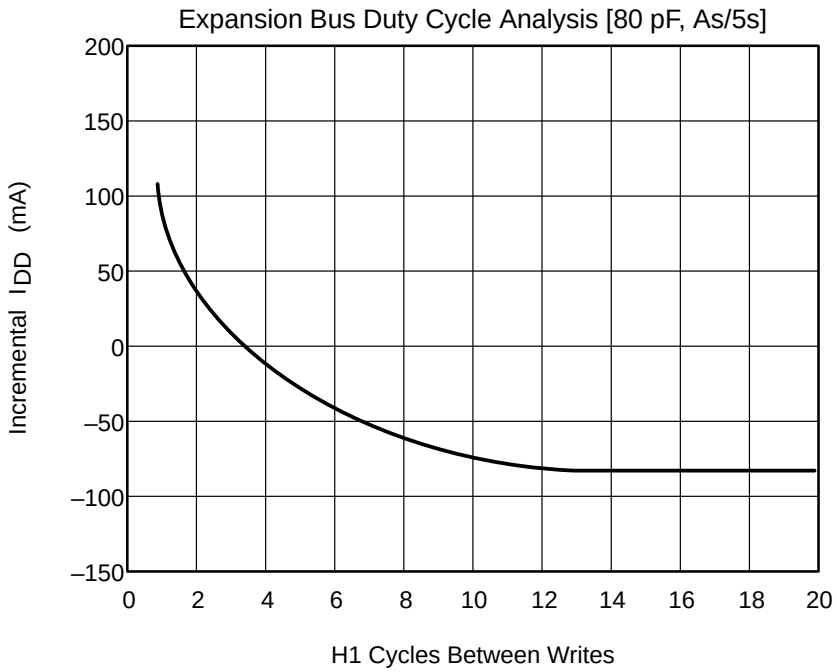


Figure D–7.Expansion Bus Current Versus Transfer Rate at Zero Wait States



D.3.3 Data Dependency

Data dependency of current for the primary and expansion buses is expressed as a scale factor that is a percentage of the maximum current exhibited by either of the two buses. Data dependencies for the primary and expansion buses are shown in Figure D–8 and Figure D–9, respectively.

These two figures show normalized weighting factors that you can use to scale current requirements on the basis of patterns in data being written on the external buses. The range of possible weighting factors forms a trapezoidal pattern bounded by extremes of data values. As can be seen from Figure D–8 and Figure D–9, the minimum current is exhibited by writing all 0s, while the maximum current occurs when writing alternating 55555555h and AAAAAAAAh. This condition results in a weighting factor of 1, which corresponds to using the values from Figure D–4 and/or Figure D–5 directly.

As with internal bus operations, data dependencies for the external buses are well defined, but accurate prediction of data patterns is often either impossible or impractical. Therefore, unless you have precise knowledge of data patterns, you should use an estimate of a median or average value for scale factor. If you assume that data will be neither 5s and As nor all 0s and will be varying randomly, a value of 0.85 is appropriate. Otherwise, if you prefer a conservative approach, you can use a value of 1.0 as an upper bound.

Regardless of the approach you take for scaling, once you determine the scale factors for primary and expansion buses, apply these factors to scale the current values found by using the graphs in the previous two subsections. For example, if a nominal scale factor of 0.85 is used and the system uses zero wait states with two cycles between accesses on both the primary and expansion buses, the current contribution from the two buses is as follows:

Primary: 0.85 x 80 mA = 68 mA
Expansion: 0.85 x 40 mA = 34 mA

Figure D–8.Primary Bus Current Versus Data Complexity Derating Curve

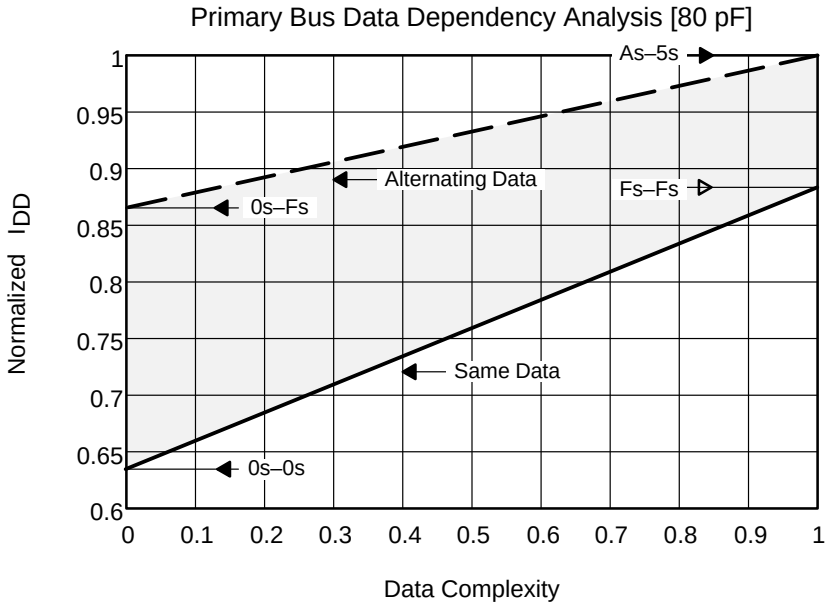
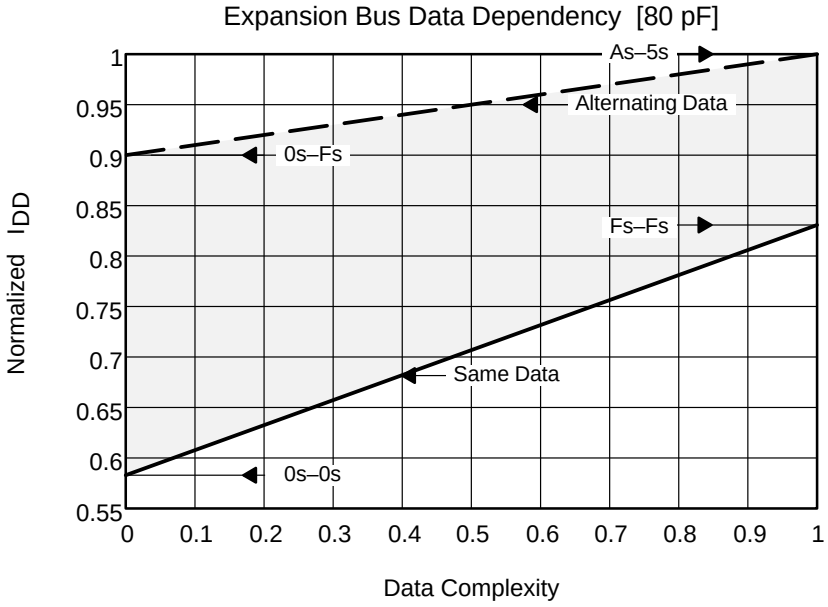


Figure D–9.Expansion Bus Current Versus Data Complexity Derating Curve



D.3.4 Capacitive Load Dependence

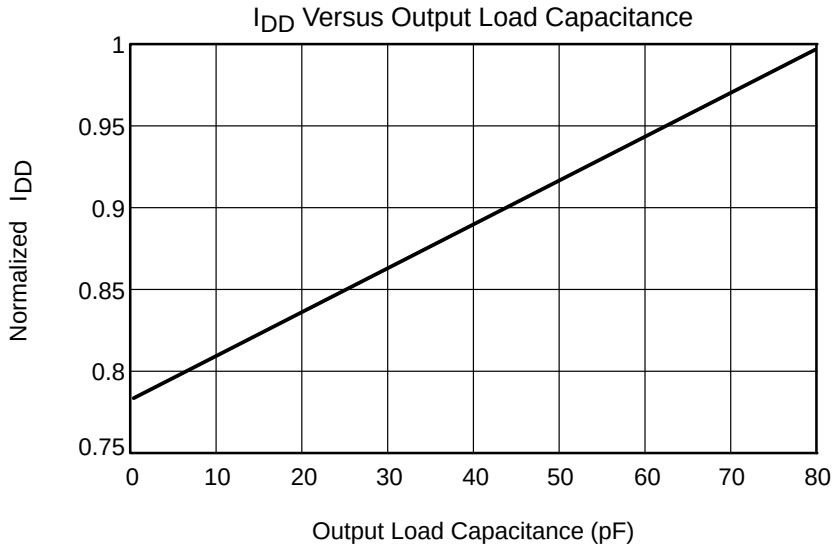
Once you account for cycle timing and data dependencies, you should include capacitive loading effects in a manner similar to that of data dependency. Figure D–10 shows the scale factor to be applied to the current values obtained above as a function of actual load capacitance if the load capacitance presented to the buses is less than 80 pF.

In the previous example, if the load capacitance is 20 pF instead of 80 pF, a scale factor of 0.84 is used, yielding:

Primary: 0.84×68 mA = 57.12 mA
Expansion: 0.84×34 mA = 28.56 mA

The slope of the load capacitance line in Figure D–10 is 0.26% normalized I_{DD} per pF. While this slope may be used to interpolate scale factors for loads greater than 80 pF, the TMS320C30 is specified to drive output loads of less than 80 pF, and interface timings cannot be guaranteed at higher loads. With data dependency and capacitive load scale factors applied to the current values for primary and expansion buses, the total supply current required for the device for a particular application can be calculated, as described in the next section.

Figure D–10. Current Versus Output Load Capacitance



D.4 Calculation of Total Supply Current

The previous sections have discussed currents contributed by several sources on the TMS320C30. Because determinations of actual current values are unique and independent for each source, each current source was discussed separately. In an actual application, however, the sum of the independent contributions from each current determines the total current requirement for the device. This total current value is exhibited as the total current supplied to the device through all of the V_{DD} inputs and returned through the V_{SS} connections.

Note that numerous V_{DD} and V_{SS} pins on the device are routed to a variety of internal connections, not all of which are common. Externally, however, all of these pins should be connected in parallel to 5 V and ground planes, respectively, with as low impedance as possible.

As mentioned previously, because different program segments inherently perform different operations that are often quite distinct from each other, it is typically appropriate to consider current for each of the different segments independently. Once this is done, peak current requirements are readily obtained. Further, you can use average current calculations to determine heating effects of power dissipation. In turn, you can use these effects to determine thermal management considerations.

D.4.1 Combining Supply Current Due to All Components

To determine the total supply current requirements for any given program activity, calculate each of the appropriate components and combine them in the following sequence:

1) Start with 110-mA quiescent current requirement.

2) Add 55 mA for internal operations unless the device is dormant, as during execution of IDLE, NOPs, or branches-to-self, or performance of internal and/or external bus operations using an RPTS instruction (see subsection D.2.2 on page D-5). Internal or external bus operations executed via RPTS do not contribute an internal operations power supply current component and hence do not add 55 mA to quiescent current. Therefore, current components in the next two steps might still be required, even though the 55 mA is omitted.

3) If significant internal bus operations are being performed (see subsection D.2.2 on page D-5), add the calculated current value.

4) If external writes are being performed at high speed (see section D.3 on page D-9), add 60 mA and then add the values calculated for primary and expansion bus current components. If only one external bus is being used, the appropriate incremental current for the unused bus should still be included because the current offsets include components required for operating both buses. Note, however, that, as discussed previously, the total current contribution for external buses, including baseline, must always be positive.

The current value resulting from summing these components is the total device current requirement for a given program activity.

D.4.2 Supply Voltage, Operating Frequency, and Temperature Dependencies

Current dependencies specific to each supply current component (such as internal or external bus operations) are discussed in subsection D.1.2 on page D-2. Supply voltage level, operating temperature, and operating frequency affect requirements for the total supply current and must be maintained within required device specifications.

Once the total current for a particular program segment has been determined, the dependencies that affect total current requirements are applied as a scale factor in the same manner as data dependencies discussed in other sections. Figure D–11 shows the relative scale factors to be applied to the supply current values as a function of both V_{DD} and operating frequency.

Power supply current consumption does not vary significantly with operating temperature. However, if desired, a scale factor of 2% normalized I_{DD} per 50°C change in operating temperature may be used to derate current within the specified range noted in the TMS320C30 data sheet. This temperature dependence is shown graphically in Figure D–12. Note that a temperature scale factor of 1.0 corresponds to current values at 25°C, which is the temperature at which all other references in the document are made.

Figure D–11. Current Versus Frequency and Supply Voltage

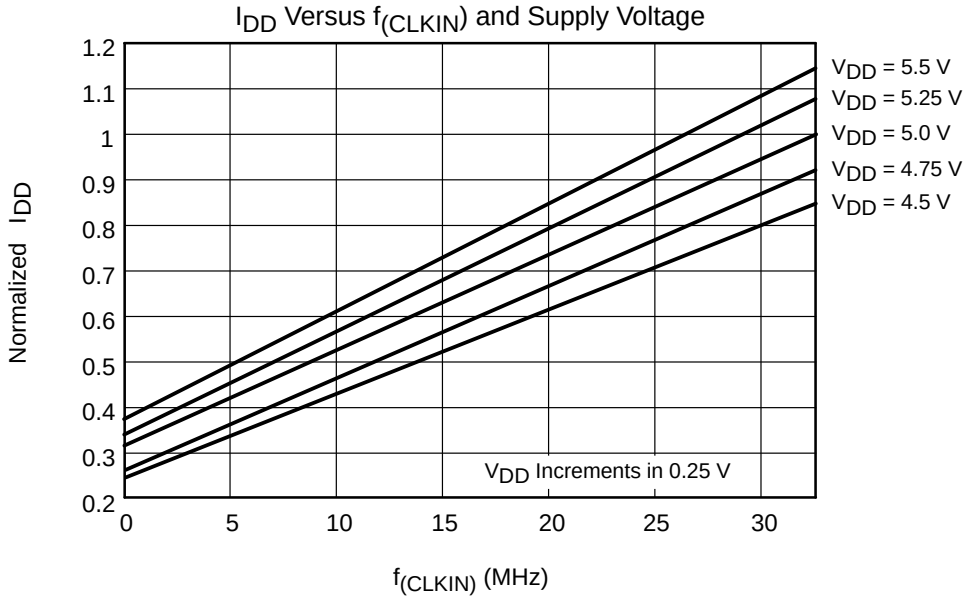
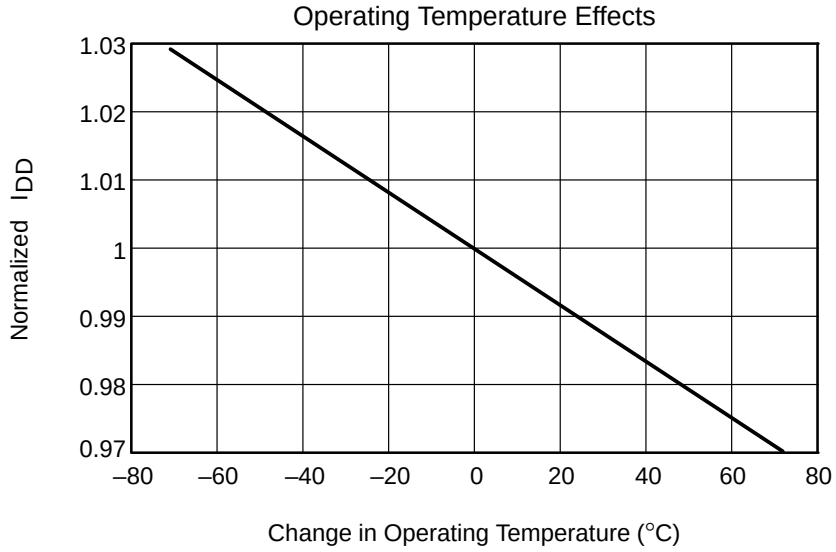


Figure D–12. Current Versus Operating Temperature Change



D.4.3 Design Equation

The procedure for determining the power supply current requirement can be summarized in the following equation:

$$I = \left(I_q + I_{iops} + I_{ibus} + I_{xbus}\right) \times FV \times T$$

where

$I_q = 110$ mA

$I_{iops} = 55$ mA

$I_{ibus} = D_1 \times f_1$ (see Table D–1)

$I_{xbus} = I_{prim} + I_{exp}$

with

$I_{base} = 60$ mA

$I_{prim} = D_2 \times C_2 \times f_2$ (see Table D–1)

$I_{exp} = D_3 \times C_3 \times f_3$ (see Table D–1)

FV is the scale factor for frequency and supply voltage, and

T is the scale factor for operating temperature.

Table D–1 describes the symbols used in the power supply current equation. The table displays figure numbers from which the value can be obtained.

Table D–1. Current Equation Symbols

Symbol	Description	Graph/Value
I_q	Quiescent Current	110 mA
I_{iops}	Internal Operations Current	55 mA
I_{ibus}	Internal Bus Operations Current	†
D_1	Internal Bus Data Scale Factor	Figure D–3
f_1	Internal Bus Current Requirement	Figure D–2
I_{xbus}	External Bus Operations Current	†
I_{base}	External Bus Base Current	60 mA
I_{prim}	Primary Bus Operations Current	†
D_2	Primary Bus Data Scale Factor	Figure D–8
C_2	Primary Bus Cap Load Scale Factor	Figure D–10
f_2	Primary Bus Current Requirement	Figure D–4 or Figure D–5
I_{exp}	Expansion Bus Operations Current	†
D_3	Expansion Bus Data Scale Factor	Figure D–9
C_3	Expansion Bus Cap Load Scale Factor	Figure D–10
f_3	Expansion Bus Current Requirement	Figure D–6 or Figure D–7
FV	Freq/Supply Voltage Scale Factor	Figure D–11
T	Temperature Scale Factor	Figure D–12

† See equation in subsection D.4.3 on page D-21.

D.4.4 Peak Versus Average Current

If current is observed over the course of an entire program, some segments will usually exhibit significantly different levels of current required for different durations of time. For example, a program may spend 80% of its time performing internal operations, drawing a current of 250 mA, and spend the remaining 20% of its time performing writes at full speed to the expansion bus, drawing 300 mA.

While knowledge of peak current levels is important in order to establish power supply requirements, some applications require information about average current. This is particularly significant if periods of high peak current are short in duration. Average current can be obtained by performing a weighted sum of the currents due to the various independent program segments over time. In the example above, the average current can be calculated as follows:

$$I = 0.8 \times 250\ mA + 0.2 \times 300\ mA = 260\ mA$$

Using this approach, average current for any number of program segments can be calculated.

D.4.5 Thermal Management Considerations

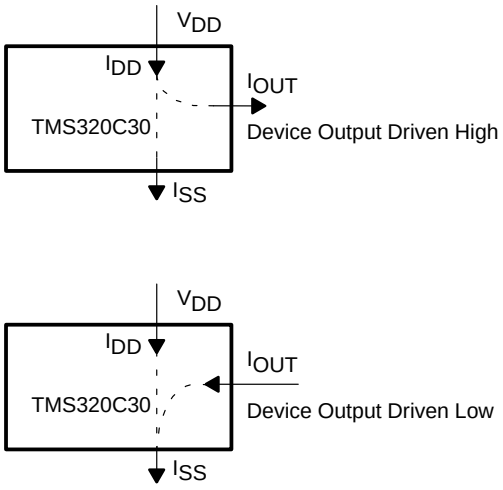
Heating characteristics of the TMS320C30 depend on power dissipation, which in turn depends on power supply current. When you make thermal management calculations, you must consider the manner in which power supply current contributes to power dissipation and to the time constant of the TMS320C30 package thermal characteristics.

Depending on sources and destinations of current on the device, some current contributions to I_{DD} do not constitute a component of power dissipation at 5 volts. Accordingly, if you use the total current flowing into V_{DD} to calculate power dissipation at 5 volts, you will obtain erroneously large values for power dissipation. Power dissipation is defined as:

$$P = I \times V$$

(where P is power, I is current, and V is voltage). If device outputs are driving any DC load to a logic high level, only a minor contribution is made to power dissipation because CMOS outputs typically drive to a level within a few tenths of a volt of the power supply rails. If this is the case, subtract these current components out of the total supply current value; then calculate their contribution to power dissipation separately and add it to the total power dissipation (see Figure D–13). If this is not done, these currents resulting from driving a logic high level into a DC load will cause unrealistically high power dissipation values. The error occurs because the currents resulting from driving a logic high level into a DC load will appear as a portion of the current used to calculate power dissipation due to V_{DD} at 5 volts.

Figure D–13. Load Currents



Furthermore, external loads draw supply-only current when outputs are being driven high, because, when outputs are in the logic 0 state, the device is sinking current that is supplied from an external source. Therefore, the power dissipation due to this current component will not have a contribution through I_{DD} but will contribute to power dissipation with a magnitude of:

$$P = V_{OL} \times I_{OL}$$

where V_{OL} is the low-level output voltage and I_{OL} is the current being sunk by the output as shown in Figure D–13. The power dissipation component due to outputs being driven low should be calculated and added to the total power dissipation.

When outputs with DC loads are being switched, the power dissipation components from outputs being driven high and outputs being driven low are averaged and added to the total device power dissipation. You should calculate power components due to DC loading of the outputs separately for each program segment before you calculate average power.

Note that any unused inputs that are left disconnected may float to a voltage level that will cause input buffer circuits to remain in the linear region and therefore contribute a significant component to power supply current. Accordingly, any unused inputs should be made inactive by being either grounded or pulled high if absolute minimum power dissipation is desired. If several unused inputs must be pulled high, they may be pulled high together through one resistor to minimize component count and board space.

When you use power dissipation values to determine thermal management considerations, you should use the average power unless the time duration of individual program segments is long. The thermal characteristics of the TMS320C30 in the 181-pin grid analysis (PGA) package are exponential in nature, with a time constant t = 4.5 minutes. Therefore, when subjected to a change in power, the temperature of the device package will, after 4.5 minutes, reach approximately 63% of the total temperature change. Accordingly, if the time duration of program segments exhibiting high power dissipation values is short (on the order of a few seconds), you can use average power, calculated in the same manner as average current (as described in subsection D.4.4 on page D-22).

Otherwise, you should calculate maximum device temperature on the basis of the actual time duration of the program segments involved. For example, if a particular program segment lasts for seven minutes, then, using the exponential function, you can calculate that a device will reach approximately 80% of the temperature due to the total power dissipation during the program segment.

Note that the average power should be determined by calculating the power for each program segment (including considerations described above) and performing a time average of these values, rather than simply multiplying the average current as determined in the previous subsection by V_{DD}.

Specific device temperature calculations are made by using the TMS320C30 thermal impedance characteristics included in Chapter 13.

D.5 Example Supply Current Calculations

A Fast Fourier Transform (FFT) represents a typical DSP algorithm. The FFT code in Section D.8 on page D-30 processes data in the RAM blocks and writes the result out to zero-wait-state external SRAM on the primary bus. The program executes out of zero-wait-state external SRAM on the primary bus, and the TMS320C30's cache is enabled. The entire algorithm consists mainly of internal bus operations and so includes quiescent and internal operations in general. At the end of processing, the 1024 results are written out on the primary bus. Therefore, the algorithm exhibits a higher current requirement during the write portion, where the external bus is being used significantly.

D.5.1 Processing

The processing portion of the algorithm is 95% of the total algorithm. During this portion, the power supply current is required only for the internal circuitry. Data is processed in several loops that compose a majority of the algorithm. During these loops, two operands are transferred on every cycle. The current required for internal bus usage, then, is 55 mA, taken from Figure D–2 on page D-7. The data is assumed to be random. A data value scale factor of 0.8 is used from Figure D–3 on page D-7. This value scales 55 mA, yielding 44 mA for internal bus operations. Adding 44 mA to the quiescent current requirement and internal operations current requirement yields a current requirement of 209 mA for the major portion of the algorithm.

$$I = I_q + I_{iops} + I_{ibus}$$

$$I = 110\ mA + 55\ mA + (55\ mA)(0.8) = 209\ mA$$

D.5.2 Data Output

The portion of the algorithm corresponding to writing out data is approximately 5% of the total algorithm. Again, the data that is being written is assumed to be random. From Figure D–3 on page D-7 and Figure D–8 on page D-15, scale factors of 0.80 and 0.85 are used for derating due to data value dependency for internal and primary buses, respectively. During the data dump portion of the code, a load and store are performed every cycle; however, the parallel load/store instruction is in an RPTS loop, so there is no contribution due to internal operations, because the instruction is fetched only once. The only internal contributions are due to quiescent and internal bus operations. Figure D–4 on page D-11 indicates a 170-mA current contribution due to back-to-back zero-wait-state writes, and Figure D–6 on page D-13 indicates a –80-mA contribution due to the expansion bus being idle (that is, with more than 18 H1 cycles between writes). Therefore, the total contribution due to this portion of the code is:

$$I = I_q + I_{ibus} + I_{xbus}$$

or,

$$I = 110 + (55\ mA)(0.8) + 60\ mA - 80\ mA + (170\ mA)(0.85)$$
$$= 278.5\ mA$$

D.5.3 Average Current

The average current is derived from the two portions of the algorithm. The processing portion took 95% of the time and required about 210 mA, and the data dump portion took the other 5% and required about 280 mA. The average is calculated as:

$$I_{avg} = (0.95)(21\ mA) + (0.05)(280\ mA) = 213.5\ mA$$

From the thermal characteristics specified in Chapter 13, it can be shown that this current level corresponds to a case temperature of 43°C. This temperature meets the maximum device specification of 85°C and hence requires no forced air cooling.

D.5.4 Experimental Results

A photograph of the power supply current for the FFT is shown Section D.7 on page D-29. During the FFT processing, the measured current varied between 180 and 220 mA. The peak of the current during external writes was 270 mA, and the average current requirement, as measured on a digital multimeter, was 200 mA. The calculations yielded results that were extremely close to the actual measured power supply current.

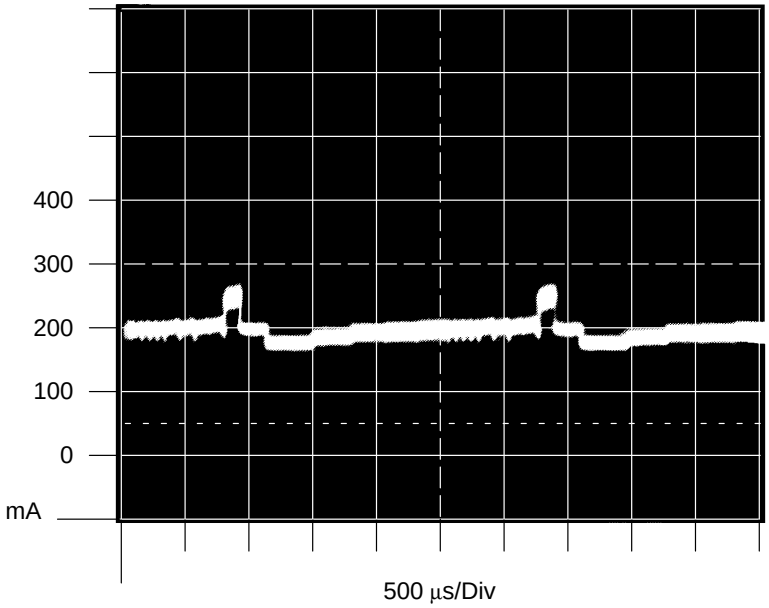
D.6 Summary

An accurate power supply current requirement for the TMS320C30 cannot be expressed simply in terms of operating frequency, supply voltage, and output load capacitance. The specification must be more complete and depends on device functionality and system parameters. The current components related to device functionality are due to quiescent current, internal operations, internal bus operations, and external bus operations. Those related to system parameters are due to operating frequency, supply voltage, output load capacitance, and operating temperature. The typical power supply current requirement is 200 mA, and the minimum, or quiescent, is 110 mA.

This application report presents information required to determine power supply specifications. Specifications are based on an algorithm's use of internal and external buses on the TMS320C30. As devices become more complex, the calculation of power dissipation becomes more critical.

The maximum current requirement is 600 mA and occurs only under *worst case* conditions: writing alternating data (AAAAAAAAh to 55555555h) out of both external buses simultaneously every cycle, with 80 pF loads and running at 33 MHz.

D.7 Photo of I_{DD} for FFT



Input Clock Frequency = 33 MHz
Voltage Level = 5.0 V_{DD}

D.8 FFT Assembly Code

```
        .GLOBL    FFT
        .GLOBL    N
        .GLOBL    M
        .GLOBL    SINE

SINTAB:                         ; setup
        .WORD     SINE
RAM0:
        .WORD     809800h
OUTBUF:
        .WORD     800h

        .TEXT

FFT:    LDP       SINTAB    ; processing portion:
                            ; quiescent, internal and
                            ; bus operations


        LDI       N,IR0
        LSH       –1,IR0

; LENGTH–TWO BUTTERFLIES

        LDI       @RAM0,AR0
        LDI       IR0,RC
        SUBI      1,RC

        RPTB      BLK1
        ADDF      *+AR0,*AR0++,R0
        SUBF      *AR0,*–AR0,R1
BLK1    STF       R0,*–AR0
||      STF       R1,*AR0++

; FIRST PASS OF THE DO–20 LOOP (STAGE K=2 IN DO–10 LOOP)

        LDI       @RAM0,AR0
        LDI       2,IR0
        LDI       N,RC
        LSH       –2,RC
        SUBI      1,RC

        RPTB      BLK2
        ADDF      *+AR0(IR0),*AR0++(IR0),R0
        SUBF      *AR0,*–AR0(IR0),R1
        NEGF      *+AR0,R0
||      STF       R0,*–AR0(IR0)
BLK2    STF       R1,*AR0++(IR0)
||      STF       R0,*+AR0

; MAIN LOOP (FFT STAGES)
```

```
      LDI     N,IR0
      LSH     -2,IR0
      LDI     3,R5
      LDI     1,R4
      LDI     2,R3
LOOP  LSH     -1,IR0
      LSH     1,R4
      LSH     1,R3

; INNER LOOP (DO-20 LOOP IN THE PROGRAM)

      LDI     @RAM0,AR5
INLOP:
      LDI     IR0,AR0
      ADDI    @SINTAB,AR0
      LDI     R4,IR1
      LDI     AR5,AR1
      ADDI    1,AR1
      LDI     AR1,AR3
      ADDI    R3,AR3
      LDI     AR3,AR2
      SUBI    2,AR2
      ADDI    R3,AR2,AR4
      LDF     *AR5++(IR1),R0
      ADDF    *+AR5(IR1),R0,R1
      SUBF    R0,*++AR5(IR1),R0
||    STF     R1,*-AR5(IR1)
      NEGF    R0
      NEGF    *++AR5(IR1),R1
||    STF     R0,*AR5
      STF     R1,*AR5

; INNERMOST LOOP

      LDI     N,IR1
      LSH     -2,IR1
      LDI     R4,RC
      SUBI    2,RC

      RPTB    BLK3
      MPYF    *AR3,*+AR0(IR1),R0
      MPYF    *AR4,*AR0,R1
      MPYF    *AR4,*+AR0(IR1),R1
||    ADDF    R0,R1,R2
      MPYF    *AR3,*AR0++(IR0),R0
      SUBF    R0,R1,R0
      SUBF    *AR2,R0,R1
      ADDF    *AR2,R0,R1
||    STF     R1,*AR3++
      ADDF    *AR1,R2,R1
||    STF     R1,*AR4--
      SUBF    R2,*AR1,R1
```

```
||    STF     R1,*AR1++
BLK3  STF     R1,*AR2--

      SUBI    @RAM0,AR5
      ADDI    R4,AR5
      CMPI    N,AR5
      BLTD    INLOP
      ADDI    @RAM0,AR5
      NOP
      NOP

      ADDI    1,R5
      CMPI    M,R5
      BLE     LOOP


DUMP  LDI     @RAM0,AR0       ; data dump portion
      LDI     @OUTBUF,AR1     ; quiescent, internal bus


      LDF     *AR0++,R0       ; ops and primary bus ops
      RPTS    N-2
      LDF     *AR0++,R0
||    STF     R0,*AR1++
      STF     R0,*AR1++

      LDI     RAM0,AR1

      LDI     @RAM0,AR0       ; swap RAM banks
      XOR     400h,AR0
      STI     AR0,*AR1

      B       FFT
      .END
```

Appendix E

SMJ320C3x Digital Signal Processor Data Sheet

This appendix contains the standalone data sheet for the military version of the ’C3x digital signal processor, the SMJ320C3x Digital Signal Processor.

Appendix F

Analog Interface Peripherals and Applications

Texas Instruments (TI) offers many products for total system solutions, including memory options, data acquisition, and analog input/output devices. This appendix describes a variety of devices that interface directly to the TMS320 DSPs in rapidly expanding applications.

Major topics discussed in this appendix are listed below.

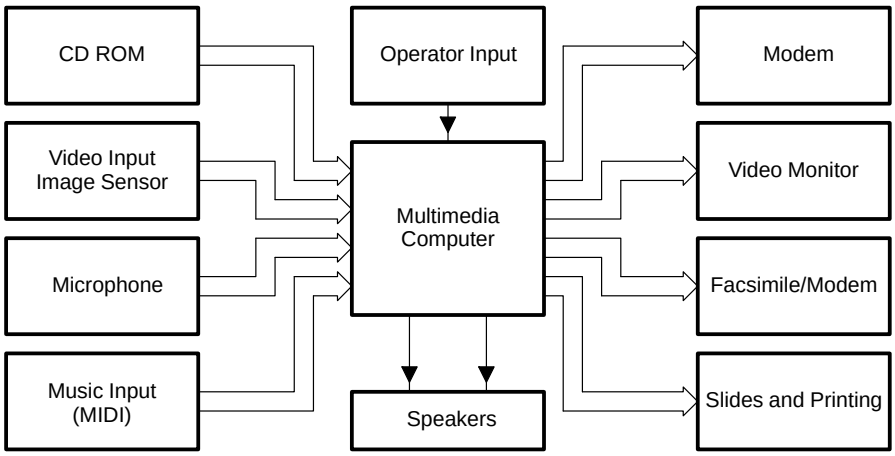
Topic **Page**

F.1 Multimedia Applications

Multimedia integrates different media through a centralized computer. These media can be visual or audio and can be input to or output from the central computer via a number of technologies. The technologies can be digital-based or analog-based (such as audio or video tape recorders). The integration and interaction of media enhance the transfer of information and can accommodate both analysis of problems and synthesis of solutions.

Figure F–1 shows both the central role of the multimedia computer and the multimedia system's ability to integrate the various media to optimize information flow and processing.

Figure F–1. System Block Diagram



F.1.1 System Design Considerations

Multimedia systems can include various grades of audio and video quality. The most popular video standard currently used (VGA) covers 640 $\times$ 480 pixels with 1, 2, 4, and 8-bit memory-mapped color. Also, 24-bit true color is supported, and 1024 $\times$ 768 (beyond VGA) resolution has emerged. There are two grades of audio. The lower grade accommodates 11.25-kHz sampling for 8-bit monaural systems, while the higher grade accommodates 44.1-kHz sampling for 16-bit stereo.

Audio specifications include a musical instrument digital interface (MIDI) with compression capability, which is based on keystroke encoding, and an input/output port with a three-disc voice synthesizer. In the media control area, video disc, CD audio, and CD ROM player interfaces are included. Figure F–2 shows a multimedia subsystem.

The TLC32047 wide-band analog interface circuit (AIC) is well suited for multimedia applications because it features wide-band audio and up to 25-kHz sampling rates. The TLC32047 is a complete analog-to-digital and digital-to-analog interface system for the TMS320 DSPs. The nominal bandwidths of the filters accommodate 11.4 kHz, and this bandwidth is programmable. The application circuit shown in Figure F–2 handles both speech encoding and modem communication functions, which are associated with multimedia applications.

Figure F–2. Multimedia Speech Encoding and Modem Communication

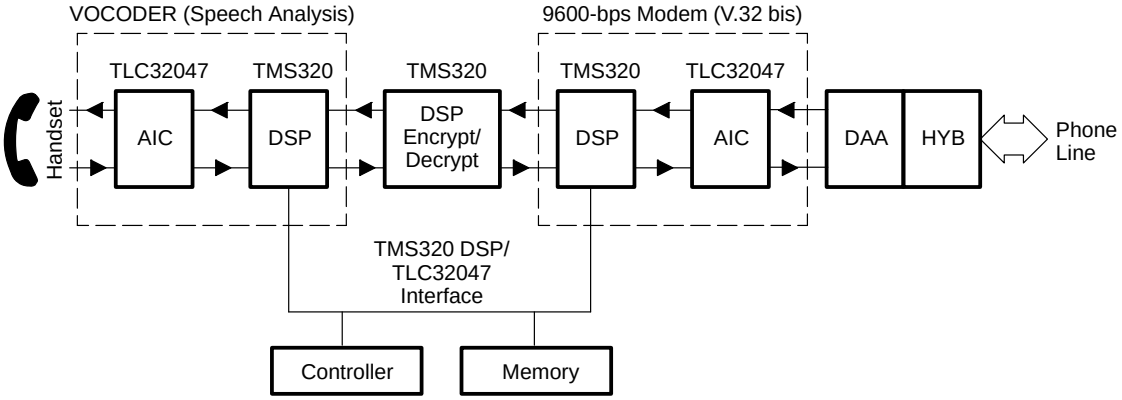
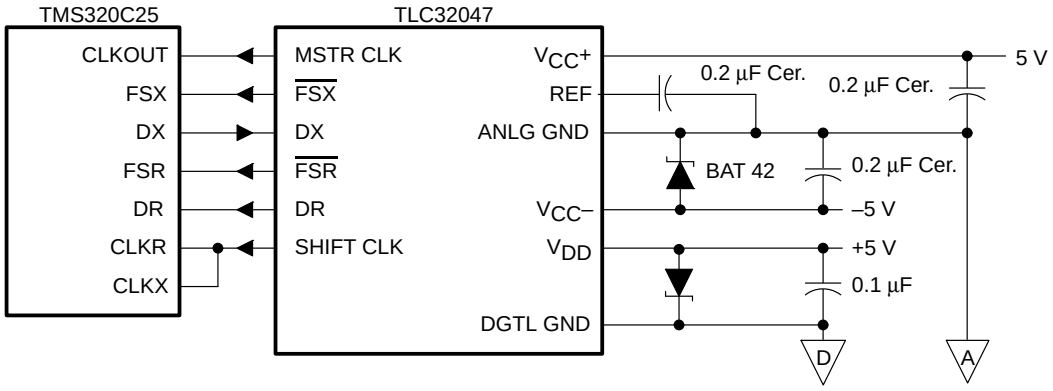


Figure F–3 shows the interfacing of the TMS320C25 DSP to the TLC32047 AIC, which constitutes a building block of the 9600-bps V.32 bis modem shown in Figure F–2.

Figure F–3. TMS320C25 to TLC32047 Interface



F.1.2 Multimedia-Related Devices

As shown in Table F–1 and Table F–2, TI provides a complete array of analog and graphics interface devices. These devices support the TMS320 DSPs for complete multimedia solutions.

Table F–1. Data Converter ICs

Device	Description	I/O	Resolution (Bits)	Conversion CLK Rate	Application
TLC320AC01	Analog interface (5 V only)	Serial	14	43.2 kHz	Portable modem and speech, multimedia
TLC32047	Analog interface (11.4 kHz BW) (AIC)	Serial	14	25 kHz	Speech, modem, and multimedia
TLC32046	Analog interface (AIC)	Serial	14	25 kHz	Speech and modems
TLC32044	Analog interface (AIC)	Serial	14	19.2 kHz	Speech and modems
TLC32040	Analog interface (AIC)	Serial	14	19.2 kHz	Speech and modems
TLC34075/6	Video palette	Parallel	Triple 8	135 MHz	Graphics
TLC34058	Video palette	Parallel	Triple 8	135 MHz	Graphics
TLC5502/3	Flash ADC	Parallel	8	20 MHz	Video
TLC5602	Video DAC	Parallel	8	20 MHz	Video
TLC5501	Flash ADC	Parallel	6	20 MHz	Video
TLC5601	Video DAC	Parallel	6	20 MHz	Video
TLC1550/1	ADC	Parallel	10	150 kHz	Servo ctrl / speech
TLC32071	Analog interface (AIC)	Parallel	8	1 MHz	Servo ctrl / disk drive
TMS57013/4	Dual audio DAC + digital filter	Serial	16/18	32, 37.8, 44.1, 48 kHz	Digital audio

Table F–2. Switched-Capacitor Filter ICs

Device	Function	Order	Roll-Off	Power Out	Power Down
TLC2470	Differential audio filter amplifier	4	5 kHz	500 mW	Yes
TLC2471	Differential audio filter amplifier	4	3.5 kHz	500 mW	Yes
TLC10/20	General-purpose dual filter	2	CLK ÷ 50 CLK ÷ 100	N/A	No
TLC04/14	Low pass, Butterworth filter	4	CLK ÷ 50 CLK ÷ 100	N/A	No

For application assistance or additional information, please call TI Linear Applications at (214) 997–3772.

F.2 Telecommunications Applications

The TI linear product line focuses on three primary telecommunications application areas:

- Subscriber instruments (telephones, modems, etc.)

  Includes the TCM508x DTMF tone encoder family, the TCM150x tone ringer family, the TCM1520 ring detector, and the TCM3105 FSK modem.

- Central office line card products

  Includes the TCM29Cxx combo (combined PCM filter plus codec) family, the TCM420x subscriber line control circuit family, and the TCM1030/60 line card transient protector.

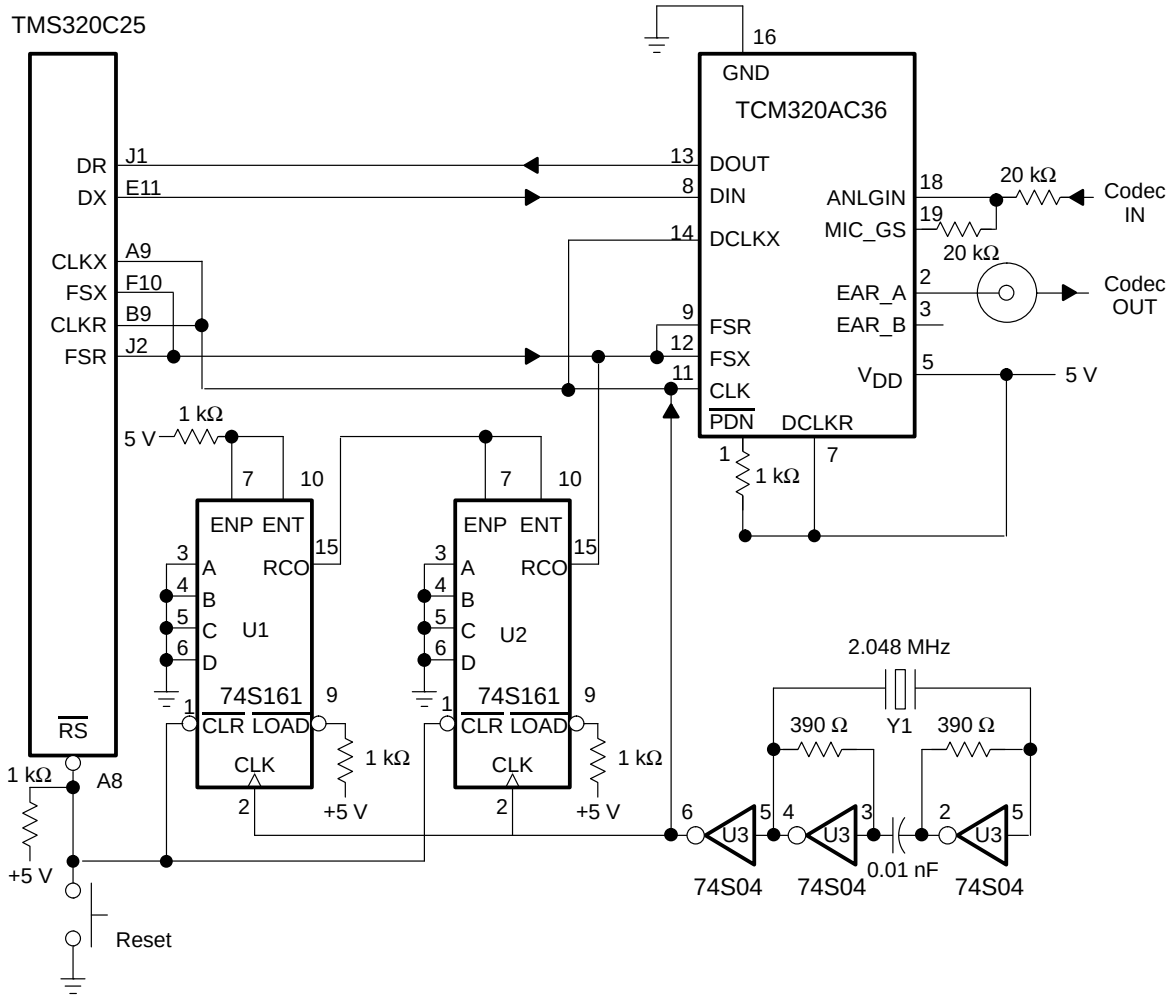
- Personal communications products

  Includes the TCM320AC3x family of 5-volt voice-band audio processors (VBAP).

TI continues to develop new telecom integrated circuits, such as a high-performance three-volt combo family for personal communications applications and an RF power amplifier family for hand-held and mobile cellular phones.

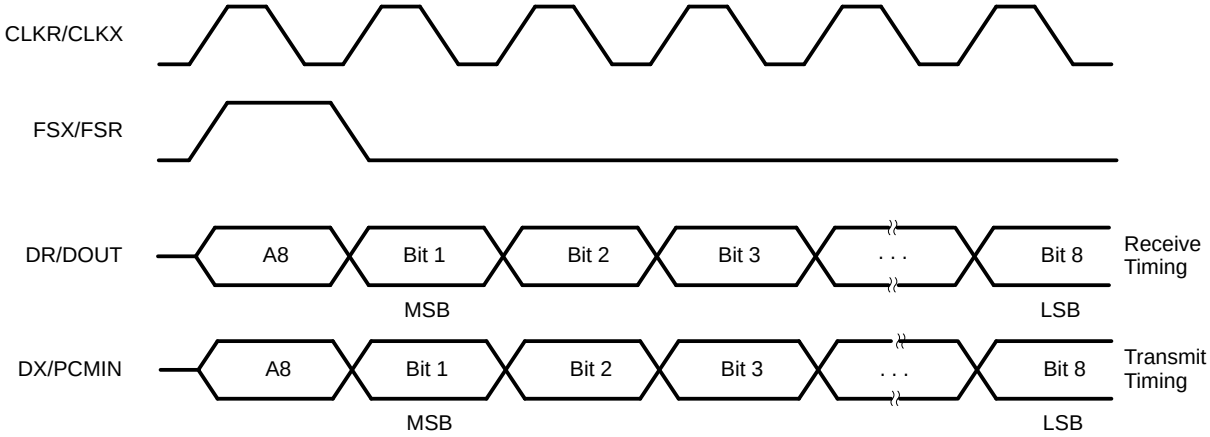
System Design Considerations. The size, network complexity, and compatibility requirements of telecommunications central office systems create demanding performance requirements. Combo voice-band filter performance is typically $\pm$ 0.15 dB in the passband. Idle channel noise must be on the order of 15 dBrnc0. Gain tracking (S/Q) and distortion must also meet stringent requirements. The key parameters for a SLIC device are gain, longitudinal balance, and return loss.

Figure F–4. Typical DSP/Combo Interface



The TCM320AC36 combo interfaces directly to the TMS320C25 serial port with a minimum of external components, as shown in Figure F–4. Half of hex inverter U3 and crystal Y1 form an oscillator that provides clock timing to the TCM320AC36. The synchronous four-bit counters U1 and U2 generate an 8-kHz frame sync signal. DCLKR on the TCM320AC36 is connected to V_{DD}, placing the combo in fixed data-rate mode. Two 20-kΩ resistors connected to ANLGIN and MIC_GS set the gain of the analog input amplifier to 1. The timing is shown in Figure F–5.

Figure F–5. DSP/Combo Interface Timing



Telecommunications-Related Devices. Data sheets for the devices in Table F–3 on page F-8 are contained in the *1991 Telecommunications Circuits Databook* (literature number SCTD001B). To request your copy, contact your nearest TI field sales office or call the Literature Response Center at (800) 477–8924.

Table F–3. Telecom Devices

Device Number	Coding Law	Clock Rates MHz†	# of Bits	Comments
		Codec/Filter		
TCM29C13	A and μ	1.544, 1.536, 2.048	8	C.O. and PBX line cards
TCM29C14	A and μ	1.544, 1.536, 2.048	8	Includes 8th-bit signal
TCM29C16	μ	2.048	8	16-pin package
TCM29C17	A	2.048	8	16-pin package
TCM29C18	μ	2.048	8	Low-cost DSP interface
TCM29C19	μ	1.536	8	Low-cost DSP interface
TCM29C23	A and μ	Up to 4.096	8	Extended frequency range
TCM29C26	A and μ	Up to 4.096	8	Low-power TCM29C23
TCM320AC36	μ and Linear	Up to 4.096	8 and 13	Single voltage (+5) VBAP
TCM320AC37	A and Linear	Up to 4.096	8 and 13	Single voltage (+5) VBAP
TCM320AC38	μ and Linear	Up to 4.096	8 and 13	Single voltage (+5) GSM
TCM320AC39	A and Linear	Up to 4.096	8 and 13	Single voltage (+5) GSM
TP3054/64	μ	1.544, 1.536, 2.048	8	National Semiconductor second source
TP3054/67	A	1.544, 1.536, 2.048	8	National Semiconductor second source
TLC320AC01	Linear	43.2 kHz	14	5-volt-only analog interface
TLC32040/1	Linear	Up to 19.2-kHz sampling	14	For high-dynamic linearity
TLC32044/5	Linear	Up to 19.2-kHz sampling	14	For high-dynamic linearity
TLC32046	Linear	Up to 25-kHz sampling	14	For high-dynamic linearity
TLC32047	Linear	Up to 25-kHz sampling	14	For high-dynamic linearity
		Transient Suppressor		
TCM1030	Transient suppressor for SLIC-based line card			(30 A max)
TCM1060	Transient suppressor for SLIC-based line card			(60 A max)

† Unless otherwise noted

Table F–4 is a list of switched-capacitor filter ICs.

Table F–4. Switched-Capacitor Filter ICs

Device	Function	Order	Roll-Off	Power Out	Power Down
TLC2470	Differential audio filter amplifier	4	5 kHz	500 mW	Yes
TLC2471	Differential audio filter amplifier	4	3.5 kHz	500 mW	Yes
TLC10/20	General-purpose dual filter	2	CLK ÷ 50 CLK ÷ 100	N/A	No
TLC04/14	Low pass, Butterworth filter	4	CLK ÷ 50 CLK ÷ 100	N/A	No

For further information on these telecommunications products, please call (214) 997–3772.

Figure F–6 and Figure F–7 show telecom applications.

Figure F–6. General Telecom Applications

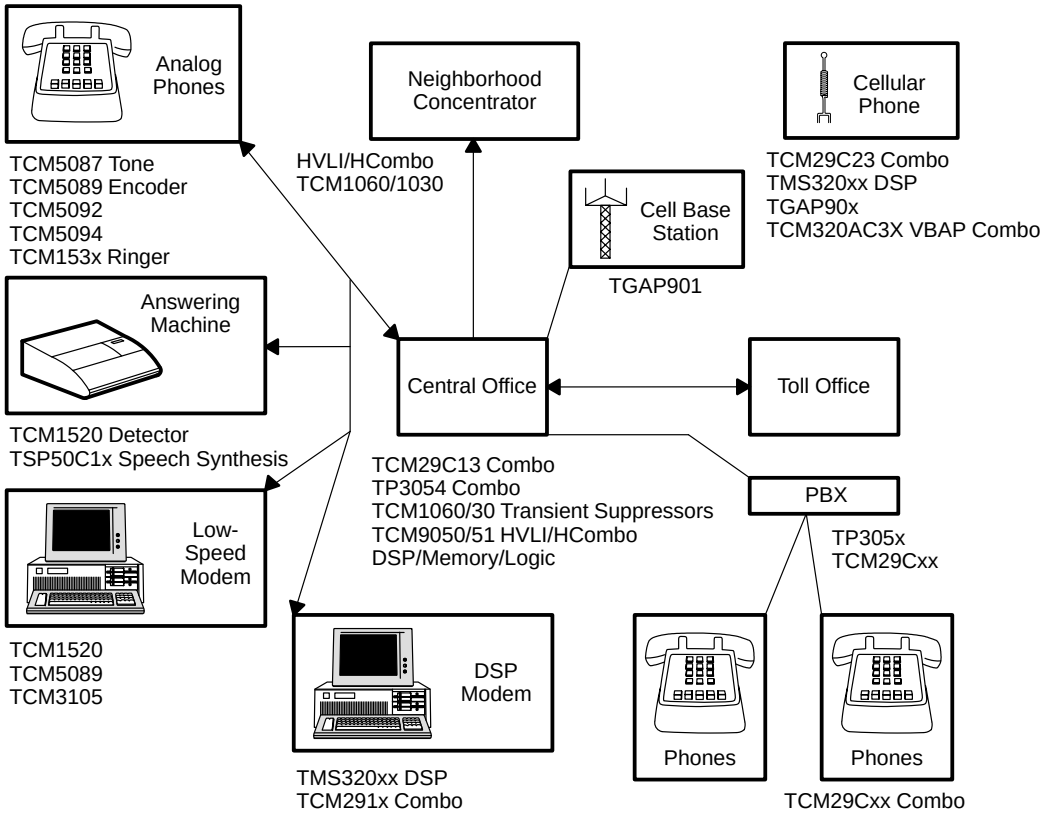
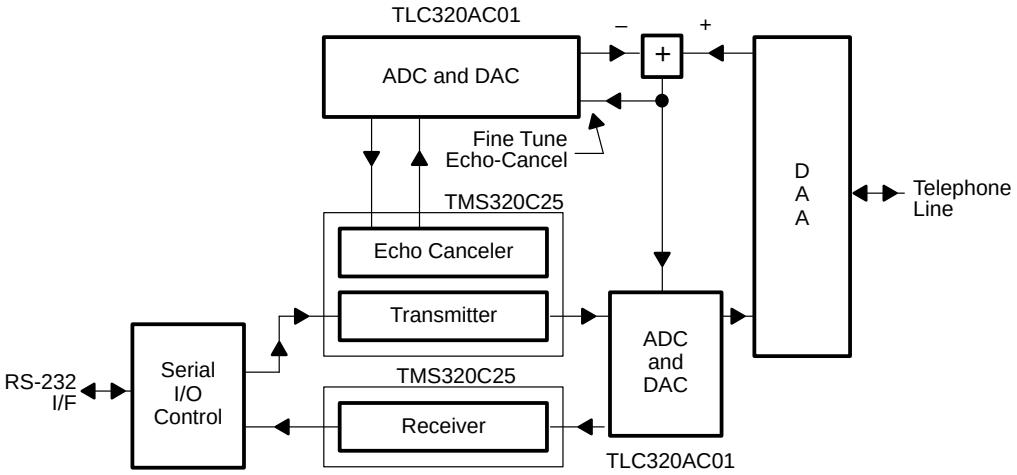


Figure F–7. Generic Telecom Applications



F.3 Dedicated Speech Synthesis Applications

For dedicated speech synthesis applications, TI offers a family of dedicated speech synthesizer chips. This speech technology has been used in a wide range of products, including games, toys, burglar alarms, fire alarms, automobiles, airplanes, answering machines, voice mail, industrial control machines, office machines, advertisements, novelty items, exercise machines, and learning aids.

Dedicated speech synthesis chips are a good alternative for low-cost applications. The speech synthesis technology provided by the dedicated chips is either linear-predictive coding (LPC) or continuously variable slope delta modulation (CVSD). Table F–5 shows the characteristics of the TI voice synthesizers.

Table F–5. TI Voice Synthesizers

Device	Microprocessor	Synthesis Method	I/O Pins	On-Chip Memory (Bits)	External Memory	Data Rate (Bits/Sec)
TSP50C4x	8-bit	LPC–10	20/32	64K/128K	VROM	1200–2400
TSP50C1x	8-bit	LPC–12	10	64K/128K	VROM	1200–2400
TSP53C30	8-bit	LPC–10	20	N/A	From host µP	1200–2400
TSP50C20	8-bit	LPC–10	32	N/A	EPROM	1200–2400
TMS3477	N/A	CVSD	2	None	DRAM	16K–32K

In addition to the speech synthesizers, TI has low-cost memories that are ideal for use with these chips. TI can also be of assistance in developing and processing the speech data that is used in these speech synthesis systems. Table F–6 shows speech memory devices of different capabilities. Additionally, audio filters are outlined in Table F–7.

Table F–6. Speech Memories

TSP60Cxx Family of Speech ROMs				
Family	**Size**	**No. of Pins**	**Interface**	**For use with:**
TSP60C18	256K	16	Parallel 4-bit	TSP50C1x
TSP60C19	256K	16	Serial	TSP50C4x
TSP60C20	256K	28	Parallel/serial 8-bit	TSP50C4x
TSP60C80	1M	28	Serial	TSP50C4x
TSP60C81	1M	28	Parallel 4-bit	TSP50C1x

Table F–7. Switched-Capacitor Filter ICs

Device	Function	Order	Roll-Off	Power Out	Power Down
TLC2470	Differential audio filter amplifier	4	5 kHz	500 mW	Yes
TLC2471	Differential audio filter amplifier	4	3.5 kHz	500 mW	Yes
TLC10/20	General-purpose dual filter	2	CLK ÷ 50 CLK ÷ 100	N/A	No
TLC04/14	Low pass, Butterworth filter	4	CLK ÷ 50 CLK ÷ 100	N/A	No

Table F–8 lists some of TI's speech synthesis development tools.

Table F–8. Speech Synthesis Development Tools

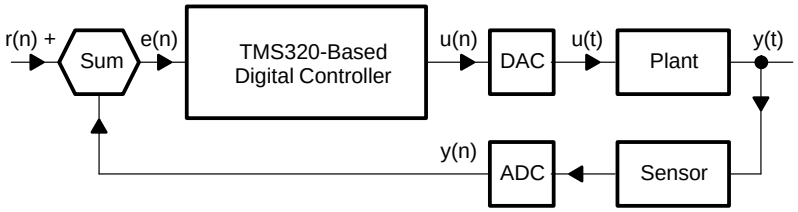
Name	Definition
(a) Software	
EVM	Code development tool
(b) Speech	
SAB	Speech audition board
SD85000	PC-based speech analysis system
(c) System	
SEB	System emulator board
SEB60Cxx	System emulator boards for speech memories

For further information, call Linear Applications at (214) 997–3772.

F.4 Servo Control/Disk Drive Applications

In the past, most servo control systems used only analog circuitry. However, the growth of digital signal processing (DSP) has made digital control theory a reality. Figure F–8 is a block diagram of a generic digital control system using a DSP, along with an analog-to-digital converter (ADC) and a digital-to-analog converter (DAC).

Figure F–8. Generic Servo Control Loop



In a DSP-based control system, the control algorithm is implemented via software. No component aging or temperature drift is associated with digital control systems. Additionally, sophisticated algorithms can be implemented and easily modified to upgrade system performance.

System Design Considerations

TMS320 DSPs have facilitated the development of high-speed digital servo control for disk drive and industrial control applications. In recent years, disk drives have increased storage capacity from 5 megabytes to over 1 gigabyte. This equates to a 23,900 percent growth in capacity. To accommodate these increasingly higher densities, the data on the servo platters, whether servo-positioning or actual storage information, must be converted to digital electronic signals at increasingly closer points in relation to the platter pick-off point. The ADC must have increasingly higher conversion rates and greater resolution to accommodate the increasing bandwidth requirements of higher storage densities. In addition, the ADC conversion rates must increase to accommodate the shorter data retrieval access time.

Figure F–9 is a block diagram of a disk drive control system.

Figure F–9. Disk Drive Control System Block Diagram

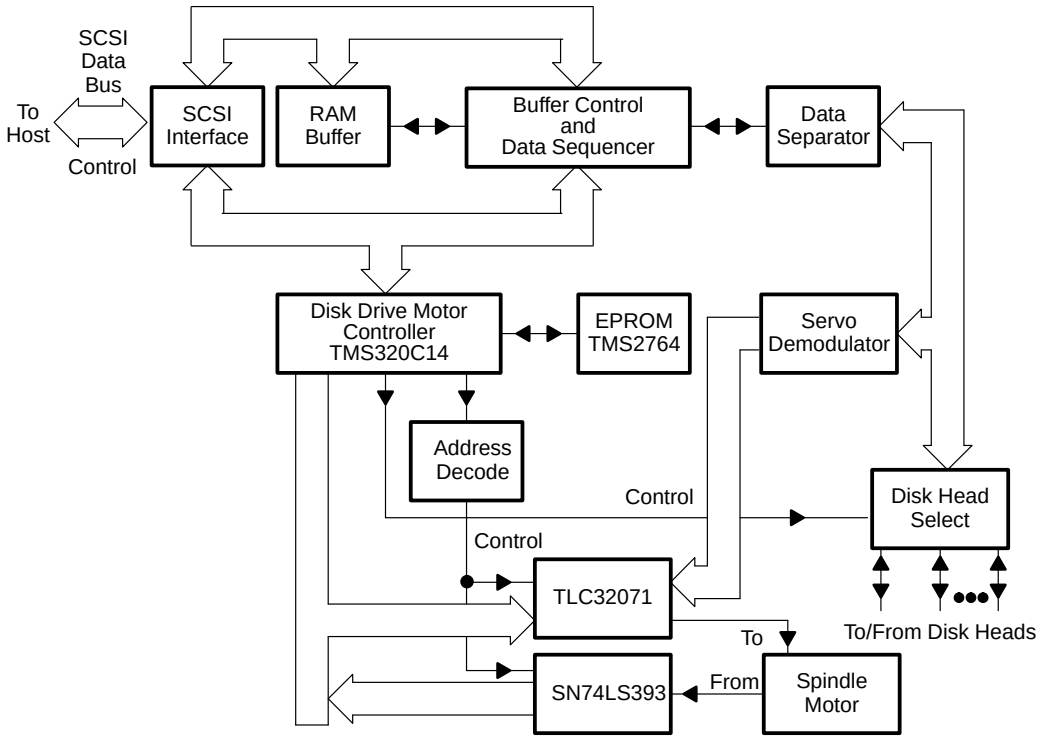


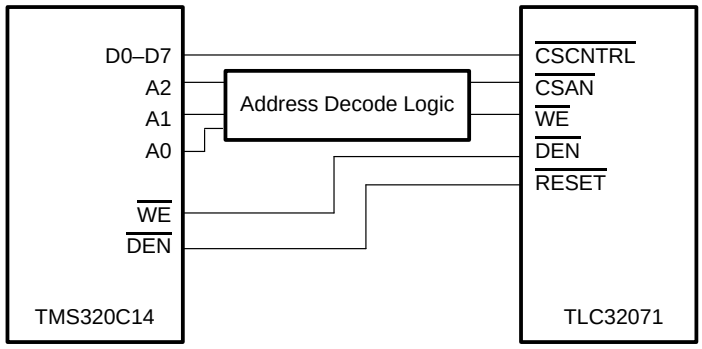
Table F–9 lists analog/digital interface devices used for servo control.

Table F–9. Control-Related Devices

Function	Device	Bits	Speed	Channels	Interface
ADC	TLC1550	10	3–5 µs	1	Parallel
	TLC1551	10	3–5 µs	1	Parallel
	TLC5502/3	8	50 ns (flash)	1	Parallel
	TLC0820	8	1.5 µs	1	Parallel
	TLC1225	13	12 µs	1 (Diff.)	Parallel
	TLC1558	10	3–5 µs	8	Parallel
	TLC1543	10	21 µs	11	Serial
	TLC1549	10	21 µs	1	Serial
DAC	TLC7524	8	9 MHz	1	Parallel
	TLC7628	8	9 MHz	(Dual)	Parallel
	TLC5602	8	30 MHz	1	Parallel
AIC	TLC32071	8 (ADC)	1 µs 9 MHz	8 1	Parallel

Figure F–10 shows the interfacing of the TMS320C14 and the TLC32071.

Figure F–10. TMS320C14–TLC32071 Interface



For further information on these servo control products, please call TI Linear Applications at (214) 997–3772.

F.5 Modem Applications

High-speed modems (9,600 bps and above) require a great deal of analog signal processing in addition to digital signal processing. Designing both high-speed capabilities and slower fall-back modes poses significant engineering challenges. TI offers a number of analog front-end (AFE) circuits to support various high-speed modem standards.

The TLC32040, TLC32044, TLC32046, TLC32047, and TLC320AC01 AICs are especially suited for modem applications by the integration of an input multiplexer, switched capacitor filters, high resolution 14-bit ADC and DAC, a four-mode serial port, and control and timing logic. These converters feature adjustable parameters, such as filtering characteristics, sampling rates, gain selection, (sin x)/x correction (TLC32044, TLC32046, and TLC32047 only), and phase adjustment. All of these parameters are software-programmable, making the AIC suitable for a variety of applications. Table F–10 has the description and characteristics of these devices.

Table F–10.Modem AFE Data Converters

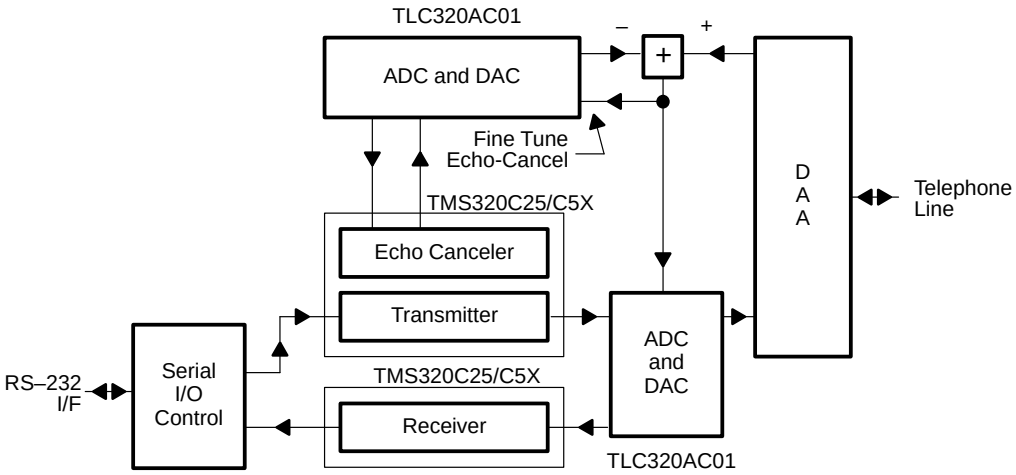
Device	Description	I/O	Resolution (Bits)	Conversion Rate
TLC32040	Analog interface chip (AIC)	Serial	14	19.2 kHz
TLC32041	AIC without on-board V_{REF}	Serial	14	19.2 kHz
TLC32044	Telephone speed/modem AIC	Serial	14	19.2 kHz
TLC32045	Low-cost version of the TLC32044	Serial	14	19.2 kHz
TLC32046	Wide-band AIC	Serial	14	25 kHz
TLC32047	AIC with 11.4-kHz BW	Serial	14	25 kHz
TLC320AC01	5-volt-only AIC	Serial	14	43.2 kHz
TCM29C18	Companding codec/filter	PCM	8	8 kHz
TCM29C23	Companding codec/filter	PCM	8	16 kHz
TCM29C26	Low-power codec/filter	PCM	8	16 kHz
TCM320AC36	Single-supply codec/filter	PCM and Linear	8	25 kHz

The AIC interfaces directly with serial-input TMS320 DSPs, which execute the modem's high-speed encoding and decoding algorithms. The TLC320C4x family performs level-shifting, filtering, and A/D and D/A data conversion. The DSP's software-programmable features provide the flexibility required for modem operations and make it possible to modify and upgrade systems easily. Under DSP control, the AIC's sampling rates permit designers to include fallback modes without additional analog hardware in most cases. Phase adjustments can be made in real time so that the A/D and D/A conversions can be synchronized with the upcoming signal. In addition, the chip has a built-in loopback feature to support modem self-test requirements.

For further information or application assistance, please call TI Linear Applications at (214) 997–3772.

Figure F–11 shows a V.32 bis modem implementation using the TMS320C25 and a TLC320AC01. The upper TMS320C25 performs echo cancellation and transmit data functions, while the lower TMS320C25 performs receive data and timing recovery functions. The echo canceler simulates the telephone channel and generates an estimated echo of the transmit data signal.

Figure F–11. High-Speed V.32 Bis and Multistandard Modem With the TLC320AC01 AIC



The TLC320AC01 performs the following functions:

- Upper TLC320AC01 D/A Path

  Converts the estimated echo, as computed by the upper TMS320C25, into an analog signal, which is subtracted from the receive signal

- Upper TLC320AC01 A/D Path

  Converts the residual echo to a digital signal for purposes of monitoring the residual echo and continuously training the echo canceler for optimum performance. The converted signal is sent to the upper TMS320C25.

- Lower TLC320AC01 D/A Path

  Converts the upper TMS320C25 transmit output to an analog signal, performs a smoothing filter function, and drives the DAC

- Lower TLC320AC01 A/D Path

  Converts the echo-free receive signal to a digital signal, which is sent to the lower TMS320C25 to be decoded

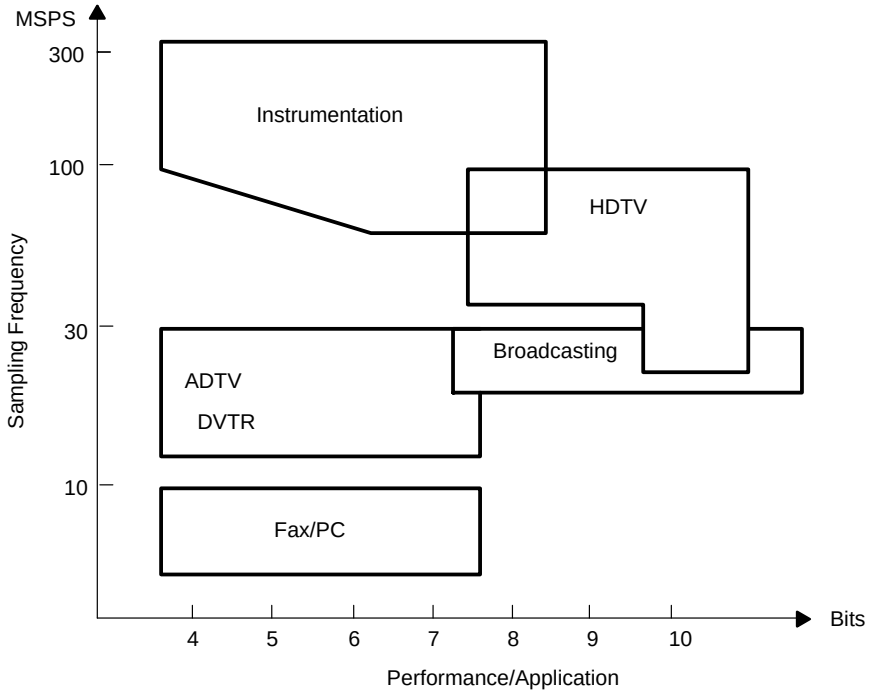
Note: Modem Functions

Figure F–11 is for illustration only. In reality, one single TMS320C5x DSP can implement high-speed modem functions.

F.6 Advanced Digital Electronics Applications for Consumers

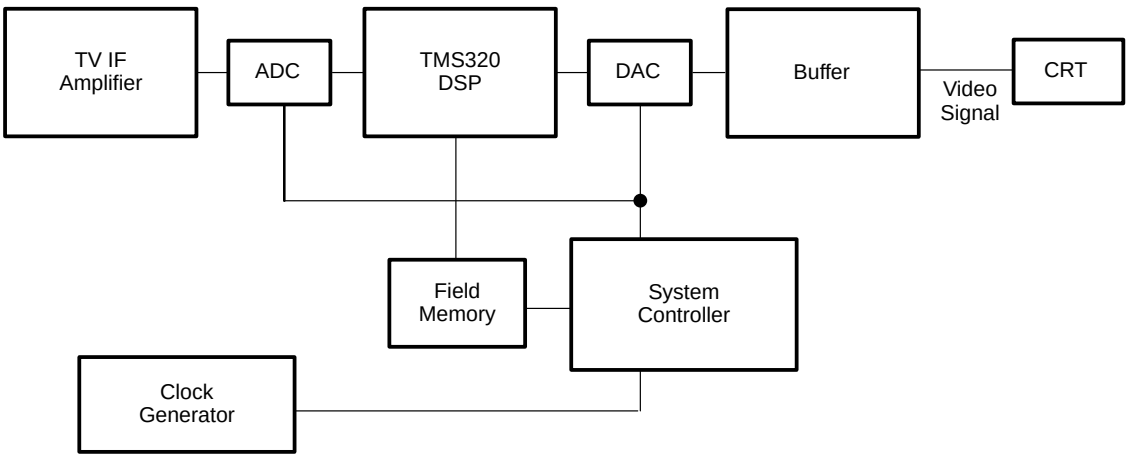
With the extensive use of the TMS320 DSPs in consumer electronics, much electromechanical control and signal processing can be done in the digital domain. Digital systems generally require some form of analog interface, usually in the form of high-performance ADCs and DACs. Figure F–12 shows the general performance requirements for a variety of applications.

Figure F–12. Applications Performance Requirements



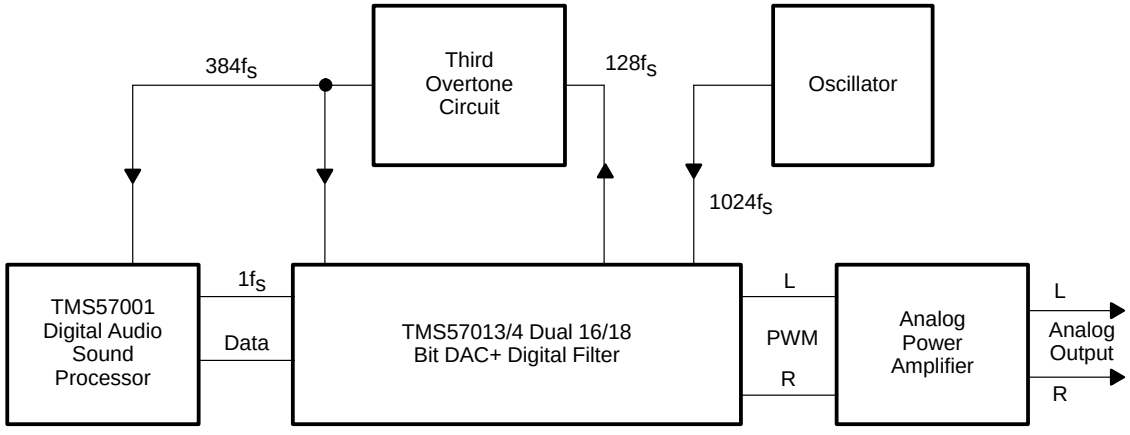
Advanced Television System Design Considerations. Advanced Digital Television (ADTV) is a technology that uses DSP to enhance video and audio presentations and to reduce noise and ghosting. Because of these DSP techniques, a variety of features can be implemented, including frame store, picture-in-picture, improved sound quality, and zoom. The bandwidth requirements remain at the existing six-MHz television allocation. From the intermediate frequency (IF) output, the video signal is converted by an eight-bit video ADC. The digital output can be processed in the digital domain to provide noise reduction, interpolation or averaging for digitally increased sharpness, and higher quality audio. The DSP digital output is converted back to analog by a video DAC, as shown in Figure F–13.

Figure F–13. Video Signal Processing Basic System



Video casette recorders (VCRs), compact disc (CD) and digital audio tape (DAT) players, and personal computers (PCs) are a few of the products that have taken a major position in the marketplace in recent years. The audio channels for compact disc and DAT require 16-bit A/D resolution to meet the distortion and noise standards. See Figure F–14 for a block diagram of a typical digital audio system.

Figure F–14. Typical Digital Audio Implementation



The motion and motor control systems usually use 8- to 10-bit ADCs for the lower frequency servo loop. Tape or disk systems use motor or motion control for proper positioning of the record or playback heads. With the storage medium compressing data into an increasingly smaller physical size, the positioning systems require more precision.

The audio processing becomes more demanding as higher fidelity is required. Better fidelity translates into lower noise and distortion in the output signal.

The TMS57013DW/57014DW one-bit DACs include an eight-times-over sampling digital filter designed for digital audio systems, such as compact disk players (CDPs), DATs, compact disks interactive (CDIs), laser disk players (LDPs), digital amplifiers, and car stereos. They are also suitable for all systems that include digital sound processing like TVs, VCRs, musical instruments, multimedia, etc.

The converters have dual channels so that the right and left stereo signals can be transformed into analog signals with only one chip. There are some functions that allow the customers to select the conditions according to their applications, such as muting, attenuation, de-emphasis, and zero data detection. These functions are controlled by external 16-bit serial data from a controller like a microcomputer.

The TMS5703DW/57014DW adopt 129-tap finite impulse response (FIR) filter and third-order $\Delta\Sigma$ modulation to get –75-dB stop band attenuation and 96-dB signal noise ratio (SNR). The output is pulse width modulation (PWM) wave, which facilitates analog signals through a low-pass filter.

Table F–11 lists TI products for analog interfacing to digital systems.

Table F–11. Audio/Video Analog/Digital Interface Devices

Function	Device	Bits	Speed	Channels	Interface
Dual audio DAC + digital filter	TMS57013/4	16/18	32, 37.8, 44.1, 48 kHz	2	Serial
Analog interface	TLC32071				
A/D		8	2 μs	8	Parallel
D/A		8	15 μs	1	Parallel
A/D	TLC1225	12	12 μs	1	Parallel
A/D	TLC1550	10	6 μs	1	Parallel
Video D/A	TLC5602	8	50 ns	1	Parallel
Video D/A	TL5602	8	50 ns	1	Parallel
Triple video D/A	TL5632	8	16 ns	3	Parallel
Triple flash A/D	TLC5703	8	70 ns	3	Parallel
Flash A/D	TLC5503	8	100 ns	1	Parallel
Flash A/D	TLC5502	8	50 ns	1	Parallel

For further information or application assistance, please call TI Linear Applications at (214) 997–3772.

Appendix G

Boot Loader Source Code

This appendix contains the source code for the TMS320C3x 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
```

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z