TEXAS INSTRUMENTS

TMS320C5x

User's Guide

User's Guide

TMS320C5x

1993

1993

Digital Signal Processing Products

TMS320C5x User's Guide

2547301-9721 revision D
January 1993

TEXAS INSTRUMENTS

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

The purpose of this user's guide is to provide the TI customer with information on 'C5x digital signal processors. This manual can also be used as a reference guide for developing hardware or software applications. The following list summarizes the contents of the chapters and appendices in this user's guide.

How to Use This Manual

This document contains the following chapters:

Chapter 1 **Introduction**
Summarizes the TMS320 family of products. Gives a general description, lists the key features, and presents some typical applications of the 'C5x devices.

Chapter 2 **Pinouts and Signal Descriptions**
Lists pin locations with associated signals, categorizes signals according to function, and describes signals.

Chapter 3 **Architecture**
Gives a general architectural overview with a functional block diagram. Describes the 'C5x design, hardware components, and device operation.

Chapter 4 **Assembly Language Instructions**
Lists instructions by function. Provides alphabetized individual instruction descriptions with examples. Includes 'C2x-to-'C5x instruction set mapping and instruction cycle times and opcodes.

Chapter 5 **Peripherals**
Describes peripheral control, serial ports, software-programmable wait states, and timing circuits.

Chapter 6 **Memory**
Discusses program/data memory operation and configuration (with memory maps), I/O space, external interface considerations, DMA operation, and memory management.

Chapter 7 **Software Applications**
Explains the use of 'C5x instruction set with particular emphasis on its new features. Includes code examples for various DSP applications.

Appendix A Electrical Specifications
Provides design documentation for the 'C5x devices. This data is based upon design goals and modeling information.

Appendix B External Interface Timing
Provides functional timing of operation on the external interface bus.

Appendix C Instruction Cycle Timings
Details the instruction cycle timings organized in different classes.

Appendix D TMS320C5x System Migration
Provides information for upgrading a 'C25 system to a 'C5x system. Includes package dimensions and pinouts, timing similarities and differences, source-code compatibility, memory maps, on-chip peripheral interfacing, and development tool enhancements.

Appendix E XDS510 Design Considerations
Provides information to meet the design requirements of the XDS510 emulator and to support XDS510 Cable #2563988–001 Rev. B.

Appendix F Analog Interface Peripherals and Applications
Describes a variety of devices that interface directly to the TMS320 DSPs for various communication and multimedia applications.

Appendix G Memories, Sockets, and Crystals
Provides product information regarding memories and sockets manufactured by Texas Instruments that are compatible with the 'C5x. Information is also given regarding crystal frequencies, specifications, and vendors.

Appendix H ROM Codes
Outlines the procedural flow for submitting code and ordering TMS320 mask-programmed ROM-based DSPs from Texas Instruments.

Appendix I Development Support
Provides a description of the 'C5x development support tools.

Related Documentation

The following books describe the TMS320 fixed-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.

TMS320C2x/C5x Optimizing C Compiler User's Guide (literature number SPRU024) describes the 'C2x/C5x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C2x and 'C5x generations of devices.

TMS320C5x C Source Debugger User's Guide (literature number SPRU055) tells you how to invoke the 'C5x emulator, SWDS, EVM, and simulator versions of the C source debugger interface. A tutorial introduces basic debugger functionality and discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints.

TMS320 Fixed-Point DSP Assembly Language Tools User's Guide (literature number SPRU018) 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 'C1x, 'C2x, and 'C5x generations of devices.

TMS320C5x Evaluation Module Technical Reference (literature number SPRU087) describes the 'C5x EVM, its features, design details and external interfaces.

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

- Digital control systems
- Digital signal processing
- Image processing
- Speech processing

Within those areas, the references appear in alphabetical order according to author. The documents contain beneficial information regarding designs, operations, and applications for general and/or specific signal-processing systems as well as circuits; all of the documents provide additional references. Therefore, Texas Instruments strongly suggests that you refer to these publications.

Digital Control Systems:

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

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

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

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

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

Digital Signal Processing:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Image Processing:

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

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

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

Speech Processing:

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

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

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

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

Style and Symbol Conventions

This document uses the following conventions.

- Program listings and program examples are shown in a `special typeface` similar to a typewriter's.

  Here is a segment of a program listing:

```
OUTPUT:
      LDP     #0          ;data page 0
      RPT     #63         ;Output 64 values from a table at 800h
       LMMR   50h,800h    ;in data memory to port 50h.
      RET
```

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

  [*label*] **BLDD** *src, dst*

  BLDD is the instruction, which has two parameters indicated by *src* and *dst*. When you use **BLDD**, the first parameter must be an actual data memory source address and *dst* a destination address. A comma and a space must separate the two addresses.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not type the brackets themselves. In the example above, instead of typing [*label*], you specify a name for the label. When you specify more than one optional parameter from a list, you separate them with a comma and a space.

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

  `ind: { * | *+ | *– | *0+ | *0– | *BRO+ | *BRO–}`

  that provides seven choices.

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

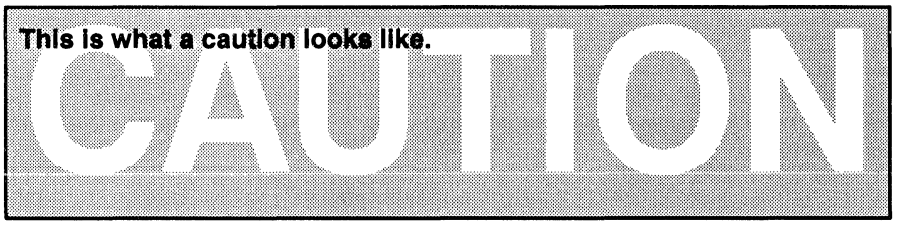
Information About Notes and Cautions

This book may contain notes and cautions.

- A **note** describes a preferred way or recommended procedure.

> **Note:**
>
> **This is what a note looks like.**

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



The information in a note or caution is provided for your protection. Please read it carefully.

Trademarks

MS-DOS and *MS-Windows* are trademarks of Microsoft Corp.
DEC, VAX, and VMS are trademarks of Digital Equipment Corp.
HP is a trademark of Hewlett Packard Co.
Macintosh and *MPW* are trademarks of Apple Computer Corp.
PC-DOS is a trademark of IBM Corp.
Sun 3 and *Sun 4* are trademarks of Sun Microsystems, Inc.
UNIX is a trademark of UNIX System Laboratories, Inc.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments Digital Signal Processing (DSP) products or oder TI documentation	Call the LRC (Literature Response Center): **(800) 477–8924**, 8:00–17:00 CST Or write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320**
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

Contents

Figures

Tables

Examples

Chapter 1

Introduction

This user's guide discusses the TMS320C5x digital signal processors (DSPs), the newest fixed-point generation in the TMS320 family. The 'C50, the 'C51, and the 'C53 are the first devices in this generation. Their central processing unit (CPU) core is based upon the 'C25's CPU core with additional architectural enhancements to greatly improve overall performance. The 'C5x generation devices are capable of executing at twice the speed of the 'C2x and are source-code upward compatible with all 'C1x and 'C2x devices. Expansion of this fixed-point generation is expected in the near future to provide even higher levels of DSP performance.

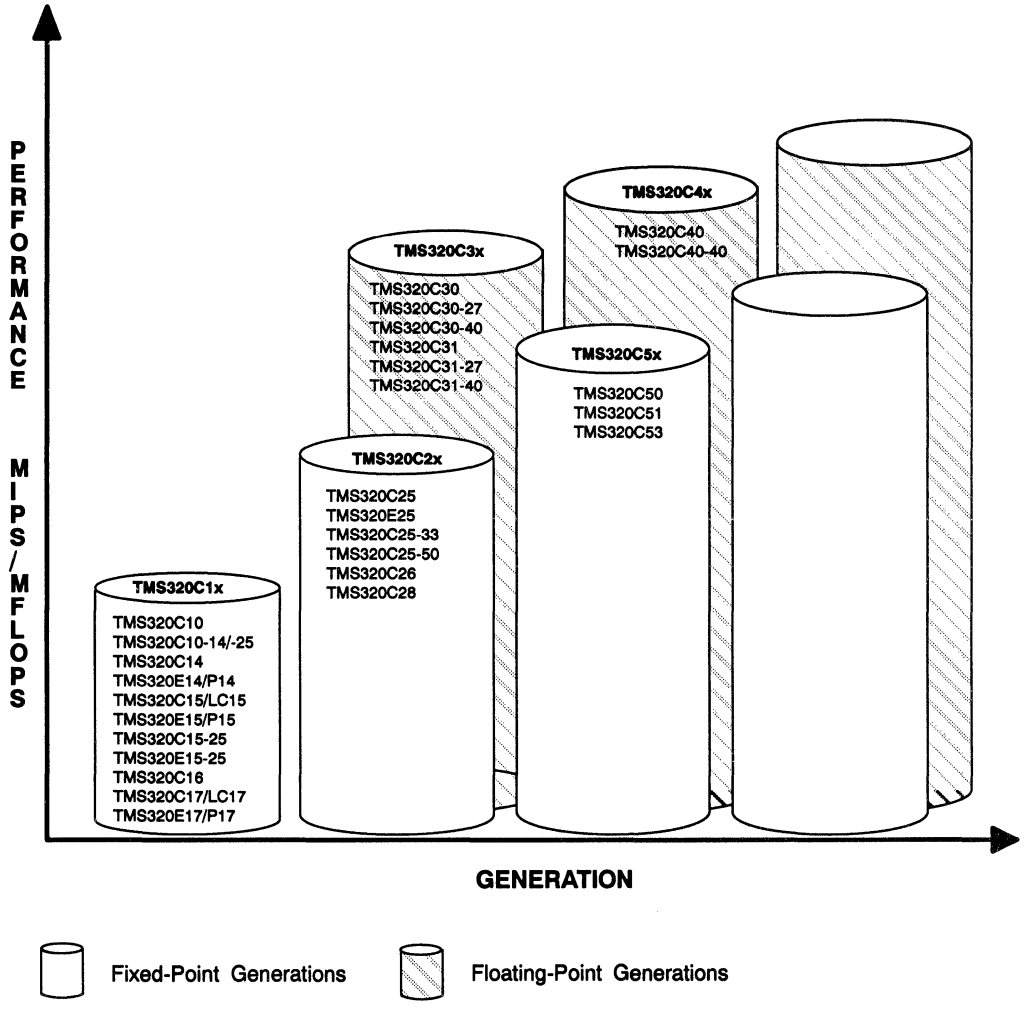
The 'C5x generation consists of the following devices:

- 'C50 is a static CMOS digital signal processor with 10K words of on-chip RAM and 2K words of on-chip ROM.
- 'C51 is a static CMOS digital signal processor with 2K words of on-chip RAM and 8K words of on-chip ROM.
- 'C53 is a static CMOS digital signal processor with 4K words of on-chip RAM and 16K words of on-chip ROM.

This chapter discusses these topics:

Topic **Page**

Figure 1–1. Evolution of the TMS320 Family



1.1 TMS320 Family Overview

The TMS320 family consists of 16-bit fixed-point and 32-bit floating-point single-chip digital signal processing devices. These processors possess the operational flexibility of high-speed controllers and the numerical capability of array processors. Combining those two qualities, the TMS320 processors are inexpensive alternatives to custom-fabricated VLSI and multichip bit-slice processors. The following qualities make this family the ideal choice for a wide range of processing applications (refer to Table 1–1 for a list of applications):

- ❑ Very flexible instruction set
- ❑ Inherent operational flexibility
- ❑ High-speed performance
- ❑ Innovative, parallel architectural design
- ❑ Cost effectiveness

In 1982, Texas Instruments introduced the first fixed-point digital signal processor in the TMS320 family, the TMS32010. Before the year had ended, the *Electronic Products* magazine awarded the TMS32010 the title **Product of the Year**. The TMS32010 became the model for future TMS320 generations.

Today, the TMS320 family consists of five generations: 'C1x, 'C2x, 'C3x, 'C4x, and 'C5x. Figure 1–1 illustrates the performance gains that the TMS320 family has made over time with successive generations. Note that the 'C1x, 'C2x, and 'C5x generations are fixed-point, and the 'C3x and 'C4x generations are floating-point. Source code is upward compatible from one fixed-point generation to the next fixed-point generation and, likewise, from one floating-point generation to the next floating-point generation. Compatibility preserves the software portion of your investment, thereby providing a convenient and cost-efficient roadmap to a higher performance, more versatile DSP system.

Each generation of TMS320 devices has an internal core CPU and a variety of memory and peripheral configurations. New combinations of on-chip memory and peripheral options can create spin-off devices. These spin-offs can satisfy a wide range of needs in the worldwide electronics market. When memory and peripherals are integrated into one processor, overall system cost is greatly reduced and board space is saved.

1.1.1 Typical Applications

With its unique versatility and real-time performance, a 'C5x-generation processor offers better, more adaptable approaches to traditional signal-processing problems such as vocoding and filtering. Furthermore, the 'C5x supports complex applications that often require several operations to be performed simultaneously. Table 1–1 lists those applications for which a 'C5x device is well suited.

Table 1–1. Typical Applications for the TMS320 Family

Automotive	Consumer	Control
Adaptive Ride Control Antiskid Brake Cellular Telephone Digital Radio Engine Control Global Positioning Navigation Vibration Analysis Voice Commands	Digital Radio/TV Educational Toys Music Synthesizer Power Tools Radar Detector Solid-State Answering Machines	Disk Drive Control Engine Control Laser Printer Control Motor Control Robotics Control Servo Control
General-Purpose	**Graphics/Imaging**	**Industrial**
Adaptive Filtering Convolution Correlation Digital Filtering Fast Fourier Transforms Hilbert Transforms Waveform Generation Windowing	3-D Rotation Animation/Digital Map Homomorphic Processing Pattern Recognition Image Enhancement Image Compression/ Transmission Robot Vision Workstations	Numeric Control Power-Line Monitoring Robotics Security Access
Instrumentation	**Medical**	**Military**
Digital Filtering Function Generation Pattern Matching Phase-Locked Loops Seismic Processing Spectrum Analysis Transient Analysis	Diagnostic Equipment Fetal Monitoring Hearing Aids Patient Monitoring Prosthetics Ultrasound Equipment	Image Processing Missile Guidance Navigation Radar Processing Radio Frequency Modems Secure Communications Sonar Processing
Telecommunications		**Voice/Speech**
1200- to 19200-bps Modems Adaptive Equalizer ADPCM Transcoder Cellular Telephone Channel Multiplexing Data Encryption Digital PBXs Digital Speech Interpolation (DSI)	DTMF Encoding/Decoding Echo Cancellation FAX Line Repeater Speaker Phone Spread Spectrum Communications Video Conferencing X.25 Packet Switching	Speech Enhancement Speech Recognition Speech Synthesis Speaker Verification Speech Vocoding Voice Mail Text-to-Speech

1.2 General Description

The 'C5x generation consists of the 'C50, the 'C51, and the 'C53 devices. These digital signal processors (DSPs) are fabricated in accordance with static CMOS integrated-circuit technology. Their architectural design is based upon that of the 'C25. The combination of an advanced Harvard architecture (separate buses for program memory and data memory), additional on-chip peripherals, more on-chip memory, and a highly specialized instruction set is the basis of the operational flexibility and speed of these DSP devices. The 'C5x devices are designed to execute more than 28 MIPS (million instructions per second). Future spin-off devices with the core CPU and customized on-chip memory and peripheral configurations may be developed for specialized areas of the electronics market.

The 'C5x generation offers these advantages:

- Enhanced TMS320 architectural design for increased performance and versatility
- Modular architectural design for fast development of spin-off devices
- Advanced IC processing technology for increased performance
- Downward source-code compatibility with 'C1x and 'C2x DSPs for fast and easy performance upgrades
- Enhanced TMS320 instruction set for faster algorithms and for optimized high-level language operation
- New static design techniques for minimizing power consumption and maximizing radiation hardness

Table 1–2 provides an overview of the 'C5x generation of digital signal processors. It shows the capacity of on-chip RAM and ROM memories, number of serial and parallel I/O ports, execution time of one machine cycle, and type of package with total pin count. The chart should help you choose the best processor for an application.

The following subsections summarize key features of the 'C5x processors. The CPU description applies to all 'C5x-generation members (current and future). Descriptions of the remaining features apply only to the 'C50, 'C51 and the 'C53. Detailed information on the CPU, on-chip peripherals, and memory, is given in Chapters 3, 5, and 6, respectively.

Table 1–2. Characteristics of the 'C5x DSP Processors

TMS320 Device	On-Chip Memory			I/O Ports		Cycle Time (ns)	Package Type QFP§
	RAM		ROM				
	Data	Data+Prog	Prog	Serial	Parallel†		
TMS320C50	1K	9K	2K	2	64K	50/35	132-pin ceramic
TMS320C51	1K	1K	8K	2	64K	50/35	132-pin plastic
TMS320C53	1K	3K	16K	2	64K	50/35	132-pin plastic

† Note that 16 of the 64K parallel I/O ports are memory-mapped.
§ QFP = Quad Flat Pack.

1.3 Key Features

Key features of the 'C5x DSPs are listed below. Where a feature is exclusive to a particular device, the device's name is enclosed within parentheses and noted after that feature.

- ❑ 35-/50-ns single-cycle fixed-point instruction execution time (28.6/20 MIPS)
- ❑ Upward source-code compatible with all 'C1x and 'C2x devices
- ❑ RAM-based memory operation ('C50)
- ❑ ROM-based memory operation ('C51)
- ❑ 9K x 16-bit single-cycle on-chip program/data RAM ('C50)
- ❑ 1K x 16-bit single-cycle on-chip program/data RAM ('C51)
- ❑ 3K x 16-bit single-cycle on-chip program/data RAM ('C53)
- ❑ 2K x 16-bit single-cycle on-chip boot ROM ('C50)
- ❑ 8K x 16-bit single-cycle on-chip program ROM ('C51)
- ❑ 16K x 16-bit single-cycle on-chip program ROM ('C53)
- ❑ 1056 x 16-bit dual-access on-chip data RAM
- ❑ 224K x 16-bit maximum addressable external memory space (64K program, 64K data, 64K I/O, and 32K global)
- ❑ 32-bit arithmetic logic unit (ALU), 32-bit accumulator (ACC), and 32-bit accumulator buffer (ACCB)
- ❑ 16-bit parallel logic unit (PLU)
- ❑ 16 x 16-bit parallel multiplier with a 32-bit product capability
- ❑ Single-cycle multiply/accumulate instructions
- ❑ Eight auxiliary registers with a dedicated arithmetic unit for indirect addressing
- ❑ Eleven context-switch registers (shadow registers) for storing strategic CPU-controlled registers during an interrupt service routine
- ❑ Eight-level hardware stack
- ❑ 0- to 16-bit left and right data barrel-shifters and a 64-bit incremental data shifter
- ❑ Two indirectly addressed circular buffers for circular addressing
- ❑ Single-instruction repeat and block repeat operations for program code
- ❑ Block memory move instructions for better program/data management
- ❑ Full-duplex synchronous serial port for direct communication between the 'C5x and another serial device
- ❑ Time-division multiple-access (TDM) serial port
- ❑ Interval timer with period, control, and counter registers for software stop, start, and reset
- ❑ 64K parallel I/O ports, 16 of which are memory mapped
- ❑ Sixteen software-programmable wait-state generators for program, data, and I/O memory spaces

- ❑ Extended hold operation for concurrent external DMA
- ❑ Four-deep pipelined operation for delayed branch, call, and return instructions
- ❑ Index-addressing mode
- ❑ Bit-reversed index-addressing mode for radix-2 FFTs
- ❑ Divide-by-one clock option
- ❑ On-chip clock generator
- ❑ JTAG boundary scan logic (IEEE standard, 1149.1)
- ❑ On-chip scan-based emulation logic
- ❑ 5-V static CMOS technology with two power-down modes
- ❑ 132-pin quad flat pack package

1.3.1 Core CPU

Enhancements to the 'C5x CPU maintain source code compatibility with the 'C1x and 'C2x generations while improving performance and versatility. Improvements include a 32-bit accumulator buffer, additional scaling capabilities, and a host of new instructions to exploit the additional hardware while supplying a more orthogonal instruction set to the user. The new control functions include an independent parallel logic unit (PLU) for performing Boolean operations and a set of context-switch registers for providing zero-latency context-switching capabilities to interrupt service routines (ISRs). Data management has been improved through the use of new block move instructions and memory-mapped register instructions. The 'C5x has 28 memory-mapped core-CPU registers and 16 memory-mapped I/O ports. See Chapter 3 for more details.

1.3.2 On-Chip ROM

The 'C50 features a 2K × 16-bit on-chip, maskable, programmable ROM. This memory is used for booting from slower external ROM or EPROM of program to fast on-chip or external SRAM. ROM can be selected during reset by driving the MP/$\overline{\text{MC}}$ pin low. Once your program has been booted into the RAM, this boot ROM can be operationally removed from the program memory space via the MP/$\overline{\text{MC}}$ bit in the PMST status register. If the ROM is not selected, the 'C50 starts its execution via an off-chip memory.

The 'C51 features an 8K × 16-bit on-chip maskable ROM. The 'C53 features a 16K × 16-bit on-chip maskable ROM. You can use this memory for your specified program. Once the development of the program has stabilized, submit a ROM code to Texas Instruments for implementation into your device. See Chapter 6 for more details.

1.3.3 On-Chip Data RAM

All 'C5x devices carry a 1056 x 16-bit on-chip data RAM. This RAM can be accessed twice per machine cycle (dual-access RAM). This block of memory is primarily intended to store data values but, when needed, can be used to store programs as well as data. It can be configured in one of two ways: either all 1056 x 16 bits as data memory or 544 x 16 bits as data memory with 512 x 16 bits as program memory. You can select the configuration with the CNF bit in status register ST1. See Chapter 6 for more details.

1.3.4 On-Chip Program/Data RAM

The 'C50 has a 9K x 16-bit on-chip RAM. The 'C51 has a 1K x 16-bit on-chip RAM. This memory is software configurable as program and/or data memory space. Code can be booted from an off-chip nonvolatile memory and then executed at full speed, once it is loaded into this RAM. See Chapter 6 for more details.

1.3.5 On-Chip Memory Security

The 'C5x generation has a maskable option to protect the contents of on-chip memories. When the related bit is set, no externally originating instruction can access the on-chip memory spaces. See Chapter 6 for more details.

1.3.6 Address-Mapped Software Wait-State Generators

Software wait-state logic is incorporated without any external hardware into 'C5x for interfacing with slower off-chip memory and I/O devices. This circuitry consists of 16 wait-state generating circuits and is user programmable to operate 0, 1, 2, 3, or 7 wait states. For off-chip memory accesses, these wait-state generators can be mapped on 16K-word boundaries in program memory, data memory, and to the I/O ports. See Chapter 5 for more details.

1.3.7 Parallel I/O Ports

Each 'C5x device has a total of 64K I/O ports, sixteen of which are memory-mapped in data memory space. These ports can be addressed by the IN instruction or the OUT instruction. The memory-mapped I/O ports can be accessed with any instruction that reads or writes data memory. An active-low $\overline{IS}$ signal indicates a read/write operation via an I/O port. Requiring minimal off-chip address-decoding circuits, the 'C5x can easily interface with external I/O devices via the I/O ports. See Chapter 5 for more details.

1.3.8 Serial I/O Ports

The 'C5x devices carry two high-speed serial ports. These serial ports are capable of operating at up to one-fourth the machine cycle rate (CLKOUT1). One

of the two circuits is a synchronous, full-duplex serial port. Its transmitter and receiver are double buffered and individually controlled by maskable external interrupt signals. Data is framed either as bytes or as words. The second circuit is a full-duplex serial port that can be configured either for synchronous or for time-division multiple-access (TDM) operations. The TDM serial port is commonly used in multiprocessor applications. See Chapter 5 for more details.

1.3.9 Hardware Timer

The 'C5x features a 16-bit timing circuit with a 4-bit prescaler. This timer clocks between one-half and one-thirty-second the machine rate of the device itself, depending upon the programmable timer's divide-down ratio. This timer can be stopped, restarted, reset, or disabled by specific status bits. See Chapter 5 for more details.

1.3.10 User-Maskable Interrupts

The 'C5x devices have four external-interrupt lines. These lines are internally latched so that asynchronous interrupt operations can be performed by the TMS320 device. Also, each device possesses five internal interrupts: the timer interrupt and four serial port interrupts. See Chapter 5 for more details.

1.3.11 JTAG Scanning Logic

The JTAG scanning logic circuitry is used for emulating and testing purposes only. The JTAG scan logic provides the boundary scan to and from the interfacing devices. Also, it can be used to test pin-to-pin continuity as well as to perform operational tests on those peripheral devices that surround the 'C5x. It is interfaced to another internal scanning logic circuitry, which has access to all of the on-chip resources. Thus, the 'C5x can perform on-board emulation by means of the JTAG serial scan pins and the emulation-dedicated pins. See IEEE Standard 1149.1 for more details.

1.3.12 Packages

The 'C5x devices are packaged in a 132-pin quad flat pack package (QFP). With consideration for the pin layout of a 'C25 package, the 'C5x package is designed to minimize printed circuit board modifications when a 'C2x-based system is upgraded to a 'C5x processing system. Signal callouts for the 'C5x appear on the same side and in the same order as those for the 'C25. See Chapter 2 for details.

Chapter 2

Pinouts and Signal Descriptions

The 'C5x DSPs are available in a 132-pin quad flat pack (QFP) package and have identical pin-to-signal relationship. The QFP package conforms to JEDEC specifications for electrical/electronic components. Electrical specifications and mechanical data for the 'C5x DSPs are in Appendix A.

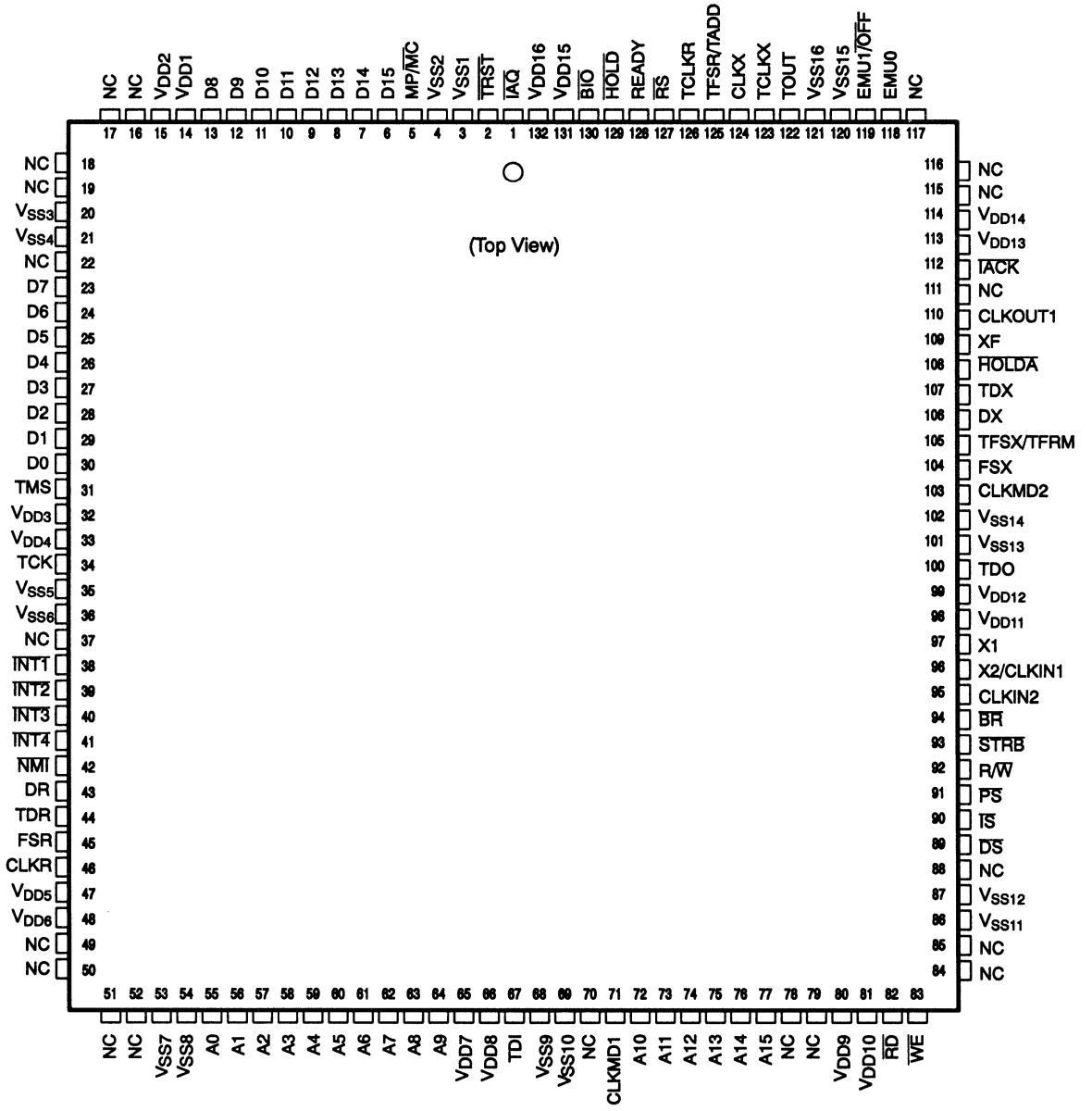
This chapter presents a simple layout of a 132-pin QFP package, with pin and signal callouts, and a table of signal definitions, in the following sections:

Topic **Page**

2.1 Pinout

The 'C5x devices are packaged in a 132-pin quad flat pack package (QFP) and have the same pin-to-signal relationship. Figure 2–1 shows the pin/signal call-outs for this package.

Figure 2–1. Signal Assignments for 'C5x 132-Pin QFP



Note: NC = No connect. (These pins are reserved.)

2.2 Signal Descriptions

The signals for the 'C5x device are described in this section. Table 2–1 lists each signal, its pin location, function, and operating mode(s), i.e., input (I), output (O), high-impedance (Z) or supply (S) state. The signals are grouped according to their functional purpose.

Table 2–1. TMS320C5x Signal Descriptions

Signal	Pin	State	Description
Address and Data Buses			
A15 (MSB) A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0 (LSB)	77 76 75 74 73 72 64 63 62 61 60 59 58 57 56 55	I/O/Z	Parallel address bus **A15** (MSB) through **A0** (LSB). Multiplexed to address external data/program memory or I/O. Placed in high-impedance state in hold mode. These signals also go into high impedance when $\overline{OFF}$ is active low. These signals are used as inputs for external DMA access of the on-chip single-access RAM. They become inputs while $\overline{HOLDA}$ is active low if the $\overline{BR}$ pin is externally driven low.
D15 (MSB) D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 (LSB)	6 7 8 9 10 11 12 13 23 24 25 26 27 28 29 30	I/O/Z	Parallel data bus **D15** (MSB) through **D0** (LSB). Multiplexed to transfer data between the core CPU and external data/program memory or I/O devices. Placed in high-impedance state when not outputting or when $\overline{RS}$ or $\overline{HOLD}$ is asserted. They also go into high impedance when $\overline{OFF}$ is active low. These signals are also used in external DMA access of the on-chip single-access RAM.

Note: All input pins that are unused should be connected to V_{DD} or an external pull-up resistor. The $\overline{BR}$ pin has an internal pull-up for performing DMA to the on-chip RAM. For emulation, TRST has an internal pull-down, and TMS, TCK, and TDI have internal pull-ups. EMU0 and EMU1 require external pull-ups to support emulation.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
			Memory Control Signals
$\overline{DS}$ $\overline{PS}$ $\overline{IS}$	89 91 90	O/Z	Data, Program, and I/O space select signals. Always high unless low level asserted for communicating to a particular external space. Placed into a high-impedance state in hold mode. These signals also go into high-impedance when $\overline{OFF}$ is active low.
READY	128	I	Data ready input. Indicates that an external device is prepared for the bus transaction to be completed. If the device is not ready (READY is low), the processor waits one cycle and checks READY again. READY also indicates a bus grant to an external device after a $\overline{BR}$ (bus request) signal.
R/$\overline{W}$	92	I/O/Z	Read/Write signal. Indicates transfer direction during communication to an external device. Normally in read mode (high), unless low level asserted for performing a write operation. Placed in high-impedance state in hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low, and it is used in external DMA access of the 9K RAM cell. While $\overline{HOLDA}$ and $\overline{IAQ}$ are active low, this signal is used to indicate the direction of the data bus for DMA reads (high) and writes (low).
$\overline{STRB}$	93	I/O/Z	Strobe signal. Always high unless asserted low to indicate an external bus cycle. Placed in high-impedance state in the hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low, and it is used in external DMA access of the on-chip single-access RAM. While $\overline{HOLDA}$ and $\overline{IAQ}$ are active low, this signal is used to select the memory access.
$\overline{RD}$	82	O/Z	Read select indicates an active, external read cycle and may connect directly to the output enable ($\overline{OE}$) of external devices. This signal is active on all external program, data, and I/O reads. Placed into high-impedance state in hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low.
$\overline{WE}$	83	O/Z	Write enable. The falling edge of this signal indicates that the device is driving the external data bus (D15–D0). Data may be latched by an external device on the rising edge of $\overline{WE}$. This signal is active on all external program, data, and I/O writes. Placed into high-impedance state in hold mode. This signal also goes into high impedance when $\overline{OFF}$ is active low.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Multiprocessing Signals			
$\overline{\text{HOLD}}$	129	I	Hold input. This signal is asserted to request control of the address, data, and control lines. When acknowledged by the 'C5x, these lines go to the high-impedance state.
$\overline{\text{HOLDA}}$	108	O/Z	Hold acknowledge signal. Indicates to the external circuitry that the processor is in a hold state and that the address, data, and memory control lines are in a high-impedance state so that they are available to the external circuitry for access of local memory. This signal also goes into high impedance when $\overline{\text{OFF}}$ is active low.
$\overline{\text{BR}}$	94	I/O/Z	Bus request signal. Asserted during access of external global data memory space. READY is asserted to the device when the global data memory is available for the bus transaction. $\overline{\text{BR}}$ can be used to extend the data memory address space by up to 32K words. It goes into high impedance when $\overline{\text{OFF}}$ is active low. $\overline{\text{BR}}$ is used in external DMA access of the on-chip single-access RAM. While $\overline{\text{HOLDA}}$ is active low, $\overline{\text{BR}}$ is externally driven low to request access to the on-chip single-access RAM.
$\overline{\text{IAQ}}$	1	O/Z	Instruction acquisition signal. This signal is asserted (active low) when there is an instruction address on the address bus and goes into high impedance when $\overline{\text{OFF}}$ is active low. $\overline{\text{IAQ}}$ is also used in external DMA access of the on-chip single-access RAM. While $\overline{\text{HOLDA}}$ is active low, $\overline{\text{IAQ}}$ acknowledges the $\overline{\text{BR}}$ request for access of the on-chip single-access RAM and stops indicating instruction acquisition.
$\overline{\text{BIO}}$	130	I	Branch control input. Samples as the BIO condition. If low, the device executes the conditional instruction. This signal must be active during the fetch of the conditional instruction.
XF	109	O/Z	External flag output (latched software-programmable signal). This signal is set high or low by specific instruction or by loading status register 1 (ST1). Used for signaling other processors in multiprocessor configurations or as a general-purpose output pin. This signal also goes into high impedance when $\overline{\text{OFF}}$ is active low. This pin is set high at reset.
$\overline{\text{IACK}}$	112	O/Z	Interrupt acknowledge signal. Indicates receipt of an interrupt and that the program counter is fetching the interrupt vector location designated by **A15–A0**. This signal also goes into high impedance when $\overline{\text{OFF}}$ is active low.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Initialization, Interrupt, and Reset Operations			
$\overline{INT4}$ $\overline{INT3}$ $\overline{INT2}$ $\overline{INT1}$	41 40 39 38	I	External user interrupt inputs. Prioritized and maskable by the interrupt mask register and interrupt mode bit. Can be polled and reset via the interrupt flag register.
$\overline{NMI}$	42	I	Nonmaskable interrupt. External interrupt that cannot be masked via the INTM or the IMR. When $\overline{NMI}$ is activated, the processor traps to the appropriate vector location.
$\overline{RS}$	127	I	Reset input. Causes the device to terminate execution and forces the program counter to zero. When $\overline{RS}$ is brought to a high level, execution begins at location zero of program memory. $\overline{RS}$ affects various registers and status bits.
MP/$\overline{MC}$	5	I	Microprocessor/Microcomputer mode select pin. If active low at reset (microcomputer mode), the pin causes the internal program ROM to be mapped into program memory space. In the microprocessor mode, all program memory is mapped externally. This pin is sampled only during reset, and the mode that is set at reset can be overridden via the software control bit MP/$\overline{MC}$ in the PMST register.
Oscillator/Timer Signals CLKIN1/2			
CLKOUT1	110	O/Z	Master clock output signal (or CLKIN2 frequency). This signal cycles at the machine-cycle rate of the CPU. The internal machine cycle is bounded by the rising edges of this signal. This signal also goes into high impedance when $\overline{OFF}$ is active low.
CLKMD1 CLKMD2	71 103	I	CLKMD1 / CLKMD2 / Clock Mode 0 / 0 / External clock with divide-by-two option. Input clock provided to X2/CLKIN1 pin. Internal oscillator and PLL disabled. 0 / 1 / Reserved for test purposes. 1 / 0 / External divide-by-one option. Input clock provided to CLKIN2. Internal oscillator disabled. Internal PLL enabled. 1 / 1 / Internal or external divide-by-two option. Input clock provided to X2/CLKIN1 pin. Internal oscillator enabled. Internal PLL disabled.
X2/CLKIN1	96	I	Input pin to internal oscillator from the crystal. If the internal oscillator is not being used, a clock may be input to the device on this pin. The internal machine cycle is half this clock rate.
X1	97	O	Output pin from the internal oscillator for the crystal. If the internal oscillator is not used, this pin should be left unconnected. This signal does not go into high impedance when $\overline{OFF}$ is active low.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Oscillator/Timer Signals (Concluded)			
CLKIN2	95	I	Divide-by-1 input clock for driving the internal machine rate.
TOUT	122	O	Timer output. This pin signals a pulse when the on-chip timer counts down past zero. The pulse is a CLKOUT1 cycle wide.
Supply Pins			
V_{DD1}	14	S	Power supply for data bus.
V_{DD2}	15	S	Power supply for data bus.
V_{DD3}	32	S	Power supply for data bus.
V_{DD4}	33	S	Power supply for data bus.
V_{DD5}	47	S	Power supply for address bus.
V_{DD6}	48	S	Power supply for address bus.
V_{DD7}	65	S	Power supply for inputs and internal logic.
V_{DD8}	66	S	Power supply for inputs and internal logic.
V_{DD9}	80	S	Power supply for address bus.
V_{DD10}	81	S	Power supply for address bus.
V_{DD11}	98	S	Power supply for memory control signals.
V_{DD12}	99	S	Power supply for memory control signals.
V_{DD13}	113	S	Power supply for inputs and internal logic.
V_{DD14}	114	S	Power supply for inputs and internal logic.
V_{DD15}	131	S	Power supply for memory control signals.
V_{DD16}	132	S	Power supply for memory control signals.
V_{SS1}	3	S	Ground for memory control signals.
V_{SS2}	4	S	Ground for memory control signals.
V_{SS3}	20	S	Ground for data bus.
V_{SS4}	21	S	Ground for data bus.
V_{SS5}	35	S	Ground for data bus.
V_{SS6}	36	S	Ground for data bus.
V_{SS7}	53	S	Ground for address bus.
V_{SS8}	54	S	Ground for address bus.
V_{SS9}	68	S	Ground for address bus.
V_{SS10}	69	S	Ground for address bus.
V_{SS11}	86	S	Ground for memory control signals.
V_{SS12}	87	S	Ground for memory control signals.
V_{SS13}	101	S	Ground for inputs and internal logic.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
Supply Pins (Concluded)			
V_{SS14}	102	S	Ground for inputs and internal logic.
V_{SS15}	120	S	Ground for inputs and internal logic.
V_{SS16}	121	S	Ground for inputs and internal logic.
Serial Port Signals			
CLKR TCLKR	46 126	I I	Receive clock inputs. External clock signal for clocking data from the DR/TDR (data receive) pins into the RSR (serial port receive shift register). Must be present during serial port transfers. If the serial port is not being used, these pins can be sampled as an input via the IN0 bit of the SPC/TSPC registers.
CLKX TCLKX	124 123	I/O/Z I/O/Z	Transmit clock. Clock signal for clocking data from the DR/TDR (data receive register) to the DX/TDX (data transmit pin). The CLKX can be an input if the MCM bit in the serial port control register is set to 0. It may also be driven by the device at 1/4 the CLKOUT1 frequency when the MCM bit is set to 1. If the serial port is not being used, this pin can be sampled as an input via the IN1 bit of the SPC/TSPC register. This signal goes into high impedance when $\overline{OFF}$ is active low.
DR TDR	43 44	I I	Serial data receive inputs. Serial data is received in the RSR (serial port receive shift register) via the DR/TDR pin.
DX TDX	106 107	O/Z	Serial port transmit outputs. Serial data transmitted from the XSR (serial port transmit shift register) via the DX/TDX pin. Placed in high-impedance state when not transmitting and also when $\overline{OFF}$ is active low.
FSR TFSR/TADD	45 125	I I/O/Z	Frame synchronization pulse for receive input. The falling edge of the FSR/TFSR pulse initiates the data receive process, beginning the clocking of the RSR. TFSR becomes an input/output (TADD) pin when the serial port is operating in TDM mode (TDM bit = 1). In TDM mode, this pin is used to output/input the address of the port. This signal goes into high impedance when $\overline{OFF}$ is active low.
FSX TFSX/TFRM	104 105	I/O/Z I/O/Z	Frame synchronization pulse for transmit input/output. The falling edge of the FSX/TFSX pulse initiates the data transmit process, beginning the clocking of the XSR. Following reset, the default operating condition of FSX/TFSX is an input. This pin may be selected by software to be an output when the TXM bit in the serial control register is set to 1. This signal goes into high impedance when $\overline{OFF}$ is active low. When operating in TDM mode (TDM bit = 1), the TFSX pin becomes TFRM, the TDM frame synch.

Table 2–1. TMS320C5x Signal Descriptions (Continued)

Signal	Pin	State	Description
			Test Signals
TCK	34	I	JTAG test clock. This is normally a free-running clock signal with a 50% duty cycle. The changes on TAP (test access port) input signals (TMS and TDI) are clocked into the TAP controller, instruction register, or selected test data register on the rising edge of TCK. Changes at the TAP output signal (TDO) occur on the falling edge of TCK.
TDI	67	I	JTAG test data input. TDI is clocked into the selected register (instruction or data) on a rising edge of TCK.
TDO	100	O/Z	JTAG test data output. The contents of the selected register (instruction or data) is shifted out of TDO on the falling edge of TCK. TDO is in high-impedance state except when scanning of data is in progress. This signal also goes into high impedance when $\overline{\text{OFF}}$ is active low.
TMS	31	I	JTAG test mode select. This serial control input is clocked into the test access port (TAP) controller on the rising edge of TCK.
$\overline{\text{TRST}}$	2	I	JTAG test reset. This signal, when active high, gives the JTAG scan system control of the operations of the device. If this signal is not connected or driven low, the device will operate in its functional mode, and the JTAG signals are ignored.
EMU0	118	I/O/Z	Emulator pin 0. When $\overline{\text{TRST}}$ is driven low , this pin must be high for activation of the $\overline{\text{OFF}}$ condition (see pin 119). When $\overline{\text{TRST}}$ is driven high, this pin is used as an interrupt to or from the emulator system and is defined as input/output via JTAG scan.

Table 2–1. TMS320C5x Signal Descriptions (Concluded)

Signal	Pin	State	Description
Test Signals (Concluded)			
EMU1/$\overline{\text{OFF}}$	119	I/O/Z	Emulator pin 1/disable all outputs. When $\overline{\text{TRST}}$ is driven high, this pin is used as an interrupt to or from the emulator system and is defined as input/output via JTAG scan. When $\overline{\text{TRST}}$ is driven low, this pin is configured as $\overline{\text{OFF}}$. The EMU1/ $\overline{\text{OFF}}$ signal, when active low, puts all output drivers into the high-impedance state. Note that $\overline{\text{OFF}}$ is used exclusively for testing and emulation purposes (not for multiprocessing applications). Thus, for $\overline{\text{OFF}}$ condition, the following conditions apply: $\overline{\text{TRST}}$=low, EMU0=high EMU1/$\overline{\text{OFF}}$=low
RESERVED	16 17 18 19 22 37 49 50 51 52 70 78 79 84 85 88 111 115 116 117	N/C	Reserved pin. These pins are reserved for future 'C5x devices. These pins should be left unconnected.

Chapter 3

Architecture

The architectural structure of a TMS320 DSP consists of three basic segments:

- Central processing unit (CPU)
- Memory
- Peripheral-interfacing circuits

This chapter describes the architecture and operation of the 'C5x core CPU; the memory and peripheral segments are not discussed except in relation to the core CPU of the 'C5x generation. This CPU is capable of performing high-speed arithmetic executions within a short instruction cycle by means of its highly parallel architectural design.

For information on the memory organization of the 'C5x, refer to Chapter 6, *Memory.* For further details about on-chip peripheral organization, refer to Chapter 5, *Peripherals.* The major topics in this chapter are:

Topic **Page**

3.1 Architectural Overview

The 'C5x high-performance digital signal processors are designed, like the 'C25, with an advanced Harvard-type architecture that maximizes the processing power by maintaining two separate memory bus structures, program and data, for full-speed execution. Instructions support data transfers between the two spaces.

The 'C5x performs 2s-complement arithmetic, using the 32-bit **arithmetic logic unit (ALU)** and accumulator. The ALU is a general-purpose arithmetic unit that uses 16-bit words taken from data memory or derived from immediate instructions, or the 32-bit result from the multiplier. In addition to arithmetic operations, the ALU can perform Boolean operations. The accumulator stores the output from the ALU and is also the second input to the ALU. The accumulator is 32 bits long and is divided into a high-order word (bits 31 through 16) and a low-order word (bits 15 through 0). Instructions are provided for storing those high- and low-order accumulator words in memory. For fast, temporary storage of the accumulator, there is a 32-bit accumulator buffer.

In addition to the main ALU, there is a **parallel logic unit (PLU)** that executes logic operations on data without affecting the contents of the accumulator. The PLU provides the bit-manipulation ability required of a high-speed controller and simplifies the bit setting, clearing, and testing required with control and status register operations.

The **multiplier** performs 16 × 16-bit 2s-complement multiplication with a 32-bit result in a single-instruction cycle. The multiplier consists of three elements: multiplier array, PREG (product register), and TREG0 (temporary register). The 16-bit TREG0 temporarily stores the multiplicand; the PREG stores the 32-bit product. The multiplier's values come from data memory, come from program memory when the MAC/MACD/MADS/MADD instructions are used, or are derived immediately from the multiply immediate instructions (MPY #). The fast on-chip multiplier allows the device to efficiently perform fundamental DSP operations such as convolution, correlation, and filtering.

The 'C5x **scaling shifter** has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. The scaling shifter produces a left shift of 0 to 16 bits on the input data, as programmed in the instruction or defined in the shift count register (TREG1). The LSBs of the output are filled with zeros, while the MSBs may be either zero-filled or sign-extended, depending upon the state of the sign-extension mode bit (SXM) of status register ST1. Additional shift capabilities enable the processor to perform numerical-scaling, bit-extraction, extended-arithmetic, and overflow-prevention operations.

Eight levels of **hardware stack** save the contents of the program counter during interrupts and subroutine calls. On interrupts, the strategic registers (ACC, ACCB, ARCR, INDX, PMST, PREG, ST0, ST1, TREGs) are pushed onto a one-deep stack and popped upon interrupt return, thus providing a zero-overhead interrupt context switch.

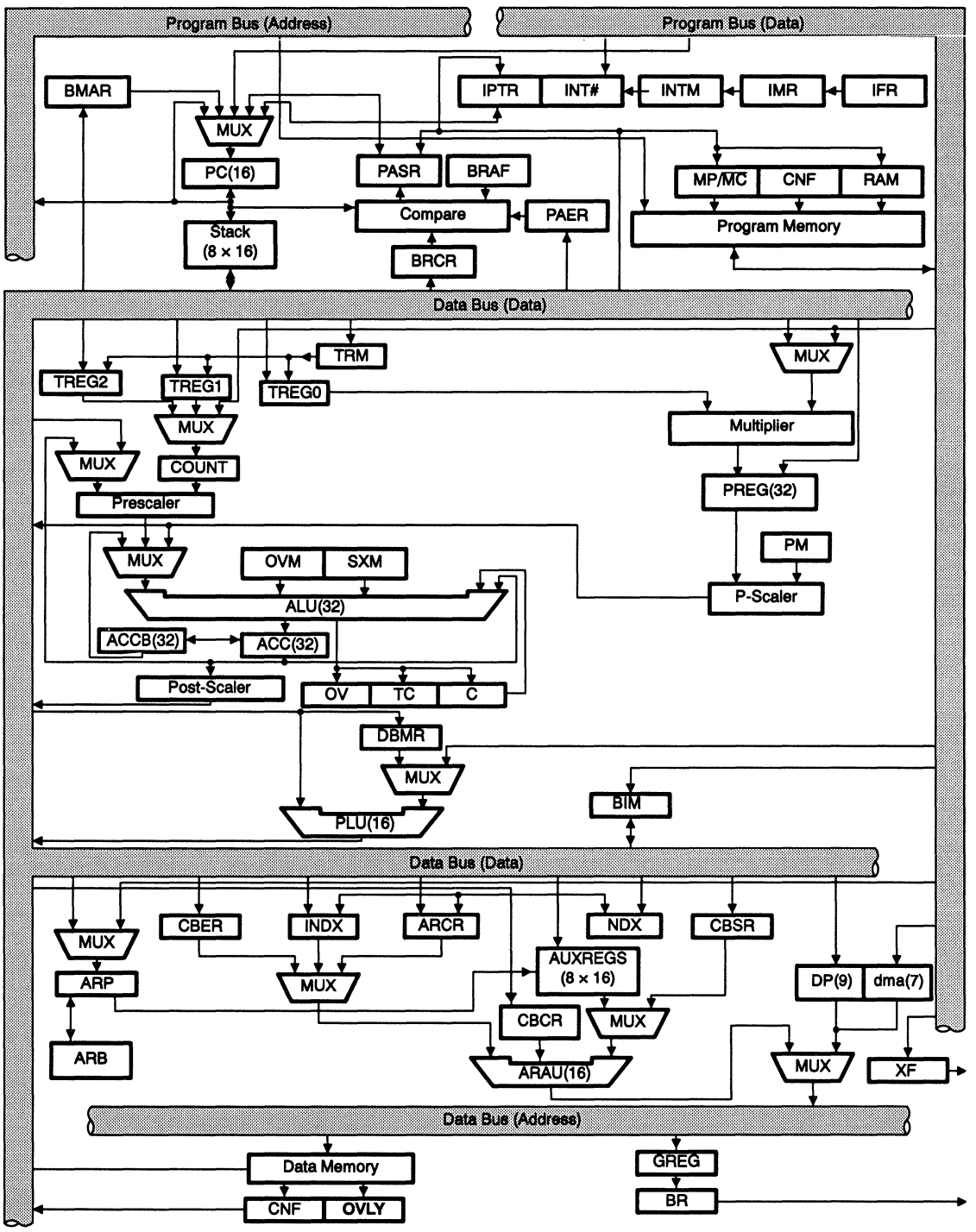
3.2 Functional Block Diagram

The functional block diagram, shown in Figure 3–1, outlines the principal blocks and data paths within the 'C5x processors. Further details of the functional blocks are provided in the succeeding sections. Refer to Section 3.3, *Internal Hardware Summary*, for definitions of the symbols used in Figure 3–1.

The 'C5x architecture is built around two major buses: the program bus and the data bus. The program bus carries the instruction code and immediate operands from program memory. The data bus interconnects various elements, such as the central arithmetic logic unit (CALU) and the auxiliary register file, to the data memory. Together, the program and data buses can carry data from on-chip data memory and internal or external program memory to the multiplier in a single cycle for multiply/accumulate operations.

The 'C5x possesses a high degree of parallelism; that is, while the data is being operated upon by the CALU, arithmetic operations may also be executed in the auxiliary register arithmetic unit (ARAU). Such parallelism results in a powerful set of arithmetic, logic, and bit-manipulation operations that may all be performed in a single machine cycle.

Figure 3–1. Block Diagram of 'C5x Internal Hardware



3.3 Internal Hardware Summary

The internal hardware of the 'C5x executes functions that other processors typically implement in software or microcode. For example, the device contains hardware for single-cycle 16 × 16-bit multiplication, data shifting, and address manipulation. This hardware-intensive approach provides computing power previously unavailable on a single chip.

Table 3–1 presents a summary of the 'C5x's internal hardware. This summary table, which includes the internal processing elements, registers, and buses, is alphabetized. All of the symbols used in the table correspond to the symbols used in Figure 3–1, the succeeding block diagrams in this chapter, and the text throughout this document.

Table 3–1. 'C5x Internal Hardware Summary

Unit	Symbol	Function
Accumulator	ACC(32) ACCH(16) ACCL(16)	A 32-bit accumulator accessible in two halves: ACCH (accumulator high) and ACCL (accumulator low). Used to store the output of the ALU. See subsection 3.5.2 for more information.
Accumulator Buffer	ACCB(32)	A register used to temporarily store the 32-bit contents of the accumulator. This register has a direct path back to the ALU and therefore can be arithmetically or logically acted upon with the ACC. See subsection 3.5.2 for more information.
Arithmetic Logic Unit	ALU	A 32-bit 2s-complement arithmetic logic unit having two 32-bit input ports and one 32-bit output port feeding the accumulator. See subsection 3.5.2 for more information.
Auxiliary Register Arithmetic Unit	ARAU	An unsigned 16-bit arithmetic unit used to calculate indirect addresses using the auxiliary, index, and compare registers as inputs. See subsection 3.4.3 for more information.
Auxiliary Register Compare	ARCR(16)	A 16-bit register used as a limit to compare indirect address against. See subsection 3.4.3 for more information.
Auxiliary Register File	AUXREGS	A register file containing eight 16-bit auxiliary registers (AR0–AR7) used for indirect data address pointers, temporary storage, or integer arithmetic processing through the ARAU. See subsection 3.4.3 for more information.
Auxiliary Register Buffer	ARB(3)	A 3-bit register that holds the previous value contained in the ARP. These bits are stored in ST1. See subsection 3.4.3 for more information.
Auxiliary Register Pointer	ARP(3)	A 3-bit register used as a pointer to the currently selected auxiliary register. These bits are stored in ST0. See subsection 3.4.3 for more information.
Block Move Address Register	BMAR(16)	A 16-bit register that holds an address value for use with block moves or multiply/accumulates. See subsection 3.4.2 for more information.
Block Repeat Active Flag	BRAF(1)	A 1-bit flag indicating that a block repeat is currently active. This bit is normally set when the RPTB instruction is executed and cleared when the BRCR register decrements below zero. This bit resides in the PMST register. See subsection 3.6.5 for more details.
Block Repeat Address End Register	PAER(16)	A 16-bit memory-mapped register containing the end address of the segment of code being repeated. See subsection 3.6.5 for more details.

Table 3–1. 'C5x Internal Hardware (Continued)

Unit	Symbol	Function
Block Repeat Address Start Register	PASR(16)	A 16-bit memory-mapped register containing the start address of the segment of code being repeated. See subsection 3.6.5 for more details.
Block Repeat Counter Register	BRCR(16)	A 16-bit memory-mapped counter register used to limit the number of times the block is to be repeated. See subsection 3.6.5 for more details.
Bus Interface Module	BIM	A buffered interface used to pass data between the internal data and program buses.
Bus Request	$\overline{BR}$	This signal indicates that a data access is mapped to global memory space as defined by the GREG register. See Section 6.4 for more details.
Carry	C	This bit stores the carry output of the ALU. This bit resides in ST1. See subsection 3.5.2 for more information.
Central Arithmetic Logic Unit	CALU	The grouping of the ALU, multiplier, accumulator, and scaling shifters. See Section 3.5 for more information.
Circular Buffer Control Register	CBCR(8)	An 8-bit register used to enable/disable the circular buffers and define which auxiliary registers are mapped to the circular buffers. See subsection 3.4.3 for more information.
Circular Buffer End Address	CBER(16) CBER1(16) CBER2(16)	Two 16-bit registers indicating circular buffer end addresses. CBER1 and CBER2 are associated with circular buffers one and two, respectively. See subsection 3.4.3 for more information.
Circular Buffer Start Address	CBSR(16) CBSR1(16) CBSR2(16)	Two 16-bit registers indicating circular buffer start addresses. CBSR1 and CBSR2 are associated with circular buffers one and two, respectively. See subsection 3.4.3 for more information.
Compare of Program Address	COMPARE	This circuit compares the current value in the PC to the value in PAER if BRAF is active. If the compare shows equal, then the PASR is loaded into the PC. See subsection 3.4.3 for more information.
Configure RAM	CNF	This bit indicates whether on-chip dual-access RAM blocks are mapped to program or data space. The CNF bit resides in ST1. See subsection 3.6.3 for more information.
Data Bus	DATA	A 16-bit bus used to route data.
Data Memory	DATA MEMORY	This block refers to data memory used with the core and defined in specific device descriptions. It refers to both on- and off-chip memory blocks in data memory space.
Data Memory Address Bus	DATA ADDRESS	A 16-bit bus that carries the address for data memory accesses.
Data Memory Address Immediate Register	dma(7)	A 7-bit register containing the immediate relative address within a 128-word data page. See subsection 3.4.2 for more information.
Data Memory Page Pointer	DP(9)	A 9-bit register containing the address of the current page. Data pages are 128 words each, resulting in 512 pages of addressable data memory space (some locations are reserved). See subsection 3.4.2 for more information.
Data RAM Map Bit	RAM(1)	This bit indicates if the single-access RAM is mapped into data space. See subsection 3.6.3 for more information.
Direct Data Memory Address Bus	DRB(16)	A 16-bit bus that carries the direct address for the data memory, which is the concatenation of the DP register and the seven LSBs of the instruction (DMA). See subsection 3.4.2 for more information.

Table 3–1. 'C5x Internal Hardware (Continued)

Unit	Symbol	Function
Dynamic Bit Manipulation Register	DBMR(16)	A 16-bit memory-mapped register used as a mask input to the PLU in the absence of a long immediate value. See Section 3.7 for more information.
Dynamic Bit Pointer	TREG2(4)	A 4-bit register that holds a dynamic bit pointer for the BITT instruction. See Section 4.3 for more information.
Dynamic Shift Count	TREG1(5)	A 5-bit register that holds a dynamic prescaling shift count for data inputs to the ALU. See Section 4.3 for more information.
External Flag	XF(1)	This bit drives the level of the external flag pin and resides in ST1. See subsection 3.6.3 for more information.
Global Memory Allocation Register	GREG(8)	An 8-bit memory-mapped register for specifying the size of the global memory space. See Section 6.4 for more details.
Hold Mode	HM(1)	This bit resides in ST1 and determines whether the CALU will stop or continue when the $\overline{\text{HOLD}}$ signal initiates a power-down mode. See Section 6.6 for more information.
Index Register	INDX(16)	This 16-bit memory-mapped register specifies increment sizes greater than 1 for indirect addressing updates. In bit-reversed addressing, the index register defines the array size. See subsection 3.4.3 for more information.
Index Register Enable	NDX(1)	This bit determines whether a modification or write to AR0 writes also to INDX and ARCR to maintain compatibility with the 'C25. This bit resides in the PMST register. See subsection 3.4.3 for more information.
Interrupt Flag Register	IFR(16)	A 16-bit flag register used to latch the active-low interrupts. The IFR is a memory-mapped register. See Section 3.8 for more information.
Interrupt Mask Bit	INTM(1)	The interrupt mask bit globally masks or enables all interrupts. This bit resides in ST0. See Section 3.8 for more information.
Interrupt Number	INT#(4)	The number of the specific interrupt being sent to the CPU to be activated. This value comes from either the interrupt-processing circuitry or, in the case of the INTR instruction, the program bus. See Section 3.8 for more information.
Interrupt Pointer	IPTR(5)	Five bits pointing to the 2K page where the interrupt vectors currently reside in the system. These bits reside in the PMST register. See Section 3.8 for more information.
Interrupt Mask Register	IMR(16)	A 16-bit memory-mapped register used to mask interrupts. See Section 3.8 for more information.
Microcall Stack	MCS (15–0)	A single-word stack that temporarily stores the contents of the PFC while the PFC is being used to address data memory with the block move (BLDD/ BLPD), multiply-accumulate (MAC/MACD), and table read/write (TBLR/ TBLW) instructions.
Microprocessor/ Microcomputer Mode	MP/$\overline{\text{MC}}$	This bit resides in the PMST register and indicates whether the on-chip ROM is mapped into program address space. See subsection 3.6.3 for more information.
Multiplexer	MUX	A bus multiplexer used to select the source of operands for a bus or execution unit, depending on the nature of the current instruction.
Multiplier	MULTIPLIER	A 16 x 16-bit parallel multiplier. See subsection 3.6.3 for more information.

Table 3–1. 'C5x Internal Hardware (Continued)

Unit	Symbol	Function
Overflow Flag	OV(1)	This bit resides in ST0 and indicates an overflow in an arithmetic operation in the ALU. See subsection 3.6.3 for more information.
Overflow Mode	OVM(1)	This bit resides in ST0 and determines whether an overflow in the ALU will wrap around or saturate. See subsection 3.6.3 for more information.
Overlay to Data Space	OVLY(1)	This bit resides in the PMST register and determines whether the on-chip single-access memory will be addressable in data address space. See subsection 3.6.3 for more information.
Parallel Logic Unit	PLU	A 16-bit logic unit that executes logic operations from either long immediate operands or the contents of the DBMR directly upon data locations without interfering with the contents of the CALU registers. See Section 3.7 for more information.
Prefetch Counter	PFC (15–0)	A 16-bit counter used to prefetch program instructions. The PFC contains the address of the instruction currently being prefetched. It is updated when a new prefetch is initiated. The PFC can also address program memory when the block move (BLPD), multiply-accumulate (MAC/MACD), and table read/write (TBLR/TBLW) instructions are used and can address data memory when the block move (BLDD) instruction is used.
Prescaler Count Register	COUNT(4)	A four-bit register that contains the value for the prescaling operation. When the register contents are used as prescaling data, this register is loaded from the dynamic shift count or from the instruction. In conjunction with the BIT and BITT instructions, this register is loaded from the dynamic bit pointer or the instruction word.
Product Register	PREG(32)	A 32-bit product register used to hold the multiplier's product. The high and low words of the PREG can be accessed individually. See subsection 3.5.3 for more information.
Program Bus	PROG DATA	A 16-bit bus used to route instructions (and data for the MAC and MACD instructions).
Program Counter	PC(16)	A 16-bit program counter used to address program memory sequentially. The PC always contains the address of the next instruction to be fetched. The PC contents are updated following each instruction decode operation.
Program Memory	PROGRAM MEMORY	This block refers to program memory used with the core and defined in specific device descriptions. It refers to both on- and off-chip memory blocks accessed in program memory space.
Program Memory Address Bus	PROG ADDRESS	A 16-bit bus that carries the program memory address.
Prescaling Shifter	PRESCALER	A 0- to 16-bit left barrel shifter used to prescale data coming into the ALU. Also used to align data for multiprecision operations. This shifter is also used as a 0- to 16-bit right barrel shifter of the ACC. See subsection 3.5.2 for more information.
Postscaling Shifter	POST-SCALER	A 0- to 7-bit left barrel shifter used to postscale data coming out of the CALU. See subsection 3.5.2 for more information.
Product Shifter	P-SCALER	A 0-, 1-, or 4-bit left shifter that can remove extra sign bits (gained in the multiply operation) when fixed-point arithmetic is used; or a 6-bit right shifter that can scale the products down to avoid overflow in the accumulation process. See subsection 3.5.3 for more information.

Table 3–1. 'C5x Internal Hardware (Continued)

Unit	Symbol	Function
Product Shifter Mode	PM(2)	These two bits define the product shifter mode; They reside in ST1. See subsection 3.6.3 for more information.
Repeat Counter	RPTC(16)	A 16-bit counter used to control the repeated execution of a single instruction. See subsection 3.6.4 for more information.
Sign Extension Mode	SXM(1)	This bit resides in ST1 and controls whether the arithmetic operation will be sign-extended or not. See subsection 3.6.3 for more information.
Stack	STACK	An 8 x 16-bit hardware stack used to store the PC during interrupts and calls. The ACCL and data memory values may also be pushed onto and popped from the stack. See Section 3.8 for more information.
Status Registers	ST0, ST1, PMST	Three 16-bit status registers that contain status and control bits. See subsection 3.6.3 for more information.
Temporary Multiplicand	TREG0(16)	A 16-bit register that temporarily holds an operand for the multiplier. See subsection 3.5.3 for more information.
Temporary Registers Enable	TRM(1)	This bit defines whether an LT(A,D,P,S) instruction loads all three of the TREGs(0,1,2) to maintain compatibility with the 'C25 or loads just TREG0. This bit resides in the PMST register. See subsection 3.6.3 for more information.
Test/Control Flag	TC(1)	This bit resides in ST1 and stores the results of ALU or PLU test bit operations. See subsection 3.6.3 for more information.

3.4 Internal Memory Organization

This section describes the memory use of the 'C5x core and the addressing modes supported by the core.

3.4.1 Core Processor Memory-Mapped Registers

Twenty-eight core processor registers are mapped into the data memory space. These are listed in Table 3–2. An additional 64 memory-mapped registers are reserved in page 0 of data space. These data memory locations are reserved for memory-mapped peripheral control and I/O port registers.

Table 3–2. Core Processor Memory-Mapped Registers

Name	Address		Description
	'C5x Dec	'C5x Hex	
—	0 – 3	0 – 3	Reserved
IMR	4	4	Interrupt mask register
GREG	5	5	Global memory allocation register
IFR	6	6	Interrupt flag register
PMST	7	7	Processor mode status register
RPTC	8	8	Repeat counter register
BRCR	9	9	Block repeat counter register
PASR	10	A	Block repeat program address start register
PAER	11	B	Block repeat program address end register
TREG0	12	C	Temporary register for multiplicand
TREG1	13	D	Temporary register for dynamic shift count
TREG2	14	E	Temporary register used as bit pointer in dynamic bit test
DBMR	15	F	Dynamic bit manipulation register
AR0	16	10	Auxiliary register zero
AR1	17	11	Auxiliary register one
AR2	18	12	Auxiliary register two
AR3	19	13	Auxiliary register three
AR4	20	14	Auxiliary register four
AR5	21	15	Auxiliary register five
AR6	22	16	Auxiliary register six
AR7	23	17	Auxiliary register seven
INDX	24	18	Index register
ARCR	25	19	Auxiliary register compare register
CBSR1	26	1A	Circular buffer 1 start address register
CBER1	27	1B	Circular buffer 1 end address register
CBSR2	28	1C	Circular buffer 2 start address register
CBER2	29	1D	Circular buffer 2 end address register
CBCR	30	1E	Circular buffer control register
BMAR	31	1F	Block move address register
—	32–79	20–4F	Memory-mapped peripheral registers. See Table 5–1.
	80–95	50–5F	Memory-mapped I/O port. See Table 5–1.

3.4.2 Memory Addressing Modes

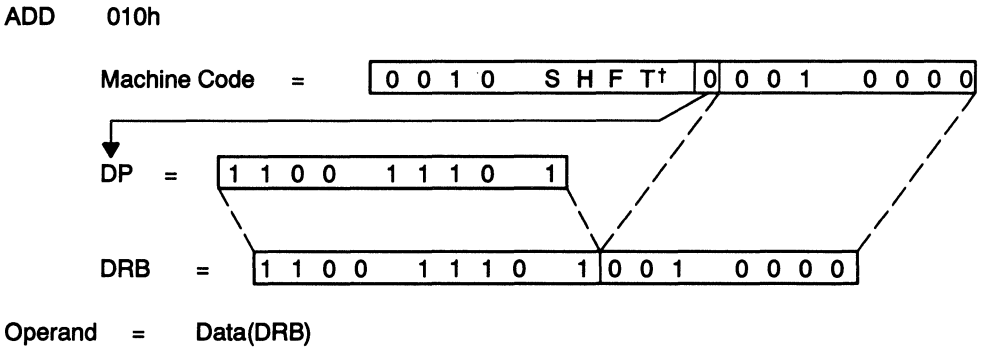
The 'C5x can address a total of 64K words of program memory and 96K words of data memory. Chapter 6 shows how the on-chip program and data memories are mapped.

The data used as instruction operands is obtained in one of the following eight ways:

- By the direct address bus (DRB) using the direct addressing mode (e.g., ADD 010h) relative to the data memory page pointer (DP)
- By the DRB using the memory-mapped addressing mode (that is, LAMM PMST) within data page zero
- By the auxiliary register file bus (AFB) using the indirect addressing mode (that is, ADD *)
- By the instruction register (IREG) in short immediate operand mode (that is, ADD #0FFh)
- By the program counter (PC) in long immediate operand mode (that is, ADD #0FFFFh)
- By the core CPU access of a register in register access mode (that is, APL *+ or MPY *+)
- By the second instruction word in long immediate address mode (that is, BLDD #TBL1,*+)
- By the block memory address register (BMAR) in registered block memory addressing mode (that is, BLDD *+)

In the direct addressing mode, the 9-bit DP points to one of 512 pages (1 page = 128 words). The data memory address (dma), specified by the seven LSBs of the instruction, points to the desired word within the page. The address on the DRB is formed by concatenating the 9-bit DP with the 7-bit dma. Figure 3–2 illustrates direct addressing mode. In the illustration, the operand is fetched from data memory space via the data bus, and the address is the concatenated value of the DP and the seven LSBs of the instruction. Note that bit 7=0 defines the addressing mode as direct.

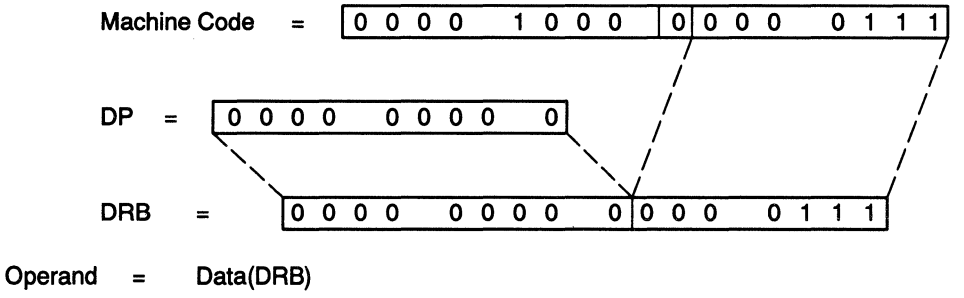
Figure 3–2. Direct Addressing Mode



† SHFT represents a 4-bit shift value.

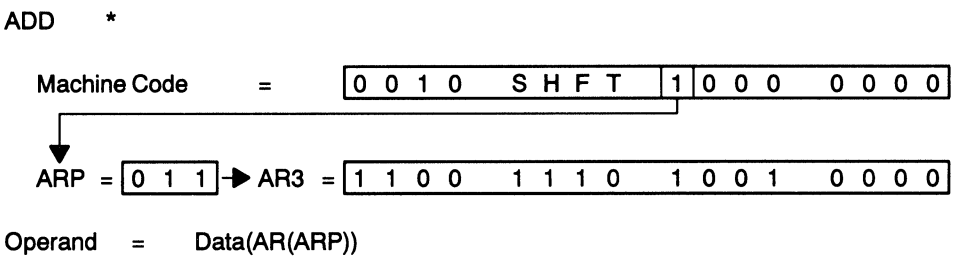
Memory-mapped addressing mode operates much like direct addressing mode except that the most significant 9 bits of the address are forced to zero instead of being loaded with the contents of the DP. This allows the user to directly address the memory-mapped registers of data page zero without the overhead of changing the DP or auxiliary register. Figure 3–3 illustrates memory-mapped addressing mode.

Figure 3–3. Memory-Mapped Addressing Mode



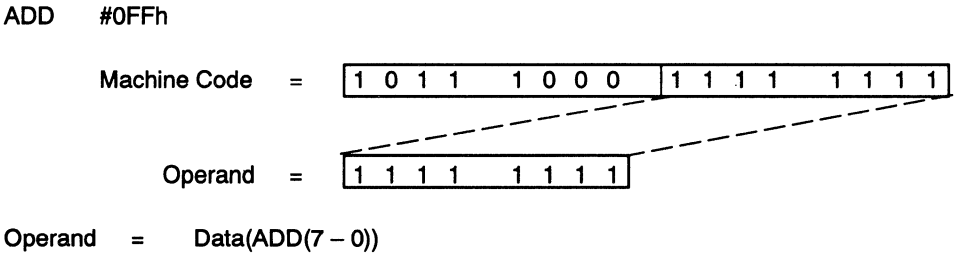
In the indirect addressing mode, the currently selected 16-bit auxiliary register AR(ARP) addresses the data memory through the auxiliary register file bus (AFB). While the selected auxiliary register provides the data memory address and the data is being manipulated by the CALU, the contents of the auxiliary register may be manipulated through the ARAU. See Figure 3–4 for an example of indirect auxiliary register addressing. Also, bit 7=1 defines this addressing mode as indirect.

Figure 3–4. Indirect Addressing Mode



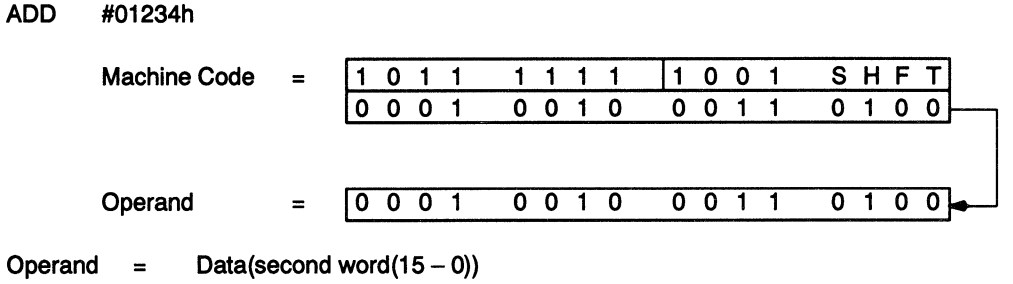
The operand may reside as part of the instruction machine code. In the case of the short immediate operand, the operand is contained in the single-word instruction. These short immediate operands vary in length from 1 bit on the SETC instruction to 13 bits on the MPY instruction. Figure 3–5 shows an example of short immediate mode. Note that, in this example, the lower eight bits are the operand and will be added to the ACC by the CALU.

Figure 3–5. Short Immediate Mode



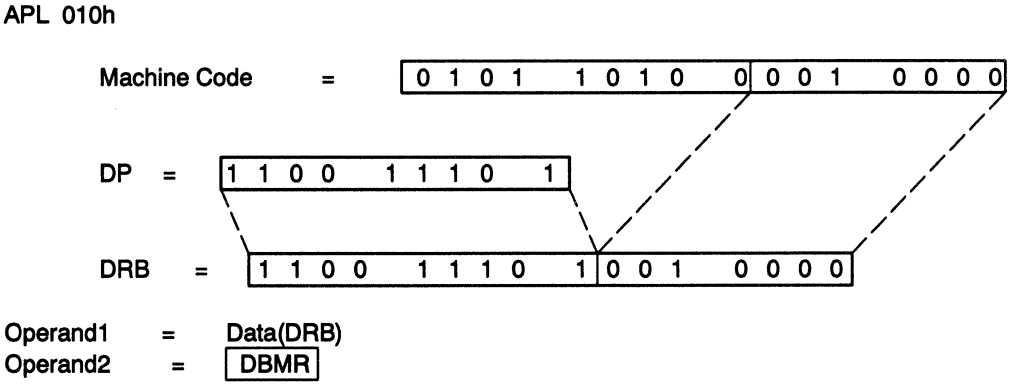
In the case of the long immediate operand, the operand immediately follows the opcode in the program sequence. The long immediate operand is 16 bits long. Figure 3–6 shows an example of long immediate mode. In this example, the second word of the two-word instruction is added to the ACC by the CALU.

Figure 3–6. Long Immediate Mode



The operand may come from a CPU register. This type of operand is used in special cases. The CALU uses this in multiplying with TREG0, in shifting with TREG1 and PM, and in bit manipulation with TREG2. The ARAU uses this with INDX and ARCR. The PLU uses this with DBMR. Figure 3–7 illustrates the use of the DBMR register as an AND mask in the APL instruction.

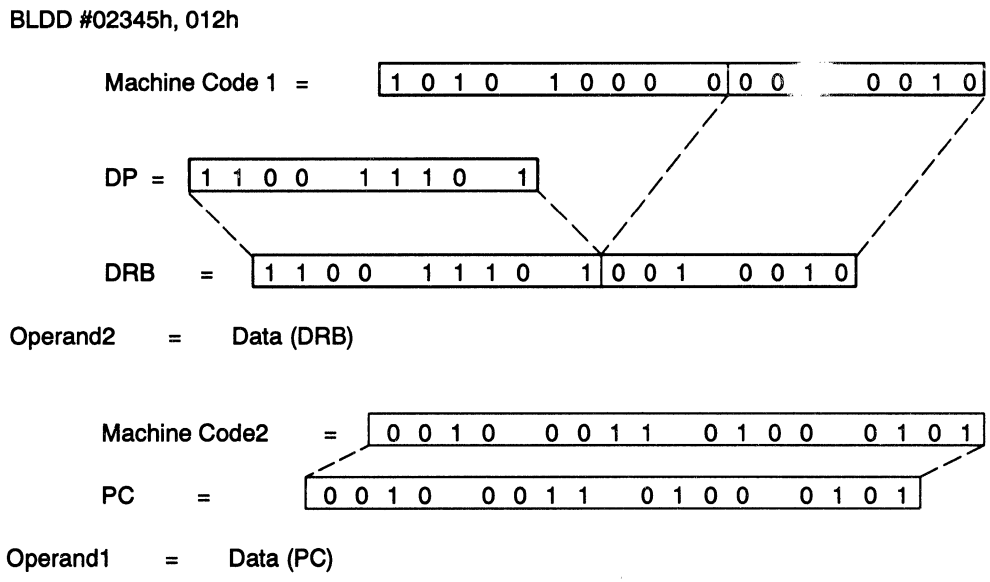
Figure 3–7. Register Access Mode



In the long immediate addressing mode, an operand is addressed by the second word of a two-word instruction. In this case, the program address/data bus (PAB) is used for the operand fetch. The PC is stored in a temporary register, and the long immediate value is loaded into the PC. Then, the PAB is used for the operand fetch or write. At the completion of the instruction, the PC is restored from the temporary register, and execution continues. This technique is used when two memory addresses are required for the execution of the in-

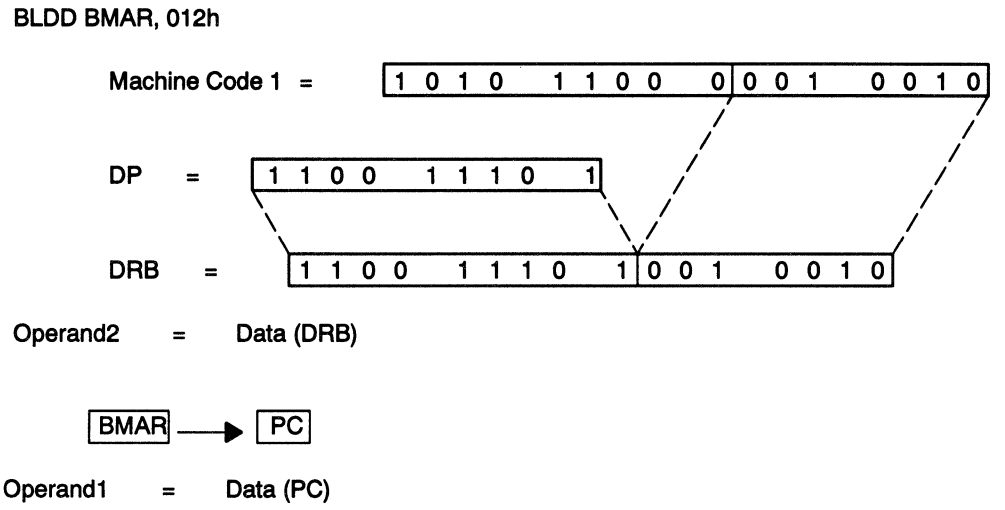
struction. The PC is used so that, when an instruction is repeated, the address generated can be autoincremented. Figure 3–8 illustrates this mode. In this illustration, the source address (OPERAND1) is fetched via PAB, and the destination address (OPERAND2) uses the direct addressing mode.

Figure 3–8. Long Immediate Addressing Mode



Registered block memory addressing mode operates like the long immediate addressing mode with the exception that the address comes from BMAR. The advantage of this technique is that the address of the block of memory to be acted upon can be changed during execution of the program. The address in long immediate addressing mode resides in the program flow and cannot be easily changed. Figure 3–9 shows an example of registered block memory addressing mode.

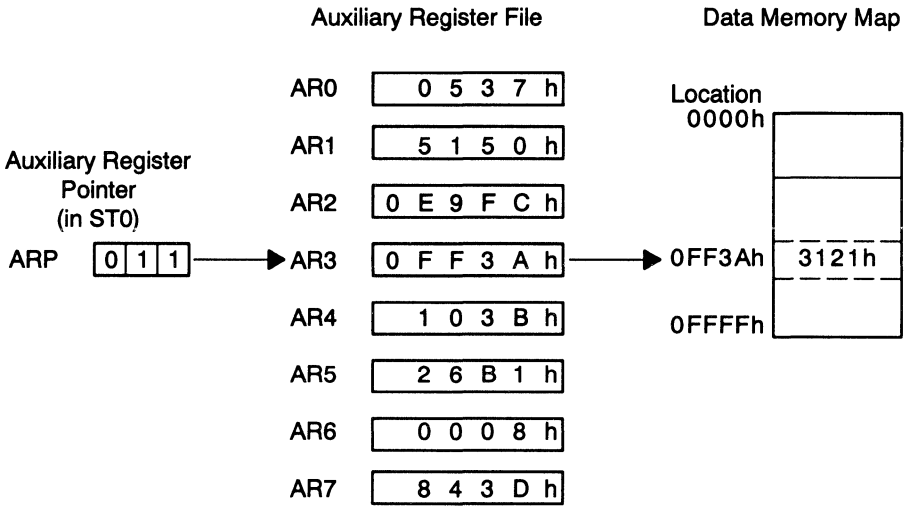
Figure 3–9. Registered Block Memory Addressing Mode



3.4.3 Auxiliary Registers

The 'C5x provides a register file containing eight auxiliary registers (AR0–AR7). The auxiliary registers may be used for indirect addressing of the data memory or for temporary data storage. Indirect auxiliary register addressing (see Figure 3–10) allows placement of the data memory address of an instruction operand into one of the auxiliary registers. These registers are pointed to by a three-bit auxiliary register pointer (ARP) that is loaded with a value from 0 through 7, designating AR0 through AR7, respectively. The auxiliary registers and the ARP may be loaded from data memory, the accumulator, the product register, or by an immediate operand defined in the instruction. The contents of these registers may also be stored in data memory or used as inputs to the CALU. These registers appear in the memory map as described in Table 3–2.

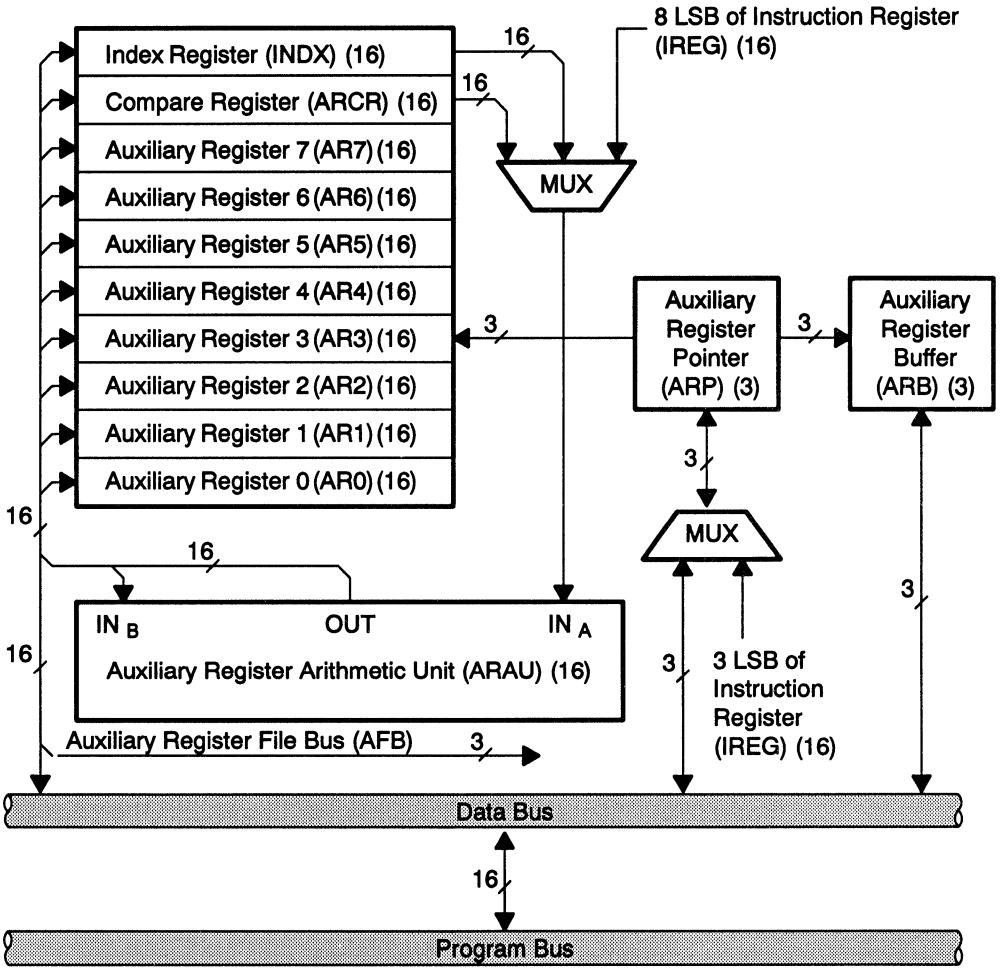
Figure 3–10. Indirect Auxiliary Register Addressing Example



The auxiliary register file (AR0–AR7) is connected to the auxiliary register arithmetic unit (ARAU), shown in Figure 3–11. The ARAU may autoindex the current auxiliary register while the data memory location is being addressed. Indexing either by ±1 or by the contents of the INDX register may be performed. As a result, accessing tables of information does not require the central arithmetic logic unit (CALU) for address manipulation; thus, the CALU is free for other operations in parallel.

If more advanced address manipulation is required, such as multidimensional array addressing, the CALU can directly read from or write to the auxiliary registers. However, the ARAU update of the ARs is done during the decode phase (second cycle) of the pipeline, while the CALU write is done during the execution phase (fourth cycle) of the pipeline. Therefore, the two instructions directly following the CALU write to an auxiliary register should not use the same auxiliary register for address generation. See subsection 3.6.2 for details.

Figure 3–11. Auxiliary Register File



As shown in Figure 3–11, the index register, compare register, or the eight LSBs of the instruction register can be used as one of the inputs of the ARAU. The other input is fed by the current AR (being pointed to by ARP). AR(ARP) refers to the contents of the current AR pointed to by ARP. The ARAU performs the functions shown in Table 3–3.

Table 3–3. Auxiliary Register Arithmetic Unit Functions

Auxiliary Register Operation	Description
AR(ARP) + INDX → AR(ARP)	Index the current AR by adding an unsigned 16-bit integer contained in INDX. Example: ADD *0+
AR(ARP) – INDX → AR(ARP)	Index the current AR by subtracting an unsigned 16-bit integer contained in INDX. Example: ADD *0–
AR(ARP) + 1 → AR(ARP)	Increment the current AR by one. Example: ADD *+
AR(ARP) – 1 → AR(ARP)	Decrement the current AR by one. Example: ADD *–
AR(ARP) → AR(ARP)	Do not modify the current AR. Example: ADD *
AR(ARP) + IR(7–0) → AR(ARP)	Add an 8-bit immediate value to current AR. Example: ADDRK *55h
AR(ARP) – IR(7–0) → AR(ARP)	Subtract an 8-bit immediate value from current AR. Example: SBRK *55h
AR(ARP) + rc(INDX) → AR(ARP)	Bit-reversed indexing; add INDX with reversed-carry (rc) propagation. Example: ADD *BR0+
AR(ARP) – rc(INDX) → AR(ARP)	Bit-reversed indexing; subtract INDX with reversed-carry (rc) propagation. Example: ADD *BR0–

If (AR(ARP)) = (ARCR) then TC = 1
If (AR(ARP)) < (ARCR) then TC = 1
If (AR(ARP)) > (ARCR) then TC = 1
If (AR(ARP)) ≠ (ARCR) then TC = 1
	Compare the current AR to ARCR and, if condition is true, then set TC bit of the status register ST1 to one. If false, then clear the TC bit. Example: CMPR 3

If (AR(ARP)) = (CBER) then AR(ARP) = CBSR	If at end of circular buffer, reload start address. The test for this condition is done prior to the execution of the auxiliary register modification. Example: ADD *+

The index register (INDX) can be added to or subtracted from AR(ARP) on any AR update cycle. This 16-bit register is one of the memory-mapped registers and is used to increment or decrement the address in steps larger than one, which is useful for operations such as addressing down a column of a matrix. The auxiliary register compare register (ARCR) is used as a limit to blocks of data and, in conjunction with the CMPR instruction, supports logical comparisons between AR(ARP) and ARCR. Note that the 'C25 uses AR0 for these two functions. After reset, a LAR load of AR0 also loads INDX and ARCR to maintain compatibility with the 'C25. The splitting of functions to the three registers is enabled by setting the NDX bit of PMST to one.

Because the auxiliary registers are memory-mapped, they can be acted upon directly by the CALU to provide for more advanced indirect addressing techniques. For example, the multiplier can be used to calculate the addresses of three-dimensional matrices. After a CALU load of the auxiliary register, there is, however, a two-instruction-cycle delay before auxiliary registers can be used for address generation. The INDX and ARCR registers are accessible via the CALU, regardless of the condition of the NDX bit (i.e., SAMM ARCR writes only to the ARCR).

In addition to its use for address manipulation in parallel with other operations, the ARAU may also serve as an additional general-purpose arithmetic unit because the auxiliary register file can directly communicate with data memory. The ARAU implements 16-bit unsigned arithmetic, whereas the CALU implements 32-bit 2s-complement arithmetic. The BANZ and BANZD instructions permit the auxiliary registers to be used as loop counters.

The 3-bit auxiliary register pointer buffer (ARB), shown in Figure 3-11, provides storage for the ARP on subroutine calls when the automatic context switch feature of the device are not used.

Two circular buffers can operate at a given time and are controlled via the circular buffer control register (CBCR). The CBCR is defined as shown in Table 3–4.

Table 3–4. Circular Buffer Control Register (CBCR)

Bit	Name	Function
0–2	CAR1	Identifies which auxiliary register is mapped to circular buffer 1.
3	CENB1	Circular buffer 1 enable=1/disable=0. Set to 0 upon reset.
4–6	CAR2	Identifies which auxiliary register is mapped to circular buffer 2.
7	CENB2	Circular buffer 2 enable=1/disable=0. Set to 0 upon reset.

Upon reset ($\overline{RS}$ rising edge), both circular buffers are disabled. To define a circular buffer, load the CBSR1/2 with the start address of the buffer and CBER1/2 with the end address, and load the auxiliary register to be used with the buffer with an address between the start and end addresses. Finally, load CBCR with the appropriate auxiliary register number and set the enable bit. Note that the same auxiliary register can not be enabled for both circular buffers, or unexpected results will occur. As the address is stepping through the circular buffer, the auxiliary register value is compared against the value contained in CBER prior to the update to the auxiliary register value. If the current auxiliary register value and the CBER are equal and an auxiliary register modification occurs, the value contained in CBSR is automatically loaded into the AR. If the values in the CBER and the auxiliary register are not equal, the auxiliary register is modified as specified.

Circular buffers can be used with either increment- or decrement-type updates. If increment is used, then the value in CBER must be greater than the value in CBSR. If decrement is used, the value in CBER must be less than the value in CBSR. The other indirect addressing modes can be used; however, the ARAU tests only for the condition AR(ARP) = CBER. The ARAU will not detect an AR update that steps over the value contained in CBER. See subsection 4.1.6 for further details.

3.4.4 Memory-to-Memory Moves

The 'C5x provides instructions for data and program block moves and for data move functions that efficiently utilize the memory spaces of the device.

The BLDD instruction moves a block within data memory, the BLPD instruction moves a block from program memory to data memory, and the BLDP instruction moves a block from data memory to program memory. One of the addresses of these instructions comes from the data address generator, while the other comes either from a long immediate constant or from the BMAR. When used with the repeat instructions (RPT and RPTZ), these instructions efficiently perform block moves from on-chip or off-chip memory.

Implemented in on-chip data RAM, the DMOV (data move) function is equivalent to that of the 'C25. DMOV copies a word from the currently addressed data memory location in on-chip RAM to the next-higher location, while the data from the addressed location is being operated upon in the same cycle (e.g., by the CALU). An ARAU operation may also be performed in the same cycle when the indirect addressing mode is used. The DMOV function can implement algorithms that use the z^{-1} delay operation, such as convolution and digital filtering, where data is being passed through a time window. The data move function is at its highest efficiency when operating in dual-access on-chip memory. When operating in single-access memory, it requires an additional cycle. It is contiguous across the boundary of blocks B0 and B1. The MACD (multiply and accumulate with data move), MADD (multiply and accumulate with data move and coefficient address contained in BMAR), DMOV (data move) and LTD (load TREG0 with data move and accumulate product) instructions make use of the data move function.

Note:

The data move operation cannot be performed on external data memory.

The TBLR/TBLW (table read/write) instructions transfer words between program and data spaces. TBLR reads words from program memory into data memory. TBLW writes words from data memory to program memory.

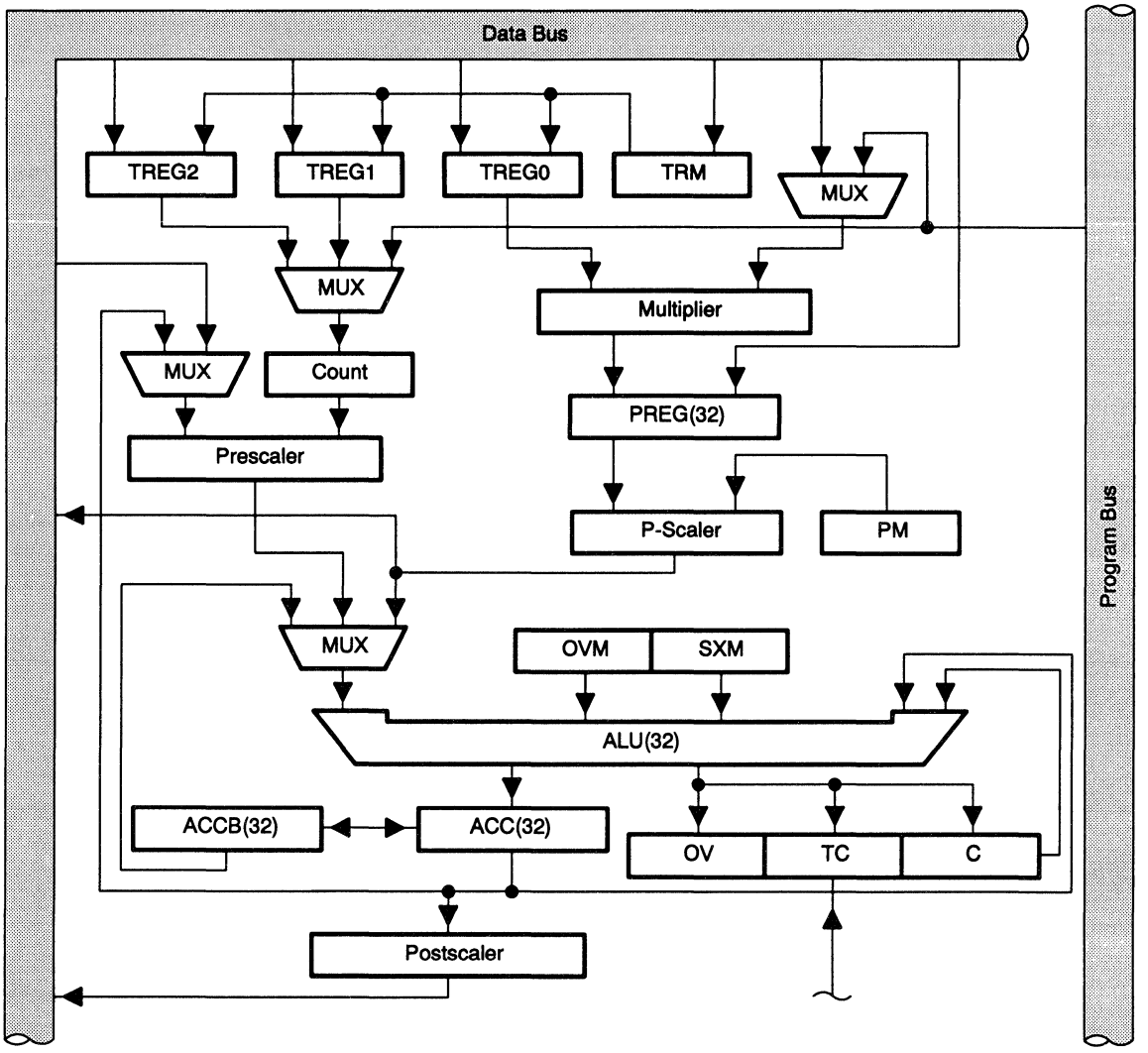
3.5 Central Arithmetic Logic Unit (CALU)

The 'C5x central arithmetic logic unit (CALU) contains a 16-bit scaling shifter, a 16 × 16-bit parallel multiplier, a 32-bit arithmetic logic unit (ALU), a 32-bit accumulator (ACC), a 32-bit accumulator buffer (ACCB), and additional shifters at the outputs of both the accumulator and the multiplier. This section describes the CALU components and their functions. Figure 3–12 is a block diagram showing the components of the CALU. The following steps occur in the implementation of a typical ALU instruction:

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

One input to the ALU is always provided by the accumulator. The other input may be transferred from the product register (PREG) of the multiplier, the accumulator buffer (ACCB), or the scaling shifter that is loaded from data memory or the accumulator (ACC).

Figure 3–12. Central Arithmetic Logic Unit



3.5.1 Scaling Shifter

The 'C5x provides a scaling shifter that has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU; see Figure 3–12. The scaling shifter produces a left shift of 0 to 16 bits on the input data. The shift count is specified by a constant embedded in the instruction word or by the value in TREG1. The LSBs of the output are filled with zeros; the MSBs may be either filled with zeros or sign-extended, depending upon the value of the SXM bit (sign-extension mode) of status register ST1.

The 'C5x also contains several other shifters that allow it to perform numerical scaling, bit extraction, extended-precision arithmetic, and overflow prevention. These shifters are connected to the output of the product register and the accumulator.

3.5.2 ALU and Accumulator

The 'C5x 32-bit ALU and accumulator implement a wide range of arithmetic and logical functions, the majority of which execute in a single clock cycle. Once an operation is performed in the ALU, the result is transferred to the accumulator where additional operations, such as shifting, may occur. Data that is input to the ALU may be scaled by the scaling shifter.

The ALU is a general-purpose arithmetic/logic unit that operates on 16-bit words taken from data memory or derived from immediate instructions. In addition to the usual arithmetic instructions, the ALU can perform Boolean operations, facilitating the bit manipulation ability required of a high-speed controller. One input to the ALU is always supplied by the accumulator, and the other input may be furnished from the product register (PREG) of the multiplier, the accumulator buffer (ACCB), or the output of the scaling shifter (that has been read from data memory or from the ACC). After the ALU has performed the arithmetic or logical operation, the result is stored in the accumulator. For the following example, assume ACC = 0, PREG = 000222200h, PM = 00, and ACCB = 000333300h:

```
LACC   #01111h,8 ;ACC = 00111100. Load ACC from prescaling
                 :shifter
APAC             ;ACC = 00333300. Add to ACC the
                 ;product register.
ADDB             ;ACC = 00666600. Add to ACC the
                 :accumulator buffer.
```

The 32-bit accumulator (ACC) can be split into two 16-bit segments for storage in data memory; see Figure 3–12. Shifters at the output of the accumulator provide a left shift of 0 to 7 places. This shift is performed while the data is being transferred to the data bus for storage. The contents of the accumulator remain unchanged. When the postscaling shifter is used on the high word of the accumulator (bits 16 – 31), the MSBs are lost and the LSBs are filled with bits shifted in from the low word (bits 0 – 15). When the postscaling shifter is used on the low word, the LSBs are zero-filled. For the following example, assume ACC = 0FF234567h:

```
SACL  TEMP1,7   ;TEMP1 = 0B380   ACC = 0FF234567.
SACH  TEMP2,7   ;TEMP2 = 91A2    ACC = 0FF234567.
```

The 'C5x supports floating-point operations for applications requiring a large dynamic range. By performing left shifts, the NORM (normalization) instruction is used to normalize fixed-point numbers contained in the accumulator. The four bits of the TREG1 define a variable shift through the scaling shifter for the ADDT/ LACT/SUBT instructions (add to / load to / subtract from accumulator

with shift specified by TREG1). These instructions are useful in denormalizing a number (converting from floating-point to fixed-point). They are also useful in execution of an automatic gain control (AGC) going into a filter.

The single-cycle 1-bit to 16-bit right shift of the accumulator can efficiently align the accumulator's contents. This, coupled with the 32-bit temporary buffer on the accumulator, enhances the effectiveness of the CALU in extended-precision arithmetic. The accumulator buffer register (ACCB) provides a temporary storage place for a fast save of the accumulator. The ACCB can also be used as an input to the ALU. The minimum or maximum value in a string of numbers can be found by comparing the contents of the ACCB with the contents of the ACC. The minimum or maximum value is placed in both registers, and, if the condition is met, the carry bit (C) is set to 1. The minimum and maximum functions are executed by the CRLT and CRGT instructions, respectively. These operations are signed arithmetic operations. For the following examples, assume ACC=012345678h and ACCB= 076543210h:

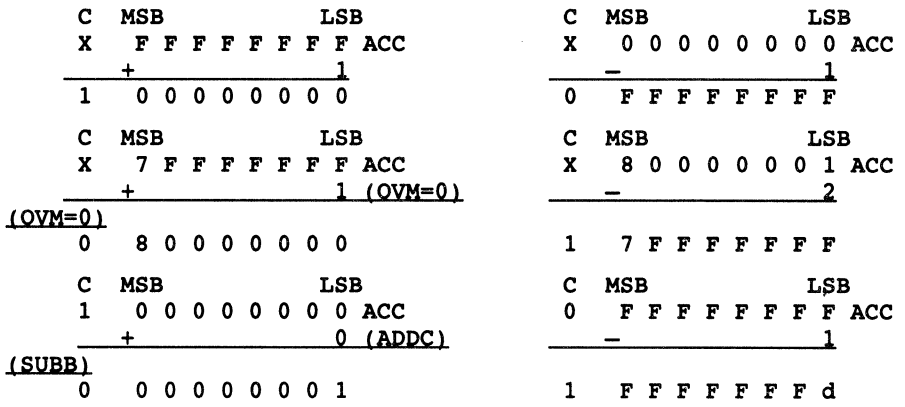
```
CRLT  ;ACC = ACCB = 12345678. C = 1.

CRGT  ;ACC = ACCB = 76543210. C = 0.
```

The accumulator's overflow saturation mode may be enabled/disabled by setting/resetting the OVM bit of ST0. When the accumulator is in the overflow saturation mode and an overflow occurs, the overflow flag is set and the accumulator is loaded with either the most positive or the most negative value representable in the accumulator, depending upon the direction of the overflow. The value of the accumulator upon saturation is 07FFFFFFh (positive) or 080000000h (negative). If the OVM (overflow mode) status register bit is reset and an overflow occurs, the overflowed results are loaded into the accumulator without modification. Note that logical operations cannot result in overflow.

The 'C5x can execute a variety of branch instructions that depend on the status of the ALU and the accumulator. For example, execution of the instruction BCND can depend on a variety of conditions in the ALU and the accumulator. The BACC instruction allows branching to an address stored in the accumulator. The bit test instructions (BITT and BIT) facilitate branching on the condition of a specified bit in data memory.

The 'C5x accumulator also has an associated carry bit that is set or reset, depending on various operations within the device. The carry bit allows more efficient computation of extended-precision products and additions or subtractions. It is quite useful in overflow management. The carry bit is affected by most arithmetic instructions as well as the single-bit shift and rotate instructions. It is not affected by loading the accumulator, logical operations, or other such non-arithmetic or control instructions. Examples of carry bit operations are shown in Figure 3–13.

Figure 3–13. Examples of Carry Bit Operations



Shown in the examples of Figure 3–13, the value added to or subtracted from the accumulator may come from the input scaling shifter, ACCB, or PREG. The carry bit is set if the result of an addition or accumulation process generates a carry; it is reset to zero if the result of a subtraction generates a borrow. Otherwise, it is cleared after an addition or set after a subtraction.

The ADDC (add to accumulator with carry) and SUBB (subtract from accumulator with borrow) instructions use the previous value of carry in their addition/subtraction operation. The ADCB (add ACCB to accumulator with carry) and the SBBB (subtract ACCB from accumulator with borrow) also use the previous value of carry.

The one exception to operation of a carry bit, as shown in Figure 3–13, is in the use of ADD with a shift count of 16 (add to high accumulator) and SUB with a shift count of 16 (subtract from high accumulator). This case of the ADD instruction can set the carry bit only if a carry is generated, and this case of the SUB instruction can reset the carry bit only if a borrow is generated; otherwise, neither instruction affects it. This feature is useful for extended precision arithmetic, as discussed in Chapter 7.

Two conditional operands, C and NC, are provided for branching, calling, returning, and conditionally executing according to the status of the carry bit. The CLRC, LST #1, and SETC instructions can also be used to load the carry bit. The carry bit is set to one on a hardware reset.

The SFL and SFR (in-place one-bit shift to the left/right) instructions and the ROL and ROR (rotate to the left/right) instructions shift or rotate the contents of the accumulator through the carry bit. The SXM bit affects the definition of the SFR (shift accumulator right) instruction. When SXM = 1, SFR performs an arithmetic right shift, maintaining the sign of the accumulator's data. When SXM = 0, SFR performs a logical shift, shifting out the LSBs and shifting in a zero for the MSB. The SFL (shift accumulator left) instruction is not affected by the SXM bit and behaves the same in both cases, shifting out the MSB and

shifting in a zero. The repeat (RPT and RPTZ) instructions may be used with the shift and rotate instructions for multiple-bit shifts.

The SFLB, SFRB, RORB, and ROLB instructions can shift or rotate the 65-bit combination of the accumulator, ACCB, and carry bit as described above.

The accumulator can also be right-shifted 0 – 31 bits in two instruction cycles or 1 – 16 bits in one cycle. The bits shifted out are lost, and the bits shifted in are either zeros or copies of the original sign bit, depending on the value of the SXM status bit. A shift count of 1 to 16 is embedded in the instruction word of the BSAR instruction. For example, let ACC = 012345678h:

```
BSAR   7        ;ACC = 02468ACE.
```

The right shift can also be controlled via TREG1. The SATL instruction shifts the ACC by 0–15 bits as defined by bits 3 – 0 of TREG1. The SATH instruction shifts the ACC 16 bits to the right if bit 4 of TREG1 is a 1. The following code sequence executes a 0- to 31-bit right shift of the ACC based on the shift count stored at SHFT. As an example, consider the value stored at SHFT = 01Bh and ACC = 012345678h:

```
LMMR   TREG1,SHFT    ;TREG1 = shift count 0 – 31. TREG1 = 1B
SATH                 ;If shift count > 15, then ACC >> 16
                     ;ACC = 00001234
SATL                 ;ACC >> shift count. ACC = 00000002
```

3.5.3 Multiplier, TREG0, and PREG

The 'C5x uses a 16 × 16-bit hardware multiplier that is capable of computing a signed or an unsigned 32-bit product in a single machine cycle. All multiply instructions, except the MPYU (multiply unsigned) instruction, perform a signed multiply operation in the multiplier. That is, two numbers being multiplied are treated as 2s-complement numbers, and the result is a 32-bit twos-complement number. As shown in Figure 3–12, the following two registers are associated with the multiplier:

- 16-bit temporary register (TREG0) that holds one of the operands for the multiplier, and
- 32-bit product register (PREG) that holds the product.

Four product shift modes (PM) are available at the PREG's output. These shift modes are useful for performing multiply/accumulate operations, performing fractional arithmetic, or justifying fractional products. The PM field of status register ST1 specifies the PM shift mode, as shown in Table 3–5.

Table 3–5. Product Shift Modes

PM	Resulting Shift
00	No shift
01	Left shift of 1 bit
10	Left shift of 4 bits
11	Right shift of 6 bits

The product is shifted one bit to compensate for the extra sign bit gained in multiplying two 16-bit 2s-complement numbers (MPY). The four-bit shift is used in conjunction with the MPY instruction with a short immediate value (13 bits or less) to eliminate the four extra sign bits gained in multiplying a16-bit number times a 13-bit number. The output of PREG can, instead, be right-shifted 6 bits to enable the execution of up to 128 consecutive multiply/accumulates without the possibility of overflow. Note that, when the right shift is specified, the product is always sign-extended, regardless of the value of SXM.

The LT (load TREG0) instruction normally loads TREG0 to provide one operand (from the data bus), and the MPY (multiply) instruction provides the second operand (also from the data bus). A multiplication can also be performed with a short or long immediate operand by using the MPY instruction with an immediate operand. A product can be obtained every two cycles except when a long immediate operand is used.

Four multiply/accumulate instructions (MAC, MACD, MADD, and MADS) fully utilize the computational bandwidth of the multiplier, allowing both operands to be processed simultaneously.The data for these operations can be transferred to the multiplier each cycle via the program and data buses. This facilitates single-cycle multiply/accumulates when used with repeat (RPT and RPTZ) instructions. In these instructions, the coefficient addresses are generated by the PC, while the data addresses are generated by the ARAU. This allows the repeated instruction to sequentially access the values from the coefficient table and step through the data in any of the indirect addressing modes. The RPTZ instruction also clears the accumulator and the product register to initialize the multiply/accumulate operation. As an example, consider multiplying the row of one matrix times the column of a second matrix. For this example, consider 10 × 10 matrices, MTRX1 points to the beginning of the first matrix, INDX = 10, and AR(ARP) points to the beginning of the second matrix:

```
RPTZ   #9          ;For i = 0, i < 10, i++
MAC    MTRX1,*0+   ;PREG=DATA(MTRX1+i) x DATA[MTRX2 + (ixINDX)]
                   ;ACC += PREG.
APAC               ;ACC += PREG.
```

The MAC and MACD instructions obtain their coefficient pointer from a long immediate address and are, therefore, two-word instructions. The MADS and MADD instructions obtain their coefficient pointer from the BMAR and are, therefore, one-word instructions. The use of the BMAR as a source to the coefficient table enables one block of code to support multiple applications and makes it unnecessary to modify executable code to change the long immediate address. The MACD and MADD instructions also include a data move (DMOV) operation that, in conjunction with the fetch of the data multiplicand, writes the data value to the next-higher data address.

The MACD and MADD instructions, when repeated, support filter constructs (weighted running averages) so that as the sum-of-products is executed, the

sample data is shifted in memory to make room for the next sample and to throw away the oldest sample. Circular addressing with MAC and MADS instructions may also be used to support filter implementation.

For the example below, AR(ARP) points to the oldest of the samples. BMAR points to the coefficient table. In addition to initiating the repeat operation, the RPTZ instruction also clears the accumulator and the product register. In this example, the PC is stored in a temporary register while the repeated operation is executed. Next, the PC is loaded with the value stored in BMAR. The program bus is used to address the coefficients and, as the MADD is repeatedly executed, the PC increments to step through the coefficient table. The ARAU generates the address of the sample data. Indirect addressing with decrement steps the sample data, starting with the oldest data. As the data is fetched, it is also written to the next higher location in data memory. This operation aligns the data for the next execution of the filter by moving the oldest sample out past the end of the sample's array and making room for the new sample at the beginning of the sample array. The previous product (PREG) is added to the accumulator (ACC), while the two fetched values are multiplied and the product loaded into the PREG. Note that the DMOV portion of the MACD and MADD instructions will not function with external data memory addresses.

```
RPTZ   #9      ;ACC = PREG = 0. For I = 9 TO 0 Do
MADD   *-      ;SUM A(I) x X(I).  X(I+1) = X(I).
APAC           ;FINAL SUM.
```

The MPYU instruction performs an unsigned multiplication, which greatly facilitates extended-precision arithmetic operations. The unsigned contents of TREG0 are multiplied by the unsigned contents of the addressed data memory location, with the result placed in PREG. This allows operands of greater than 16 bits to be broken down into 16-bit words and processed separately to generate products of greater than 32 bits. The SQRA (square/add) and SQRS (square/subtract) instructions pass the same value to both inputs of the multiplier for squaring a data memory value.

After the multiplication of two 16-bit numbers, the 32-bit product is loaded into the 32-bit product register (PREG). The product from the PREG may be transferred to the ALU or to data memory via the SPH (store product high) and SPL (store product low).

3.6 System Control

System control on the 'C5x is provided by the program counter, hardware stack, PC-related hardware, external reset signal, interrupts (see Section 3.8), status registers, and repeat counters. The following subsections describe the function of each of these components in system control and pipeline operation.

3.6.1 Program Address Generation and Control

The 'C5x has a 16-bit program counter (PC) and an eight-deep hardware stack for PC storage. The program counter addresses internal and external program memory in fetching instructions. The stack is used during interrupts and subroutines.

The program counter addresses program memory, either on-chip or off-chip, via the program address bus (PAB). Through the PAB, an instruction is addressed in program memory and loaded into the instruction register (IREG). When the IREG is loaded, the PC is ready to start the next instruction fetch cycle.

The PC can be loaded in a number of ways. When code is sequentially executed, the PC is loaded with PC + 1. When a branch is executed, the PC is loaded with the long immediate value directly following the branch instruction. In the case of a subroutine call, the PC+2 is pushed onto the stack and then loaded with the long immediate value directly following the call instruction. The return instructions pop the stack back into the PC to return to the calling or interrupting sequence of code. In the case of a software trap or interrupt trap, the PC is loaded with the address of the appropriate trap vector. The contents of the accumulator may be loaded into the PC to implement computed GOTO operations. This can be accomplished with the BACC (branch to address in accumulator) or CALA (call subroutine at location specified by ACC) instructions.

The PAB bus can also address data stored in either program or data space. This makes it possible, in repeated instructions, to fetch a second operand in parallel with the data bus for two-operand operations. When repeated, the array addressed by the PAB is sequentially accessed via the incrementing of the PC. The block transfer instructions (BLDD, BLDP, and BLPD) use both buses so that, when repeated, the pipeline structure can be reading the next operand while writing the current one. The BLPD instruction loads the PC with either the long immediate address following the BLPD or with the contents of the block move address register (BMAR). The PAB bus is then used to fetch the source data from program space in this block move operation. The BLDP executes in the same way, except that the PAB bus is used for the destination operation. The BLDD instruction uses the PAB bus to address data space.

The TBLR and TBLW instructions operate much like the BLPD and BLDP instructions, respectively, except that the PC is loaded with the low 16 bits of the

accumulator instead of the BMAR or long immediate address. This facilitates calculated table look-up operations. The multiply/accumulate operations (MAC, MACD, MADD, and MADS) use the PAB bus to address their coefficient table. The MAC and MACD instructions load the PC with the long immediate address following the instruction. The MADD and MADS instructions load the PC with the contents of BMAR.

To start a new fetch cycle, the PC is loaded either with PC +1 or with a branch address (for instructions such as branches, calls, and interrupts). In the case of conditional branches where the branch is not taken, the PC is incremented once more beyond the location of the branch immediate address. In addition to the conditional branches, the 'C5x has a full complement of conditional calls, executes, and returns. These instructions execute according to the following conditions:

Operand	Condition	Description
EQ	ACC = 0	Accumulator equal to zero
NEQ	ACC ≠ 0	Accumulator not equal to zero
LT	ACC < 0	Accumulator less than zero
LEQ	ACC ≤ 0	Accumulator less than or equal to zero
GT	ACC > 0	Accumulator greater than zero
GEQ	ACC ≥ 0	Accumulator greater than or equal to zero
C	C = 1	Accumulator carry set to one
NC	C = 0	Accumulator carry set to zero
OV	OV = 1	Accumulator overflow detected
NOV	OV = 0	No accumulator overflow detected
BIO	$\overline{\text{BIO}}$ is low	$\overline{\text{BIO}}$ signal is low
TC	TC = 1	Test/control flag set to one
NTC	TC = 0	Test/control flag set to zero
UNC	none	Unconditional operation

Multiple conditions can be defined in the operands of the conditional instructions. If multiple conditions are defined, all conditions must be met. For example,

```
BCND BRANCH,LT,NOV ;If ACC <  0 and no overflow.
```

In this example, both conditions must be met (that is, OV = 0 and ACC < 0) for the branch to be taken.

The conditional branch is a two-word instruction. The conditions for the branch are not stable until the fourth cycle of the branch instruction pipeline execution, because the previous instruction must have completely executed for the accumulator's status bits to be accurate. Therefore, the pipeline controller stops the decode of instructions following the branch until the conditions are valid. If the conditions defined in the operands of the instruction are met, then the PC is

loaded with the second word and the core CPU starts refilling the pipeline with instructions at the branch address. Because the pipeline has been flushed, the branch instruction has an effective execution time of four cycles if the branch is taken. If, however, any of the conditions are not met, the pipeline controller allows the next instruction (already fetched) to be decoded. This means that if the branch is not taken, the effective execution time of the branch is two cycles.

The subroutine call can also be executed conditionally. The CC instruction operates like the BCND except that the PC pointing to the instruction following the CC is pushed onto the PC stack. This sets up the return (by RET) to pop the stack to return to the calling sequence. A subroutine or function can have multiple return paths based upon the data being processed. Using conditional returns (RETC) avoids the need for conditionally branching around the return. For example,

```
        CC      OVER_FLOW,OV  ;If overflow,then execute the
        .                     ;overflow-handling routine.
        .
        .
OVER_FLOW                     ;Overflow-handling routine.
        .
        .
        .
        RETC    GEQ           ;If ACC >= 0, then return.
        .
        .
        .
        RET                   ;Return.
```

In the example, an overflow-handling subroutine is called if the main algorithm causes an overflow condition. During the subroutine, the ACC is checked and, if it is positive, the subroutine returns to the calling sequence. If not, additional processing is necessary before the return. Note that RETC, like RET, is a single-word instruction. However, because of the potential PC discontinuity, it still operates with the same effective execution time as BCND and CC.

To avoid flushing the pipeline and causing extra cycles, the 'C5x has a full set of delayed branches, calls, and returns. In the delayed operation of branches, calls, or returns, the two-instruction words following the delayed instruction are executed while the instructions at and following the branch address are being fetched—therefore, giving an effective two-cycle branch instead of flushing the pipeline. If the instruction following the delayed branch is a two-word instruction, only that instruction is executed before the branch is taken. For example,

```
OPL     #030h,PMST
BCND    NEW_ADRS,EQ
```

or

```
BCNDD   NEW_ADRS,EQ
OPL     #030h,PMST.
```

The first code segment takes six cycles to execute (two for the OPL and four for the BCND). The second code segment takes four cycles because the two dead cycles following the BCNDD are filled with the OPL instruction. Note that the condition tested on the branch is not affected by the OPL instruction, thus, allowing it to be executed after the branch.

In cases where the conditional branch is used to skip over one or two words of code, the branch can be replaced with the conditional execute instruction. For example,

```
        BCND SUM,NC
        ADD  ONE
SUM     APAC
```

or

```
        XC   1,C
        ADD  ONE
        APAC
```

The first code segment takes six cycles. The second code segment takes three cycles. If the condition is met in the second code segment, the ADD is executed. If the condition is not met, then a NOP is forced in the instruction register over the ADD. Note that the condition must be stable one full cycle before the XC instruction is executed. This is to assure that the decision is made on the condition before the instruction following the XC is decoded (auxiliary register updates occur during the decode phase of an instruction, so the instruction must be stopped before the decode to make sure it is not executed). The user should avoid affecting the XC test conditions one instruction word before the XC. Without interrupts, this instruction will have no effect on the XC. However, with interrupts, an interrupt can trap between the instruction and the XC so that the condition is affected prior to the XC execution. The following examples show this cycle dependency:

```
LACL    #0              ;ACC = 0.
ADD     TEMP1           ;ACC = TEMP1.
XC      2,EQ            ;If ACC == 0,
SPLK    #0EEEEh,TEMP2   ;Then TEMP2 = EEEE.
```

or

```
LACL    #0              ;ACC = 0.
ADD     #01234h         ;ACC = 00001234.
XC      2,EQ            ;If ACC == 0,
SPLK    #0EEEEh,TEMP2   ;Then TEMP2 is unmodified.
```

In the first code segment, TEMP2 = EEEE. The NEQ status, caused by the ADD instruction, is not established at the time the decision is made by the XC instruction. Therefore, the previous condition of EQ, caused by the ZAC instruction, determines the conditional execute. Since this condition is met, TEMP2 is loaded by the SPLK instruction. Note that interrupts can trap before the XC and after the ADD so that the SPLK will not execute. In the second code

segment, TEMP2 is not set to EEEE. The NEQ status, caused by the ADD instruction, is established one full cycle before the XC execution phase because the long immediate value (#01234h) used in the ADD caused it to be a two-cycle instruction. Since the condition is not met, a NOP is forced over both words of the two-word SPLK instruction, and, therefore, TEMP2 is not affected. Note that interrupts have no effect on this instruction sequence.

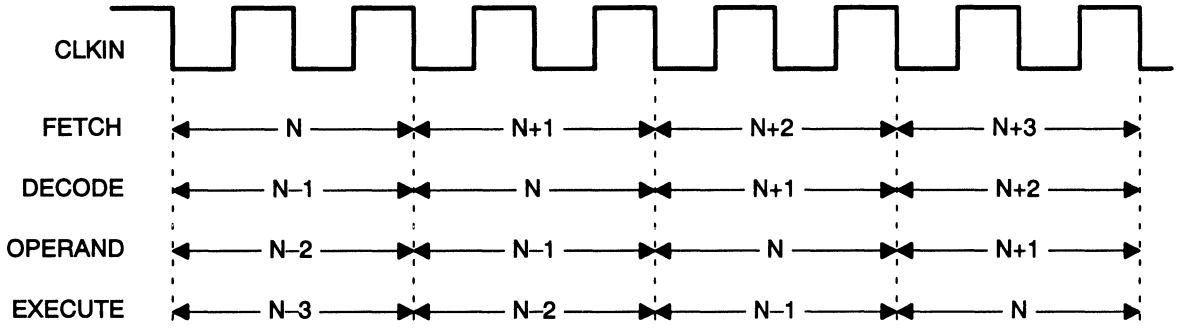
The 'C5x also has a feature that allows the execution of a single instruction N + 1 times where N is the value loaded in a 16-bit repeat counter (RPTC). If the repeat feature is used, the instruction is executed and the RPTC is decremented until the RPTC goes to zero. This feature is useful with many instructions, such as NORM (normalize contents of accumulator), MACD (multiply and accumulate with data move), and SUBC (conditional subtract). As instructions repeat, the program address and data buses are freed to fetch a second operand in parallel with the data address and data buses. This allows instructions such as MACD and BLPD to effectively execute in a single cycle when they repeat. See Section 7.6, *Single Instruction Repeat Loops*, for details on these instructions.

The stack is 16 bits wide and eight levels deep. The PC stack is accessible through the use of the PUSH and POP instructions. Whenever the contents of the PC are pushed onto the top of the stack, the previous contents of each level are pushed down, and the bottom (eighth) location of the stack is lost. Therefore, data will be lost if more than eight successive pushes occur before a pop. The reverse happens on pop operations. Any pop after seven sequential pops yields the value at the bottom stack level, and all of the stack levels then contain the same value. Two additional instructions, PSHD and POPD, push a data memory value onto the stack or pop a value from the stack to data memory. These instructions allow a stack to be built in data memory for the nesting of subroutines/interrupts beyond eight levels. See Section 7.3, *Software Stack*, for details on software stack.

3.6.2 Pipeline Operation

Instruction pipelining consists of the sequence of bus operations that occur during instruction execution. In the operation of the pipeline, the instruction fetch, decode, operand fetch, and execute operations are independent, which allows overall instruction executions to overlap. Thus, during any given cycle, one to four different instructions can be active, each at a different stage of completion, resulting in a four-deep pipeline. Figure 3–14 shows the operation of the four-level pipeline for single-word single-cycle instructions executing with no wait states. The pipeline is essentially invisible to the user except in some cases, such as auxiliary register updates, memory-mapped accesses of the CPU registers, the NORM instruction, and memory configuration commands.

Figure 3–14. Four-Level Pipeline Operation



ARAU updates of auxiliary registers execute during the decode (second phase) of the pipeline. This allows the address to be generated before the operand fetch phase. However, memory-mapped accesses (e.g., SAMM, LMMR, SACL, or SPLK) of these registers happen on the execute phase of the pipeline. This means that the next two instructions after a memory-mapped load of the auxiliary register should not use this auxiliary register. In addition, modifications to the memory-mapped registers INDX and ARCR also occur in the execution phase of the pipeline. Therefore, any auxiliary register updates using the INDX register or auxiliary register compares using the ARCR register must occur at least two cycles after a load of these registers. The following code examples illustrate the effects of a memory-mapped write to an auxiliary register:

```
EXAM1  LAR AR2,#067h     ;AR2 = 67.
       LACC  #064h       ;ACC = 00000064.
       SAMM  AR2         ;This update is overridden by *– up–
                         ;dates on the next two instructions.
       LACC  *–          ;AR2 = 66.
       ADD   *–          ;AR2 = 65.
```

or

```
EXAM2  LAR AR2,#067h     ;AR2 = 67.
       LACC  #064h       ;ACC = 00000064.
       SAMM  AR2         ;LACC *– update happens before
                         ;SAMM write.
       LACC  *–          ;AR2 = 66.
       NOP               ;AR2 = 64 {SAMM write to AR2 happens
                         ;in parallel with the NOP.
       ADD   *–          ;AR2 = 63.
```

or

```
EXAM3  LAR    AR2,#067h ;AR2 = 67.
       LACC   #064h     ;ACC = 00000064.
       SAMM   AR2       ;AR2 = 64.
       NOP              ;Pipeline protection.
       NOP              ;Pipeline protection.
       LACC   *-        ;AR2 = 63.
       ADD    *-        ;AR2 = 62.
```

In EXAM1, the decode phase of the ADD instruction is on the same cycle as the execute (write) phase of the SAMM instruction. Both of these instructions are trying to load AR2. The ADD *– update does load AR2, while the SAMM execution is voided. In EXAM2, a NOP is strategically placed to avoid the conflict between the ADD *– update of the AR2 and the SAMM write to AR2. In this code's sequence:

AR2 = 67 → 66 → 64 → 63

Note that the LACC address is based on the value in AR2 before the SAMM write to AR2. In EXAM3, the SAMM write to AR2 is completed before either the LACC or the ADD have updated AR2. Any two instruction words that do not update AR2 can be used in place of the two NOP instructions. This could be two one-word instructions or one two-word instruction. The results obtained by EXAM1 and EXAM2 code examples may be different if the code is interruptible. The user should avoid writing code similar to EXAM1 and EXAM2.

The pipeline effect described above requires writes to memory-mapped registers to allow for a latency between the write and an access of that register. These registers can be accessed by 'C5x instructions in the decode and operand fetch phases of the pipeline. Table 3–6 outlines the latency required between an instruction that writes the register via its memory-mapped address, and the access of that register by subsequent instructions. Note that all direct accesses to the registers that do not use memory-mapped addressing (such as all 'C25-compatible instructions, like LAR, LT, etc) are pipelined-protected and, hence, do not require any latency.

Table 3–6. Latencies Required

Name	Description	Words	Affects
GREG	Global memory allocation register	1	Next 1 word uses previous map
PMST	Processor mode status register	2	Next 2 words use previous map
TREG1	Dynamic shift count	1	Next 1 word uses old shift count
TREG2	Dynamic bit address	1	Next 1 word uses old bit address
ARx	Auxiliary registers 0–7	2	Next word uses previous value; second word update gets over written
INDX	Index register	2	Next 2 words use previous value
ARCR	Auxiliary register compare register	2	Next 2 words use previous value
CBSR	Circular buffer start registers 1 and 2	2	Next 2 words use previous value
CBER	Circular buffer end registers 1 and 2	2	Next 2 words cannot be end of buffer
CBCR	Circular buffer control register	2	Next 2 words cannot be end of buffer
BMAR	Block move address register	1	Next 1 word uses previous value
PDWSR	Program/data S/W wait state register	1	Next 1 word uses previous count
IOWSR	I/O space S/W wait state register	1	Next 1 word uses previous count
CWSR	S/W wait state control register	1	Next 1 word uses previous modes
CNF	Configuration bit in ST1 register	2	Next 2 words use previous map

The NORM instruction affects AR(ARP) during its execute phase of the pipeline. The same pipeline management, as described above, works in this case. The assembler can detect an auxiliary register update or store (SAR) directly after a NORM instruction and insert NOP instructions automatically to maintain source-code compatibility with the 'C25 (–p option).

The 'C5x core CPU supports the reconfiguration of memory segments, both internal and external to the device. The reconfiguration operations happen during the execute phase of the pipeline. Therefore, before an instruction uses the new configuration, at least two instruction words should follow the instruction that reconfigures memory. In the following example, assume AR(ARP) = 0200h and RAMB0(0) = 1.

```
CLRC   CNF        ;Map RAM B0 to data space.
LACC   #01234h    ;ACC = 00001234.
ADD    *          ;ACC = 00001235.
```

Notice the use of the LACC #01234h to fill the two-word requirement. Because a long immediate operand is used, this is a two-word instruction and, therefore, meets the requirement. This also applies to memory configurations controlled by the PMST register.

If main code is running in the B0 block (CNF=1) and an ISR not in B0 changes CNF to 0, a RETE will not restore CNF in time to fetch the next instruction from the B0 block. Thus, in the ISR, the CNF bit should be set to 1 at least two words before the RETE.

3.6.3 Status and Control Registers

There are four key status and control registers for the 'C5x core. ST0 and ST1 contain the status of various conditions and modes compatible with the 'C25, while PMST and CBCR contain extra status and control information for control of the enhanced features of the 'C5x core. These registers can be stored into data memory and loaded from data memory, thus allowing the status of the machine to be saved and restored for subroutines. ST0, ST1, and PMST each have an associated one-deep stack for automatic context-saving when an interrupt trap is taken. The stack is automatically popped upon a return from interrupt (RETI or RETE). Note that the XF bit in ST1 is not saved on the one-deep stack or restored from that stack on an automatic context save. This feature allows the XF pin to be toggled in an interrupt service routine while still allowing automatic context saves.

The PMST and CBCR registers reside in the memory-mapped register space in page zero of data memory space. Therefore, they can be acted upon directly by the CALU and the PLU. They can be saved in the same way as any other data memory location. Note that the CALU and the PLU operations change the bits of these status registers during the execute phase of the pipeline. The next two instruction words, following an update of these status registers, may not be affected by the reconfiguration caused by the status update, as shown in Table 3–6.

The LST instruction writes to ST0 and ST1, and the SST instruction reads from them, except that the INTM bit is not affected by the LST instruction. Unlike the PMST and CBCR registers, the ST0 and ST1 registers do not reside in the memory map and, therefore, cannot be handled by using the PLU instructions. The individual bits of these registers can be set or cleared with the SETC and CLRC instructions. For example, the sign-extension mode is set with SETC SXM or cleared with CLRC SXM.

Figure 3–15 shows the organization of the four status registers, indicating all status bits contained in each. Several bits in the status registers are reserved and read as logic ones. Table 3–7 defines all the status/control bits.

Figure 3–15. Status and Control Register Organization

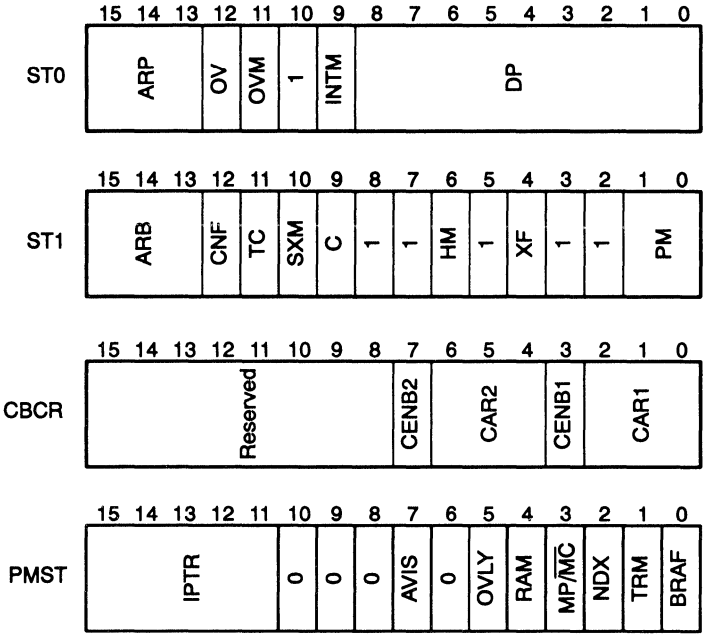


Table 3–7. Status Register Field Definitions

Field	Function
ARB	Auxiliary Register Pointer Buffer. Whenever the ARP is loaded, the old ARP value is copied to the ARB except during an LST instruction. When the ARB is loaded via an LST #1 instruction, the same value is also copied to the ARP. This is useful when restoring context (when not using the automatic context save) in a subroutine that modifies the current ARP.
ARP	Auxiliary Register Pointer. This three-bit field selects the AR to be used in indirect addressing. When the ARP is loaded, the old ARP value is copied to the ARB register. ARP may be modified by memory-reference instructions when indirect addressing is used, and by the MAR and LST instructions. The ARP is also loaded with the same value as ARB when an LST #1 instruction is executed.
AVIS	Address VISibility Mode. This mode allows the internal program address to appear at the pins of the device so that the internal program address can be traced and the interrupt vector can be decoded in conjunction with $\overline{IACK}$ when the interrupt vectors reside in on-chip memory. The internal program address is driven to the pins when AVIS = 0. The address lines do not change with internal program when AVIS = 1. Note that the control lines and data lines are not effected when AVIS = 0 and the address bus is driven with the last address on the bus. The AVIS bit is set to zero at reset.
BRAF	Block Repeat Active Flag. This bit indicates whether block repeat is currently active. Writing a zero to this bit deactivates block repeat. BRAF is set to zero upon reset.
C	Carry Bit. This bit is set to 1 if the result of an addition generates a carry, or is reset to 0 if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition or is set after a subtraction, unless the instruction is ADD or SUB with a 16-bit shift. In these cases, the ADD can only set and the SUB only reset the carry bit, but they cannot affect it otherwise. The single-bit shift and rotate instructions, as well as the SETC, CLRC, and LST #1 instructions also affect this bit. C is set to 1 on a reset.

Table 3–7. Status Register Field Definitions (Continued)

Field	Function
CAR1	Circular Buffer 1 Auxiliary Register. These three bits identify which auxiliary register is assigned to circular buffer 1.
CAR2	Circular Buffer 2 Auxiliary Register. These three bits identify which auxiliary register is assigned to circular buffer 2.
CENB1	Circular Buffer 1 Enable. This bit, when set to 1, enables circular buffer 1. When CENB1 is set to 0, circular buffer 1 is disabled. CENB1 is set to zero upon reset.
CENB2	Circular Buffer 2 Enable. This bit, when set to 1, enables circular buffer 2. When CENB2 is set to 0, circular buffer 2 is disabled. CENB2 is set to zero upon reset.
CNF	On-chip RAM Configuration Control Bit. If this bit is set to 0, the reconfigurable-data dual-access RAM blocks are mapped to data space; otherwise, they are mapped to program space. The CNF may be modified by the SETC CNF, CLRC CNF, and LST #1 instructions. $\overline{RS}$ sets the CNF to 0.
DP	Data Memory Page Pointer. The 9-bit DP register is concatenated with the 7 LSBs of an instruction word to form a direct memory address of 16 bits. DP may be modified by the LST and LDP instructions.
HM	Hold Mode Bit. When HM = 1, the processor halts internal execution when acknowledging an active $\overline{HOLD}$. When HM = 0, the processor may continue execution out of internal program memory but puts its external interface in a high-impedance state. This bit is set to 1 by reset.
INTM	Interrupt Mode Bit. When this bit is set to 0, all unmasked interrupts are enabled. When it is set to 1, all maskable interrupts are disabled. INTM is set and is reset by the SETC INTM and CLRC INTM instructions. $\overline{RS}$ and $\overline{IACK}$ also set INTM. INTM has no effect on the unmaskable $\overline{RS}$ and $\overline{NMI}$ interrupts. Note that INTM is unaffected by the LST instruction. This bit is set to 1 by reset. It is also set to 1 when a maskable interrupt trap is taken. It is reset to 0 when a RETE (return from interrupt with interrupt enable) is executed.
IPTR	Interrupt Vector Pointer. These five bits point to the 2K page where the interrupt vectors reside. This allows you to remap the interrupt vectors to RAM for boot-loaded operations. At reset, these bits are all set to zero. Therefore, the reset vector always resides at zero in the program memory space.
MP/$\overline{MC}$	Microprocessor/Microcomputer Bit. When this bit is set to zero, the on-chip ROM is enabled. When it is set to one, the on-chip ROM is not addressable. This bit is set to the value corresponding to the logic level on the MP/$\overline{MC}$ pin at reset. The level on the MP/$\overline{MC}$ pin is sampled at device reset only and can have no effect until the next reset.
NDX	Enable Extra Index Register. This bit configures indexed indirect addressing and auxiliary address register compare to operate either in a 'C2x-compatible mode (NDX = 0) or in a 'C5x-enhanced mode (NDX = 1). When NDX = 0, any 'C2x-compatible instruction that modifies or loads AR0, also modifies/loads the INDX and ARCR registers in addition to AR0. This is because the 'C2x devices use AR0 for indexing and AR compare operations. When NDX = 1, INDX and ARCR are not affected by any 'C2x-compatible instruction. NDX = 0 at reset.
OV	Overflow Flag Bit. As a latched overflow signal, OV is set to 1 when overflow occurs in the ALU. Once an overflow occurs, the OV remains set until a reset, BCND(D) on OV/NOV, or LST instruction clears OV.
OVLY	RAM Overlay Bit. This bit enables on-chip single-access program RAM cells to be mapped into data space. If OVLY is set to one, the block of memory is mapped into data space. If it is set to 0, the memory block is not addressable in data space. See Table 3–8 for the mappings of specific 'C5x devices. This bit is set to zero at reset.

Table 3–7. Status Register Field Definitions (Concluded)

Field	Function
OVM	Overflow Mode Bit. When OVM is set to 0, overflowed results overflow normally in the accumulator. When set to 1, the accumulator is set to either its most positive or most negative value upon encountering an overflow. The SETC and CLRC instructions set and reset this bit, respectively. LST may also be used to modify the OVM.
PM	Product Shift Mode. If these two bits are 00, the multiplier's 32-bit product is not shifted when transferred to the ALU. If PM = 01, the PREG output is left-shifted one place when transferred to the ALU, with the LSB zero-filled. If PM = 10, the PREG output is left-shifted by four bits when transferred to the ALU, with the LSBs zero-filled. PM = 11 produces a right shift of six bits, sign-extended. Note that the PREG contents remain unchanged. The shift also takes place when the contents of the PREG are stored to the data memory. PM is loaded by the SPM and LST #1 instructions. The PM bits are cleared by $\overline{RS}$.
RAM	Program RAM Enable. This bit enables mapping of on-chip single-access RAM blocks into program space. RAM set to 1 maps the memory block in program space. RAM set to 0 removes the memory block from the program space. See Table 3–8 for the mappings of specific 'C5x devices. This bit is set to zero at reset.
SXM	Sign-Extension Mode Bit. SXM = 1 produces sign extension on data as it is passed into the accumulator through the scaling shifter. SXM = 0 suppresses sign extension. SXM does not affect the definitions of certain instructions; e.g., the ADDS instruction suppresses sign extension, regardless of SXM. This bit is set by the SETC SXM, reset by the CLRC SXM instructions, and may be loaded by the LST #1. SXM is set to 1 by reset.
TC	Test/Control Flag Bit. The TC bit is affected by the BIT, BITT, CMPR, LST #1, NORM, CPL, XPL, OPL, and APL instructions. The TC bit is set to a 1 if (1) a bit tested by BIT or BITT is a 1, (2) a compare condition tested by CMPR exists between ARCR and another AR pointed to by ARP, (3) the exclusive-OR function of the two MSBs of the accumulator is true when tested by a NORM instruction, (4) the long immediate value is equal to the data value on the CPL instruction, or (5) the result of the logical function (XPL, OPL or APL) is zero. The TC bit can influence the execution of the conditional branch, call, and return instructions.
TRM	Enable Multiple TREGs. This bit sets the 'C5x to operate in either 'C2x-compatible mode (TRM = 0) or 'C5x-enhanced mode (TRM=1) in conjunction with the use of the TREG0, TREG1, and TREG2 registers. This bit affects the operation of all 'C2x-compatible instructions that modify TREGO. The 'C2x CPU uses TREGx as a shift count for the prescaling shifter and as a bit address in the BITT instruction. When TRM=0, all 'C2x-compatible instructions write to all three of the 'C5x TREGs to maintain source compatibility with the 'C2x devices. When TRM = 1, the LT instructions affect only TREG0. TRM = 0 upon reset.
XF	XF Pin Status Bit. This bit indicates the state of the XF pin, a general-purpose output pin. XF is set by the SETC XF and reset by the CLRC XF instructions. XF is set to 1 by reset. This bit is not saved or restored on an automatic context save during interrupt service routines.

Table 3–8. On-Chip Single-Access RAM Configuration Control

OVLY	RAM	On-Chip SARAM Configuration
0	0	Disabled
0	1	Mapped into program space
1	0	Mapped into data space
1	1	Mapped into both program and data spaces

3.6.4 Repeat Counter

RPTC is a 16-bit repeat counter, which, when loaded with a number N, causes the next single instruction to be executed N + 1 times. The RPTC register is loaded by either the RPT or the RPTZ instruction. This results in a maximum of 65,536 executions of a given instruction. RPTC is cleared by reset. The RPTZ instruction clears both ACC and PREG before the next instruction starts repeating. Once a repeat instruction (RPT or RPTZ) is decoded, all interrupts including NMI (except reset) are masked until the completion of the repeat loop. However, the device responds to the HOLD signal while executing an RPT/RPTZ loop. The RPTC register resides in the CPU's memory-mapped register space; however, you should avoid writing to this register.

The repeat function can be used with instructions such as multiply/accumulates (MAC and MACD), block moves (BLDD and BLPD), I/O transfers (IN/OUT), and table read/writes (TBLR/TBLW). These instructions, although normally multicycle, are pipelined when the repeat feature is used, and they effectively become single-cycle instructions. For example, the table read instruction may take three or more cycles to execute, but when the instruction is repeated, a table location can be read every cycle. Note that not all instructions can be repeated. Table 3–9 through Table 3–11 list all 'C5x instructions, according to their repeatability.

Table 3–9. Repeatable Instructions

Repeatable Instructions		Description
ADCB		Add ACCB to ACC with carry
ADD	dma,shft	Add to ACC direct addressed with shift
ADD	*,shft	Add to ACC indirect addressed with shift
ADDB		Add ACCB to ACC
ADDC		Add to ACC direct/indirect with carry
ADDS		Add to low ACC direct/indirect with sign suppressed
ADDT		Add to ACC direct/indirect with shift specified by TREG1
APAC		Add PREG to ACC
APL		AND DBMR to direct/indirect addressed
BLDD		Block move from data memory to data memory
BLDP		Block move from data memory to program memory
BLPD		Block move from program memory to data memory
BSAR		Barrel-shift ACC right
DMOV		Move direct/indirect addressed data one location up in memory
IN		Read from I/O space
LMMR		Load memory-mapped register
LTA		Load TREG0 direct/indirect and add PREG to ACC
LTD		Load TREG0 direct/indirect with data move and add PREG to ACC

Table 3–9. Repeatable Instructions (Continued)

Repeatable Instructions	Description
LTS	Load TREG0 direct/indirect and subtract PREG
MAC	Add PREG to ACC and multiply immediate addressed by direct/indirect
MACD	Add PREG to ACC and multiply immediate addressed by direct/indirect with data move
MADD	Add PREG to ACC and multiply BMAR addressed by direct/indirect with data move
MADS	Add PREG to ACC and multiply BMAR addressed by direct/indirect
MPYA	Add PREG to ACC and multiply TREG0 by direct/indirect
MPYS	Subtract PREG from ACC and multiply TREG0 by direct/indirect
MAR	Modify AR
NOP	No operation
NORM	Normalize ACC
OPL	OR DBMR to direct/indirect addressed
OUT	Write to I/O space
POP	Pop the PC stack to low ACC
POPD	Pop the PC stack to direct/indirect addressed
PSHD	Push direct/indirect addressed to the PC stack
PUSH	Push low ACC to the PC stack
ROL	Rotate ACC left once
ROLB	Rotate combined ACC and ACCB left once
ROR	Rotate ACC right once
RORB	Rotate combined ACC and ACCB right once
SACH	Store high ACC with shift
SACL	Store low ACC with shift
SAMM	Store low ACC direct/indirect to data page 0
SAR AR,*	Store AR indirect addressed
SATH	Shift ACC right 0 or 16 bits as specified by TREG1(4)
SATL	Shift ACC right 0 to 15 bits as specified by TREG1(0–3)
SBB	Subtract ACCB from ACC
SBBB	Subtract ACCB from ACC with borrow
SFL	Shift ACC left once
SFLB	Shift combined ACC and ACCB left once
SFR	Shift ACC right once
SFRB	Shift combined ACC and ACCB right once
SMMR	Store memory-mapped register
SPAC	Subtract PREG from ACC
SPH	Store high PREG to direct/indirect addressed
SPL	Store low PREG to direct/indirect addressed

Table 3–9. Repeatable Instructions (Concluded)

Repeatable Instructions		Description
SQRA		Add PREG to ACC and square direct/indirect addressed
SQRS		Subtract PREG from ACC and square direct/indirect addressed
SST		Store status registers
SUB	dma,shft	Subtract from ACC direct addressed with shift
SUB	*,shft	Subtract from ACC indirect addressed with shift
SUBB		Subtract from ACC direct/indirect with borrow
SUBC		Conditional subtract from ACC direct/indirect
SUBS		Subtract from low ACC direct/indirect with sign suppressed
SUBT		Subtract from ACC direct/indirect with shift specified by TREG1
TBLR		Read from program space to data space
TBLW		Write from data space to program space
XPL		XOR DBMR to direct/indirect addressed

Table 3–10. Instructions Not Meaningful to Repeat

Instructions Not Meaningful to Repeat		Description
ABS		Absolute value of ACC
AND		AND to low ACC direct/indirect
ANDB		AND ACCB to ACC
BIT		Test bit in data word
BITT		Test bit (specified by TREG2) in data word
CLRC		Clear status bit
CMPL		Complement ACC
CMPR		Compare AR(ARP) to ARCR
CPL		Compare DBMR to direct/indirect addressed
CRGT		Compare ACC to ACCB and match larger value
CRLT		Compare ACC to ACCB and match smaller value
EXAR		Exchange ACC with ACCB
LACB		Load ACC with ACCB
LACC	dma,shft	Load ACC direct addressed with shift
LACC	*,shft	Load ACC indirect addressed with shift
LACL		Load low ACC direct/indirect and zero high ACC
LACT		Load ACC direct/indirect with shift specified by TREG1
LAMM		Load low ACC direct/indirect from data page 0
LAR	dma,AR	Load AR direct addressed
LAR	*,AR	Load AR indirect addressed
LDP	dma	Load DP direct addressed
LDP	*	Load DP indirect addressed

Table 3–10. Instructions Not Meaningful to Repeat (Continued)

Instructions Not Meaningful to Repeat	Description
LPH	Load high PREG with direct/indirect addressed
LST	Load status registers
LT	Load TREG0 with direct/indirect addressed
LTP	Load TREG0 direct/indirect and load ACC with PREG
MPY	Multiply TREG0 by direct/indirect
MPYU	Multiply TREG0 by direct/indirect unsigned
NEG	Negate ACC
OR	OR to low ACC direct/indirect
ORB	OR ACCB to ACC
PAC	Load ACC with PREG
SACB	Store ACC in ACCB
SAR AR,dma	Store AR direct addressed
SETC	Set status bit
SPM	Set PREG shift mode
XOR	XOR to low ACC direct/indirect
XORB	XOR ACCB to ACC
ZALR	Zero low ACC, load high ACC with rounding
ZAP	Zero ACC and PREG
ZPR	Zero PREG

Table 3–11. Nonrepeatable Instructions

Nonrepeatable Instructions	Description
ADD *#k*	Add to ACC short immediate
ADD *#lk,shift*	Add to ACC long immediate with shift
ADRK	Add to AR short immediate
AND *#lk,shft*	AND to ACC long immediate with shift
APL *#lk*	AND long immediate to direct/indirect addressed
B[D]	Branch [delayed] unconditionally
BACC[D]	Branch [delayed] to address specified in low ACC
BANZ[D]	Branch [delayed] on AR(ARP) not zero
BCND[D]	Branch [delayed] conditionally
CALA[D]	Call [delayed] to address specified in low ACC
CALL[D]	Call [delayed] subroutine
CC[D]	Call [delayed] subroutine conditionally
CPL *#lk*	Compare long immediate to direct/indirect addressed
IDLE	Idle CPU

Table 3–11 .Nonrepeatable Instructions (Continued)

Nonrepeatible Instructions		Description
IDLE2		Idle until interrupt — low power mode
INTR		Soft interrupt
LACC	*#lk,shft*	Load ACC long immediate
LACL	*#k*	Load ACC short immediate
LAR	*#lk*	Load AR with long immediate
LDP	*#k*	Load DP short immediate
NMI		Non-maskable interrupt
OPL	*#lk*	OR long immediate to direct/indirect addressed
OR	*#lk,shft*	OR to ACC long immediate with shift
RCND[D]		Return [delayed] from subroutine conditionally
RET		Return from subroutine
RETE		Return from interrupt service routine with automatic global enable
RETI		Return from interrupt service routine
RPT		Repeat next instruction N + 1 times
RPTB		Repeat block
RPTZ		Zero ACC and PREG and repeat next instruction N + 1 times
SBRK		Subtract from AR short immediate
SPLK	*#lk*	Store long immediate to direct/indirect addressed
SUB	*#k*	Subtract from ACC short immediate
SUB	*#lk,shft*	Subtract from ACC long immediate with shift
TRAP		Software interrupt
XC		Execute next instruction conditionally
XOR	*#lk,shft*	XOR to ACC long immediate with shift
XPL	*#lk*	XOR long immediate to direct/indirect addressed

3.6.5 Block Repeat

The block repeat feature provides zero-overhead looping for implementation of FOR and DO loops. The function is controlled by three registers (PASR, PAER, and BRCR) and the BRAF bit in the PMST register. The block repeat counter register (BRCR) is loaded with a loop count of 0 to 65,535. Then, execution of the RPTB (repeat block) instruction loads the program address start register (PASR) with the address of the instruction following the RPTB instruction and loads the program address end register (PAER) with its long immediate operand. The long immediate operand is the address of the instruction following the last instruction in the loop minus one. Note that the repeat block must contain at least three instruction words. Execution of the RPTB instruction automatically sets active the BRAF bit. With each PC update, the PAER

is compared to the PC. If they are equal, the BRCR contents are compared to zero. If the BRCR is greater than zero, it is decremented, and the PASR is loaded into the PC, thus starting the loop over. If not, the BRAF bit is set low, and the processor resumes execution past the end of the code's loop. For example,

```
    SPLK   #010h,BRCR.    ;Set loop count to 16.
    RPTB   END_LOOP-1     ;For I = BRCR; I >=0; I--.
*
    ZAP                   ;ACC = PREG = 0.
    SQRA   *,AR2          ;PREG = X2.
    SPL    SQRX           ;Save X2.
    MPY    *              ;PREG = b x X.
    LTA    SQRX           ;ACC = bX.  TREG = X2.
    MPY    *              ;PREG = aX2.
    APAC                  ;ACC = aX2 + bX.
    ADD    *,0,AR3        ;ACC = aX2 + bX + c = Y.
    SACL   *,0,AR1        ;Save Y.
    CRGT                  ;Save MAX.
END_LOOP
```

The example implements 16 executions of $Y = aX^2 + bX + c$ and saves the maximum value in ACCB. Note that the initialization of the auxiliary registers is not shown in the coded example. PAER is loaded with the address of the last word in the code segment. The label END_LOOP is placed after the last instruction, and the RPTB instruction long immediate is defined as END_LOOP–1 in case the last word in the loop is a two-word instruction.

There is only one set of block repeat registers, so multiple block repeats cannot be nested without saving the context of the outside block or using BANZD. The simplest method of executing nested loops is to use the RPTB for only the innermost loop and using BANZD for all the outer loops. This is still a valuable cycle-saving operation because the innermost loop is repeated significantly more times than the outer loops. Block repeats can be nested by storing the context of the outer loop before initiating the inner loop, then restoring the outer loop's context after completing the inner loop. The context save and restore are shown in the following example:

```
    SMMR   BRCR,TEMP1        ;Save block repeat counter
    SMMR   PASR,TEMP2        ;Save block start address
    SMMR   PAER,TEMP3        ;Save block end address

    SPLK   #NUM_LOOP,BRCR    ;Set inner loop count
    RPTB   END_INNER         ;For I = 0; I<=BRCR; I++
      .
      .
      .
END_INNER
    OPL    #1,PMST           ;Set BRAF to continue outer loop
    LMMR   BRCR,TEMP1        ;Restore block repeat counter
    LMMR   PASR,TEMP2        ;Restore block start address
    LMMR   PAER,TEMP3        ;Restore block end address
```

In this example, the context save and restore operations take 14 cycles. Note that repeated single and BANZ/BANZD loops can also be inside a block repeat. The repeated code can include subroutine calls. Upon returning, the block repeat resumes. Repeated blocks can be interrupted. When an enabled interrupt occurs during a repeated block of code, the CALU traps to the interrupt and, when the ISR returns, the block repeat resumes.

Be extremely careful when interrupting block repeats. If the interrupt service routine uses block repeats, check whether a block repeat has been interrupted and, if so, save the context of the block repeat as shown in the previous example. Smaller external loops can be implemented with the BANZD-looping method that takes two extra cycles per loop (that is, if the loop count is less than 8, it may be more efficient to use the BANZD technique). Single-cycle instructions can be repeated within a block repeat by using the RPT or RPTZ instructions.

While a block is being repeated, the block repeat active flag (BRAF) of the PMST register is set to a one. This flag is set by the execution of the RPTB instruction and is reset when the PC = PAER and BRCR = 0. This flag can be cleared and/or reset via the PMST register. WHILE loops can be implemented with the RPTB instruction and a conditional reset of the BRAF bit. The following code example clears BRAF so that the processor will drop out of the code loop and continue to sequentially access instructions past the end of the loop if an overflow occurs:

```
XC     2,OV           ;If overflow,
APL    #0FFFEh,PMST   ;then turn off block repeat.
```

The equivalent of a WHILE loop can be implemented by setting the BRAF bit to zero if the exit condition is met. If this is done, the program completes the current pass through the loop but does not go back to the top. To exit, the bit must be reset at least four instruction words before the end of the loop. You can exit block repeat loops and return to them without stopping and restarting the loop. Branches, calls, and interrupts do not necessarily affect the loop. When program control is returned to the loop, loop execution is resumed. The following example illustrates the block repeat with a small loop of code that executes a series of tasks. The tasks are stored in a table addressed by TEMP0F. The number of tasks to be executed is defined at NUM_TASKS.

```
   BLKP   NUM_TASKS,BRCR       ;Set loop count.
   SPLK   #(TASKS-1),TEMP0F    ;TEMP0F points to list of tasks.
   RPTB   ENDCALL-1            ;For I = 0, I <= NUM_TASKS; I++.
TASK_HANDLER
   LACC   TEMP0F               ;ACC points to task table.
   ADD    #1                   ;Increment pointer to next task.
   SACL   TEMP0F               ;Save for next pass of loop.
   TBLR   TEMP0E               ;Get task address.
   LACC   TEMP0E               ;ACC = task address.
   CALA                        ;Call task.
ENDCALL
```

In the setup for the example, the block repeat counter (BRCR) is loaded with the number of tasks to be executed, minus 1. Next, the address of the task table is loaded into a temporary register. The block repeat is started with the execution of the RPTB instruction. The PASR register is loaded with the address of the LACC TEMP0F instruction. The PAER register is loaded with the address of the last word of the table. Notice that the label marking the end of the loop is placed after the last instruction, then the PAER is loaded with that label, minus 1. It is possible to place the label before the CALA instruction, then load the PAER with the label address because this is a one-word instruction. However, if the last instruction in this loop had been a two-word instruction, the second word of the instruction would not be read, and the long immediate operand would be substituted with the first instruction in the loop.

Inside the loop, the pointer to the task table is incremented and saved. Then, the task address is read from the table and loaded into the accumulator. Next, the task is called by the CALA instruction. Notice that, when the task returns to the task handler, it returns to the top of the loop. This is because the PC has already been loaded with the PASR before the CALA executes the PC discontinuity. Therefore, when the CALA is executed, the address of the top of the loop is pushed onto the PC stack.

The last two words of a repeat-block loop are not interruptible. In other words, the interrupt path will not be taken while the last two instruction words of a repeat block are being fetched.

Example 3–1. Interrupt Operation With a Single-Word Instruction at the End of an RPTB

```
          RPTB    END_LOOP-1
          SAR     AR0,*           ← interrupt path taken here
                                    if not the last loop iteration
          .
          .
          .
          LACC    *+
          SACL    *               ← interrupt occurs here
ENDLOOP:
          MAR     *,AR1           ← Interrupt path taken here if interrupt
                                    occurs during last two instruction words
                                    of the last loop iteration
```

Example 3–2. Interrupt Operation With a Two-Word Instruction at the End of an RPTB

```
          RPTB    END_LOOP-1
          SAR     AR0,*           ← interrupt path taken here
                                    if not the last loop iteration
          .
          .
          .
          LACC    *+
          SPLK    #1234h,*        ← interrupt occurs here
ENDLOOP:
          MAR     *,AR1           ← Interrupt path taken here if interrupt
                                    occurs during last two instruction words
                                    of the last loop iteration
```

Note that any incoming interrupt is latched by the 'C5x as soon as it meets the interrupt timing requirement. However, the PC does not branch to the corresponding interrupt service routine vector if it is fetching the last two words of a repeat-block loop. This behavior is functionally equivalent to disabling interrupts before fetching the last two instruction words, and re-enabling interrupts afterward. Interrupt operation with repeat blocks potentially increases the worst-case interrupt latency time.

3.6.6 Power-Down Mode

In the power-down mode, the 'C5x core enters a dormant state and dissipates considerably less power than normal. Power-down mode is invoked either by executing the IDLE/ IDLE2 instructions or by driving the $\overline{\text{HOLD}}$ input low with the HM status bit set to one.

While the 'C5x is in power-down mode, all its internal contents are maintained; this allows operation to continue unaltered when power-down mode is terminated. All CPU activities are halted when the IDLE instruction is executed but the CLKOUT1 pin remains active. The peripheral circuits continue to operate, allowing the peripherals such as serial ports and timers to take the CPU out of its powered-down state. Power-down mode, when initiated by an IDLE instruction, is terminated upon receipt of an interrupt. If INTM = 0, then the processor enters the interrupt service routine when IDLE is terminated. If INTM = 1, then the processor continues with the instruction following IDLE.

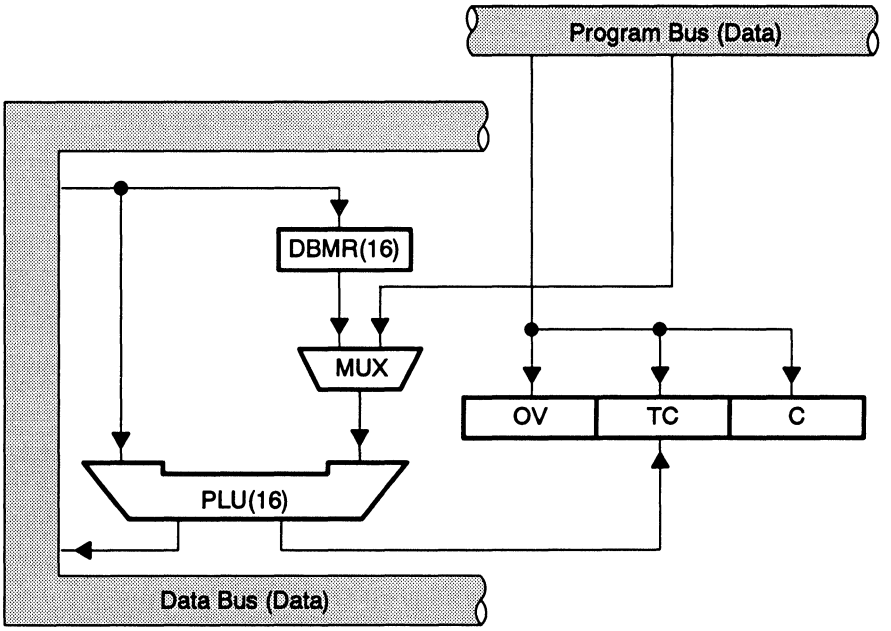
The IDLE2 instruction is used for a complete shutdown of the core CPU as well as all on-chip peripherals. Because the on-chip peripherals are stopped with this power-down mode, they cannot be used to generate the interrupt to wake the device as described above on the IDLE mode. However, the power is significantly reduced because the complete device is stopped. This power-down mode is terminated by activating any of the external interrupt pins ($\overline{\text{RS}}$, $\overline{\text{NMI}}$, $\overline{\text{INT1}}$, $\overline{\text{INT2}}$, $\overline{\text{INT3}}$, and $\overline{\text{INT4}}$) for at least five machine cycles. Once again, if INTM = 0, then the processor enters the interrupt service routine when the IDLE2 instruction is terminated. If INTM =1, then the processor continues with the instruction following the IDLE2. It is advisable to reset peripherals when IDLE2 terminates execution, especially if they are externally clocked.

Power-down mode can also be initiated by the $\overline{\text{HOLD}}$ signal. When the $\overline{\text{HOLD}}$ signal initiates power-down and HM=1, the CPU stops executing; also, address and control lines go into high impedance for further power reduction. If HM=0 when $\overline{\text{HOLD}}$ initiates power-down, address and memory control signal drivers still go into high impedance, but the CPU continues to execute internally. If external memory accesses are not currently required in the system, the HM=0 mode can be used. The device continues to operate normally unless an off-chip access is required by an instruction, at which time the processor halts until the hold is removed. When the $\overline{\text{HOLD}}$ signal initiates the power-down mode, power-down mode is terminated when $\overline{\text{HOLD}}$ goes inactive. $\overline{\text{HOLD}}$ does not stop operation of on-chip peripherals (i.e., on-chip timers and serial ports continue to operate, regardless of the level on $\overline{\text{HOLD}}$ or the condition of the HM bit).

3.7 Parallel Logic Unit (PLU)

The parallel logic unit (PLU) can directly set, clear, test, or toggle multiple bits in a control/status register or any data memory location. The PLU, shown in the block diagram in Figure 3–16, provides a direct logic operation path to data memory values without affecting the contents of the accumulator or product register. It can be used to set or clear multiple bits in a control register or to test multiple bits in a flag register.

Figure 3–16. Parallel Logic Unit Block Diagram



The PLU executes a read-modify-write operation on data stored in data space. The PLU operation begins with the fetching of one operand from data memory space and the fetching of the second from either long immediate on the program bus or the dynamic bit manipulation register (DBMR). Then, the PLU executes a logical operation defined by the instruction on the two operands. The result is written to the same data memory location from which the first operand was fetched.

The PLU allows the direct manipulation of bits in any location in data memory space. This direct bit manipulation is done by ANDing, ORing, XORing, or loading a 16-bit long immediate value to a data location. For example, to use AR1 for circular buffer 1 and AR2 for circular buffer 2 but not enable the circular buffers, initialize the circular buffer control register (CBCR) by executing this:

```
SPLK #021h,CBCR  ;Store peripheral long immediate (DP = 0).
```

To later enable circular buffers 1 and 2, execute

```
OPL #088h,CBCR   ;Set bit 7 and bit 3 in CBCR.
```

Test for individual bits in a specific register or data word via the BIT instruction; however, test against a pattern with the CPL (compare parallel long immediate) instruction. If the data value is equal to the long immediate value, then the TC bit is set to 1. The TC bit is set if the result of any PLU instruction is zero.

The bit set, clear, and toggle functions can also be executed with a 16-bit dynamic register value instead of the long immediate value. This is done with the following three instructions: APL (AND DBMR register to data), OPL (OR DBMR register to data), and XPL (XOR DBMR register to data).

The TC bit in ST1 is also set by the APL, OPL, XPL instructions if the result of the PLU operation (value written back into data memory) is zero. This allows bits to be tested and cleared simultaneously. For example,

```
APL    #0FF00h,TEMP          ;Clear low byte and check for
                             ;bits set in high byte.
BCND   HIGH_BITS_SET,NTC     ;If bits active in high byte,
                             ;then branch.
```

or

```
XPL    #1,TEMP    ;Toggle bit 0.
BCND BIT_SET,TC   ;If bit was set, branch. If not, bit set now.
```

In the first example, the low byte of a flag word is cleared while the high byte is checked for any active flags (bits = 1). If none of the flags in the high byte are set, then the resulting APL operation yields a zero to TEMP and the TC bit is set to 1. If any of the flags in the high byte are set, then the resulting APL operation yields a nonzero value to TEMP and the TC bit is set to 0. Therefore, the conditional branch (BCND) following the APL instruction branches if any of the bits in the high byte are nonzero. The second example tests the flag. If low, it is set high; if high, it is cleared and the branch is taken. The PLU instructions can operate anywhere in data address space, so they can be used to operate with flags stored in RAM locations as well as control registers for both on- and off-chip peripherals.

3.8 Interrupts

The 'C5x core CPU supports sixteen user-maskable interrupts ($\overline{INT16}$–$\overline{INT1}$). However, each 'C5x DSP does not necessarily use all 16. For example, the 'C50, 'C51, and 'C53 use only nine of these interrupts (the others are tied high inside the device). Interrupts can be generated by the serial ports (RINT, XINT, TRNT, and TXNT), by the timer (TINT), and by the software interrupt (TRAP and INTR) instructions. The reset ($\overline{RS}$) interrupt has the highest priority, and the $\overline{INT16}$ interrupt has the lowest priority.

3.8.1 Reset

Reset ($\overline{RS}$) is a nonmaskable external interrupt that can be used at any time to put the 'C5x into a known state. Reset is typically applied after power-up when the machine is in an unknown state.

Driving the $\overline{RS}$ signal low causes the 'C5x to terminate execution and forces the program counter to zero. $\overline{RS}$ affects various registers and status bits. At power-up, the state of the processor is undefined. For correct system operation after power-up, a reset signal must be asserted low for several clock cycles so that data lines are put into the high-impedance state and address lines are driven low (see Appendix A for specific timings). The device will latch the reset pulse and generate an internal reset pulse long enough to guarantee a reset of the device. Several clock cycles after deasserting reset (see Appendix A), the reset vector at program address zero is fetched.

When the $\overline{RS}$ signal is received, the following actions occur:

1) A logic 0 is loaded into the CNF (configuration control) bit in status register ST1, mapping dual-access RAM block 0 into data address space.

2) The program counter (PC) is set to 0. The address bus (lines A15 – A0) is unknown while $\overline{RS}$ is low. IF $\overline{HOLD}$ is asserted while $\overline{RS}$ is low, $\overline{HOLDA}$ is generated. In this case, the address lines are placed into a high-impedance state until $\overline{HOLD}$ is brought back high.

3) All interrupts are disabled by setting the INTM bit (interrupt mode) to 1; note that $\overline{RS}$ and $\overline{NMI}$ are nonmaskable. The interrupt flag register (IFR) is cleared.

4) Status bits are set as follows:
   0 → OV, 1 → XF, 1 → SXM, 0 → PM, 1 → HM, 0 → BRAF,
   0 → TRM, 0 → NDX, 0 → CENB1, 0 → CENB2, 0 → IPTR,
   0 → OVLY, 0 → AVIS, 0 → RAM, 0 → BIG, 0 → CNF,
   1 → INTM, MP/$\overline{MC}$ (Pin) → PMST (MP/$\overline{MC}$), and 1 → C,

   Note that the remaining status bits remain undefined and should be initialized appropriately.

5) The global memory allocation register (GREG) is cleared to make all memory local.

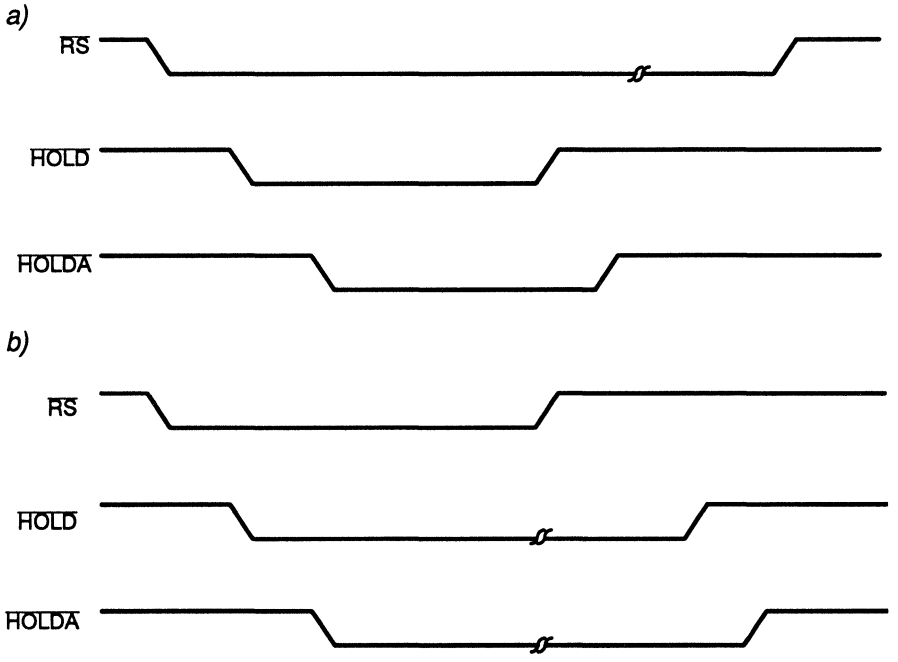
6) The repeat counter (RPTC) is cleared.

7) The $\overline{\text{IACK}}$ (interrupt acknowledge) signal is generated in the same manner as a maskable interrupt.

8) A synchronized reset ($\overline{\text{SRESET}}$) signal is sent to the peripheral circuits to initialize them. See subsection 5.1.3 for peripheral reset information.

Execution starts from location 0 of program memory when the $\overline{\text{RS}}$ signal is taken high. Note that if $\overline{\text{HOLD}}$ is asserted while $\overline{\text{RS}}$ is low, normal reset operation occurs internally, but all buses and control lines remain in a high-impedance state and $\overline{\text{HOLDA}}$ is asserted, as shown in Figure 3–17(a) and (b). However, if $\overline{\text{RS}}$ is asserted while $\overline{\text{HOLD}}$/$\overline{\text{HOLDA}}$ are low, the CPU comes out of the hold mode momentarily by deasserting $\overline{\text{HOLDA}}$. This condition should be avoided. Upon release of $\overline{\text{HOLD}}$ and $\overline{\text{RS}}$, execution starts from location zero. Figure 3–17 (a) and (b) shows two valid ways of exiting reset and hold.

Figure 3–17. $\overline{\text{RS}}$ and $\overline{\text{HOLD}}$ Interaction



3.8.2 Interrupt Operation

This subsection explains interrupt organization and management. Vector relative locations and priorities for all internal and external interrupts are shown in Table 3–12 .

The TRAP instruction (software interrupts) is not prioritized but is included here because it has its own vector location. Each interrupt address has been spaced apart by two locations so that branch instructions can be accommodated in those locations. To make vectors stored in ROM reprogrammable, use the following code:

```
LAMM TEMP0     ;ACC = ISR address.
BACC           ;Branch to ISR.
```

TEMP0 resides in B2 and holds the address of the interrupt service routine (ISR). Note that the ISR addresses must be loaded into B2 before interrupts are enabled. Further information regarding interrupt operation, with respect to specific devices in the 'C5x generation, is located in Chapter 5, *Peripherals*.

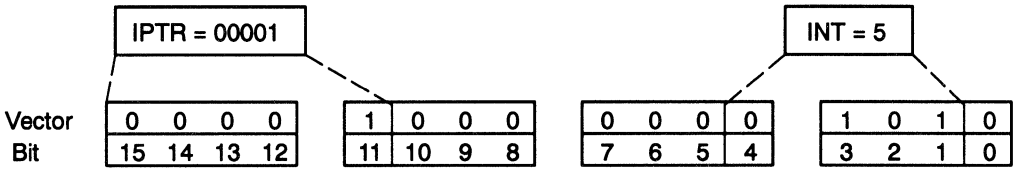
The interrupt vectors can be remapped to the beginning of any 2K-word page in program memory. The interrupt vector address is generated by concatenating the IPTR bits of the PMST with the interrupt vector number (1–16) shifted by one as shown in Figure 3–18.

Table 3–12. Interrupt Locations and Priorities

Name †	Location		Priority	Function
	Dec	Hex		
$\overline{RS}$	0	0	1 (highest)	reset signal
$\overline{INT1}$	2	2	3	user interrupt #1
$\overline{INT2}$	4	4	4	user interrupt #2
$\overline{INT3}$	6	6	5	user interrupt #3
$\overline{INT4}$	8	8	6	user interrupt #4
$\overline{INT5}$	10	A	7	user interrupt #5
$\overline{INT6}$	12	C	8	user interrupt #6
$\overline{INT7}$	14	E	9	user interrupt #7
$\overline{INT8}$	16	10	10	user interrupt #8
$\overline{INT9}$	18	12	11	user interrupt #9
$\overline{INT10}$	20	14	12	user interrupt #10
$\overline{INT11}$	22	16	13	user interrupt #11
$\overline{INT12}$	24	18	14	user interrupt #12
$\overline{INT13}$	26	1A	15	user interrupt #13
$\overline{INT14}$	28	1C	16	user interrupt #14
$\overline{INT15}$	30	1E	17	user interrupt #15
$\overline{INT16}$	32	20	18	user interrupt #16
TRAP	34	22	N/A	TRAP instruction vector
$\overline{NMI}$	36	24	2	nonmaskable interrupt

† The interrupt numbers here do not correspond to any specific 'C5x device. The definitions of the interrupts, specific to particular 'C5x devices, are covered in Chapter 5.

Figure 3–18. Interrupt Vector Address Generation



Upon reset, the IPTR bits are all set to zero, thus mapping the vectors to page zero in program memory space. This means the reset vector always resides at zero. The interrupt vectors can be moved to another location by loading a nonzero value into the IPTR bits. For example, the interrupt vectors can be moved to start at location 0800h by loading the IPTR with 1.

When an interrupt occurs, a flag is activated in the 16-bit interrupt flag register (IFR). This happens regardless of whether the interrupt is enabled or disabled. Each interrupt is stored in the IFR until it is recognized by the CPU. Any of the following four events clears the interrupt flag:

1) Device reset ($\overline{RS}$ is active low),

2) Program takes the interrupt trap,

3) Program writes a one to the appropriate bit in IFR, or

4) Execution of the INTR instruction with the appropriate interrupt number.

The IFR is located at address 6 in data memory space and can be read to identify active interrupts and written to clear interrupts.

A logic one in an IFR bit position indicates a pending interrupt. A one can be written to a specific bit to clear the corresponding interrupt. All pending interrupts can be cleared by writing the current contents of the IFR back into the IFR. The following example clears these two vectors without affecting any other flags that may have been set:

```
SPLK #5,IFR  ;Clear flags for INT1 and INT3.
```

An interrupt flag is automatically cleared when the corresponding interrupt trap is taken. When the CPU accepts the interrupt, it jams the instruction bus with an INTR instruction. This instruction forces the PC to the appropriate address and fetches the soft vector. While fetching the first word of the soft vector, it generates an interrupt acknowledge ($\overline{IACK}$) signal that clears the appropriate interrupt flag bit. The number of the specific interrupt being taken is indicated by address bits A1 – A5 on the falling edge of $\overline{IACK}$. If the interrupt vectors reside in on-chip memory, the device should be operating in address visibility mode (AVIS = 0) for the interrupt number to be decoded. A hardware reset ($\overline{RS}$ is active low) clears all pending interrupt flags. If an interrupt occurs while the device is in HOLD and HM = 0, the address will not be present when the $\overline{IACK}$ goes active low.

The 'C5x has a memory-mapped interrupt mask register (IMR) for masking external and internal interrupts. A 1 in bit positions 15 through 0 of the IMR enables the corresponding interrupt, provided that INTM = 0. The IMR is accessible with both read and write operations. Note that neither $\overline{NMI}$ nor $\overline{RS}$ is included in the IMR; therefore, the IMR has no affect on the nonmaskable interrupt or reset.

The INTM (global enable) bit, which is bit 9 of status register ST0, enables or disables all interrupts. INTM = 0 enables all the unmasked interrupts, and INTM = 1 disables these interrupts. The INTM is set to 1 automatically when an interrupt trap is taken. If the interrupt service routine is exited via the RETE instruction (return from interrupt with automatic re-enable), then the INTM bit is re-enabled (set to zero). It can also be set to 1 with a hardware reset ($\overline{RS}$ is low) or by executing a disable interrupt (SETC INTM) instruction. This bit is reset to a zero by executing the enable interrupt instruction (CLRC INTM). The INTM does not actually modify the IMR or IFR.

The interrupt latency of 'C5x depends on the current contents of the pipeline. The device always completes all instructions in the pipeline before executing the soft vector. The following example, Example 3–3, illustrates the minimum latency from the time an interrupt occurs externally to the interrupt acknowledge ($\overline{IACK}$). The minimum interrupt acknowledge time is defined as 8 cycles:

- 3 cycles to externally synchronize the interrupt
- 1 cycle for the interrupt to be recognized by the CPU
- 4 cycles to execute the INTR instruction and flush the pipeline

On the ninth cycle, the interrupt vector is fetched and the $\overline{IACK}$ is generated.

Example 3–3. Minimum Interrupt Latency

```
Interrupt occurs                    Interrupt                This instruction will be
before the fetch of                 written to               refetched after return from
this instruction  ↓                       IFR  ↓            ↓ interrupt

Fetch     Main1  Main2  Main3  Main4  Main5  Main6  Dummy  Dummy  Dummy  Vec1  Vec2  Dummy  Dummy  ISR1
Decode           Main1  Main2  Main3  Main4  Main5  INTR   Dummy  Dummy  Dummy Vec1  Vec2   Dummy  Dummy
Read                    Main1  Main2  Main3  Main4  Main5  INTR   Dummy  Dummy Dummy Vec1   Vec2   Dummy
Execute                        Main1  Main2  Main3  Main4  Main5  INTR   Dummy Dummy Dummy  Vec1   Vec2

                               ↑ Interrupt            ↑ INTR jammed          ↑ IACK
                               latched external       into the pipe-         generated
                               to the CPU             line                   here
```

The maximum latency is a function of what is in the pipeline. Multicycle instructions add additional cycles to empty the pipeline. This applies to instructions that are extended via wait-state insertion on memory accesses. The wait states required for interrupt vector accesses also affect the latency. The repeat next instruction N times (RPT and RPTZ) also locks out interrupts (including NMI, but not reset), and the repeated instruction completes all executions before allowing the interrupt to execute. This is to protect the context of the re-

peated instructions because when repeated, the instructions run more parallel operations in the pipeline, and the context of these additional parallel operations cannot be saved in an ISR. The $\overline{\text{HOLD}}$ function takes precedence over interrupts and also can delay the interrupt trap. If an interrupt happens during an active-$\overline{\text{HOLD}}$ state, the interrupt is taken at the completion of the $\overline{\text{HOLD}}$ state, that is, when $\overline{\text{HOLDA}}$ is deasserted. However, if the processor is in concurrent hold mode (HM bit of ST1 is 0) and the interrupt vector table is located in internal memory, then the CPU takes the interrupt, regardless of $\overline{\text{HOLD}}$ status.

Interrupts cannot be processed between CLRC INTM and the next instruction in a program sequence. For example, if an interrupt occurs during an CLRC INTM instruction execution, the device always completes CLRC INTM as well as the following instruction before the pending interrupt is processed. This ensures that a return (RET) can be executed in an ISR before the next interrupt is processed—thus protecting against PC stack overflow. If the ISR is exited via a RETE (return from ISR with enable), the CLRC INTM is unnecessary. Of course, after a SETC INTM instruction, the following instruction will not be interrupted.

3.8.3 Interrupt Context Save

When an interrupt trap is executed, certain strategic registers are saved automatically. When the return from interrupt instruction (RETE or RETI) is executed, these registers are automatically restored. The program counter (PC) is saved on an 8-deep hardware stack. This stack is also used for subroutine calls. Therefore, the device supports subroutine calls within the interrupt service routine (ISR) as long as the 8-level stack is not exceeded. Also, there is a one-deep stack (or shadow registers) for each of the following registers:

ACC	accumulator
ACCB	accumulator buffer
PREG	product register
ST0	status register 0
ST1	status register 1
PMST	processor mode status register
TREG0	temporary register for multiplier
TREG1	temporary register for shift count
TREG2	temporary register for bit test
INDX	indirect address index register
ARCR	auxiliary register compare register

When the interrupt trap is taken, all these registers are each pushed onto a one-level stack, with the exception of the XF bit in ST1 and the INTM bit in ST0. On an interrupt, the INTM bit is always set to 1 to disable interrupts. The values in the registers at the time of the trap are still available to the ISR but are also protected in the stack. The stack is popped when the return from interrupt

(RETI or RETE) is executed. This system allows the CPU to be used without requiring context save and restore overhead in the ISR.

With only a one-level stack for the above 11 registers, this hardware does not support nested interrupts. In most cases, this is not a problem, because without the context save and restore overhead, serial processing of the interrupts is so efficient that nested interrupt handling is less effective. If the application does require nested interrupts, they can be handled by using a software stack. Software compatibility with the 'C25 is maintained because the RET instruction, used to return from the ISR on a 'C25, does not pop these registers. Interrupts are not re-enabled unless an RETE or a CLRC INTM instruction is executed.

In a case where the ISR needs to modify values in these registers with respect to the interrupted code, these registers can be popped from the stack as shown in the following example and modified:

```
ISR
    LACC #ISR_RE_ENTER  ;ACC = address of reentry point.
    PUSH                ;Top of stack = reentry point.
    RETI                ;Pop all the stacks.
ISR_RE_ENTER
    .
    .
    .
    CLRC INTM
    RET                 ;Return to interrupted code.
```

In the example, the address of the reentry point within the ISR is pushed onto the PC stack. The RETI instruction pops all the stacks, including the PC stack, and resumes execution. At the end of the ISR, a standard return is executed because the stack is already popped.

Not all of the 16 core CPU interrupts are necessarily used on any given 'C5x device. The vectors for the interrupts not tied to specific external pins or internal peripherals can be used as software interrupts. To use the corresponding interrupt vectors as software traps with full context save and restore, execute the INTR instruction with the appropriate interrupt number as an operand. These traps are protected from other interrupts in the same way the ISR is protected; all interrupts are globally masked via the INTM bit. To execute the context restore, these trap routines must be exited via the RETI or RETE instruction. For example,

```
INTR   15     ;Software trap to address 01Eh.
```

In this example, the processor traps to the vector relatively located at 01Eh.

3.8.4 Nonmaskable Interrupt

The core of the 'C5x has two nonmaskable interrupts, $\overline{RS}$ (reset) and $\overline{NMI}$. Reset is discussed in subsection 3.8.1 $\overline{NMI}$ is a soft reset. It is different from a

standard interrupt because it is not maskable, and it does not invoke the automatic context save. The context save is not invoked, because it is possible to take the $\overline{\text{NMI}}$ even during an interrupt service routine. In addition, interrupts are globally disabled during an NMI instruction. The $\overline{\text{NMI}}$ is different from reset in that it does not affect any of the modes of the device. Note that some 'C5x devices may not make the $\overline{\text{NMI}}$ available externally. The $\overline{\text{NMI}}$ is also delayed by multicycle instructions (including RPT) and by $\overline{\text{HOLD}}$, as described in subsection 3.8.2. The $\overline{\text{NMI}}$ trap can also be initiated via software using the NMI instruction. This instruction forces the PC to the NMI trap location.

Chapter 4

Assembly Language Instructions

The 'C5x instruction set supports numerically intensive signal-processing operations as well as general-purpose applications, such as multiprocessing and high-speed control. The instruction set is a superset of the 'C1x and 'C2x instruction sets and is source-code upward compatible with both devices. This chapter describes the assembly language instructions for the 'C5x digital signal processor. Included in this chapter are the following major topics:

Topic **Page**

4.1 Memory Addressing Modes

The 'C5x instruction set provides six basic memory addressing modes:

- Direct addressing mode
- Indirect addressing mode
- Immediate addressing mode
- Dedicated register addressing mode
- Memory-mapped register addressing mode
- Circular addressing mode

Both direct and indirect addressing can be used to access data memory. Direct addressing concatenates seven bits of the instruction word with the nine bits of the data memory page pointer to form the 16-bit data memory address. Indirect addressing accesses data memory through one of eight auxiliary registers. In immediate addressing, the data is based on a portion of the instruction word(s). Two types of immediate addressing modes are available: short and long. In short immediate addressing, an 8-/ 9-/13-bit operand is included in the instruction word. Long immediate addressing mode uses as its operand a 16-bit word following the instruction. Dedicated register addressing refers to the block move instructions in which the BMAR register addresses program or data memory and the parallel logic unit (PLU) instructions in which operands are obtained from the DBMR register. Memory-mapped register addressing mode is used to load and store memory-mapped registers. Circular addressing is an additional mode of indirect addressing that automatically wraps to the beginning of a block of data when the end of the block is reached. The following subsections describe each addressing mode and give the opcode formats and some examples for each mode.

4.1.1 Direct Addressing Mode

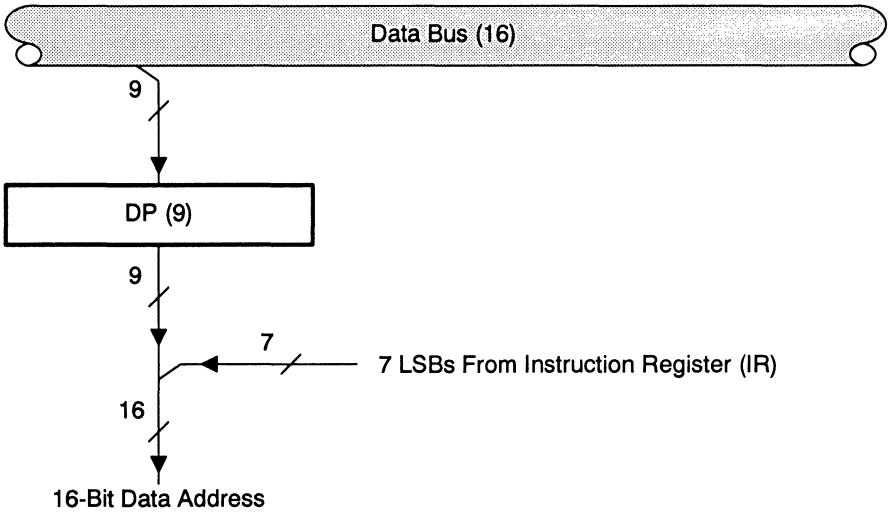
In the direct memory addressing mode, the instruction contains the lower seven bits of the data memory address (dma). This field is concatenated with the nine bits of the data memory page pointer (DP) register to form the full 16-bit data memory address. Thus, the DP register points to one of 512 possible 128-word data memory pages, and the 7-bit address in the instruction points to the specific location within that data memory page. The DP register is loaded by using the LDP (load data memory page pointer) or the LST #0 (load status register ST0) instructions.

Note:

The data page pointer is not initialized by reset and, therefore, is undefined after power-up. The 'C5x development tools, however, utilize default values for many parameters, including the data page pointer. Because of this, programs that do not explicitly initialize the data page pointer may execute improperly, depending on whether they are executed on a 'C5x device or with a development tool. Thus, it is critical that all programs initialize the data page pointer in software.

Figure 4-1 illustrates how the 16-bit data address is formed.

Figure 4–1. Direct Addressing Block Diagram



The direct addressing format is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode								0	dma						

Bits 15 through 8 contain the opcode. Bit 7 = 0 defines the addressing mode as direct, and bits 6 through 0 contain the data memory address (dma).

Example of direct addressing format:

```
ADD 9h,5  ;The contents of data address 9h is
          ;left-shifted 5 bits and added to the
          ;contents of the accumulator.
```

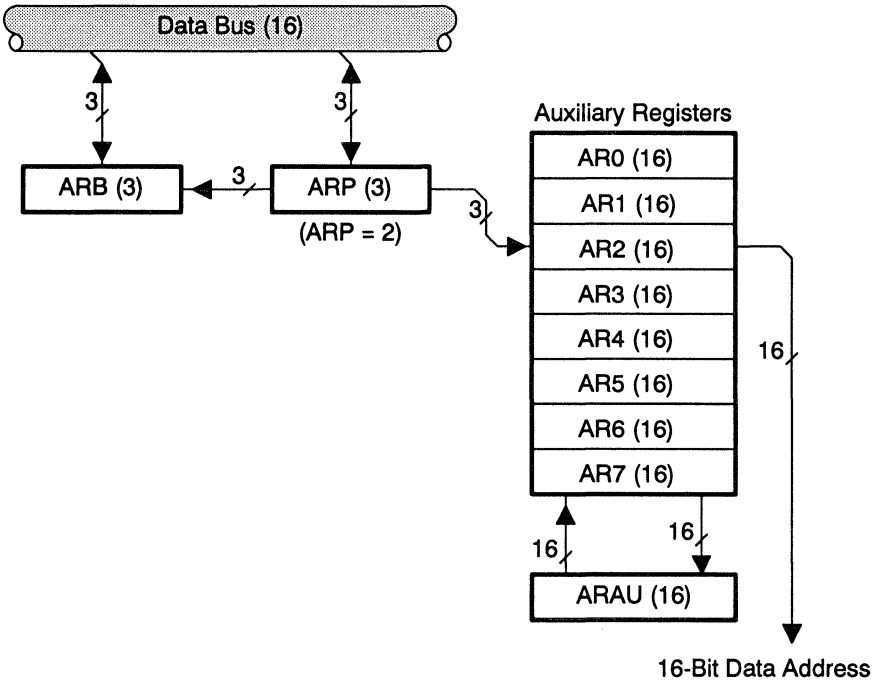
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	1

The opcode of the ADD 9h,5 instruction is 25h and appears in bits 15 through 8. The shift count of 5 appears in bits 11 through 8 of the opcode. The data memory address 09h appears in bits 6 through 0.

4.1.2 Indirect Addressing Mode

Eight auxiliary registers (AR0–AR7) provide flexible and powerful indirect addressing on the 'C5x. To select a specific auxiliary register, load the auxiliary register pointer (ARP) with a value from 0 through 7, designating AR0 through AR7, respectively (see Figure 4–2).

Figure 4–2. Indirect Addressing Block Diagram



The contents of the auxiliary registers may be operated upon by the auxiliary register arithmetic unit (ARAU), which implements unsigned16-bit arithmetic. The ARAU performs auxiliary register arithmetic operations in the decode phase of the pipeline. This allows the address to be generated before the decode phase of the next instruction. The AR is incremented or decremented after it is used in the current instruction.

In indirect addressing, any location in the 64K data memory space can be accessed via a 16-bit address contained in an auxiliary register. The LAR instruction loads the address into the register. The auxiliary registers on the 'C5x may

be modified by ADRK (add to auxiliary register short immediate) or SBRK (subtract from auxiliary register short immediate); they may also be modified by the MAR (modify auxiliary register) instruction or, equivalently, by the indirect addressing field of any instruction supporting indirect addressing. AR(ARP) denotes that the auxiliary register is to be selected by ARP. The auxiliary registers can also be loaded via the data bus by using memory-mapped writes to the auxiliary registers. The following instructions can write to the memory-mapped auxiliary registers: APL, BLDD, LMMR, OPL, SACH, SACL, SAMM, SMMR, SPLK, and XPL. Be careful when using these memory-mapped loads of the auxiliary registers because in this case the memory-mapped auxiliary registers are modified in the execute phase of the pipeline. This causes a pipeline conflict if one of the next two instruction words modifies that auxiliary register. For further information on the pipeline and possible pipeline conflicts, see subsection 3.6.2.

The following symbols are used in indirect addressing, including bit-reversed (BR) addressing:

*	Contents of AR(ARP) are used as the data memory address.
*–	Contents of AR(ARP) are used as the data memory address and decremented after the access.
*+	Contents of AR(ARP) are used as the data memory address and incremented after the access.
*0–	Contents of AR(ARP) are used as the data memory address, and the contents of INDX are subtracted from it after the access.
*0+	Contents of AR(ARP) are used as the data memory address, and the contents of INDX are added to it after the access.
*BR0–	Contents of AR(ARP) are used as the data memory address, and the contents of INDX are subtracted, with reverse carry (rc) propagation, from it after the access.
*BR0+	Contents of AR(ARP) are used as the data memory address, and the contents of INDX added, with reverse carry (rc) propagation, to it after the access.

There are two primary types of indirect addressing with indexing:

- Regular indirect addressing with increment or decrement, and
- Indirect addressing with indexing based on the value of INDX:
  - Indexing by adding or subtracting the contents of INDX, or
  - Indexing by adding or subtracting the contents of INDX with the carry propagation reversed (for FFTs on the 'C5x).

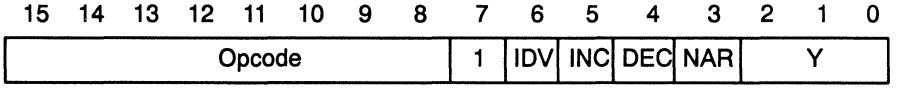
In either case, the contents of the auxiliary register pointed to by the ARP register are used as the address of the data memory operand. Then, the ARAU per-

forms the specified mathematical operation on the indicated auxiliary register. Additionally, the ARP may be loaded with a new value. All indexing operations are performed on the current auxiliary register in the same cycle as the original instruction decode phase of the pipeline.

Indirect auxiliary register addressing allows for post-access adjustments of the auxiliary register pointed to by the ARP. The adjustment may be an increment or decrement by one or may be based upon the contents of the INDX register. To maintain compatibility with the 'C2x devices, set the NDX bit in the PMST register to 0. In the 'C2x architecture, the current auxiliary register can be incremented or decremented by the value in the AR0 register. When the NDX bit is set to 0, every AR0 modification or LAR write also writes the ARCR and INDX registers with the same value. Subsequent modifications of the current auxiliary registers using indexed addressing will use the INDX register, therefore maintaining compatibility with existing 'C2x code. The NDX bit is set to 0 at reset.

Bit-reversed addressing modes on the 'C5x allow efficient I/O to be performed by the resequencing of data points in a radix-2 FFT program. The direction of carry propagation in the ARAU is reversed when this mode is selected, and INDX is added to/subtracted from the current auxiliary register. Typical use of this addressing mode requires that INDX first be set to a value corresponding to one-half of the array's size, and that AR(ARP) be set to the base address of the data (the first data point).

Indirect addressing can be used with all instructions except those with immediate operands or with no operands. The indirect addressing format is as follows:



Bits 15 through 8 contain the opcode, and bit 7 = 1 defines the addressing mode as indirect. Bits 6 through 0 contain the indirect addressing control bits.

Bit 6 contains the increment/decrement value (IDV). The IDV bit determines whether the INDX register will be used to increment or decrement the current auxiliary register. If bit 6 = 0, an increment or decrement (if any) by one occurs to the current auxiliary register. If bit 6 = 1, the INDX register is added to or subtracted from the current auxiliary register as defined by bits 5 and 4.

Bits 5 and 4 control the arithmetic operation to be performed with AR(ARP) and the INDX register. When set, bit 5 indicates that an increment is to be performed. If bit 4 is set, a decrement is to be performed. Table 4–1 shows the correspondence of bit pattern and arithmetic operation.

Table 4–1. Indirect Addressing Arithmetic Operations

Bits 6	5	4	Arithmetic Operation
0	0	0	No operation on AR(ARP)
0	0	1	AR(ARP) – 1 → AR(ARP)
0	1	0	AR(ARP) + 1 → AR(ARP)
0	1	1	Reserved
1	0	0	AR(ARP) – INDX → AR(ARP) [reverse carry propagation]
1	0	1	AR(ARP) – INDX → AR(ARP)
1	1	0	AR(ARP) + INDX → AR(ARP)
1	1	1	AR(ARP) + INDX → AR(ARP) [reverse carry propagation]

Bit 3 and bits 2 through 0 control the auxiliary register pointer (ARP). Bit 3 (NAR) determines whether new value is loaded into the ARP. If bit 3 = 1, the contents of bits 2 through 0 (Y = next ARP) are loaded into the ARP. If bit 3 = 0, the contents of the ARP remain unchanged. If the ARP is loaded with a new value, the old value is loaded into the auxiliary register buffer (ARB) in the ST1 status register.

Table 4–2 shows the bit fields, notation, and operation used for indirect addressing.

Table 4–2. Bit Fields for Indirect Addressing

Instruction Field Bits 15 – 8 7 6 5 4 3 2 1 0	Notation	Operation
← Opcode → 1 0 0 0 0 ← Y →	*	No manipulation of ARx/ARP
← Opcode → 1 0 0 0 1 ← Y →	*,Y	Y → ARP
← Opcode → 1 0 0 1 0 ← Y →	*–	AR(ARP) – 1 → AR(ARP)
← Opcode → 1 0 0 1 1 ← Y →	*–,Y	AR(ARP) – 1 → AR(ARP) Y → ARP
← Opcode → 1 0 1 0 0 ← Y →	*+	AR(ARP) + 1 → AR(ARP)
← Opcode → 1 0 1 0 1 ← Y →	*+,Y	AR(ARP) + 1 → AR(ARP) Y → ARP
← Opcode → 1 1 0 0 0 ← Y →	*BR0–	AR(ARP) – *rc*INDX → AR(ARP) †
← Opcode → 1 1 0 0 1 ← Y →	*BR0–,Y	AR(ARP) – *rc*INDX → AR(ARP) Y → ARP †
← Opcode → 1 1 0 1 0 ← Y →	*0–	AR(ARP) – INDX → AR(ARP)
← Opcode → 1 1 0 1 1 ← Y →	*0–,Y	AR(ARP) – INDX → AR(ARP) Y → ARP
← Opcode → 1 1 1 0 0 ← Y →	*0+	AR(ARP) + INDX → AR(ARP)
← Opcode → 1 1 1 0 1 ← Y →	*0+,Y	AR(ARP) + INDX → AR(ARP) Y → ARP
← Opcode → 1 1 1 1 0 ← Y →	*BR0+	AR(ARP) + *rc*INDX → AR(ARP) †
← Opcode → 1 1 1 1 1 ← Y →	*BR0+,Y	AR(ARP) + *rc*INDX → AR(ARP) Y → ARP †

† BR = bit-reversed addressing mode and *rc* = reverse carry propagation

The CMPR (compare auxiliary register with ARCR) and TC/NTC conditions facilitate conditional branches, calls, returns, or conditional executes according to comparisons between the contents of ARCR and the contents of AR(ARP). To maintain compatibility with the TMS320C2x devices, set the NDX bit in the PMST register to 0. In the 'C2x architecture, the auxiliary register compare function is performed by comparing AR0 with the current auxiliary register. When the NDX bit is set to 0, every load to AR0 loads the ARCR register with the same value. Subsequent compares of the current auxiliary register will use the ARCR register, therefore maintaining compatibility with existing 'C2x code. The NDX bit is set to 0 at reset. The auxiliary registers may also be used for temporary storage via the load and store auxiliary register instructions, LAR and SAR, respectively, or via any instruction that can load and store the memory-mapped auxiliary registers.

The following examples illustrate the indirect addressing format:

Example 1 **ADD *+,8**

Add to the accumulator the contents of the data memory address defined by the contents of the current auxiliary register. This data is left-shifted 8 bits before being added. The current auxiliary register is autoincremented by one. The instruction word is 028A0h.

Example 2 **ADD *,8**

As in Example 1, but with no autoincrement; the instruction word is 02880h.

Example 3 **ADD *–,8**

As in Example 1, except that the current auxiliary register is decremented by one; the instruction word is 02890h.

Example 4 **ADD *0+,8**

As in Example 1, except that the contents of register INDX are added to the current auxiliary register; the instruction word is 028E0h.

Example 5 **ADD *0–,8**

As in Example 1, except that the contents of register INDX are subtracted from the current auxiliary register; the instruction word is 028D0h.

Example 6 **ADD *+,8,AR3**

As in Example 1, except that the auxiliary register pointer (ARP) is loaded with the value **3** for subsequent instructions; the instruction word is 028ABh.

Example 7 **ADD *BR0–,8**

The contents of register INDX are subtracted from the current auxiliary register, with reverse carry propagation; the instruction word is 028C0h.

Example 8 **ADD *BR0+,8**

The contents of register INDX are added to the current auxiliary register, with reverse carry propagation; the instruction word is 028F0h.

4.1.3 Immediate Addressing Mode

In immediate addressing, the instruction word(s) contains the value of the immediate operand. The 'C5x has both single-word (8-bit, 9-bit, and 13-bit constant) short immediate instructions and two-word (16-bit constant) long immediate instructions. In short immediate instructions, the immediate operand is contained within the instruction word itself. In long immediate instructions, the word following the instruction word is used as the immediate operand.

The 'C5x instructions listed in Table 4–3 support immediate addressing.

Table 4–3. Instructions That Support Immediate Addressing

8-Bit Immediate	9-Bit Immediate	13-Bit Immediate	16-Bit Immediate
ADD	LDP	MPY	ADD
ADRK			AND
LACL			APL
LAR			CPL
RPT			LACC
SBRK			LAR
SUB			MPY
			OPL
			OR
			RPT
			RPTZ
			SPLK
			SUB
			XOR
			XPL

Example code for the RPT instruction with short immediate addressing:

```
RPT #99      ;Execute the instruction after RPT 100 times.
```

In this example, the immediate operand is contained as a part of the RPT instruction opcode. The instruction word format for RPT with short immediate addressing is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	1	8-bit constant							

For long immediate instructions, the constant is a 16-bit value in the word following the opcode. The 16-bit value can be optionally used as an absolute constant or as a 2s-complement value.

The following is an example code and the instruction word format for the RPT instruction with long immediate addressing:

```
RPT #0FFFh   ;Execute instruction after RPT 1000h times.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	0
16-bit constant															

4.1.4 Dedicated Register Addressing

Nine instructions in the 'C5x instruction set can use one of two special-purpose memory-mapped registers in the core CPU. These two registers are the block move address register (BMAR) and the dynamic bit manipulation register (DBMR). The APL, OPL, CPL, and XPL parallel logic unit (PLU) instructions use the contents of the DBMR register when an immediate value is not specified as one of the operands. The BLDD, BLDP, and BLPD instructions can use the BMAR register to point at the source or destination space of a block move. The MADD and MADS also use the BMAR register to address an operand in program memory for a multiply-accumulate operation.

The syntax for dedicated register addressing can be stated in one of two ways:

1) Specifying BMAR by its predefined symbol as shown below:

```
BLDD BMAR,DAT100 ;DP = 0. BMAR contains the value 200h.
```

The contents of data memory location 200h are copied to data memory location 100 on the current data page. The opcode for this instruction is 0AC64h.

2) Excluding the immediate value from parallel logic unit instructions as shown below. The BMAR register is implied by the MADD and MADS instruction mnemonics.

```
OPL    DAT10 ;DP = 6. DBMR contains the value 0FFF0h.
             ;Address 030Ah contains the value 01h
```

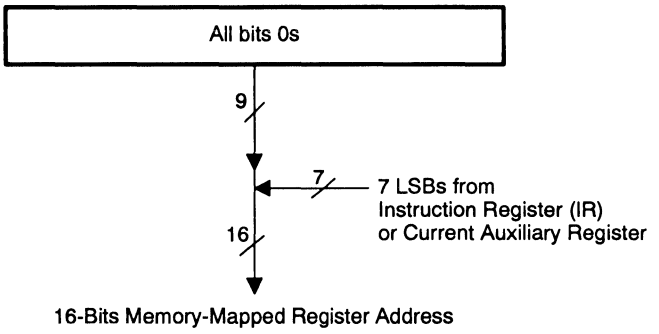
The contents of data memory location 030Ah are ORed with the contents of DBMR. The resulting value 0FFF1h is stored back in memory location 030Ah. The opcode for this instruction is 590Ah.

4.1.5 Memory-Mapped Register Addressing

Memory-mapped register addressing is used for modifying the memory-mapped registers without affecting the current data page pointer value. In addition, any scratch pad RAM location or data page 0 can be modified

by using this addressing mode. Figure 4–3 illustrates how this is done by forcing the 9 MSBs of the data memory address to zero, regardless of the current value of the DP when direct addressing is used or of the current auxiliary register value when indirect addressing is used. The use of these instructions does not affect the contents of the DP.

Figure 4–3. Memory-Mapped Register Addressing Block Diagram



This addressing mode allows greater flexibility for dealing with memory-mapped registers. The overhead required to perform operations involving a memory-mapped register is greatly reduced because the data page pointer (DP) does not need to be modified before and after the operation. The following instructions operate in the memory-mapped register addressing mode:

- LAMM – Load accumulator with memory-mapped register
- SAMM – Store accumulator in memory-mapped register
- LMMR – Load memory-mapped register
- SMMR – Store memory-mapped register

The following examples illustrate the use of these instructions in the direct and indirect addressing modes.

```
LMMR CBCR,#0800h        ;DP = 6. Load CBCR memory-mapped register.
```

The CBCR memory-mapped register is loaded with the value at location 0800h. The instruction word for this instruction is 0891Eh, followed by the 16-bit word 0800h.

```
SAMM  *+                ;Store accumulator to PMST register.
```

If the auxiliary register pointer ARP = 3 and auxiliary register AR3 = FF07h, the contents of the accumulator is stored to the PMST register (address 07h) pointed at by the last 7 bits of AR3. The instruction word for this instruction is 08890h.

4.1.6 Circular Addressing

Many algorithms such as convolution, correlation, and FIR filters can make use of circular buffers in memory. In these algorithms, a circular buffer is used to implement a sliding window, which contains the most recent data to be processed. The 'C5x supports two concurrent circular buffers operating via the auxiliary registers. The following five memory-mapped registers control the circular buffer operation:

- CBSR1 – Circular Buffer One Start Register
- CBSR2 – Circular Buffer Two Start Register
- CBER1 – Circular Buffer One End Register
- CBER2 – Circular Buffer Two End Register
- CBCR – Circular Buffer Control Register

The 8-bit circular buffer control register enables and disables the circular buffer operation. The CBCR is defined as follows:

Bit	Name	Function
0–2	CAR1	Identifies which auxiliary register is mapped to circular buffer 1.
3	CENB1	Circular buffer 1, enable=1/disable=0. Set to 0 upon reset.
4–6	CAR2	Identifies which auxiliary register is mapped to circular buffer 2.
7	CENB2	Circular buffer 2, enable=1/disable=0. Set to 0 upon reset.

In order to define circular buffers, the start and end addresses should first be loaded into the corresponding buffer registers; next, a value between the start and end registers for the circular buffer is loaded into an auxiliary register. The proper auxiliary register value is loaded, and the corresponding circular buffer enable bit is set in the control register. Note that the same auxiliary register can not be enabled for both circular buffers, or unexpected results occur.The algorithm for circular buffer addressing is as follows (note that the test of the auxiliary register value is performed before any modifications):

If (ARn = CBER) and (any AR modification),
Then: ARn = CBSR.
Else: ARn = ARn + step.

In addition, note that if ARn=CBER and no AR modification occurs, the current AR is not modified and is still equal to CBER.Note that when the current auxiliary register = CBER, any AR modification (increment or decrement) will set the current AR = CBSR. The following examples illustrate the operation:

```
splk   #200h,CBSR1  ; Circular buffer start register
splk   #203h,CBER1  ; Circular buffer end register
splk   #0eh,CBCR    ; Enable AR6 pointing to buffer 1

lar    ar6,#200h    ; Case 1
lacc   *            ; AR6 = 200h
```

```
lar   ar6,#203h    ; Case 2
lacc  *            ; AR6 = 203h

lar   ar6,#200h    ; Case 3
lacc  *+           ; AR6 = 201h

lar   ar6,#203h    ; Case 4
lacc  *+           ; AR6 = 200h

lar   ar6,#200h    ; Case 5
lacc  *-           ; AR6 = 1ffh

lar   ar6,#203h    ; Case 6
lacc  *-           ; AR6 = 200h

lar   ar6,#202h    ; Case 7
adrk  2            ; AR6 = 204h

lar   ar6,#203h    ; Case 8
adrk  2            ; AR6 = 200h
```

In circular addressing, the step is the quantity that is being added to or subtracted from the specified auxiliary register. Take care when using a step of greater than one to modify the auxiliary register pointing to an element of the circular buffer. If an update to an auxiliary register generates an address outside the range of the circular buffer, the ARAU does not detect this situation, and the buffer does not wrap around. Auxiliary register updates are performed as described in subsection 4.1.2. Note that there is a two-cycle latency between configuring the circular buffer control registers and performing AR modifications due to the pipeline.

Circular buffers can be used in increment- or decrement-type updates. For incrementing the value in the auxiliary register, the value in CBER must be greater than the value in CBSR. For decrementing the value in the auxiliary register, the CBSR register value must be greater than the value in the CBER register.

4.2 Instruction Set

The 'C5x assembly language instruction set supports both DSP-specific and general-purpose applications. This section lists and groups the 'C5x instruction set according to the following functional headings:

- Accumulator memory reference instructions
- Auxiliary registers and data page pointer instructions
- Parallel logic unit instructions
- T register, P register, and multiply instructions
- Branch instructions
- I/O and data memory operations
- Control instructions

Section 4.1 covers the addressing modes associated with the instruction set, and Section 4.3 describes individual instructions in more detail.

4.2.1 Symbols and Abbreviations

Table 4–4 lists symbols and abbreviations used in the instruction set summary (Table 4–4) and the individual instruction descriptions (Section 4.3).

Table 4–4. Instruction Symbols

Symbol	Meaning
A	Address
ACC	Accumulator
addr	16-bit data memory address
ARB	Auxiliary register pointer buffer
ARn	Auxiliary register n ($0 \leq n \leq 7$)
ARP	Auxiliary register pointer
B	4-bit field specifying bit code
BIO	Branch control input
BMAR	Block move address register
C	Carry bit
CM	2-bit field specifying compare mode
CNF	On-chip RAM configuration control bit
D	Data memory address field
DATn	Label assigned to data memory location n
DBMR	Dynamic bit manipulation register
dma	7-bit data memory address
DP	Data page pointer
FO	Format status bit
FSM	Frame synchronization mode bit
HM	Hold mode bit
I	Addressing mode bit
ind	Indirect addressing operands
INTM	Interrupt mode flag bit
K	Immediate operand field
lK	Long immediate operand field
MCS	Microcall stack
nnh	Indicates that nn represents a hexadecimal number
OV	Overflow bit
OVM	Overflow mode bit
P	Product register
PAn	Port address n ($0 \leq n \leq 65535$)
PC	Program counter
PFC	Prefetch counter
PGMn	Label assigned to program memory location n
PM	2-bit field specifying P register output shift code
pma	Program memory address
R	3-bit field specifying auxiliary register
RPTC	Repeat counter
S	4-bit left-shift code
STn	Status register n (n = 0 or 1)
SXM	Sign-extension mode bit
TREGn	Temporary register n (n = 0, 1, or 2)
TC	Test control bit
TOS	Top of stack
TRM	Control bit to enable multiple TREGs
TXM	Transmit mode bit
XF	XF pin status bit
→	Is assigned to
| x |	Absolute value of x
italics	User-defined items
[ ]	Optional items
( )	Contents of
{ }	Alternative items; one of which must be entered
#	Prefix of constants used in immediate addressing

4.2.2 Instruction Set Summary

Table 4–5 is a summary of the instruction set for the 'C5x digital signal processors. This instruction set is a superset of the 'C1x and 'C2x instruction sets.

The instruction set summary is arranged according to function and is alphabetized within each functional grouping. The number of words that an instruction occupies in program memory is specified in column four of the table. Several instructions specify two values, separated by a slash mark "/" for the number of words. Different forms of the instruction occupy a different number of words. For example, the ADD instruction occupies one word when the operand is a short immediate value or two words if the operand is a long immediate value. The number of cycles that an instruction requires to execute is in column four of the table. All instructions are assumed to be executed from internal program memory (RAM) and internal data dual-access memory. The cycle timings are for single-instruction execution, not for repeat mode. Additional information is presented in the Individual Instruction Descriptions in Section 4.3. Bold typeface indicates instructions that are new for the 'C5x instruction set.

CAUTION

A read or write access to any peripheral memory-mapped register in data memory locations 20h–4Fh will add one cycle to the cycle-time shown. This is due to the fact that all peripherals perform these accesses over the TI Bus.

Section 4.4 includes a table that maps 'C2x instructions to 'C5x instructions. Note that the Texas Instruments 'C5x assembler accepts 'C2x instructions as well as 'C5x instructions.

Table 4–5. Instruction Set Summary

Accumulator Memory Reference Instructions			
Mnemonic	**Description**	**Words**	**Cycles**
ABS	Absolute value of ACC	1	1
ADCB	**Add ACCB to ACC with carry**	**1**	**1**
ADD	Add to ACC	1/2	1 2 (long immediate value specified)
ADDB	**Add ACCB to ACC**	**1**	**1**
ADDC	Add to ACC with carry	1	1
ADDS	Add to low ACC with sign-extension suppressed	1	1
ADDT	Add to ACC with shift specified by TREG1	1	1
AND	AND with ACC	1/2	1 2 (long immediate value specified)
ANDB	**AND ACCB with ACC**	**1**	**1**
BSAR	**Barrel-shift ACC right**	**1**	**1**
CMPL	Complement ACC	1	1
CRGT	**Test for ACC > ACCB**	**1**	**1**
CRLT	**Test for ACC < ACCB**	**1**	**1**
EXAR	**Swap ACCB with ACC**	**1**	**1**
LACB	**Load ACC with ACCB**	**1**	**1**
LACC	Load ACC with shift	1/2	1 2 (long immediate value specified
LACL	Load low word of ACC	1	1
LACT	Load ACC with shift specified by TREG1	1	1
LAMM	**Load ACC with contents of memory-mapped register**	1	**1 (processor memory-mapped register)** **2 (peripheral memory-mapped registers)**
NEG	Negate accumulator	1	1
NORM	Normalize contents of ACC	1	1
OR	OR with accumulator	1/2	1 2 (long immediate value specified)
ORB	**OR ACCB with ACC**	**1**	**1**
ROL	Rotate ACC left	1	1
ROLB	**Rotate ACCB and ACC left**	**1**	**1**
ROR	Rotate ACC right	1	1
RORB	**Rotate ACCB and ACC right**	**1**	**1**

Table 4–5. Instruction Set Summary (Continued)

Accumulator Memory Reference Instructions (Concluded)			
Mnemonic	**Description**	**Words**	**Cycles**
SACB	**Store ACC in ACCB**	**1**	**1**
SACH	Store high ACC with shift	1	1
SACL	Store low ACC with shift	1	1
SAMM	**Store ACC to memory-mapped register**	1	**1 (processor memory-mapped register)** **2 (peripheral memory-mapped registers)**
SATH	**Barrel-shift ACC right 0 or 16 bits as specified by TREG1**	1	1
SATL	**Barrel-shift ACC right 0 to 15 bits as specified by TREG1**	1	1
SBB	**Subtract ACCB from ACC**	1	1
SBBB	**Subtract ACCB from ACC with borrow**	**1**	**1**
SFL	Shift ACC left	1	1
SFLB	**Shift ACCB and ACC left**	**1**	**1**
SFR	Shift ACC right	1	1
SFRB	**Shift ACCB and ACC right**	**1**	**1**
SUB	Subtract from ACC	1/2	1 2 (long immediate value specified)
SUBB	Subtract from ACC with borrow	1	1
SUBC	Conditional subtract	1	1
SUBS	Subtract from low ACC with sign-extension suppressed	1	1
SUBT	Subtract from ACC with shift specified by TREG1	1	1
XOR	Exclusive-OR with ACC	1/2	1 2 (long immediate value specified)
XORB	**Exclusive-OR ACCB with ACC**	**1**	**1**
ZALR	Zero low ACC and load high ACC with rounding	1	1
ZAP	Zero ACC and PREG	1	1
Auxiliary Registers and Data Page Pointer Instructions			
Mnemonic	**Description**	**Words**	**Cycles**
ADRK	Add to ARn short immediate	1	1
CMPR	Compare ARn with ARCR	1	1
LAR	Load ARn	1/2	2
LDP	Load data page pointer	1	2
MAR	Modify ARn	1	1
SAR	Store ARn	1	1
SBRK	Subtract from ARn short immediate	1	1

Table 4–5. Instruction Set Summary (Continued)

Mnemonic	Description	Words	Cycles
Parallel Logic Unit Instructions			
APL	**AND DBMR or constant with data memory value**	**1/2**	**1 (second operand DBMR) 2 (second operand long immediate)**
CPL	**Compare DBMR or constant with data memory value**	**1/2**	**1 (second operand DBMR) 2 (second operand long immediate)**
OPL	**OR DBMR or constant with data memory value**	**1/2**	**1 (second operand DBMR) 2 (second operand long immediate)**
SPLK	**Store long immediate to data memory location**	**2**	**2**
XPL	**XOR DBMR or constant with data memory value**	**1/2**	**1 (second operand DBMR) 2 (second operand long immediate)**
T Register, P Register, and Multiply Instructions			
Mnemonic	**Description**	**Words**	**Cycles**
APAC	Add PREG to ACC	1	1
LPH	Load high PREG	1	1
LT	Load TREG0	1	1
LTA	Load TREG0 & accumulate previous product	1	1
LTD	Load TREG0, accumulate previous product, and move data	1	1
LTP	Load TREG0 & store PREG in accumulator	1	1
LTS	Load TREG0 and subtract previous product	1	1
MAC	Multiply and accumulate	2	3
MACD	Multiply and accumulate with data move	2	3
MADD	**Multiply and accumulate with source pointed at by BMAR**	**1**	**3**
MADS	**Multiply and accumulate both with source pointed at by BMAR and with data move**	**1**	**3**
MPY	Multiply	1/2	1 2 (long immediate value specified)
MPYA	Multiply and accumulate previous product	1	1
MPYS	Multiply and subtract previous product	1	1
MPYU	Multiply unsigned	1	1
PAC	Load ACC with PREG	1	1
SPAC	Subtract PREG from ACC	1	1
SPH	Store high PREG	1	1
SPL	Store low PREG	1	1
SPM	Set PREG output shift mode	1	1
SQRA	Square and accumulate previous product	1	1
SQRS	Square and subtract previous product	1	1
ZPR	**Zero product register**	**1**	**1**

Table 4–5. Instruction Set Summary (Continued)

Branch Instructions			
Mnemonic	**Description**	**Words**	**Cycles**
B[D]	Branch unconditionally	2	4 (2 if delayed)
BACC[D]	Branch to address specified by ACC	1	4 (2 if delayed)
BANZ[D]	Branch on ARn not-zero	2	4 (conditions true, 2 if delayed) 2 (conditions false)
BCND[D]	Branch conditionally	2	4 (conditions true, 2 if delayed) 2 (at least one condition false)
CALA[D]	Call subroutine indirect	1	4 (2 if delayed)
CALL[D]	Call subroutine	2	4 (2 if delayed)
CC[D]	Call conditionally	2	4 (conditions true, 2 if delayed) 2 (at least one condition false)
INTR	Soft interrupt	1	4
NMI	Nonmaskable interrupt	1	4
RET[D]	Return from subroutine	1	4 (2 if delayed)
RETC[D]	Return conditionally	1	4 (conditions true, 2 if delayed) 2 (at least one condition false)
RETE	Return with context switch & global interrupt enable	1	4
RETI	Return with context switch	1	4
TRAP	Software interrupt	1	4
XC	Execute next instruction(s) conditionally	1	1
I/O and Data Memory Operations			
Mnemonic	**Description**	**Words**	**Cycles**
BLDD	Block move from data memory to data memory	1/2	2 (operand specified by BMAR) 3 (operand specified by long immediate)
BLDP	Block move from data memory to program memory	1	2
BLPD	Block move from program memory to data memory	1/2	2 (operand specified by BMAR) 3 (operand specified by long immediate)
DMOV	Data move in data memory	1	1
IN	Input data from port	2	2
LMMR	**Load memory-mapped register**	2	**2 (processor memory-mapped register)** **3 (peripheral memory-mapped register)**
OUT	Output data to port	2	3
SMMR	**Store memory-mapped register**	2	**2 (processor memory-mapped register)** **3 (peripheral memory-mapped register)**
TBLR	Table read	1	3
TBLW	Table write	1	3

Table 4–5. Instruction Set Summary (Continued)

Control Instructions			
Mnemonic	**Description**	**Words**	**Cycles**
BIT	Test bit	1	1
BITT	Test bit specified by TREG2	1	1
CLRC	**Clear control bit**	**1**	**1**
IDLE	**Idle until interrupt**	**1**	**1**
IDLE2	**Idle until interrupt — low power mode**	**1**	**1**
LST	Load status register	1	2
NOP	No operation	1	1
POP	Pop top of stack to low ACC	1	1
POPD	Pop top of stack to data memory	1	1
PSHD	Push data memory value on stack	1	1
PUSH	Push low ACC onto stack	1	1
RPT	Repeat next instruction	1/2	2
RPTB	**Repeat block**	**2**	**2**
RPTZ	**Repeat next instruction and clear ACC and PREG**	**2**	**2**
SETC	**Set control bit**	**1**	**1**
SST	Store status register	1	1

Note that all writes to external memory require two cycles. Reads require one cycle. Any write access immediately before or after a read cycle will require three cycles (refer to Appendix B). In addition, if two pipelined instructions try to access the same 2K-word long single-access memory block simultaneously, one extra cycle is required. For example, the DMOV instruction, when repeated with RPT, requires one cycle in the dual-access RAM but takes two cycles in the single-access RAM. Wait states are added to all external accesses according to the configuration of the software wait-state registers described in Section 5.3.

4.3 Individual Instruction Descriptions

This section furnishes detailed information on the instruction set for the 'C5x family; see Table 4–4, *Instruction Set Summary*, for a complete list of available instructions. Each instruction presents the following information:

- Assembler syntax
- Operands
- Opcode
- Execution
- Description
- Words
- Cycles
- Examples

The **EXAMPLE** instruction is provided to familiarize you with the instruction format and explain the contents of the instruction manual pages.

Syntax

Direct:	[*label*] **EXAMPLE** *dma* [,*shift*]
Indirect:	[*label*] **EXAMPLE** {*ind*} [,*shift*[,*next ARP*]]
Short Immediate:	[*label*] **EXAMPLE** [*#k*]
Long Immediate:	[*label*] **EXAMPLE** [*#lk*]

Each instruction begins with an assembler syntax expression. Labels may be placed either before the command (instruction mnemonic) on the same line or on the preceding line in the first column. An optional comment field may conclude the syntax expression. Spaces are required between each field (label, command, operand, and comment fields).

Operands

$0 \le$ dma ≤ 127
$0 \le$ pma ≤ 65535
$0 \le$ next ARP ≤ 7
$0 \le$ k ≤ 255
$0 \le$ lk ≤ 65535
$0 \le$ shift ≤ 15

ind: {* | *+ | *– | *0+ | *0– | *BR0+ | *BR0–}

The above set of operands is not comprehensive; however, they are the most frequently used in the instruction set. Operands may be constants or assembly-time expressions referring to memory, I/O ports, register addresses, pointers, shift counts, and a variety of other constants.

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

The opcode breaks down the various bit fields that make up each instruction word.

Execution

(PC) + 1 → PC
(ACC) + (dma) → ACC; 0 → C

Affected by OVM; affects OV and C. Not affected by SXM.

The instruction operation sequence describes the processing that takes place when the instruction is executed. Conditional effects of status register specified modes are also given. Those bits in the 'C5x status registers affected by the instruction are also listed.

Description

Instruction execution and its effect on the rest of the processor or memory contents are 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 execution block.

Words

This field specifies the number of memory words required to store the instruction and its extension words.

Cycles

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Instruction				
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

The table shows the number of cycles required for a given 'C5x instruction to execute in a given memory configuration when executed as a single instruc-tiion or in the repeat (RPT) mode. The column headings in the table indicate the program source location (PR, PDA, PSA, PE), defined as follows:

PR The instruction executes from internal program ROM.

PDA The instruction executes from internal dual-access program RAM.

PSA The instruction executes from internal single-access program RAM.

PE The instruction executes from external program memory.

If an instruction requires memory operand(s), row divisions in the table indicate the location(s) of the operand(s), as defined below:

DARAM The operand is in internal dual-access RAM.

SARAM The operand is in internal single-access RAM.

Ext The operand is in external memory.

ROM The operand is in internal program ROM.

MMR The operand is a memory-mapped register.

MMPORT The operand is a memory-mapped io port.

The number of cycles required for each instruction is given in terms of the processor machine cycles (CLKOUT1 period). For the RPT mode execution, *n* indicates the number of times a given instruction is repeated by an RPT or RPTZ instruction. The additional wait states for program/data memory and I/O accesses are defined below. Note that these additional cycles can be generated by the on-chip software wait state generator or by the external READY signal.

p Program memory wait states. Represents the number of additional clock cycles the device waits for external program memory to respond to an access.

d Data memory wait states. Represents the number of additional clock cycles the device waits for external data memory to respond to an access.

io I/O wait states. Represents the number of additional clock cycles the device waits for an external I/O to respond to an access.

n Repetitions (where $n > 2$ to fill the pipeline). Represents the number of times a repeated instruction is executed.

The above variables can also use the subscripts *src, dst,* and *code* to indicate source, destination, and code, respectively.

Note that the internal single-access memory on each 'C5x processor is divided into 1K- or 2K-word blocks contiguous in address space:

'C50	**Data Address Range**
Four 2K-word block	0800h–0FFFh 1000h–17FFh 1800h–1FFFh 2000h–27FFh
One 1K-word block	2800h–2BFFh
'C51	**Data Address Range**
One 1K-word block	0800h–0BFFh
'C53	**Data Address Range**
One 2K-word block	0800h–0FFFh
One 1K-word block	1000h–13FFh

All 'C5x processors support parallel accesses to these internal single-access RAM blocks. However, one single access block allows only one access per cycle. In other words, the processor can read/write on single-access RAM block while accessing another single-access RAM block at the same time.

Note that all external reads take at least one machine cycle while all external writes take at least two machine cycles. However, if an external write is immediately followed or preceded by an external read cycle, then the external write requires three cycles. See Appendix B for details. If the on-chip wait state generator is used to add m ($m > 0$) wait states to an external access, then both the external reads and the external writes require $m+1$ cycles, assuming that the external READY line is driven high. In case the READY input line is used to add m additional cycles to an external access, then external reads require

m+1 cycles, and external write accesses require *m*+2 cycles. See Chapter 6 for the dicussion on software wait states and Appendix A for READY electrical specifications.

The instruction cycle timings are based on the following assumptions:

- At least the next four instructions are fetched from the same memory section (internal or external) that was used to fetch the current instruction (except in case of PC discontinuity instructions like B, CALL, etc.)
- In the single execution mode, there is no pipeline conflict between the current instruction and the instructions immediately preceding or following that instruction. The only exception is the conflict between the fetch phase of the pipeline and the memory read/write (if any) access of the instruction under consideration. See Chapter 3 for pipeline operation.
- In the repeat execution mode, all conflicts caused by the pipelined execution of an instruction are considered.

Refer to Appendix C for further information on instruction cycle classifications and timings.

Example

Example code is included for each instruction. The effect of the code on memory and/or registers is summarized.

Syntax [*label*] **ABS**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0

Execution

(PC) + 1 → PC
|(ACC)| → ACC; 0 → C

Affected by OVM; affects OV and C.
Not affected by SXM.

Description

If the contents of the accumulator are greater than or equal to zero, the accumulator is unchanged by the execution of ABS. If the contents of the accumulator are less than zero, the accumulator is replaced by its 2s-complement value. The carry bit (C) on the 'C5x is always reset to zero by the execution of this instruction.

Note that 80000000h is a special case. When the overflow mode is not set (OVM = 0), the ABS of 80000000h is 80000000h. When the overflow mode is set (OVM = 1), the ABS of 80000000h is 7FFFFFFFh. In either case, the OV status bit is set.

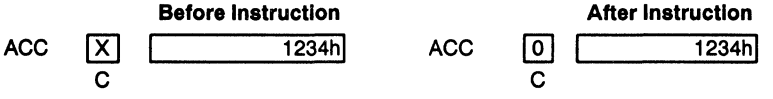
Words 1

Cycles

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
ABS
```

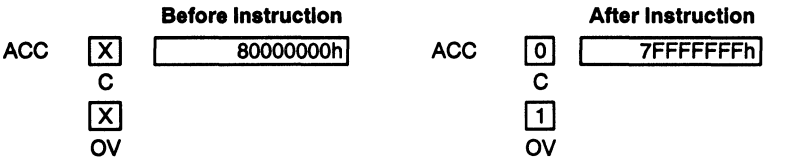


Example 2

```
ABS
```

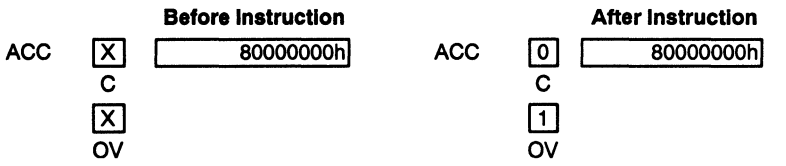
Example 3

```
ABS    ;(OVM = 1)
```



Example 4

```
ABS    ;(OVM = 0)
```



Syntax [*label*] **ADCB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	0	0	1

Execution

(PC) + 1 → PC
(ACC) + (ACCB) + (C) → ACC

Affected by OVM; affects OV and C

Description The contents of the accumulator buffer (ACCB) and the value of the carry bit (C) are added to the accumulator. The carry bit is set to one if the result of the addition generates a carry from the MSB position of the accumulator.

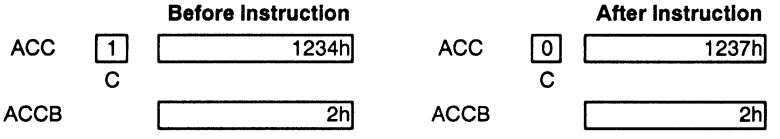
Words 1

Cycles

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ADCB
```



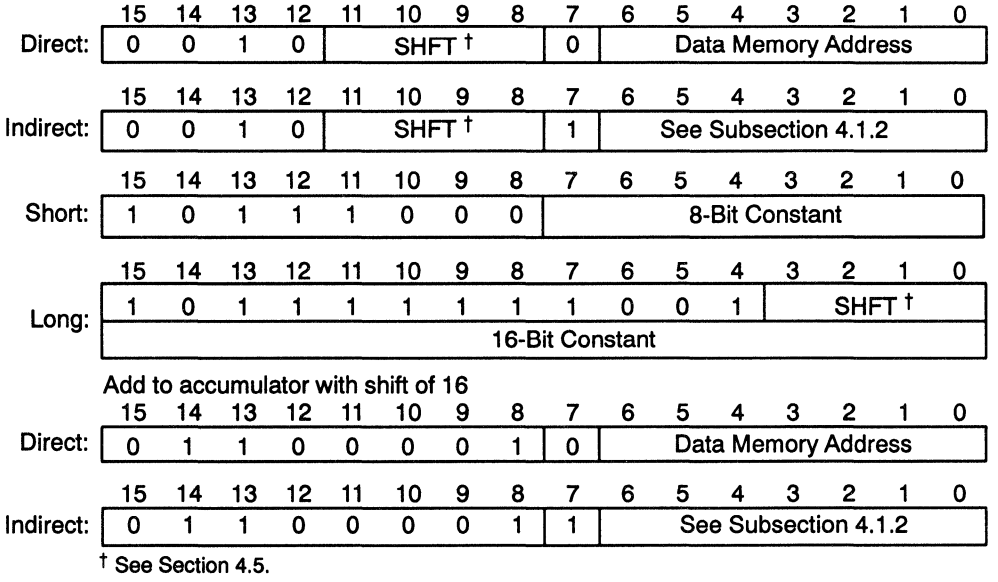
Syntax

Direct: [*label*] **ADD** *dma* [,*shift1*]
Indirect: [*label*] **ADD** {*ind*} [,*shift1* [,*nextARP*]]
Short Immediate: [*label*] **ADD** *#k*
Long Immediate: [*label*] **ADD** *#lk* [,*shift2*]

Operands

0 ≤ dma ≤ 127
0 ≤ shift1 ≤16 (defaults to 0)
0 ≤ next ARP ≤ 7
0 ≤ k ≤ 255
–32768 ≤ lk ≤ 32767
0 ≤ shift2 ≤ 15 (defaults to 0)

Opcode



† See Section 4.5.

Execution

Direct or Indirect Addressing:

(PC) + 1 → PC
(ACC) + [(dma) × 2^{shift1}] → ACC
Affected by SXM and OVM; affects C and OV.

Short Immediate Addressing:

(PC) + 1 → PC
(ACC) + k → ACC
Affected by OVM; affects C and OV

Long Immediate Addressing:

(PC) + 2 → PC
(ACC) + lk × 2^{shift2} → ACC
Affected by SXM and OVM; affects C and OV.

Description

The contents of the addressed data memory location or an immediate constant are left-shifted and added to the accumulator. During shifting, low-order bits are zero-filled. High-order bits are sign-extended if SXM = 1 and zero-filled if SXM = 0. The result is stored in the accumulator. When short immediate addressing is used, the addition is unaffected by SXM and is not repeatable. Note that when the ARP is updated during indirect addressing, a shift operand must be specified. If no shift is desired, a 0 may be entered for this operand.

When adding with a shift of 16, the carry bit is set if the results of the addition generates a carry; otherwise, the carry bit is unaffected. This allows the accumulation to generate the proper single carry when adding a 32-bit number to the accumulator.

Words

1 (Direct, indirect, or short immediate addressing)

2 (Long immediate addressing)

Cycles

Direct: [*label*] **ADD** *dma* [,*shift1*]
Indirect: [*label*] **ADD** {*ind*} [,*shift1* [,*nextARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Short Immediate: [*label*] **ADD** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

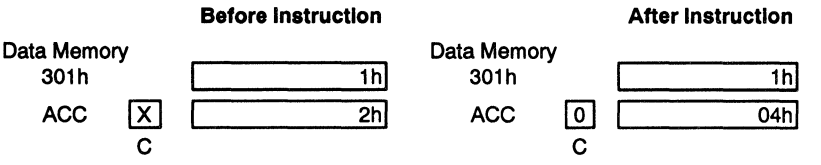
Cycles

Long Immediate: [*label*] **ADD** *#lk* [,*shift2*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

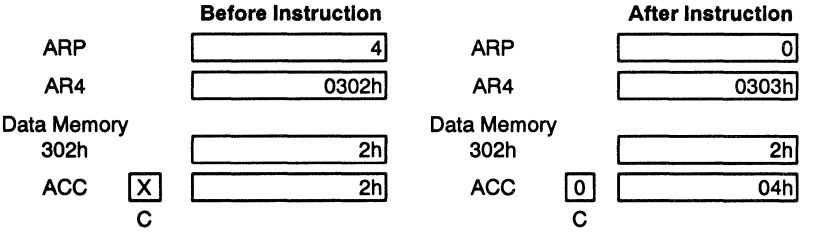
Example 1

```
ADD    DAT1,1  ;(DP = 6)
```



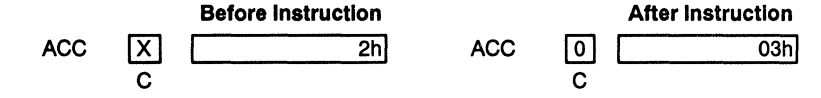
Example 2

```
ADD    *+,0,AR0
```



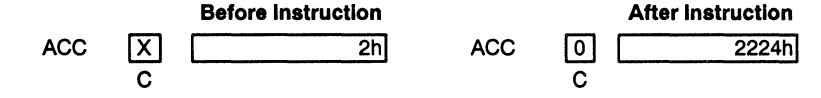
Example 3

```
ADD    #1h     ;Add short immediate
```



Example 4

```
ADD    #1111h,1 ;Add long immediate with shift of 1
```



Syntax [*label*] **ADDB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	0	0	0

Execution

(PC) + 1 → PC
(ACC) + (ACCB) → ACC

Affected by OVM; affects C and OV.

Description The contents of the accumulator buffer (ACCB) are added to the accumulator.

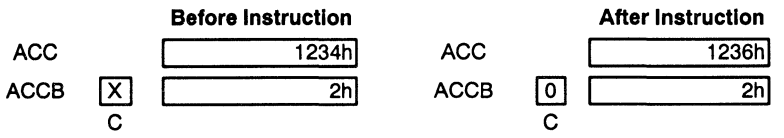
Words 1

Cycles [*label*] **ADDB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ADDB
```



Syntax

Direct: [*label*] **ADDC** *dma*
Indirect: [*label*] **ADDC** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	0	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	0	0	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) + (dma) + (C) → ACC

Affected by OVM; affects OV and C. Not affected by SXM.

The contents of the addressed data memory location and the value of the carry bit are added to the accumulator with sign extension suppressed. The carry bit is then affected in the normal manner.

The ADDC instruction can be used in performing multiple-precision arithmetic.

Words

1

Cycles

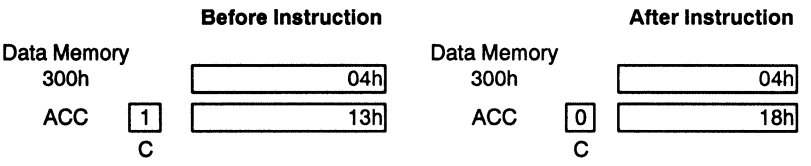
Direct: [*label*] **ADDC** *dma*
Indirect: [*label*] **ADDC** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

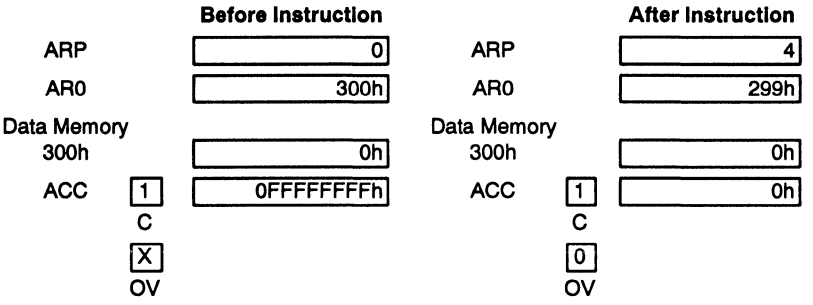
Example 1

```
ADDC   DAT0   ;(DP = 6)
```



Example 2

```
ADDC    *-,AR4 ;(OVM = 0)
```



Syntax

Direct: [*label*] **ADDS** *dma*
Indirect: [*label*] **ADDS** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6 5 4 3 2 1 0
Direct:	0	1	1	0	0	0	1	0	0	Data Memory Address
Indirect:	0	1	1	0	0	0	1	0	1	See Subsection 4.1.2

Execution

(PC) + 1 → PC
(ACC) + (dma) → ACC
(dma) is an unsigned16-bit number

Affected by OVM; affects OV and C.
Not affected by SXM.

Description

The contents of the specified data memory location are added to the accumulator with sign-extension suppressed. The data is treated as an unsigned 16-bit number, regardless of SXM. The accumulator contents are treated as a signed number. Note that ADDS produces the same results as an ADD instruction with SXM = 0 and a shift count of 0.

Words

1

Cycles

Direct: [*label*] **ADDS** *dma*
Indirect: [*label*] **ADDS** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
ADDS   DAT0   ;(DP = 6)
```

	Before Instruction	After Instruction
Data Memory 300h	0F006h	0F006h
ACC	00000003h	0000F009h
C	X	0

Example 2

```
ADDS   *
```

	Before Instruction	After Instruction
ARP	0	0
AR0	0300h	0300h
Data Memory 300h	0FFFFh	0FFFFh
ACC	7FFF0000h	7FFFFFFFh
C	X	0

Syntax

Direct: [*label*] **ADDT** *dma*
Indirect: [*label*] **ADDT** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	0	1	1	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	0	0	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) + [(dma) × $2^{TREG1(3–0)}$] → (ACC)
If SXM = 1:
Then (dma) is sign-extended.
If SXM = 0:
Then (dma) is not sign-extended.

Affected by SXM and OVM; affects OV and C.

Description

The data memory value is left-shifted and added to the accumulator, with the result replacing the accumulator contents. The left-shift is defined by the four LSBs of the TREG1, resulting in shift options from 0 to 15 bits. Sign extension on the data memory value is controlled by SXM. The carry bit is set when a carry is generated out of the MSB of the accumulator.

Software compatibility with the 'C25 can be maintained by setting the TRM bit of the PMST status register to zero. This causes any 'C25 instruction that loads TREG0 to write to all three TREGs. Subsequent calls to the ADDT instruction will shift the value by the TREG1 value (which is the same as TREG0), maintaining object-code compatibility.

Words

1

Cycles

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
ADDT   DAT127  ;(DP = 4. SXM = 0)
```

	Before Instruction		After Instruction	
Data Memory 027Fh		09h	Data Memory 027Fh	09h
TREG1		0FF94h	TREG1	0FF94h
ACC	X	0F715h	ACC	0F7A5h
	C			C

Example 2

```
ADDT   *-,AR4 ;(SXM = 0)
```

	Before Instruction		After Instruction	
ARP		0	ARP	4
AR0		027Fh	AR0	027Eh
Data Memory 027Fh		09h	Data Memory 027Fh	09h
TREG1		0FF94h	TREG1	0FF94h
ACC	X	0F715h	ACC	0F7A5h
	C			C

Syntax [*label*] **ADRK** *#k*

Operands $0 \le k \le 255$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	0	1	1	1	1	0	0	0	8-Bit Constant							

Execution

(PC) + 1 → PC
AR(ARP) + 8-bit positive constant → AR(ARP)

Description The 8-bit immediate value is added, right-justified, to the currently selected auxiliary register (as specified by the current ARP) with the result replacing the auxiliary register contents. The addition takes place in the ARAU, with the immediate value treated as an 8-bit positive integer. Note that all arithmetic operations on the auxiliary registers are unsigned.

Words 1

Cycles [*label*] **ADRK** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
ADRK   #80h
```

	Before Instruction		After Instruction
ARP	5	ARP	5
AR5	4321h	AR5	43A1h

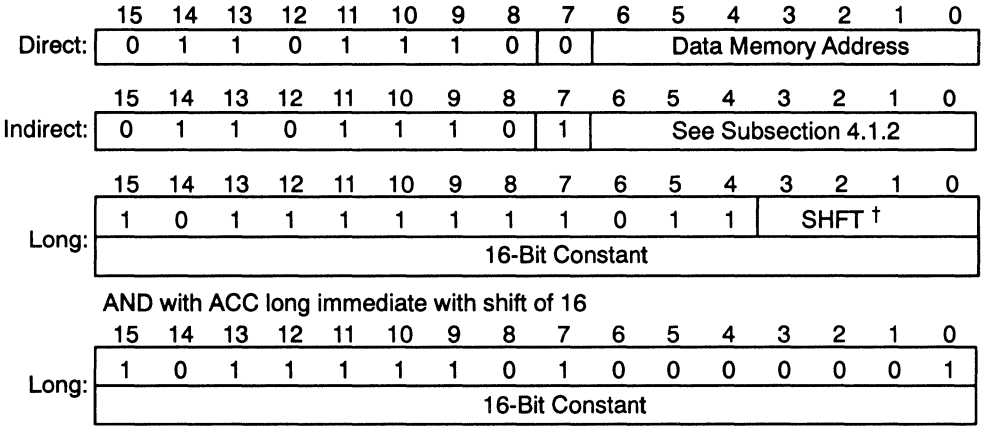
Syntax

Direct:	[*label*] **AND** *dma*
Indirect:	[*label*] **AND** {*ind*} [,*next ARP*]
Long Immediate:	[*label*] **AND** *#lk* [,*shift*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
lk: 16-bit constant
0 ≤ shift ≤ 16

Opcode



† See Section 4.5.

Execution

Direct or Indirect Addressing:
(PC) + 1 → PC
(ACC(15–0)) **AND** (dma) → ACC(15–0)
0 → ACC(31–16)

Immediate Addressing:
(PC) + 2 → PC
(ACC(30–0)) **AND** lk × 2^{shift} → ACC
Not affected by SXM

Description

If direct or indirect addressing is used, the low word of the accumulator is ANDed with a data memory value, and the result is placed in the low word position in the accumulator. The high word of the accumulator is zeroed. If immediate addressing is used, the long immediate constant is shifted, and the low-order bits below and high-order bits above the shifted value are zeroed. The resulting value is ANDed with the accumulator contents.

Words

1 (Direct or indirect addressing)

2 (Long immediate addressing)

Cycles

Direct: [*label*] **AND** *dma*
Indirect: [*label*] **AND** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Long Immediate: [*label*] **AND** *#lk* [,*shift*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
AND    DAT16  ;(DP = 4)
```

	Before Instruction		After Instruction
Data Memory 0210h	00FFh	Data Memory 0210h	00FFh
ACC	12345678h	ACC	00000078h

Example 2

```
AND    *
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	0301h	AR0	0301h
Data Memory 0301h	0FF00h	Data Memory 0301h	0FF00h
ACC	12345678h	ACC	00005600h

Example 3

```
AND    #00FFh,4
```

	Before Instruction		After Instruction
ACC	12345678h	ACC	00000670h

Syntax [*label*] **ANDB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	0	1	0

Execution

(PC) + 1 → PC
(ACC) **AND** (ACCB) → ACC

Description The contents of the accumulator are ANDed with the contents of the accumulator buffer (ACCB). The result is placed in the accumulator while the accumulator buffer is unaffected.

Words 1

Cycles [*label*] **ANDB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ANDB
```

	Before Instruction		After Instruction
ACC	0F0FFFFFh	ACC	05055555h
ACCB	55555555h	ACCB	55555555h

Syntax [*label*] **APAC**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	1	0	0

Execution

(PC) + 1 → PC
(ACC) + (shifted P register) → ACC

Affected by PM and OVM; affects OV and C.
Not affected by SXM.

Description The contents of the P register are shifted as defined by the PM status bits and added to the contents of the accumulator. The result is placed in the accumulator. APAC is not affected by the SXM bit of the status register; the P register is always sign-extended.The APAC instruction is a subset of the LTA, LTD, MAC, MACD, MADS, MADD, MPYA, and SQRA instructions.

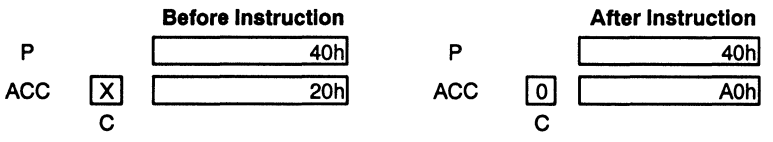
Words 1

Cycles [*label*] **APAC**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
APAC   ;(PM = 01)
```



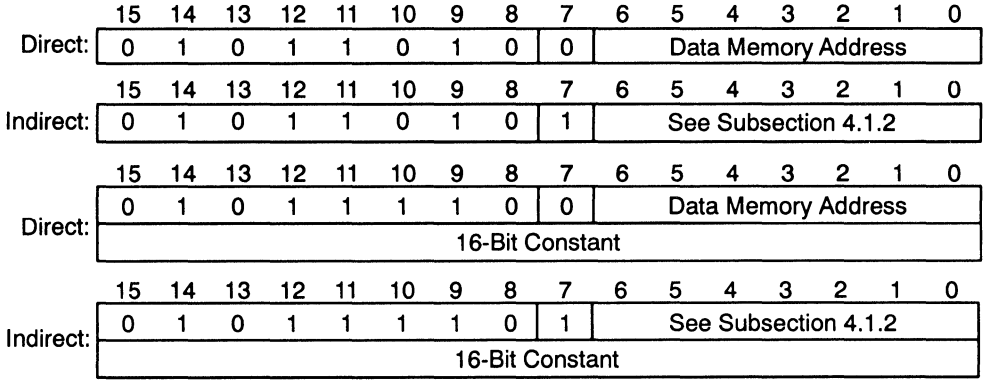
Syntax

Direct: [*label*] **APL** [#*lk*,] *dma*
Indirect: [*label*] **APL** [#*lk*,] {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
lk: 16-bit constant
0 ≤ next ARP ≤ 7

Opcode



Execution

lk unspecified:
(PC) + 1 → PC
(dma) **AND** (DBMR) → dma

lk specified:
(PC) + 2 → PC
(dma) **AND** lk → dma

Affects TC.

Description

If a long immediate constant is specified, it is ANDed with the data memory value dma. Otherwise, the data memory value is ANDed with the contents of the dynamic bit manipulation register (DBMR). In either case, the result is written directly back to the data memory location, while the contents of the accumulator are unaffected. If the result of the AND operation is 0, then the TC bit is set to 1. Otherwise, the TC bit is set to 0.

Words

1 (Second operand DBMR)

2 (Second operand long immediate)

Cycles

Direct: [*label*] **APL** [#*lk*,] *dma*
Indirect: [*label*] **APL** [#*lk*,] {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 3†	1+p
Operand Ext	2+2d	2+2d	2+2d	5+2d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p

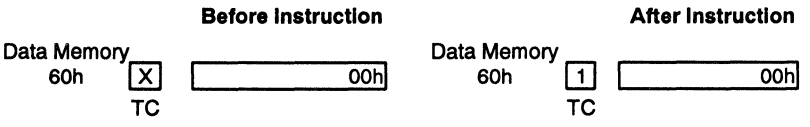
Direct: [*label*] **APL** [*#lk,*] *dma*
Indirect: [*label*] **APL** [*#lk,*] {*ind*} [*,next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2	2+2p
Operand Ext	3+2d	3+2d	3+2d	6+2d+2p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n+1	n+1	n+1	n+1+2p
Operand SARAM	2n–1	2n–1	2n–1 2n+2†	2n–1+2p
Operand Ext	4n–1+2nd	4n–1+2nd	4n–1+2nd	4n+2+2nd+2p

† If the operand and the code reside in same SARAM block.

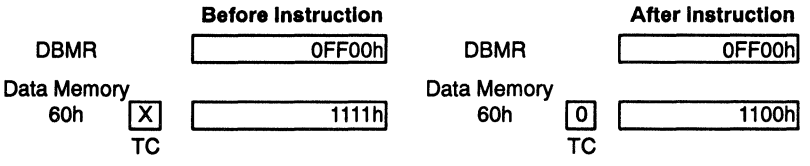
Example 1

```
APL    #0023h,DAT96  ;(DP = 0)
```



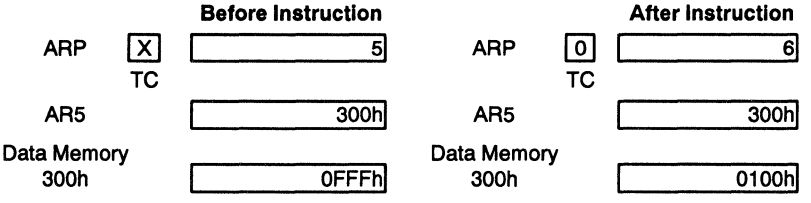
Example 2

```
APL    DAT96  ;(DP = 0)
```



Example 3

```
APL    #0100h,*,AR6
```



Example 4

```
APL   *,AR7
```

	Before Instruction		After Instruction
ARP	X	6	7
TC			
AR6		310h	310h
DBMR		0303h	0303h
Data Memory 310h		0EFFh	0203h

Syntax [*label*] **B**[*D*] *pma* [, {*ind*} [,*next ARP*]]

Operands 0 ≤ pma ≤ 65535
0 ≤ next ARP ≤ 7

Opcode

Branch unconditional with AR update

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	1	1	See Subsection 4.1.2						
16-Bit Constant															

Branch unconditional delayed with AR update

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	1	See Subsection 4.1.2						
16-Bit Constant															

Execution pma → PC
Modify AR(ARP) and ARP as specified.

Description The current auxiliary register and ARP are modified as specified, and control is passed to the designated program memory address (pma). Pma can be either a symbolic or numeric address. The one two-word instruction or two one-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken, if the branch is a delayed branch (specified by the *D* suffix).

Words 2

Cycles [*label*] **B**[*D*] *pma* [, {*ind*} [,*next ARP*]]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+4p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example 1

```
B       191,*+,AR1
```

The value 191 is loaded into the program counter, and the program continues executing from that location. The current auxiliary register is incremented by 1, and ARP is set to auxiliary register 1.

Example 2

```
BD      191
MAR     *+,AR1
LDP     #5
```

After the current AR, ARP, and DP are modified as specified, program execution continues from location 191.

Syntax [*label*] **BACC**[*D*]

Operands None

Opcode

BACC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0

BACCD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	1

Execution ACC(15–0) → PC

Description Control is passed to the 16-bit address residing in the lower half of the accumulator.The one two-word instruction or two one-word instructions following the branch instruction are fetched from program memory and executed before the branch is taken, if the branch is a delayed branch (specified by the *D* suffix).

Words 1

Cycles BACC

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

BACCD (delayed)

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
BACC   ;(ACC contains the value 191)
```

The value 191 is loaded into the program counter, and the program continues executing from that location.

Example 2

```
BACCD          ;(ACC contains the value 191)
MAR    *+,AR1
LDP    #5
```

After the current AR, ARP, and DP are modified as specified, program execution continues from location 191.

Syntax

[label] **BANZ**[*D*] *pma* [, {*ind*} [,*next ARP*]]

Operands

0 ≤ pma ≤ 65535
0 ≤ next ARP ≤ 7

Opcode

BANZ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	1	1	1	See Subsection 4.1.2						
16-Bit Constant															

BANZD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

If AR(ARP)≠0
Then pma → PC
Else (PC) + 2 → PC
Modify AR(ARP) as specified

Description

Control is passed to the designated program memory address (pma) if the contents of the current auxiliary register are not zero. Otherwise, control passes to the next instruction.The default modification to AR(ARP) is a decrement by one. N loop iterations may be executed by initializing an auxiliary register loop counter to N–1 prior to loop entry. The program memory address (pma) can be either a symbolic or a numeric address.

The two one-word instructions or one two-word instruction following the branch instruction are fetched from program memory and executed before the branch is taken, if the branch is a delayed branch (specified by the D suffix).

Words

2

Cycles

[label] **BANZ** *pma* [, {*ind*} [,*next ARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	4	4	4	4+4p†
Condition False	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

[*label*] **BANZD** *pma* [, {*ind*} [,*next ARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	2	2	2	2+2p
Condition False	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

Example 1

```
BANZ   PGM0
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	5h	AR0	4h

0 is loaded into the program counter, and the program continues executing from that location.

or

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	0h	AR0	0FFFFh

The program counter (PC) is incremented by 2, and execution continues from that location.

Example 2

```
BANZD PGM0
LACC   #01h
LDP    #5
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	5h	AR0	4h
DP	4	DP	5
ACC	00h	ACC	01h

After the current DP and ACC are modified as specified, program execution continues from location 0.

Example 3

```
       MAR   *,AR0
       LAR   AR1,#3
       LAR   AR0,#60h
PGM191 ADD   *+,AR1
       BANZ  PGM191,AR0
```

The contents of data memory locations 60h–63h are added to the accumulator.

Syntax [*label*] **BCND**[*D*] *pma*, [*cond1*] [,*cond2*] [,...]

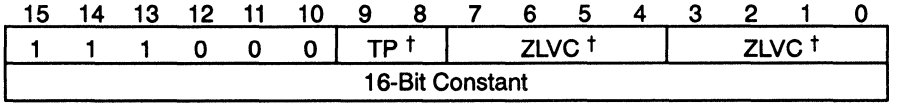
Operands $0 \leq pma \leq 65535$

Conditions:

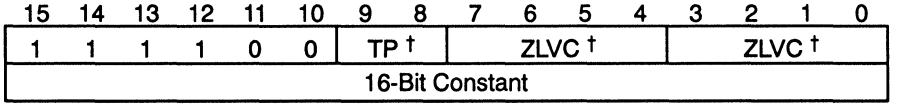
ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
$\overline{BIO}$ low	BIO
TC=0	NTC
TC=1	TC
Unconditionally	UNC

Opcode

BCND



BCNDD



† See Section 4.5.

Execution

If (condition(s))
Then pma → PC
Else PC + 2 → PC

Description

A branch is taken to program memory address pma if the specified conditions are met. Note that not all combinations of conditions are meaningful. Also, note that testing BIO is mutually exclusive to testing TC.

The two one-word instructions or one two-word instruction following the branch are fetched from program memory and executed before the branch is taken, if the branch is a delayed branch (specified by the D suffix). If the delayed instruction is specified, the two instruction words following the BCNDD instruction have no effect on the conditions being tested.

Words 2

Cycles [*label*] **BCND** *pma*, [*cond1*] [,*cond2*] [,...]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	4	4	4	4+4p†
Condition False	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

[*label*] **BCNDD** *pma*, [*cond1*] [,*cond2*] [,...]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	2	2	2	2+2p
Condition False	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

Example 1

```
BCND   PGM191,LEQ,C
```

If the accumulator contents are less than or equal to zero and the carry bit is set, program address 191 is loaded into the program counter, and the program continues executing from that location. If these conditions do not hold, execution continues from location PC + 2.

Example 2

```
BCNDD  PGM191,OV
MAR    *,AR1
LDP    #5
```

After the current AR, ARP, and DP are modified as specified, program execution continues at location 191 if the overflow flag (OV) in status register ST0 is set. If the flag is not set, execution continues at the instruction following the LDP instruction.

Syntax

Direct: [*label*] **BIT** *dma, bit code*
Indirect: [*label*] **BIT** {*ind*} , *bit code* [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ bit code ≤15

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	0	BITX †				0	Data Memory Address						
Indirect:	0	1	0	0	BITX †				1	See Subsection 4.1.2						

† See Section 4.5.

Execution

(PC) + 1 → PC
(dma bit at bit address (15 – bit code)) → TC

Affects TC.

Description

The BIT instruction copies the specified bit of the data memory value to the TC bit of status register ST1. Note that the BITT, CMPR, LST1, APL, CPL, OPL, XPL, and NORM instructions also affect the TC bit in status register ST1. A bit code value is specified that corresponds to a certain bit address in the instruction, as given by the following table:

Bit Address	Bit Code
(LSB) 0	1 1 1 1
1	1 1 1 0
2	1 1 0 1
3	1 1 0 0
4	1 0 1 1
5	1 0 1 0
6	1 0 0 1
7	1 0 0 0
8	0 1 1 1
9	0 1 1 0
10	0 1 0 1
11	0 1 0 0
12	0 0 1 1
13	0 0 1 0
14	0 0 0 1
(MSB) 15	0 0 0 0

Words

1

Cycles

Direct: [*label*] **BIT** *dma, bit code*
Indirect: [*label*] **BIT** {*ind*} , *bit code* [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
BIT    0h,15  ;(DP = 6).Test LSB at 300h
```

	Before Instruction		**After Instruction**
Data Memory 300h	4DC8h	Data Memory 300h	4DC8h
TC	0	TC	0

Example 2

```
BIT    *,0,AR1 ;Test MSB at 310h
```

	Before Instruction		**After Instruction**
ARP	0	ARP	1
AR0	310h	AR0	310h
Data Memory 310h	8000h	Data Memory 310h	8000h
TC	0	TC	1

Syntax

Direct: [*label*] **BITT** *dma*
Indirect: [*label*] **BITT** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	1	1	1	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	0	1	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma bit at bit address (15 –TREG2(3–0))) → TC

Affects TC.

Description

The BITT instruction copies the specified bit of the data memory value to the TC bit of status register ST1. Note that the BITT, CMPR, LST1, CPL, OPL, APL, XPL, and NORM instructions also affect the TC bit in status register ST1. The bit address is specified by a bit code value contained in the 4 LSBs of the TREG2, as given by the table below.

Software compatibility with the 'C25 can be maintained by setting the TRM bit of the PMST status register to zero. This causes any 'C25 instructions that load TREG0 to write to all three TREGs. Subsequent calls to the BITT instruction will use TREG1 value (which is the same as TREG0), maintaining 'C25 object-code compatibility.

Bit Address	Bit Code
(LSB) 0	1 1 1 1
1	1 1 1 0
2	1 1 0 1
3	1 1 0 0
4	1 0 1 1
5	1 0 1 0
6	1 0 0 1
7	1 0 0 0
8	0 1 1 1
9	0 1 1 0
10	0 1 0 1
11	0 1 0 0
12	0 0 1 1
13	0 0 1 0
14	0 0 0 1
(MSB) 15	0 0 0 0

Words 1

Cycles

Direct: [*label*] **BITT** *dma*
Indirect: [*label*] **BITT** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
BITT   00h    ;(DP = 6). Test bit 14 of data at 300h
```

	Before Instruction		After Instruction
Data Memory 300h	4DC8h	Data Memory 300h	4DC8h
TREG2	1h	TREG2	1h
TC	0	TC	1

Example 2

```
BITT   *   ;Test bit 1 of data at 310h
```

	Before Instruction		After Instruction
ARP	1	ARP	1
AR1	310h	AR1	310h
Data Memory 310h	8000h	Data Memory 310h	8000h
TREG2	0Eh	TREG2	0Eh
TC	0	TC	0

Syntax

General syntax: [*label*] **BLDD** *src, dst*

All valid cases have the general syntax:

Direct K/DMA:	[*label*] **BLDD** *#addr, dma*
Indirect K/DMA:	[*label*] **BLDD** *#addr, {ind}* [*,next ARP*]
Direct DMA/K:	[*label*] **BLDD** *dma, #addr*
Indirect DMA/K:	[*label*] **BLDD** *{ind}, #addr* [*,next ARP*]
Direct BMAR/DMA:	[*label*] **BLDD BMAR**, *dma*
Indirect BMAR/DMA:	[*label*] **BLDD BMAR**, *{ind}* [*,next ARP*]
Direct DMA/BMAR:	[*label*] **BLDD** *dma*, **BMAR**
Indirect DMA/BMAR:	[*label*] **BLDD** *{ind}*, **BMAR** [*,next ARP*]

Operands

$0 \le addr \le 65535$
$0 \le dma \le 127$
$0 \le next\ ARP \le 7$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	0	0	0	0	Data Memory Address						
	16-Bit Constant															
Indirect:	1	0	1	0	1	0	0	0	1	See Subsection 4.1.2						
	16-Bit Constant															

Block move data to data DEST long immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	0	0	1	0	Data Memory Address						
	16-Bit Constant															
Indirect:	1	0	1	0	1	0	0	1	1	See Subsection 4.1.2						
	16-Bit Constant															

Block move data to data with SRC in BMAR

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	1	0	0	0	Data Memory Address						
Indirect:	1	0	1	0	1	1	0	0	1	See Subsection 4.1.2						

Block move data to data with DEST in BMAR

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	1	0	1	0	Data Memory Address						
Indirect:	1	0	1	0	1	1	0	1	1	See Subsection 4.1.2						

Execution

(PFC) → MCS

If long immediate:
(PC) + 2 → PC
#lk → PFC
Else:
(PC) + 1 → PC
(BMAR) → PFC

While (repeat counter) ≠ 0:
(src, addressed by PFC) → dst or src → (dst, addressed by PFC)
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC
(repeat counter) –1 → repeat counter.
(src, addressed by PFC) →dst or src → (dst, addressed by PFC)
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Description

The word in data memory pointed at by *src* is copied to a data memory space pointed at by *dst*. The word of the source and/or destination space can be pointed at with a long immediate value, with the contents of the BMAR register, or by a data memory address. Note that not all src/dst combinations of pointer types are valid.

RPT can be used with the BLDD instruction in indirect addressing mode to move consecutive words in data memory. The number of words to be moved is one greater than the number contained in the repeat counter RPTC at the beginning of the instruction. The source or destination address for the BLDD instruction specified by the long immediate address or BMAR register contents are automatically incremented in repeat mode. If a direct memory address is specified, its address is not automatically incremented in repeat mode. Note that the source and destination blocks do not have to be entirely on-chip or off-chip. Interrupts are inhibited during a *BLDD* operation used with the RPT instruction. When used with RPT, BLDD becomes a single-cycle instruction once the RPT pipeline is started.

CAUTION

Neither the long immediate nor the BMAR can be used as the address to the on-chip memory-mapped registers. The direct or indirect addressing mode can be used to address the on-chip memory-mapped core processor and peripheral registers.

Words

1 (One source or destination is specified by the BMAR register)

2 (One source or destination is specified by a long immediate value)

Cycles

Direct K/DMA: [*label*] **BLDD** #*addr, dma*
Indirect K/DMA: [*label*] **BLDD** #*addr*, {*ind*} [,*next ARP*]
Direct DMA/K: [*label*] **BLDD** *dma*, #*addr*
Indirect DMA/K: [*label*] **BLDD** {*ind*}, #*addr* [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	2	2	2	2+p
Source DARAM Destination DARAM	2	2	2	2+p
Source SARAM Destination DARAM	2	2	2	2+p
Source Ext Destination DARAM	2+d_{src}	2+d_{src}	2+d_{src}	2+d_{src}+p
Source DARAM Destination SARAM	2	2	2 3†	2+p
Source SARAM Destination SARAM	2	2	2 3†	2+p
Source Ext Destination SARAM	2+d_{src}	2+d_{src}	2+d_{src} 3+d_{src}†	2+d_{src}+p
Source DARAM Destination Ext	3+d_{dst}	3+d_{dst}	3+d_{dst}	5+d_{dst}+p
Source SARAM Destination Ext	3+d_{dst}	3+d_{dst}	3+d_{dst}	5+d_{dst}+p
Source Ext Destination Ext	3+d_{src}+d_{dst}	3+d_{src}+d_{dst}	3+d_{src}+d_{dst}	5+d_{src}+d_{dst}+p
Cycle Timings for a Repeat (RPT) Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+1	n+1	n+1	n+1+p
Source SARAM Destination DARAM	n+1	n+1	n+1	n+1+p
Source Ext Destination DARAM	n+1+nd_{src}	n+1+nd_{src}	n+1+nd_{src}	n+1+nd_{src}+p
Source DARAM Destination SARAM	n+1	n+1	n+1 n+3†	n+1+p
Source SARAM Destination SARAM	n+1 2n–1‡	n+1 2n–1‡	n+1 2n–1‡ n+3† 2n+1§	n+1+p 2n–1+p‡

Cycle Timings for a Repeat (RPT) Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source Ext Destination SARAM	$n+1+nd_{src}$†	$n+1+nd_{src}$	$n+1+nd_{src}$ $n+3+nd_{src}$†	$n+1+nd_{src}+p$
Source DARAM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p$
Source SARAM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p$
Source Ext Destination Ext	$4n-1+nd_{src}+nd_{dst}$	$4n-1+nd_{src}+nd_{dst}$	$4n-1+nd_{src}+nd_{dst}$	$4n+1+nd_{src}+nd_{dst}+p$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Direct BMAR/DMA: [*label*] **BLDD BMAR**, *dma*
Indirect BMAR/DMA: [*label*] **BLDD BMAR**, {*ind*} [,*next ARP*]
Direct DMA/BMAR: [*label*] **BLDD** *dma*, **BMAR**
Indirect DMA/BMAR: [*label*] **BLDD** {*ind*}, **BMAR** [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	3	3	3	3+2p
Source SARAM Destination DARAM	3	3	3	3+2p
Source Ext Destination DARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$	$3+d_{src}+2p$
Source DARAM Destination SARAM	3	3	3 4†	3+2p
Source SARAM Destination SARAM	3	3	3 4†	3+2p
Source Ext Destination SARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$ $4+d_{src}$†	$3+d_{src}+2p$
Source DARAM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p$
Source SARAM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p$
Source Ext Destination Ext	$4+d_{src}+d_{dst}$	$4+d_{src}+d_{dst}$	$4+d_{src}+d_{dst}$	$6+d_{src}+d_{dst}+2p$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+2	n+2	n+2	n+2+2p

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination DARAM	n+2	n+2	n+2	n+2+2p
Source Ext Destination DARAM	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src}
Source DARAM Destination SARAM	n+2	n+2	n+2 n+4†	n+2+2p
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ n+4† 2n+2§	n+2+2p 2n+2p‡
Source Ext Destination SARAM	n+2nd_{src}	n+2nd_{src}	n+2nd_{src} n+4+nd_{src}†	n+2+nd_{src}+2p
Source DARAM Destination Ext	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst} +2p
Source SARAM Destination Ext	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}+2p
Source Ext Destination Ext	4n+nd_{src}+nd_{dst}‡	4n+nd_{src}+nd_{dst}	4n+nd_{src}+nd_{dst}	4n+2+nd_{src}+nd_{dst}+2p

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Example 1

```
BLDD   #300h,20h    ;(DP = 6)
```

	Before Instruction		After Instruction
Data Memory 300h	0h	Data Memory 300h	0h
320h	0Fh	320h	0h

Example 2

```
BLDD   *+,#321h,AR3
```

	Before Instruction		After Instruction
ARP	2	ARP	3
AR2	301h	AR2	302h
Data Memory 301h	01h	Data Memory 301h	01h
321h	0Fh	321h	01h

Example 3

```
BLDD   BMAR,*
```

	Before Instruction		After Instruction
ARP	2	ARP	2
BMAR	320h	BMAR	320h
AR2	340h	AR2	340h
Data Memory 320h	01h	Data Memory 320h	01h
340h	0Fh	340h	01h

Example 4

```
BLDD   00h,BMAR   ;(DP = 6)
```

	Before Instruction		After Instruction
Data Memory 300h	0Fh	Data Memory 300h	0Fh
BMAR	320h	BMAR	320h
Data Memory 320h	01h	Data Memory 320h	0Fh

Example 5

```
RPTK   2
BLDD   #300h,*+
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	320h	AR0	323h
300h	7F98h	300h	7F98h
301h	0FFE6h	301h	0FFE6h
302h	9522h	302h	9522h
320h	8DEEh	320h	7F98h
321h	9315h	321h	0FFE6h
322h	2531h	322h	9522h

Syntax

Direct: [*label*] **BLDP** *dma*
Indirect: [*label*] **BLDP** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	1	1	1	0	Data Memory Address						
Indirect:	0	1	0	1	0	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(PFC) → MCS
(BMAR) → PFC

While (repeat counter) ≠ 0:
    dma → (dst, addressed by PFC)
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (repeat counter) –1 → repeat counter.
dma → (dst, addressed by PFC)
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Description

A word in data memory is copied to a word in program memory space pointed at by the BMAR register. The RPT instruction used with the BLDP instruction can move consecutive words pointed at indirectly in data memory to a contiguous program memory space pointed at by the BMAR register. The BMAR register is automatically updated in the repeat mode. Note that the source and destination blocks do **not** have to be entirely on-chip or off-chip. When used with RPT, BLDP becomes a single-cycle instruction once the RPT pipeline is started. Interrupts are inhibited during a BLDP operation used with RPT.

Words

1

Cycles

Direct: [*label*] **BLDP** *dma*
Indirect: [*label*] **BLDP** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction

	PR	PDA	PSA	PE
Source DARAM Destination DARAM	2	2	2	2+p
Source SARAM Destination DARAM	2	2 3¶	2	2+p
Source Ext Destination DARAM	2+d_{src}	2+d_{src}	2+d_{src}	3+d_{src}+p_{code}
Source DARAM Destination SARAM	2	2	2 3†	2+p

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination SARAM	2	2	2 3† or ¶ 4§	2+p
Source Ext Destination SARAM	2+d_{src}	2+d_{src}	2+d_{src} 3+d_{src}†	3+d_{src}+p_{code}
Source DARAM Destination Ext	3+p_{dst}	3+p_{dst}	3+p_{dst}	4+p_{dst}+p_{code}
Source SARAM Destination Ext	3+p_{dst}	3+p_{dst}	3+p_{dst} 4+p_{dst}¶	4+p_{dst}+p_{code}
Source Ext Destination Ext	3+d_{src}+p_{dst}	3+d_{src}+p_{dst}	3+d_{src}+p_{dst}	5+d_{src}+p_{dst}+p_{code}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+1	n+1	n+1	n+1+p_{code}
Source SARAM Destination DARAM	n+1	n+1	n+1 n+2¶	n+1+p_{code}
Source Ext Destination DARAM	n+1+nd_{src}	n+1+nd_{src}	n+1+nd_{src}	n+2+nd_{src}+p_{code}
Source DARAM Destination SARAM	n+1	n+1	n+1 n+2†	n+1+p_{code}
Source SARAM Destination SARAM	n+1 2n–1‡	n+1 2n–1‡	n+1 2n–1‡ n+2† or ¶ 2n+1§	n+1+p_{code} 2n–1+p_{code}‡
Source Ext Destination SARAM	n+1+nd_{src}	n+1+nd_{src}	n+1+nd_{src} n+2+np_{src}†	n+2+nd_{src}+p_{code}
Source DARAM Destination Ext	2n+1+np_{dst}	2n+1+np_{dst}	2n+1+np_{dst}	2n+2+np_{dst}+p_{code}
Source SARAM Destination Ext	2n+1+np_{dst}	2n+1+np_{dst}	2n+1+np_{dst} 2n+2+np_{dst}¶	2n+2+np_{dst}+p_{code}
Source Ext Destination Ext	4n–1+nd_{src}+ np_{dst}	4n–1+nd_{src}+ np_{dst}	4n–1+nd_{src}+ np_{dst}	4n+1+nd_{src}+np_{dst}+p *code*

† If the destination operand and the code are in the same SARAM block.

‡ If both the source and the destination operands are in the same SARAM block.

§ If both operands and the code are in the same SARAM block.

¶ If the source operand and the code are in the same SARAM block.

Example 1

```
BLDP   00h    ;(DP=6)
```

	Before Instruction		After Instruction
Data Memory 300h	0A089h	Data Memory 300h	0A089h
BMAR	2800h	BMAR	2800h
Program Memory 2800h	1234h	Program Memory 2800h	0A089h

Example 2

```
BLDP   *,AR0
```

	Before Instruction		After Instruction
ARP	7	ARP	0
AR7	310h	AR7	310h
Data Memory 310h	0F0F0h	Data Memory 310h	0F0F0h
BMAR	2800h	BMAR	2800h
Program Memory 2800h	1234h	Program Memory 2800h	0F0F0h

Syntax

General syntax: [*label*] **BLPD** *src, dst*

All valid cases have the general syntax:

Direct K/DMA: [*label*] **BLPD** #*pma, dma*
Indirect K/DMA: [*label*] **BLPD** #*pma*, {*ind*} [,*next ARP*]
Direct BMAR/DMA: [*label*] **BLPD BMAR**, *dma*
Indirect BMAR/DMA: [*label*] **BLPD BMAR**, {*ind*} [,*next ARP*]

Operands

0 ≤ pma ≤ 65535
0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	0	1	0	1	0	Data Memory Address						
	16-Bit Constant															

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	0	1	0	1	1	See Subsection 4.1.2						
	16-Bit Constant															

Block move prog to data with source in BMAR

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	0	1	0	0	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	0	1	0	0	1	See Subsection 4.1.2						

Execution

If long immediate:
(PC) + 2 → PC
(PFC) → MCS
lk → PFC
Else:
(PC) + 1 → PC
(PFC) → MCS
(BMAR) → PFC

While (repeat counter) ≠ 0:
(pma, addressed by PFC) → dst
Modify AR(ARP) and ARP as specified,
(PFC) + 1 → PFC
(repeat counter) –1 → repeat counter.

(pma, addressed by PFC) → dst,
Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Description

A word in program memory pointed at by the *src* is copied to data memory space pointed at by *dst*. The first word of the source space can be pointed at with a long immediate value or the contents of the BMAR register. The data memory destination space is always pointed at by a data memory address or

auxiliary register pointer. Note that not all src/dst combinations of pointer types are valid.

RPT can be used with the BLPD instruction if more than one word is to be moved. The number of words to be moved is one greater than the number contained in the repeat counter, RPTC, at the beginning of the instruction. The source address specified by the long immediate or BMAR value is automatically incemented in repeat mode. Note that the source and destination blocks do **not** have to be entirely on-chip or off-chip. Interrupts are inhibited during a repeated BLPD instruction. When used with RPT, BLPD becomes a single-cycle instruction once the RPT pipeline is started.

Words

1 (Source is specified by the BMAR register)

2 (Source is specified by a long immediate)

Cycles

Direct K/DMA: [*label*] **BLPD** #*pma*, *dma*
Indirect K/DMA: [*label*] **BLPD** #*pma*, {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	2	2	2	$2+p_{code}$
Source SARAM Destination DARAM	2	2	2	$2+p_{code}$
Source Ext Destination DARAM	$2+p_{src}$	$2+p_{src}$	$2+p_{src}$	$2+p_{src}+p_{code}$
Source DARAM/ROM Destination SARAM	2	2	2 3†	$2+p_{code}$
Source SARAM Destination SARAM	2	2	2 3†	$2+p_{code}$
Source Ext Destination SARAM	$2+p_{src}$	$2+p_{src}$	$2+p_{src}$ $3+p_{src}$†	$2+p_{src}+2p_{code}$
Source DARAM/ROM Destination Ext	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+p_{code}$
Source SARAM Destination Ext	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+p_{code}$
Source Ext Destination Ext	$3+p_{src}+d_{dst}$	$3+p_{src}+d_{dst}$	$3+p_{src}+d_{dst}$	$5+p_{src}+d_{dst}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Source SARAM Destination DARAM	n+1	n+1	n+1	$n+1+p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source Ext Destination DARAM	$n+1+np_{src}$	$n+1+np_{src}$	$n+1+np_{src}$	$n+1+np_{src}+p_{code}$
Source DARAM/ROM Destination SARAM	$n+1$	$n+1$	$n+1$ $n+3$†	$n+1+p_{code}$
Source SARAM Destination SARAM	$n+1$ $2n-1$‡	$n+1$ $2n-1$‡	$n+1$ $2n-1$‡ $n+3$† $2n+1$§	$n+1+p_{code}$ $2n-1+p_{code}$‡
Source Ext Destination SARAM	$n+1+np_{src}$	$n+1+np_{src}$	$n+1+np_{src}$ $n+3+np_{src}$†	$n+1+np_{src}+p_{code}$
Source DARAM/ROM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p_{code}$
Source SARAM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p_{code}$
Source Ext Destination Ext	$4n-1+np_{src}+nd_{dst}$	$4n-1+np_{src}+nd_{dst}$	$4n-1+np_{src}+nd_{dst}$	$4n+1+np_{src}+nd_{dst}+p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Direct BMAR/DMA: [*label*] **BLPD BMAR**, *dma*
Indirect BMAR/DMA: [*label*] **BLPD BMAR**, {*ind*} [,*next ARP*]

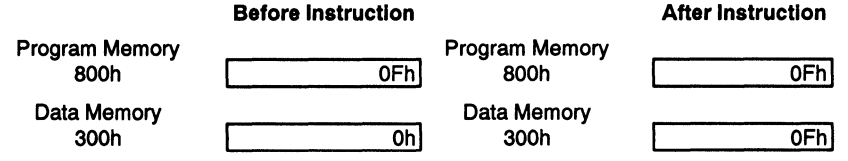
Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	3	3	3	$3+2p_{code}$
Source SARAM Destination DARAM	3	3	3	$3+2p_{code}$
Source Ext Destination DARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$	$3+p_{src}+2p_{code}$
Source DARAM/ROM Destination SARAM	3	3	3 4†	$3+2p_{code}$
Source SARAM Destination SARAM	3	3	3 4†	$3+2p_{code}$
Source Ext Destination SARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$ $4+p_{src}$†	$3+p_{src}+2p_{code}$
Source DARAM/ROM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p_{code}$

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p_{code}$
Source Ext Destination Ext	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$6+p_{src}+d_{dst}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Source SARAM Destination DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Source Ext Destination DARAM	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}+2p_{code}$
Source DARAM/ROM Destination SARAM	n+2	n+2	n+2 n+4†	$n+2+2p_{code}$
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ n+4† 2n+2§	$n+2+2p_{code}$ $2n+2p_{code}$‡
Source Ext Destination SARAM	$n+2+np_{src}$†	$n+2+np_{src}$	$n+2+np_{src}$ $n+4+np_{src}$†	$n+2+np_{src}+2p_{code}$
Source DARAM/ROM Destination Ext	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}+2p_{code}$
Source SARAM Destination Ext	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}+2p_{code}$
Source Ext Destination Ext	$4n+np_{src}+nd_{dst}$†	$4n+np_{src}+nd_{dst}$	$4n+np_{src}+nd_{dst}$	$4n+2+np_{src}+nd_{dst}+2p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Example 1

```
BLPD   #800h,00h ;(DP=6)
```



Example 2

```
BLPD   #800h,*,AR7
```

	Before Instruction		After Instruction
ARP	0	ARP	7
AR0	310h	AR0	310h
Program Memory 800h	1111h	Program Memory 800h	1111h
Data Memory 310h	0100h	Data Memory 310h	1111h

Example 3

```
BLPD   BMAR,00h ;(DP=6)
```

	Before Instruction		After Instruction
BMAR	800h	BMAR	800h
Program Memory 800h	0Fh	Program Memory 800h	0Fh
Data Memory 300h	0h	Data Memory 300h	0Fh

Example 4

```
BLPD   BMAR,*+,AR7
```

	Before Instruction		After Instruction
ARP	0	ARP	7
AR0	300h	AR0	301h
BMAR	810h	BMAR	810h
Program Memory 810h	4444h	Program Memory 810h	4444h
Data Memory 300h	0100h	Data Memory 300h	4444h

Syntax [*label*] **BSAR** *shift*

Operands 1 ≤ shift ≤ 16

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	SHFT †			

† See Section 4.5.

Execution

(PC) + 1 → PC
(ACC) / 2^{shift} → ACC

Affected by SXM.

Description The BSAR instruction executes a 1- to 16-bit right-barrel arithmetic shift of the accumulator in a single cycle. The sign extension is determined by the sign-extension mode bit in status register 1 (ST1).

Words 1

Cycles [*label*] **BSAR** *shift*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
BSAR 16          ;(SXM=0)
```

	Before Instruction		After Instruction
ACC	00010000h	ACC	00000001h

Example 2

```
BSAR 4           ;(SXM=1)
```

	Before Instruction		After Instruction
ACC	0FFF10000h	ACC	0FFFF1000h

Syntax [*label*] **CALA**[*D*]

Operands None

Opcode

CALA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0

CALLD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	1	1	0	1

Execution

Nondelayed: PC + 1 → TOS
Delayed: PC + 3 → TOS
ACC(15–0) → PC

Description The current program counter (PC) is incremented and pushed onto the top of the stack (TOS). Then, the contents of the lower half of the accumulator are loaded into the PC. Execution continues at this address. If the call is a delayed call (specified by the D suffix), the one two-word instruction or two one-word instructions following the call instruction are fetched from program memory and executed before the call is executed.

The CALA instruction is used to perform computed subroutine calls.

Words 1

Cycles [*label*] **CALA**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken these two instruction words are discarded.

[*label*] **CALAD**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
CALA
```

	Before Instruction		After Instruction
PC	25h	PC	83h
ACC	83h	ACC	83h
TOS	100h	TOS	26h

Example 2

```
CALAD
MAR *+,AR1
LDP #5
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	8	AR0	9
DP	0	DP	5
PC	25h	PC	83h
ACC	83h	ACC	83h
TOS	100h	TOS	28h

After the current AR, ARP, and DP are modified as specified, the address of the instruction following the LDP instruction is pushed onto the stack, and program execution continues from location 83h.

Syntax [*label*] **CALL**[*D*] *pma* [,{*ind*} [,*next ARP*]]

Operands 0 ≤ pma ≤ 65535
0≤ next ARP ≤ 7

Opcode

CALL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

CALLD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution
Nondelayed: PC + 2 → TOS
Delayed: PC + 4 → TOS
pma → PC
Modify AR(ARP) and ARP as specified.

Description The current program counter (PC) is incremented and pushed onto the top of the stack (TOS). Then, the contents of the program memory address (pma), either a symbolic or numeric address, are loaded into the PC. Execution continues at this address. The current auxiliary register and ARP are modified as specified. If the call is a delayed call (specified by the "D" suffix), the one two-word instruction or two one-word instructions following the call instruction are fetched from program memory and executed before the call is executed.

Words 2

Cycles [*label*] **CALL** *pma* [,{*ind*} [,*next ARP*]]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+4p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Cycles [*label*] **CALLD** *pma* [,{*ind*} [,*next ARP*]]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
CALL   PRG191,*+,AR0
```

	Before Instruction		After Instruction
ARP	1	ARP	0
AR1	05h	AR1	06h
PC	30h	PC	0BFh
TOS	100h	TOS	32h

0BFh is loaded into the program counter, and the program continues executing from that location.

Example 2

```
CALLD  PRG191
MAR    *+,AR1
LDP    #5
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	09h	AR0	0Ah
DP	1	DP	5
PC	30h	PC	0BFh
TOS	100h	TOS	34h

After the current AR, ARP, and DP are modified as specified, the address of the instruction following the LDP instruction is pushed onto the stack, and program execution continues from location 0BFh.

Syntax

[*label*] **CC**[*D*] *pma* [*cond1*] [,*cond2*] [,...]

Operands

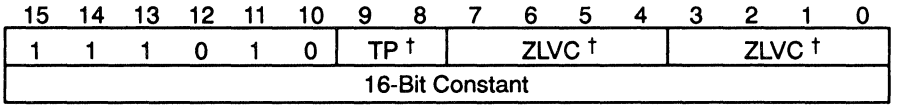
0 ≤ pma ≤ 65535

Conditions:

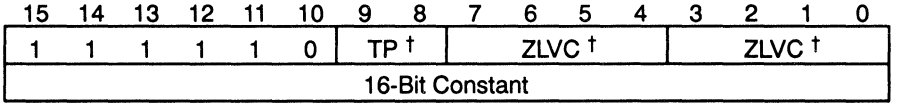
ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
TC=0	NTC
TC=1	TC
$\overline{\text{BIO}}$ low	BIO
Unconditionally	UNC

Opcode

CC



CCD



† See Section 4.5.

Execution

```
If(condition(s))
    Then
        Nondelayed: PC + 2 → TOS
        Delayed:  PC + 4 → TOS
        pma → PC
    Else
        PC + 2 → PC
```

Description

Control is passed to the program memory address pma if the specified conditions are met. Note that not all combinations of conditions are meaningful. In addition, the NTC, TC, and BIO conditions are mutually exclusive. If the call is a delayed call (specified by the "D" suffix), the two one-word instructions or the one two-word instruction following the call are fetched from program memory and executed before the call is executed. The CC instruction operates like the CALL instruction if all conditions are true.

Words

2

Cycles

[*label*] **CC** *pma* [*cond1*] [,*cond2*] [,...]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	4	4	4	4+4p†
Condition False	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken these two instruction words are discarded.

[*label*] **CCD** *pma* [*cond1*] [,*cond2*] [,...]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	2	2	2	2+2p
Condition False	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

Example 1

```
CC      PGM191,LEQ,C
```

If the accumulator contents are less than or equal to zero and the carry bit is set, 0BFh is loaded into the program counter, and the program continues executing from that location. If the conditions are not met, execution continues at the instruction following the CC instruction.

Example 2

```
CCD     PGM191,LEQ,C
MAR *+,AR1
LDP #5
```

The current AR, ARP, and DP are modified as specified. If the accumulator contents are less than or equal to zero and the carry bit is set, the address of the instruction following the LDP instruction is pushed onto the stack and program execution continues from location 0BFh. If the conditions are not met, execution continues at the instruction following the LDP instruction.

Syntax [*label*] **CLRC** *control bit*

Operands *Control bit*: ST0, ST1 bit (from: {C, CNF, HM, INTM, OVM, TC, SXM, XF})

Opcode

Reset overflow mode (OVM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	0	1	0

Reset sign extension mode (SXM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	0

Reset hold mode (HM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	0	0	0

Reset TC bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	0	1	0

Reset carry (C)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	1	1	0

Reset CNF bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	1	0	0

Reset INTM bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0

Reset XF pin

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	1	0	0

Execution (PC) + 1 → PC
0 → control bit

Description The specified control bit is set to a logic zero. Note that the LST instruction may also be used to load ST0 and ST1. See subsection 3.6.3, *Status and Control Registers*, for more information on each of these control bits.

Words 1

Cycles [*label*] **CLRC** *control bit*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
CLRC TC    ;TC is bit 11 of ST1
```

	Before Instruction		After Instruction
ST1	x9xxh	ST1	x1xxh

Syntax [*label*] **CMPL**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1

Execution (PC) + 1 → PC
$(\overline{ACC})$ → ACC

Description The contents of the accumulator are replaced with its logical inversion (ones complement). The carry bit is unaffected.

Words 1

Cycles [*label*] **CMPL**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
CMPL
```

Syntax [*label*] **CMPR** *constant*

Operands $0 \le CM \le 3$

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	1	CM †	

† See Section 4.5.

Execution (PC) + 1 → PC
Compare AR(ARP) to ARCR, placing result in TC bit of status register ST1.

Affects TC; affected by NDX.
Not affected by SXM; does not affect SXM.

Description The CMPR instruction performs a comparison specified by the value of CM:

If CM = 00, test if AR(ARP) = ARCR
If CM = 01, test if AR(ARP) < ARCR
If CM = 10, test if AR(ARP) > ARCR
If CM = 11, test if AR(ARP) ≠ ARCR

If the condition is true, the TC bit is set to 1. If the condition is false, the TC bit is set to 0.

Software compatibility with 'C25 can be maintained by resetting the NDX bit in the PMST register to 0. This causes any 'C25 instruction that loads auxiliary register 0 (AR0) to load the ARCR register also. This allows source-code compatibility with the 'C25. Note that the auxiliary registers are treated as unsigned integers in the comparisons.

Words 1

Cycles [*label*] **CMPR** *constant*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
CMPR  2
```

	Before Instruction		After Instruction
ARP	4	ARP	1
ARCR	0FFFFh	ARCR	0FFFFh
AR4	7FFFh	AR4	7FFFh
TC	1	TC	0

Syntax

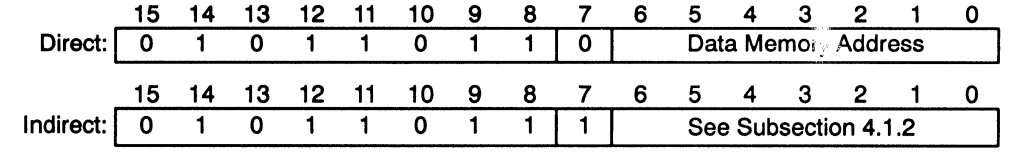
Direct: [*label*] **CPL** [,#*lk*] *dma*
Indirect: [*label*] **CPL** [,#*lk*] {*ind*} [,*next ARP*]

Operands

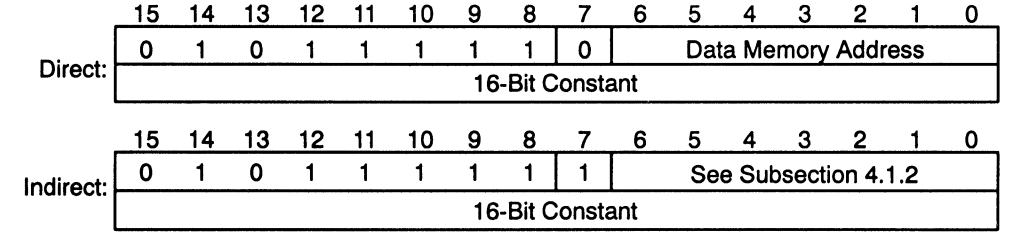
0 ≤ dma ≤ 127
lk: 16-bit constant
0 ≤ next ARP ≤ 7

Opcode

Compare DBMR to data value



Compare data with long immediate



Execution

lk unspecified:
(PC) + 1 → PC
Compare DBMR contents to (dma).
If (DBMR) = (dma),
    TC = 1;
Else,
    TC = 0.

lk specified:
(PC) + 2 → PC

Compare lk to (dma).
If lk = (dma),
    TC = 1;
Else
    TC = 0.

Affects TC.
Not affected by SXM.

Description

If the two quantities involved in the comparison are equal, the TC bit is set to one. TC is set to zero otherwise.

Words 1 (If long immediate value is not specified)

Words 2 (If long immediate value is specified)

Cycles

Direct: [*label*] **CPL** *dma*
Indirect: [*label*] **CPL** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Cycles

Direct: [*label*] **CPL** *#lk dma*
Indirect: [*label*] **CPL** *#lk* {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2 3†	2+2p
Operand Ext	2+d	2+d	2+d	3+d+2p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n+1	n+1	n+1	n+1+2p
Operand SARAM	n+1	n+1	n+1 n+2†	n+1+2p
Operand Ext	n+1	n+1	n+1	n+2+2p

† If the operand and the code are in the same SARAM block.

Example 1

```
CPL    #060h,60h
```

	Before Instruction		**After Instruction**
Data Memory 60h	066h	Data Memory 60h	066h
TC	1	TC	0

Example 2

```
CPL   60h
```

	Before Instruction	After Instruction
Data Memory 60h	066h	066h
DBMR	066h	066h
TC	0	1

Example 3

```
CPL   #0F1h,*,AR6
```

	Before Instruction	After Instruction
ARP	7	6
AR7	300h	300h
Data Memory 300h	0F1h	0F1h
TC	1	1

Example 4

```
CPL   *,AR7
```

	Before Instruction	After Instruction
ARP	6	7
AR6	300h	300h
Data Memory 300h	0F1h	0F1h
DBMR	0F0h	0F0h
TC	0	0

Syntax [*label*] **CRGT**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	1	1

Execution

(PC) + 1 → PC
If (ACC) > (ACCB)
  Then (ACC) → ACCB; 1 → C
If (ACC) < (ACCB)
  Then (ACCB) → ACC; 0 → C
If (ACC) = (ACCB)
  Then 1 → C

Affects C.

Description The contents of the accumulator (ACC) are compared to the contents of the accumulator buffer (ACCB). The larger value (signed) is loaded into both registers. If the contents of the accumulator are greater than or equal to the contents of the accumulator buffer, the carry bit is set to 1. Otherwise, it is set to 0.

Words 1

Cycles [*label*] **CRGT**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
CRGT
```

	Before Instruction		After Instruction
ACCB	4h	ACCB	5h
ACC	5h	ACC	5h
C	0	C	1

Example 2

```
CRGT
```

	Before Instruction		After Instruction
ACCB	5h	ACCB	5h
ACC	5h	ACC	5h
C	0	C	1

Syntax [*label*] **CRLT**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	1	1

Execution

(PC) + 1 → PC
If (ACC) < (ACCB)
  Then (ACC) → ACCB; 1 → C
If (ACC) > (ACCB)
  Then (ACCB) → ACC; 0 → C
If (ACC) = (ACCB)
  Then 0 → C

Affects C.

Description The contents of the accumulator (ACC) are compared to the contents of the accumulator buffer (ACCB). The smaller (signed) value is loaded into both registers. If the contents of the accumulator are less than the contents of the accumulator buffer, the carry bit is set to 1. Otherwise it is set to 0.

Words 1

Cycles [*label*] **CRLT**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
CRLT
```

	Before Instruction		After Instruction
ACCB	5h	ACCB	4h
ACC	4h	ACC	4h
C	0	C	1

Example 2

```
CRLT
```

	Before Instruction		After Instruction
ACCB	4h	ACCB	4h
ACC	4h	ACC	4h
C	1	C	0

Syntax

Direct: [*label*] **DMOV** *dma*
Indirect: [*label*] **DMOV** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	1	1	1	0	Data Memory Address						
Indirect:	0	1	1	1	0	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → dma + 1

Affected by CNF and OVLY.

Description

The contents of the specified data memory address are copied into the contents of the next higher address. DMOV works only within on-chip data RAM blocks. It works within any configurable RAM block if that block is configured as data memory. In addition, the data move function is continuous across block boundaries. The data move function cannot be used on external data memory or memory-mapped registers. If used on external memory or memory-mapped registers, DMOV will read the specified memory location but will perform no operations.

When data is copied from the addressed location to the next higher location, the contents of the addressed location remain unaltered.

The data move function is useful in implementing the z^{-1} delay encountered in digital signal-processing. The DMOV function is included in the LTD, MACD, and MADD instructions (see the LTD, MACD, and MADD instructions for more information).

Words

1

Cycles

Direct: [*label*] **DMOV** *dma*
Indirect: [*label*] **DMOV** {*ind*} [,*next ARP*

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 3†	1+p
Operand Ext	2+2d	2+2d	2+2d	5+2d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	2n–2	2n–2	2n–2 2n+1†	2n–2+p
Operand Ext	4n–2+2nd	4n–2+2nd	4n–2+2nd	4n+1+2nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
DMOV   DAT8   ;(DP = 6)
```

	Before Instruction		After Instruction
Data Memory 308h	43h	Data Memory 308h	43h
Data Memory 309h	2h	Data Memory 309h	43h

Example 2

```
DMOV   *,AR1
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR1	30Ah	AR1	30Ah
Data Memory 30Ah	40h	Data Memory 30Ah	40h
Data Memory 30Bh	41h	Data Memory 30Bh	40h

Syntax [*label*] **EXAR**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	1	0	1

Execution

(PC) + 1 → PC
(ACCB) ↔ (ACC)

Description The contents of the accumulator is exchanged (switched) with the contents of the accumulator buffer (ACCB).

Words 1

Cycles [*label*] **EXAR**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
EXAR
```

	Before Instruction		After Instruction
ACC	043h	ACC	02h
ACCB	02h	ACCB	043h

Syntax [*label*] **IDLE**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	1	0

Execution (PC) + 1 → PC

Affected by INTM.

Description The IDLE instruction forces the program being executed to wait until an unmasked interrupt (external or internal) or reset occurs. The PC is incremented only once, and the device remains in an idle state until interrupted.

The idle state is exited by an unmasked interrupt even if INTM is 1. If INTM is 1, the program scontinue executing at the instruction following the IDLE. If INTM is 0, the program branches to the corresponding interrupt service routine. Execution of the IDLE instruction causes the 'C5x to enter the power-down mode. During the idle mode, the timer and serial port peripherals are still active. Therefore, timer and peripheral interrupts, as well as reset or external interrupts, will remove the processor from the idle mode.

Words 1

Cycles [*label*] **IDLE**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
IDLE   ;The processor idles until a reset or unmasked
       ;interrupt occurs.
```

Syntax [*label*] **IDLE2**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	1	1

Execution (PC) + 1 → PC

Affected by INTM.

Description The IDLE2 instruction removes the functional clock input from the internal device. This allows for an extremely low power mode. The PC is incremented only once, and the device remains in an idle state until interrupted by reset or an unmasked interrupt.

The low power mode is exited by an unmasked interrupt even if INTM is high. If INTM is high, the program continues executing at the instruction following the IDLE2. If INTM is low, then the program branches to the corresponding interrupt service routine. Execution of the IDLE2 instruction causes the 'C5x to enter the power-down mode. Unlike the idle mode, in the idle2 mode the peripherals (serial ports or timer) are not active.

The idle2 mode is exited by a low logic level on an external interrupt ($\overline{INT1}$–$\overline{INT4}$), $\overline{RS}$, or NMI with a duration of at least five machine cycles since interrupts are not latched as in normal device operation.

Words 1

Cycles [*label*] **IDLE2**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
IDLE2  ;The processor idles until a reset or unmasked external
       ;interrupt occurs.
```

Syntax

Direct: [*label*] **IN** *dma* , *PA*
Indirect: [*label*] **IN** {*ind*} ,*PA* [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤7
0 ≤ **PA** ≤ 65535

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	1	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	1	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
While (repeat counter) ≠ 0
  Port address → address bus **A15–A0**
  Data bus **D15–D0** → dma
  Port address → dma
  Port address + 1 → Port address
  (repeat counter – 1) → repeat counter

Description

The IN instruction reads a 16-bit value from an external I/O port into the specified data memory location. The $\overline{IS}$ line goes low to indicate an I/O access, and the $\overline{STRB}$, $\overline{RD}$, and READY timings are the same as for an external data memory read. Note that port addresses 50h–5Fh are memory-mapped (see subsection 5.1.1), but the other port addresses are not.

RPT can be used with the IN instruction to read in consecutive words from I/O space to data space. In the repeat mode, the port address (PA) is incremented after each access.

Words

2

Cycles

Direct: [*label*] **IN** *dma* , *PA*
Indirect: [*label*] **IN** {*ind*} ,*PA* [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Destination DARAM	$2+io_{src}$	$2+io_{src}$	$2+io_{src}$	$3+io_{src}+2p_{code}$
Destination SARAM	$2+io_{src}$	$2+io_{src}$	$2+io_{src}$ $3+io_{src}$ †	$3+io_{src}+2p_{code}$
Destination Ext	$3+d_{dst}+io_{src}$	$3+d_{dst}+io_{src}$	$3+d_{dst}+io_{src}$	$6+d_{dst}+io_{src}+2p_{code}$

Cycle Timings for a Repeat (RPT) Execution				
Destination DARAM	2n+nio$_{src}$	2n+nio$_{src}$	2n+nio$_{src}$	2n+1+nio$_{src}$+2p$_{code}$
Destination SARAM	2n+nio$_{src}$	2n+nio$_{src}$	2n+nio$_{src}$ 2n+2+nio$_{src}$†	2n+1+nio$_{src}$+2p$_{code}$
Destination Ext	4n–1+nd$_{dst}$+ nio$_{src}$	4n–1+nd$_{dst}$+ nio$_{src}$	4n–1+nd$_{dst}$+ nio$_{src}$	4n+2+nd$_{dst}$+nio$_{src}$+ 2p$_{code}$

† If the destination operand and the code are in the same SARAM block.

Example 1

```
IN    DAT7,PA5    ;Read in word from peripheral on port
                  ;address 5. Store in data memory location
                  ;307h (DP=6).
```

Example 2

```
IN    *,PA0       ;Read in word from peripheral on port
                  ;address 0. Store in data memory location
                  ;specified by current auxiliary register.
```

Syntax

[*label*] **INTR** *k*

Operands

$0 \le k \le 31$

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	1	INTR# †				

† See Section 4.5.

Execution

(PC) + 1 → stack
corresponding *interrupt* vector → PC

Not affected by INTM.
Affects INTM.

Description

The INTR instruction is a software interrupt that transfers program control to the program memory address specified by k (see the following table). The instruction allows any interrupt service routine to be executed from your software. During execution of the instruction, the contents of PC + 1 is pushed onto the stack. Note that the interrupt mask has no effect on the INTR instruction. An INTR interrupt for the external interrupts ($\overline{INT1}$–$\overline{INT4}$) looks exactly like an external interrupt (an interrupt acknowledge is generated, the appropriate bit in the IFR is cleared, interrupts are globally disabled (INTM = 1), and context is automatically saved). See subsection 5.1.2 for a complete description of interrupt operation.

k	Interrupt	Location	k	Interrupt	Location
0	$\overline{RS}$	0h	16	Reserved	20h
1	$\overline{INT1}$	2h	17	TRAP	22h
2	$\overline{INT2}$	4h	18	$\overline{NMI}$	24h
3	$\overline{INT3}$	6h	19	Reserved	26h
4	TINT	8h	20	User-defined	28h
5	RINT	Ah	21	User-defined	2Ah
6	XINT	Ch	22	User-defined	2Ch
7	TRNT	Eh	23	User-defined	2Eh
8	TXNT	10h	24	User-defined	30h
9	$\overline{INT4}$	12h	25	User-defined	32h
10	Reserved	14h	26	User-defined	34h
11	Reserved	16h	27	User-defined	36h
12	Reserved	18h	28	User-defined	38h
13	Reserved	1Ah	29	User-defined	3Ah
14	Reserved	1Ch	30	User-defined	3Ch
15	Reserved	1Eh	31	User-defined	3Eh

CAUTION

The reserved interrupt vectors may be used for the 'C50, 'C51, and 'C53. However, software compatibility with future fifth generation devices is not guaranteed.

Words 1

Cycles [*label*] **INTR** *k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example

```
INTR   3    ;Control is passed to program memory location 6h
            ;PC + 1 is pushed onto the stack.
```

Syntax [*label*] **LACB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	1	1	1

Execution

(PC) + 1 → PC
(ACCB) → ACC

Description The accumulator is loaded with the contents of the accumulator buffer (ACCB).

Words 1

Cycles [*label*] **LACB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
LACB
```

	Before Instruction		After Instruction
ACC	01376h	ACC	5555AAAAh
ACCB	5555AAAAh	ACCB	5555AAAAh

Syntax

Direct: [*label*] **LACC** *dma* [,*shift1*]
Indirect: [*label*] **LACC** {*ind*} [,*shift1* [,*next ARP*]]
Immediate: [*label*] **LACC** *#lk* [,*shift2*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ shift1 ≤ 16 (defaults to 0)
–32768 ≤ lk ≤ 32767
0 ≤ shift2 ≤ 15 (defaults to 0)

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	1	SHFT†				0	Data Memory Address						
Indirect:	0	0	0	1	SHFT†				1	See Subsection 4.1.2						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	1	1	0	0	0	SHFT †			
	16-Bit Constant															

Load ACC with shift of 16

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	1	0	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	1	0	1	See Subsection 4.1.2						

† See Section 4.5.

Execution

Direct or Indirect Addressing:

(PC) + 1 → PC
(dma) × 2^{shift1} → ACC

Long Immediate Addressing:
(PC) + 2 → PC
lk × 2^{shift2} → ACC

Affected by SXM.

Description

The contents of the specified data memory address or a 16-bit constant are left-shifted and loaded into the accumulator. During shifting, low-order bits are zero-filled. High-order bits are sign-extended if SXM = 1 and zeroed if SXM = 0.

Words

1 (Direct or indirect addressing)

2 (Long immediate addressing)

Cycles

Direct: [*label*] **LACC** *dma* [,*shift1*]
Indirect: [*label*] **LACC** {*ind*} [,*shift1* [,*next ARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

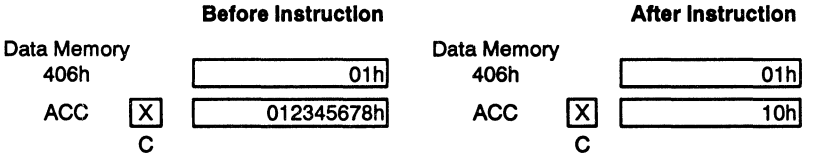
† If the operand and the code are in the same SARAM block.

Immediate: [*label*] **LACC** *#lk* [,*shift2*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

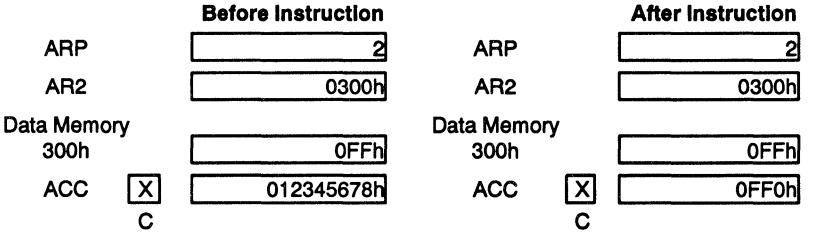
Example 1

```
LACC   DAT6,4 ;(DP = 8. SXM = 0)
```



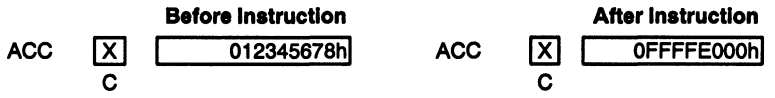
Example 2

```
LACC   *,4     ;(SXM = 0)
```



Example 3

```
LACC   #F000h,1 ;(SXM = 1)
```



Syntax

Direct: [*label*] **LACL** *dma*
Indirect: [*label*] **LACL** {*ind*} [,*next ARP*]
Immediate: [*label*] **LACL** *#k*

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7
$0 \le k \le 255$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	0	1	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	0	1	1	See Subsection 4.1.2						
Short Immediate:	1	0	1	1	1	0	0	1	8-Bit Constant							

Execution

(PC) + 1 → PC

Direct or Indirect Addressing:

0 → ACC(31–16)
(dma) → ACC(15–0)

Short Immediate Addressing:

0 → ACC(31–8)
k → ACC(7–0)

Not affected by SXM.

Description

The contents of the addressed data memory location or a zero-extended 8-bit constant are loaded into the 16 low-order bits of the accumulator. The upper half of the accumulator is zeroed. The data is treated as an unsigned 16-bit number rather than a 2s-complement number. There is no sign-extension of the operand with this instruction, regardless of the state of SXM.

Words

1

Cycles

Direct: [*label*] **LACL** *dma*
Indirect: [*label*] **LACL** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Immediate: [*label*] **LACL** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
LACL   DAT1   ;(DP = 6)
```

	Before Instruction	After Instruction
Data Memory 301h	0h	0h
ACC	7FFFFFFFh	0h
X		
C		

Example 2

```
LACL *-,AR4
```

	Before Instruction	After Instruction
ARP	0	4
AR0	401h	400h
Data Memory 401h	00FFh	00FFh
ACC	7FFFFFFFh	0FFh
X		
C		

Example 3

```
LACL #10h
```

	Before Instruction	After Instruction
ACC	7FFFFFFFh	010h
X		
C		

Syntax

Direct: [*label*] **LACT** *dma*
Indirect: [*label*] **LACT** {*ind*} [,*next ARP*]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	1	1	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) × $2^{\text{TREG1}(3\text{–}0)}$ → ACC

If SXM = 1:
Then (dma) is sign-extended.
If SXM = 0:
Then (dma) is not sign-extended.

Affected by SXM.

Description

The LACT instruction loads the accumulator with a data memory value that has been left-shifted. The left-shift is specified by the four LSBs of TREG1, resulting in shift options from 0 to 15 bits. Using TREG1's contents as a shift code provides a dynamic shift mechanism. During shifting, the high-order bits are sign-extended if SXM = 1 and zeroed if SXM = 0.

LACT may be used to denormalize a floating-point number if the actual exponent is placed in the four LSBs of the T register and the mantissa is referenced by the data memory address. Note that this method of denormalization can be used only when the magnitude of the exponent is four bits or less.

Software compatibility with the 'C25 can be maintained by setting the TRM bit of the PMST status register to zero. This causes any 'C25 instruction that loads TREG0 to write to all three TREGs. Subsequent calls to LACT will contain the correct shift value in TREG1, maintaining object-code compatibility.

Words

1

Cycles

Direct: [*label*] **LACT** *dma*
Indirect: [*label*] **LACT** {*ind*} [,*next ARP*]

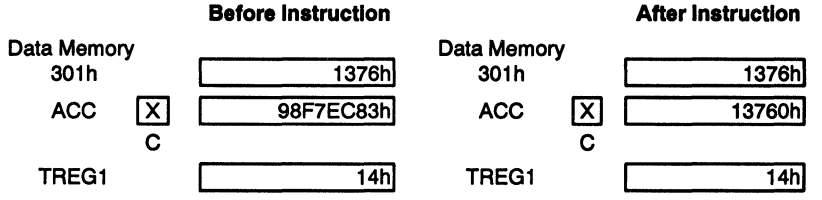
Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 $2^{\dagger}$	1+p
Operand Ext	1+d	1+d	1+d	2+d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

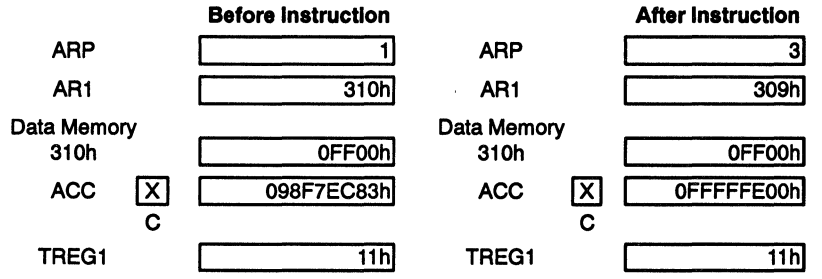
Example 1

```
LACT   DAT1   ;(DP = 6. SXM = 0)
```



Example 2

```
LACT   *-,AR3  ;(SXM = 1)
```



Syntax

Direct: [*label*] **LAMM** *dma*
Indirect: [*label*] **LAMM** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	0	0	0	0	Data Memory Address						
Indirect:	0	0	0	0	1	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → ACC

Not affected by SXM.

Description

The lower half of the accumulator is loaded with the contents of the addressed memory-mapped register. The upper half of the accumulator is zeroed. The 9 MSBs of the data memory address are set to zero, regardless of the current value of DP or the upper 9 bits of AR(ARP). This instruction allows any location on data page zero to be loaded into the accumulator without modifying the DP field in status register ST0.

Words

1

Cycles

Direct: [*label*] **LAMM** *dma*
Indirect: [*label*] **LAMM** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand MMR†	1	1	1	1+p
Operand MMPORT	$1+io_{src}$	$1+io_{src}$	$1+iod_{src}$	$1+2+p+iod_{src}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand MMR‡	n	n	n	n+p
Operand MMPORT	$n+mio_{src}$	$n+mio_{src}$	$n+mio_{src}$	$n+p+mio_{src}$

† Add one more cycle for peripheral memory mapped access.
‡ Add *n* more cycles for peripheral memory mapped access.

Example 1

```
LAMM BMAR      ;(DP = 6)
```

	Before Instruction		After Instruction
ACC	22221376h	ACC	5555h
BMAR	5555h	BMAR	5555h
Data Memory 31Fh	1000h	Data Memory 31Fh	1000h

Example 2

```
LAMM  *
```

	Before Instruction		After Instruction
ARP	1	ARP	1
AR1	325h	AR1	325h
ACC	22221376h	ACC	0Fh
PRD	0Fh	PRD	0Fh
Data Memory 325h	1000h	Data Memory 325h	1000h

Note that the value in data memory location 325h is not loaded into the accumulator. The value at data memory location 25h (address of the PRD register) is loaded.

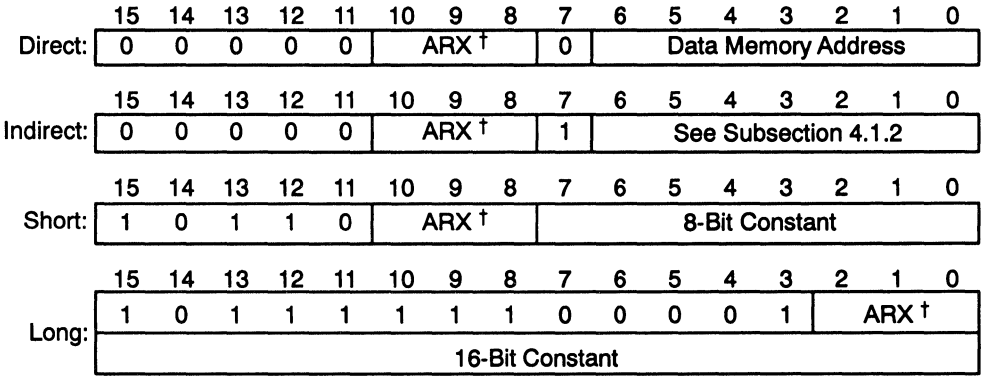
Syntax

Direct:	[*label*] **LAR** *AR, dma*
Indirect:	[*label*] **LAR** *AR, {ind}* [*,next ARP*]
Short Immediate	[*label*] **LAR** *AR, #k*
Long Immediate	[*label*] **LAR** *AR, #lk*

Operands

0 ≤ dma ≤ 127
0 ≤ auxiliary register AR ≤ 7
0 ≤ next ARP ≤ 7
0 ≤ k ≤ 255
0 ≤ lk ≤ 65535

Opcode



† See Section 4.5.

Execution

Direct or Indirect Addressing:

(PC) + 1 → PC
(dma) → auxiliary register AR

Short Immediate Addressing:

(PC) + 1 → PC
k → auxiliary register AR

Long Immediate Addressing:

(PC) + 2 → PC
lk → auxiliary register AR

Affected by NDX.

Description

The contents of the specified data memory address or an 8-bit or 16-bit constant are loaded into the designated auxiliary register (AR). The specified constant is acted upon like an unsigned integer, regardless of the value of SXM. If the NDX bit of the PMST register is 0, then ARCR and INDX registers are also loaded to maintain compatibility with the 'C2x.

The LAR and SAR (store auxiliary register) instructions can be used to load and store the auxiliary registers during subroutine calls and interrupts. If an auxiliary register is not being used for indirect addressing, LAR and SAR en-

able the register to be used as an additional storage register, especially for swapping values between data memory locations without affecting the contents of the accumulator.

Words 1 (Direct, indirect, or short immediate addressing)

2 (Long immediate addressing)

Cycles Direct: [*label*] **LAR** *AR, dma*

Indirect: [*label*] **LAR** *AR, {ind}* [*,next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	2	2	2	$2+p_{code}$
Source SARAM	2	2	2 3†	$2+p_{code}$
Source Ext	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
Source DARAM	2n	2n	2n	$2n+p_{code}$
Source SARAM	2n	2n	2n 2n+1†	$2n+p_{code}$
Source Ext	$2n+nd_{src}$	$2n+nd_{src}$	$2n+nd_{src}$	$2n+1+nd_{src}+p_{code}$

† If the source operand and the code are in the same SARAM block.

Short Immediate [*label*] **LAR** *AR, #k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	$2+p_{code}$
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Long Immediate [*label*] **LAR** *AR, #lk*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
LAR    AR0,DAT16 ;(DP = 6)
```

	Before Instruction		**After Instruction**
Data Memory 310h	18h	Data Memory 310h	18h
AR0	6h	AR0	18h

Example 2

```
LAR   AR4,*-
```

	Before Instruction		After Instruction
ARP	4	ARP	4
Data Memory 300h	32h	Data Memory 300h	32h
AR4	300h	AR4	32h

Note:

LAR in the indirect addressing mode ignores any AR modifications if the AR specified by the instruction is the same as that pointed to by the ARP. Therefore, in Example 2, AR4 is not decremented after the LAR instruction.

Example 3

```
LAR   AR4,#01h
```

	Before Instruction		After Instruction
AR4	0FF09h	AR4	01h

Example 4

```
LAR   AR4,#3FFFh
```

	Before Instruction		After Instruction
AR4	0h	AR4	3FFFh

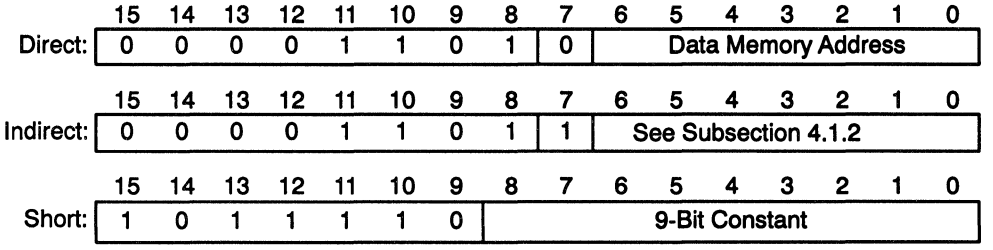
Syntax

Direct:	[*label*]	**LDP** *dma*
Indirect:	[*label*]	**LDP** {*ind*} [,*next ARP*]
Short Immediate:	[*label*]	**LDP** *#k*

Operands

$0 \le dma \le 127$
$0 \le next\ ARP \le 7$
$0 \le k \le 511$

Opcode



Execution

(PC) + 1 → PC

Direct or Indirect Addressing:
Nine LSBs of (dma) → data page pointer (DP) status bits

Short Immediate Addressing:
k → data page pointer register (DP) status bits

Affects DP.

Description

The nine LSBs of the contents of the addressed data memory location or a 9-bit immediate value are loaded into the DP register. The DP and 7-bit data memory address are concatenated to form 16-bit data memory addresses. The DP can also be loaded by the LST instruction.

Words

1

Cycles

Direct:	[*label*]	**LDP** *dma*
Indirect:	[*label*]	**LDP** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction

	PR	PDA	PSA	PE
Source DARAM	2	2	2	$2+p_{code}$
Source SARAM	2	2	2 3†	$2+p_{code}$
Source Ext	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	2n	2n	2n	2n+p_{code}
Source SARAM	2n	2n	2n 2n+1†	2n+p_{code}
Source Ext	2n+nd_{src}	2n+nd_{src}	2n+nd_{src}	2n+1+nd_{src}+p_{code}

† If the source operand and the code are in the same SARAM block.

Short Immediate: [*label*] **LDP** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+p_{code}
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
LDP    DAT127  ;(DP = 511)
```

	Before Instruction		After Instruction
Data Memory 0FFFFh	0FEDCh	Data Memory 0FFFFh	0FEDCh
DP	1FFh	DP	0DCh

Example 2

```
LDP    #0h
```

	Before Instruction		After Instruction
DP	1FFh	DP	0h

Example 3

```
LDP    *,AR5
```

	Before Instruction		After Instruction
ARP	4	ARP	5
AR4	300h	AR4	300h
Data Memory 300h	06h	Data Memory 300h	06h
DP	1FFh	DP	06h

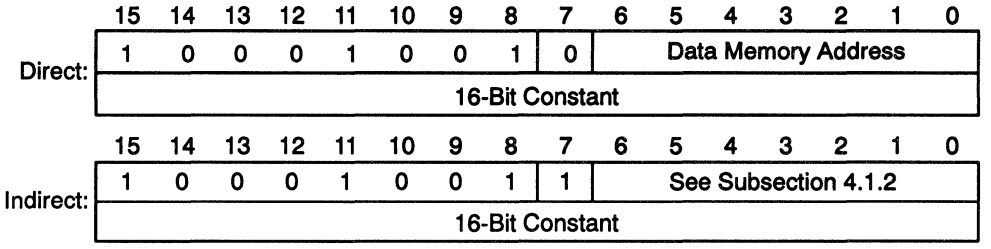
Syntax

Direct: [*label*] **LMMR** *dma, #addr*
Indirect: [*label*] **LMMR** {*ind*}, *#addr* [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ addr ≤ 65535

Opcode



Execution

PFC → MCS
(PC) + 2 → PC
lk → PFC
While (repeat counter ≠ 0):
    (src, addressed by PFC) → (dst, specified by lower 7 bits of dma)
    (PFC) + 1 → PFC
    (repeat counter) – 1 → repeat counter
MCS → PFC

Description

The memory-mapped register pointed at by the lower 7 bits of the directly or indirectly addressed data memory value is loaded with the contents of the data memory location addressed by the 16-bit address, addr. The 9 MSBs of the data memory address are set to zero, regardless of the current value of the data page pointer (DP) or the upper 9 bits of AR(ARP). This instruction allows any memory location on data page zero to be accessed without modifying the DP field in status register ST0.

When using the LMMR instruction with the RPT instruction, the source address, *#addr*, is incremented after every memory-mapped load.

Words

2

Cycles

Direct: [*label*] **LMMR** *dma, #addr*
Indirect: [*label*] **LMMR** {*ind*}, *#addr* [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination MMR‡	2	2	2	$2+2p_{code}$
Source SARAM Destination MMR‡	2	2	2 3†	$2+2p_{code}$

Cycle Timings for a Single Instruction (continued)				
	PR	**PDA**	**PSA**	**PE**
Source Ext Destination MMR‡	2+p_{src}	2+p_{src}	2+p_{src}	3+p_{src}+2p_{code}
Source DARAM Destination MMPORT	3+io_{dst}	3+io_{dst}	3+io_{dst}	5+2p_{code}+io_{dst}
Source SARAM Destination MMPORT	3+io_{dst}	3+io_{dst}	3+io_{dst} 4†	5+2p_{code}+io_{dst}
Source Ext Destination MMPORT	3+p_{src}+io_{dst}	3+p_{src}+io_{dst}	3+p_{src}+io_{dst}	6+p_{src}+2p_{code}+io_{dst}
Cycle Timings for a Repeat (RPT) Execution				
Source DARAM Destination MMR§	2n	2n	2n	2n+2p_{code}
Source SARAM Destination MMR§	2n	2n	2n 2n+1†	2n+2p_{code}
Source Ext Destination MMR§	2n+nd_{src}	2n+nd_{src}	2n+nd_{src}	2n+1+nd_{src}+2p_{code}
Source DARAM Destination MMPORT	3n+nio_{dst}	3n+nio_{dst}	3n+nio_{dst}	3n+3+nio_{dst}+2p_{code}
Source SARAM Destination MMPORT	3n+nio_{dst}	3n+nio_{dst}	3n+nio_{dst} 3n+1+nio_{dst}†	3n+3+nio_{dst}+2p_{code}
Source Ext Destination MMPORT	4n–1+nd_{src}+ nio_{dst}	4n–1+nd_{src}+ nio_{dst}	4n–1+nd_{src}+ nio_{dst}	4n+2+nd_{src}+ nio_{dst}+2p_{code}

† If the source operand and the code are in the same SARAM block.
‡ Add one more cycle if peripheral memory mapped register access.
§ Add *n* more cycles if peripheral memory mapped register access.

Example 1

```
LMMR  DBMR,#300h
```

	Before Instruction		After Instruction
Data Memory 300h	1376h	Data Memory 300h	1376h
DBMR	5555h	DBMR	1376h

Example 2

```
LMMR  *,#300h,AR4   ;CBCR = 1Eh
```

	Before Instruction		After Instruction
ARP	0	ARO	4h
AR0	31Eh	AR0	31Eh
Data Memory 300h	20h	Data Memory 300h	20h
CBCR	0h	CBCR	20h

Syntax

Direct: [*label*] **LPH** *dma*
Indirect: [*label*] **LPH** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	1	0	1	0	Data Memory Address						
Indirect:	0	1	1	1	0	1	0	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → P register (31–16)

Description

The P register high-order bits are loaded with the contents of data memory. The low-order P register bits are unaffected.

The LPH instruction can be used for restoring the high-order bits of the P register after interrupts and subroutine calls if automatic context save is not used.

Words

1

Cycles

Direct: [*label*] **LPH** *dma*
Indirect: [*label*] **LPH** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
LPH    DAT0   ;(DP = 4)
```

	Before Instruction		**After Instruction**
Data Memory 200h	0F79Ch	Data Memory 200h	0F79Ch
P	30079844h	P	0F79C9844h

Example 2

```
LPH   *,AR6
```

	Before Instruction		After Instruction
ARP	5	ARP	6
AR5	200h	AR5	200h
Data Memory 200h	0F79Ch	Data Memory 200h	0F79Ch
P	30079844h	P	0F79C9844h

Syntax

Direct: [*label*] **LST** *#n, dma*
Indirect: [*label*] **LST** *#n, {ind}* [*,next ARP*]

Operands

0 ≤ dma ≤ 127
n = 0,1
0 ≤ next ARP ≤ 7

Opcode

LST #0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	1	1	0	0	Data Memory Address						
Indirect:	0	0	0	0	1	1	1	0	1	See Subsection 4.1.2						

LST #1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	1	1	1	0	Data Memory Address						
Indirect:	0	0	0	0	1	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → status register STn
dma (bits 13–15) → ARP (regardless of n)

Affects ARB, ARP, OV, OVM, DP, CNF, TC, SXM, C, HM, XF, and PM.
Does not affect INTM.

Description

Status register STn is loaded with the addressed data memory value. Note that the INTM bit is unaffected by LST #0. In addition, the LST #0 instruction does not affect the ARB field in the ST1 register even though a new ARP is loaded. If a next ARP value is specified via the indirect addressing mode, the specified value is ignored. Instead, ARP is loaded with the value contained within the addressed data memory word.

Note:

When ST1 is loaded, the value loaded into ARB is also loaded into ARP.

The LST instruction can be used for restoring the status registers after subroutine calls and interrupts.

Words

1

Cycles

Direct: [*label*] **LST** *#n, dma*
Indirect: [*label*] **LST** *#n, {ind} [,next ARP]*

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	2	2	2	$2+p_{code}$
Source SARAM	2	2	2 3†	$2+p_{code}$
Source Ext	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	2n	2n	2n	$2n+p_{code}$
Source SARAM	2n	2n	2n 2n+1†	$2n+p_{code}$
Source Ext	$2n+nd_{src}$	$2n+nd_{src}$	$2n+nd_{src}$	$2n+1+nd_{src}+p_{code}$

† If the source operand and the code are in the same SARAM block.

Example 1

```
MAR  *,AR0
LST  #0,*,AR1  ;The data memory word addressed by the contents
               ;of auxiliary register AR0 is loaded into
               ;status register ST0,except for the INTM bit.
               ;Note that even though a next ARP value is
               ;specified, that value is ignored, and the
               ;old ARP is not loaded into the ARB.
```

Example 2

```
LST   #0,60h  ;(DP = 0)
```

	Before Instruction		**After Instruction**
Data Memory 60h	2404h	Data Memory 60h	2404h
ST0	6E00h	ST0	2604h
ST1	0580h	ST1	0580h

Example 3

```
LST   #0,*-,AR1
```

	Before Instruction		**After Instruction**
ARP	4	ARP	1
AR4	3FFh	AR4	3FEh
Data Memory 3FFh	0EE04h	Data Memory 3FFh	0EE04h
ST0	0EE00h	ST0	0EE04h
ST1	0F780h	ST1	0F780h

Example 4

```
LST   #1,00h      ;(DP = 6)
```

	Before Instruction		After Instruction
Data Memory 300h	0E1BCh	Data Memory 300h	0E1BC
ST0	0406h	ST0	E406
ST1	09A0	ST1	0E1BCh

Syntax

Direct: [*label*] **LT** *dma*
Indirect: [*label*] **LT** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	0	1	1	0	Data Memory Address						
Indirect:	0	1	1	1	0	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → TREG0

If TRM = 0:
(dma) → TREG1
(dma) → TREG2

Affected by TRM.

Description

TREG0 is loaded with the contents of the specified data memory address (dma). The LT instruction may be used to load TREG0 in preparation for multiplication. See the LTA, LTD, LTP, LTS, MPY, MPYA, MPYS,and MPYU instructions. If the TRM bit of the PMST register is 0, then TREG1 and TREG2 are also loaded to maintain compatibility with the 'C25. The TREGs are memory-mapped registers and may be read and written with any instruction that accesses data memory. Note that TREG1 is only 5 bits and TREG2 is only 4 bits.

Words

1

Cycles

Direct: [*label*] **LT** *dma*
Indirect: [*label*] **LT** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution‡				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
LT    DAT24  ;(DP = 8. TRM = 1).
```

	Before Instruction	After Instruction
Data Memory 418h	62h	62h
TREG0	3h	62h

Example 2

```
LT    *,AR3  ;(TRM = 0)
```

	Before Instruction	After Instruction
ARP	2	3
AR2	418h	418h
Data Memory 418h	62h	62h
TREG0	3h	62h
TREG1	4h	62h
TREG2	5h	62h

Syntax

Direct: [*label*] **LTA** *dma*
Indirect: [*label*] **LTA** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	0	0	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	1	0	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → TREG0
(ACC) + (shifted P register) → ACC

Affected by OVM, PM, and TRM; affects OV and C.

Description

TREG0 is loaded with the contents of the specified data memory address (dma). The contents of the product register, shifted as defined by the PM status bits, are added to the accumulator, with the result left in the accumulator. If the TRM bit of the PMST register is 0, then TREG1 and TREG2 are loaded with the same value as TREG0 to maintain compatibility with the 'C25. Note that TREG1 is only 5 bits and TREG2 is only 4 bits.

The function of the LTA instruction is included in the LTD instruction.

Words

1

Cycles

Direct: [*label*] **LTA** *dma*
Indirect: [*label*] **LTA** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
LTA    DAT36  ;(DP = 6, PM = 0, TRM = 1)
```

	Before Instruction	After Instruction
Data Memory 324h	62h	62h
TREG0	3h	62h
P	0Fh	0Fh
ACC	5h	14h
C	X	0

Example 2

```
LTA    *,5    ;(TRM = 0)
```

	Before Instruction	After Instruction
ARP	4	5
AR4	324h	324h
Data Memory 324h	62h	62h
TREG0	3h	62h
TREG1	4h	62h
TREG2	5h	62h
P	0Fh	0Fh
ACC	5h	14h
C	X	0

Syntax

Direct: [*label*] **LTD** *dma*
Indirect: [*label*] **LTD** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	0	1	0	0	Data Memory Address						
Indirect:	0	1	1	1	0	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → TREG0
(dma) → dma + 1
(ACC) + (shifted P register) → ACC

Affected by OVM, PM, and TRM; affects C and OV.

Description

TREG0 is loaded with the contents of the specified data memory address (dma). The contents of the P register, shifted as defined by the PM status bits, are added to the accumulator, and the result is placed in the accumulator. The contents of the specified data memory address are also copied to the next higher data memory address. If the TRM bit of the PMST register is 0, then TREG1 and TREG2 are also loaded to maintain compatibility with the 'C25. Note that TREG1 is only 5 bits and TREG2 is only 4 bits.

This instruction is valid for all blocks of on-chip RAM configured as data memory. The data move function is continuous across the boundaries of contiguous blocks of memory but cannot be used with external data memory or memory-mapped registers. This function is described under the instruction DMOV. Note that if LTD is used with external data memory, its function is identical to that of LTA.

Words

1

Cycles

Direct: [*label*] **LTD** *dma*
Indirect: [*label*] **LTD** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 3†	1+p
Operand Ext	2+2d	2+2d	2+2d	5+2d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	2n–2	2n–2	2n–2 2n+1†	2n–2+p
Operand Ext	4n–2+2nd	4n–2+2nd	4n–2+2nd	4n+1+2nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
LTD    DAT126  ;(DP = 7, PM = 0, TRM = 1).
```

	Before Instruction	After Instruction
Data Memory 3FEh	62h	62h
Data Memory 3FFh	0h	62h
TREG0	3h	62h
P	0Fh	0Fh
ACC	5h	14h
C	X	0

Example 2

```
LTD    *,AR3  ;(TRM = 0)
```

	Before Instruction	After Instruction
ARP	1	3
AR1	3FEh	3FEh
Data Memory 3FEh	62h	62h
Data Memory 3FFh	0h	62h
TREG0	3h	62h
TREG1	4h	62h
TREG2	5h	62h
P	0Fh	0Fh
ACC	5h	14h
C	X	0

Syntax

Direct: [*label*] **LTP** *dma*
Indirect: [*label*] **LTP** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	0	0	1	0	Data Memory Address						
Indirect:	0	1	1	1	0	0	0	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → TREG0
(shifted P register) → ACC

Affected by PM and TRM.

Description

TREG0 is loaded with the contents of the addressed data memory location, and the product register is stored in the accumulator. The shift at the output of the product register is controlled by the PM status bits. If the TRM bit of the PMST register is 0, then TREG1 and TREG2 are also loaded to maintain compatibility with the 'C25. Note that TREG1 is only 5 bits and TREG2 is only 4 bits.

Words

1

Cycles

Direct: [*label*] **LTP** *dma*
Indirect: [*label*] **LTP** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
LTP    DAT36  ;(DP = 6, PM = 0, TRM = 1)
```

	Before Instruction	After Instruction
Data Memory 324h	62h	62h
TREG0	3h	62h
P	0Fh	0Fh
ACC	5h	0Fh
C	X	X

Example 2

```
LTP    *,AR5  ;(PM = 0, TRM = 0)
```

	Before Instruction	After Instruction
ARP	2	5
AR2	324h	324h
Data Memory 324h	62h	62h
TREG0	3h	62h
TREG1	4h	62h
TREG2	5h	62h
P	0Fh	0Fh
ACC	5h	0Fh
C	X	X

Syntax

Direct: [*label*] **LTS** *dma*
Indirect: [*label*] **LTS** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	1	0	0	0	Data Memory Address						
Indirect:	0	1	1	1	0	1	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(dma) → TREG0
ACC – (shifted P register) → ACC

Affected by PM, TRM, and OVM; affects OV and C.

Description

TREG0 is loaded with the contents of the addressed data memory location. The contents of the product register, shifted as defined by the contents of the PM status bits, are subtracted from the accumulator. The result is placed in the accumulator. If the TRM bit of PMST is set to 0, the value is also loaded into TREG1 and TREG2 to maintain compatibility with the 'C25. Note that TREG1 is only 5 bits and TREG2 is only 4 bits.

Words

1

Cycles

Direct: [*label*] **LTS** *dma*
Indirect: [*label*] **LTS** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
LTS    DAT36  ;(DP = 6, PM = 0, TRM = 1)
```

	Before Instruction	After Instruction
Data Memory 324h	62h	62h
TREG0	3h	62h
P	0Fh	0Fh
ACC	05h	0FFFFFFF6h
C	X	0

Example 2

```
LTS    *,AR2  ;(TRM = 0)
```

	Before Instruction	After Instruction
ARP	1	2
AR1	324h	324h
324h	62h	62h
TREG0	3h	62h
TREG1	4h	62h
TREG2	5h	62h
P	0Fh	0Fh
ACC	05h	0FFFFFFF6h
C	X	0

Syntax

Direct: [*label*] **MAC** *pma, dma*
Indirect: [*label*] **MAC** *pma,* {*ind*} [,*next ARP*]

Operands

0 ≤ pma ≤ 65535
0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified
    (PFC) + 1 → PFC
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified
(MCS) → PFC

Affected by OVM, TRM, and PM; affects C and OV.

Description

The MAC instruction multiplies a data memory value (specified by dma) by a program memory value (specified by pma). It also adds the previous product, shifted as defined by the PM status bits, to the accumulator.

The data and program memory locations on the 'C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When the MAC instruction is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction.

When the MAC instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This makes it possible to access a series of operands in memory. MAC is useful for long sum-of-products operations because it becomes a single-cycle instruction, once the RPT pipeline is started.

If the TRM bit of the PMST register is 0, then TREG1 and TREG2 are loaded with the same value as TREG0 to maintain compatibility with the 'C2x. Note that TREG1 and TREG2 are only 5-bit, and 4-bit long, respectively.

Words 2

Cycles

Direct: [*label*] **MAC** *pma, dma*
Indirect: [*label*] **MAC** *pma,* {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	3	3	3	$3+2p_{code}$
Operand1 SARAM Operand2 DARAM	3	3	3	$3+2p_{code}$
Operand1 Ext Operand2 DARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	3	3	3	$3+2p_{code}$
Operand1 SARAM Operand2 SARAM	3 4†	3 4†	3 4†	$3+2p_{code}$ $4+2p_{code}$†
Operand1 Ext Operand2 SARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 Ext	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand1 SARAM Operand2 Ext	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand1 Ext Operand2 Ext	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand1 SARAM Operand2 DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand1 Ext Operand2 DARAM	$n+2+np_{op1}$	$n+2+np_{op1}$	$n+2+np_{op1}$	$n+2+np_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	n+2	n+2	n+2	$n+2+2p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 SARAM	n+2 2n+2†	n+2 2n+2†	n+2 2n+2†	n+2+2p_{code} 2n+2†
Operand1 Ext Operand2 SARAM	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 Ext	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}
Operand1 SARAM Operand2 Ext	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}
Operand1 Ext Operand2 Ext	2n+2+np_{op1}+nd_{op2}	2n+2+np_{op1}+nd_{op2}	2n+2+np_{op1}+nd_{op2}	2n+2+np_{op1}+nd_{op2}+2p_{code}

† If both operands are in the same SARAM block.

Example 1

```
MAC    0FF00h,02h ;(DP = 6, PM = 0, CNF = 1)
```

	Before Instruction	After Instruction
Data Memory 302h	23h	23h
Program Memory FF00h	4h	4h
TREG0	45h	23h
P	458972h	08Ch
ACC	723EC41h	76975B3h
C	X	0

Example 2

```
MAC    0FF00h,*,AR5 ;(PM = 0, CNF = 1)
```

	Before Instruction	After Instruction
ARP	4	5
AR4	302h	302h
Data Memory 302h	23h	23h
Program Memory FF00h	4h	4h
TREG0	45h	23h
P	458972h	8Ch
ACC	723EC41h	76975B3h
C	X	0

Syntax

Direct: [*label*] **MACD** *pma, dma*
Indirect: [*label*] **MACD** *pma,* {*ind*} [,*next ARP*]

Operands

0 ≤ pma ≤ 65535
0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	1	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	1	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
(PFC) → MCS
(pma) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (dma) → (dma) + 1
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register
    (dma) → (dma) + 1
    Modify AR(ARP) and ARP as specified,
(MCS) → PFC

Affected by OVM and PM; affects C and OV.

Description

The MACD instruction multiplies a data memory value (specified by dma) by a program memory value (specified by pma). It also adds the previous product, shifted as defined by the PM status bits to the accumulator. The data and program memory locations on the 'C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When MACD is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction. If MACD addresses one of the memory-mapped registers or external memory as a data memory location, the effect of the instruction will be that of a MAC instruction (see the DMOV instruction description).

If the TRM bit of the PMST register is 0, TREG1 and TREG2 are loaded with the same value as TREG0 to maintain compatibility with the 'C2x. Note that TREG1 and TREG2 are only 5 bits and 4 bits long, respectively.

MACD functions in the same manner as MAC, with the addition of data move for on-chip RAM blocks. Otherwise, the effects are the same as for MAC. This feature makes MACD useful for applications such as convolution and transversal filtering.

When the MACD instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This permits accessing a series of operands in memory. When used with RPT, MACD becomes a single-cycle instruction once the RPT pipeline is started.

Words 2

Cycles

Direct: [*label*] **MACD** *pma, dma*
Indirect: [*label*] **MACD** *pma,* {*ind*} [,*next ARP*]

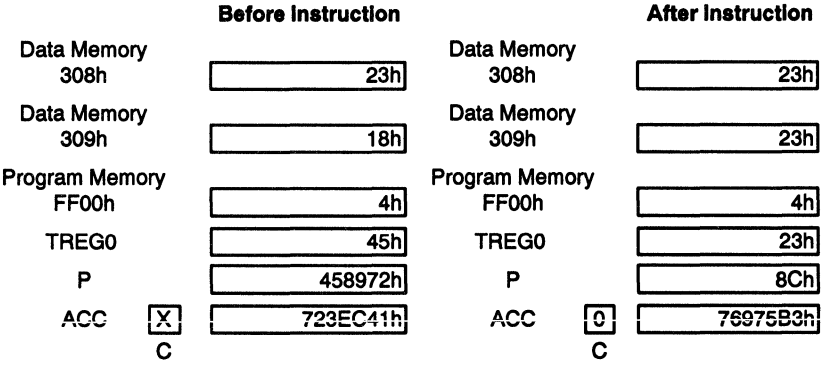
Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 DARAM	3	3	3	$3+2p_{code}$
Operand1 DARAM/ROM Operand2 DARAM	3	3	3	$3+2p_{code}$
Operand1 Ext Operand2 DARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	3	3	3	$3+2p_{code}$
Operand1 SARAM Operand2 SARAM	3	3	3 4‡ 5§	$3+2p_{code}$ $4+2p_{code}$‡
Operand1 Ext Operand2 SARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 Ext¶	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand1 SARAM Operand2 Ext¶	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand1 Ext Operand2 Ext¶	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+2	n+2	n+2	$n+2+2p_{code}$

Cycle Timings for a Repeat (RPR) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand1 Ext Operand2 DARAM	$n+2+np_{op1}$	$n+2+np_{op1}$	$n+2+np_{op1}$	$n+2+np_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	2n	2n	2n 2n+2†	$2n+2p_{code}$
Operand1 SARAM Operand2 SARAM	2n 3n‡	2n 3n‡	2n 2n+2† 3n‡ 3n+2§	$2n+2p_{code}$ 3n‡
Operand1 Ext Operand2 SARAM	$2n+np_{op1}$	$2n+np_{op1}$	$2n+np_{op1}$ $2n+2+np_{op1}$†	$2n+np_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 Ext¶	$n+2+nd_{op2}$	$n+2+nd_{op2}$	$n+2+nd_{op2}$	$n+2+nd_{op2}+2p_{code}$
Operand1 SARAM Operand2 Ext¶	$n+2+nd_{op2}$	$n+2+nd_{op2}$	$n+2+nd_{op2}$	$n+2+nd_{op2}+2p_{code}$
Operand1 Ext Operand2 Ext¶	$2n+2+np_{op1}+nd_{op2}$	$2n+2+np_{op1}+nd_{op2}$	$2n+2+np_{op1}+nd_{op2}$	$2n+2+np_{op1}+nd_{op2}+2p_{code}$

† If operand2 and code are in the same SARAM block.
‡ If both operands are in the same SARAM block.
§ If both operands and code are in the same SARAM block.
¶ Data move operation is not performed when operand2 is in external data memory.

Example 1

```
MACD   0FF00h,08h ;(DP = 6, PM = 0, CNF = 1).
```



Example 2

```
MACD   0FF00h,*,AR6 ;(PM = 0, CF = 1)
```

	Before Instruction	After Instruction
ARP	5	6
AR5	308h	308h
Data Memory 308h	23h	23h
Data Memory 309h	18h	23h
Program Memory FF00h	4h	4h
TREG0	45h	23h
P	458972h	8Ch
ACC	723EC41h	76975B3h
C	X	0

Note: The data move function for MACD can occur only within on-chip data memory RAM blocks.

Syntax

Direct: [*label*] **MADD** *dma*
Indirect: [*label*] **MADD** {*ind*} [,*next ARP*]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	0	1	1	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	1	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 2 → PC
(PFC) → MCS
(BMAR) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (dma) → (dma) + 1
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register
    (dma) → (dma) + 1
    Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Affected by OVM, TRM, and PM; affects C and OV.

Description

The MADD instruction multiplies a data memory value (specified by the dma) by a program memory value. The program memory address is contained in the BMAR register; it is not specified by a long immediate constant. This facilitates dynamic addressing of coefficient tables. In addition, the previous product, shifted as defined by the PM status bits, is added to the accumulator. The data and program memory locations on the 'C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When the MADD instruction is used in direct addressing mode, the dma cannot be modified during repetition of the instruction. If MADD addresses one of the memory-mapped registers or external memory as a data memory location, the effect of the instruction is that of a MADS instruction (see the DMOV instruction description).

MADD functions in the same manner as MADS, with the addition of *data move* for on-chip RAM blocks. Otherwise, the effects are the same as for MADS. This

feature makes MADD useful for applications such as convolution and transversal filtering.

If the TRM bit of the PMST register is 0, TREG1 and TREG2 are loaded with the same value as TREG0 to maintain compatibility with the 'C2x. Note that TREG1 and TREG2 are only 5 bits and 4 bits long, respectively.

When the MADD instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This enables accessing a series of operands in memory. When used with RPT, MADD becomes a single-cycle instruction, once the RPT pipeline is started.

Words 1

Cycles

Direct: [*label*] **MADD** *dma*
Indirect: [*label*] **MADD** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	2	2	2	$2+p_{code}$
Operand1 SARAM Operand2 DARAM	2	2	2	$2+p_{code}$
Operand1 Ext Operand2 DARAM	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	2	2	2	$2+p_{code}$
Operand1 SARAM Operand2 SARAM	2	2	2 3‡ 4§	$2+p_{code}$ $3+p_{code}$‡
Operand1 Ext Operand2 SARAM	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 Ext¶	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}+p_{code}$
Operand1 SARAM Operand2 Ext¶	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}+p_{code}$
Operand1 Ext Operand2 Ext¶	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Operand1 SARAM Operand2 DARAM	n+1	n+1	n+1	$n+1+p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 Ext Operand2 DARAM	$n+1+np_{op1}$	$n+1+np_{op1}$	$n+1+np_{op1}$	$n+1+np_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	2n–1	2n–1	2n–1 2n+1†	$2n-1+p_{code}$
Operand1 SARAM Operand2 SARAM	2n–1 3n–1‡	2n–1 3n–1‡	2n–1 2n+1† 3n–1‡ 3n+1§	$2n-1+p_{code}$ 3n–1‡
Operand1 Ext Operand2 SARAM	$2n-1+np_{op1}$	$2n-1+np_{op1}$	$2n-1+np_{op1}$ $2n+1+np_{op1}$†	$2n-1+np_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 Ext¶	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}+p_{code}$
Operand1 SARAM Operand2 Ext¶	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}+p_{code}$
Operand1 Ext Operand2 Ext¶	$2n+1+np_{op1}+nd_{op2}$	$2n+1+np_{op1}+nd_{op2}$	$2n+1+np_{op1}+nd_{op2}$	$2n+1+np_{op1}+nd_{op2}+p_{code}$

† If operand2 and code reside in same SARAM block.
‡ If both operands reside in same SARAM block.
§ If both operands and code reside in same SARAM block.
¶ Data move operation is not performed when operand2 is in external data memory.

Example 1

```
MADD   DAT7   ;(DP = 6, PM = 0, CNF = 1)
```

	Before Instruction		After Instruction	
Data Memory 307h	8h		Data Memory 307h	8h
Data Memory 308h	9h		Data Memory 308h	8h
BMAR	0FF00h		BMAR	0FF00h
TREG0	4Eh		TREG0	8h
FF00h	2h		FF00h	2h
P	458972h		P	10h
ACC	723EC41h	X (C)	ACC	76975B3h 0 (C)

Example 2

```
MADD   *,3    ;(PM = 0, CNF = 1)
```

	Before Instruction		After Instruction
ARP	2	ARP	3
AR2	307h	AR2	307h
Data Memory 307h	8h	Data Memory 307h	8h
Data Memory 308h	9h	Data Memory 308h	8h
BMAR	0FF00h	BMAR	0FF00h
TREG0	4Eh	TREG0	8h
FF00h	2h	FF00h	2h
P	458972h	P	10h
ACC	723EC41h	ACC	76975B3h
C	X	C	0

Note: The data move function for MADD can occur only within on-chip data memory RAM blocks.

Syntax

Direct: [*label*] **MADS** *dma*
Indirect: [*label*] **MADS** {*ind*} [,*next ARP*]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	1	0	1	0	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	1	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(PFC) → MCS
(BMAR) → PFC

If (repeat counter) ≠ 0:
  Then (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (repeat counter) – 1 → repeat counter.
  Else (ACC) + (shifted P register) → ACC,
    (dma) → TREG0
    (dma) × (pma, addressed by PFC) → P register,
    Modify AR(ARP) and ARP as specified,
(MCS) → PFC

Affected by OVM, TRM, and PM; affects C and OV.

Description

The MADS instruction multiplies a data memory value (specified by dma) by a program memory value (specified by pma). It also adds the previous product, shifted as defined by the PM status bits, to the accumulator. The *pma* is specified by the contents of the BMAR register, rather than by a long immediate constant. This allows for dynamic addressing of coefficient tables.

The data and program memory locations on the 'C5x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, then the CNF bit must be set to one. When MADS is used in the direct addressing mode, the dma cannot be modified during repetition of the instruction.

When the MADS instruction is repeated, the program memory address contained in the PFC is incremented by one during its operation. This makes it possible to access a series of operands in memory. MADS is useful for long sum-of-products operations because this instruction becomes a single-cycle instruction, once the RPT pipeline is started.

If the TRM bit of the PMST register is 0, TREG1 and TREG2 are loaded with the same value as TREG0 to maintain compatibility with the 'C2x. Note that TREG1 and TREG2 are only 5 bits and 4 bits long, respectively.

Words 1

Cycles

Direct: [*label*] **MADS** *dma*
Indirect: [*label*] **MADS** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	2	2	2	$2+p_{code}$
Operand1 SARAM Operand2 DARAM	2	2	2	$2+p_{code}$
Operand1 Ext Operand2 DARAM	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}$	
Operand1 DARAM/ROM Operand2 SARAM	2	2	2	$2+p_{code}$
Operand 1 SARAM Operand2 SARAM	2 3†	2 3†	2 3†	$2+p_{code}$ $3+p_{code}$†
Operand1 Ext Operand2 SARAM	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 Ext	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}+p_{code}$
Operand1 SARAM Operand2 Ext	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}+p_{code}$
Operand1 Ext Operand2 Ext	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Operand1 SARAM Operand2 DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Operand1 Ext Operand2 DARAM	$n+1+np_{op1}$	$n+1+np_{op1}$	$n+1+np_{op1}$	$n+1+np_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	n+1	n+1	n+1	$n+1+p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 SARAM	n+1 2n+1†	n+1 2n+1†	n+1 2n+1†	n+1+p_{code} 2n+1†
Operand1 Ext Operand2 SARAM	n+1+np_{op1}	n+1+np_{op1}	n+1+np_{op1}	n+1+np_{op1}+p_{code}
Operand1 DARAM/ROM Operand2 Ext	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}+p_{code}
Operand1 SARAM Operand2 Ext	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}+p_{code}
Operand1 Ext Operand2 Ext	2n+1+np_{op1}+ nd_{op2}	2n+1+np_{op1}+ nd_{op2}	2n+1+np_{op1}+ nd_{op2}	2n+1+np_{op1}+nd_{op2}+ p_{code}

† If both operands are in the same SARAM block.

Example 1

```
MADS   DAT12  ;(DP = 6, PM = 0, CNF = 1).
```

	Before Instruction	After Instruction
Data Memory 30Ch	8h	8h
BMAR	0FF00h	0FF00h
TREG0	4Eh	8h
Program Memory FF00h	2h	2h
P	458972h	10h
ACC	723EC41h	76975B3h
C	X	0

Example 2

```
MADS   *,AR3  ;(PM = 0, CNF = 1)
```

	Before Instruction	After Instruction
ARP	2	3
AR2	30Ch	30Ch
Data Memory 30Ch	8h	8h
BMAR	0FF00h	0FF00h
TREG0	4Eh	8h
Program Memory FF00h	2h	2h
P	458972h	10h
ACC	723EC41h	76975B3h
C	X	0

Syntax

Direct: [*label*] **MAR** *dma*
Indirect: [*label*] **MAR** {*ind*} [,*next ARP*]

Operands

0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	0	1	1	0	Data Memory Address						

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	0	0	1	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC

Modifies ARP, AR(ARP) as specified by the indirect addressing field. Acts as a NOP in direct addressing mode.

Affected by NDX.

Description

In the indirect addressing mode, the auxiliary registers and the ARP are modified; however, no use is made of the memory being referenced. Note that if the NDX bit of the PMST register is 0 and the auxiliary register 0 (AR0) is modified, then the ARCR and INDX registers are also modified in the same way to maintain compatibility with the 'C2x. Note that TREG1 and TREG2 are only 5 bits and 4 bits long, respectively. MAR modifies the auxiliary registers or the ARP, and the old ARP is copied to the ARB field of the status register ST1. Any operation that MAR performs can also be performed with any instruction that supports indirect addressing. ARP can also be loaded by an LST instruction. The instruction LARP from the 'C25 instruction set is a subset of MAR (that is, MAR *,4 performs the same function as LARP 4).

Words

1

Cycles

Direct: [*label*] **MAR** *dma*
Indirect: [*label*] **MAR** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
MAR    *,AR1  ;Load the ARP with 1.
```

	Before Instruction		After Instruction
ARP	0	ARP	1
ARB	7	ARB	0

Example 2

```
MAR   *+,AR5 ;Increment current auxiliary register
             ;(AR1) and load ARP with 5.
```

	Before Instruction		After Instruction
AR1	34h	AR1	35h
ARP	1	ARP	5
ARB	0	ARP	1

Syntax

Direct:	[*label*] **MPY** *dma*
Indirect:	[*label*] **MPY** {*ind*} [,*next ARP*]
Short Immediate:	[*label*] **MPY** #*k*
Long Immediate:	[*label*] **MPY** #*lk*

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7
$-4096 \le k \le 4095$
$-32768 \le lk \le 32767$

Opcode

Multiply data value times TREG0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	1	0	0	0	Data Memory Address						
Indirect:	0	1	0	1	0	1	0	0	1	See Subsection 4.1.2						

Multiply TREG0 by 13-bit immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	1	1	0	13-Bit Constant												

Multiply TREG0 by long immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0
	16-Bit Constant															

Execution

If indirect or direct addressing:

(PC) + 1 → PC
(TREG0) × (dma) → P register

If short immediate value specified:

(PC) + 1 → PC
(TREG0) × k → P register

If long immediate value specified:

(PC) + 2 → PC
(TREG0) × lk → P register

Description

The contents of the TREG0 register are multiplied by the contents of the addressed data memory location. The result is placed in the P register. Short immediate addressing multiplies TREG0 by a signed 13-bit constant. The short immediate value is right-justified and sign-extended before the multiplication, regardless of SXM.

Words

1 (Direct, indirect, or short immediate addressing)
2 (Long immediate addressing)

Cycles

Direct: [*label*] **MPY** *dma*
Indirect: [*label*] **MPY** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Short Immediate: [*label*] **MPY** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Long Immediate: [*label*] **MPY** *#lk*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
MPY    DAT13  ;(DP = 8)
```

	Before Instruction		**After Instruction**
Data Memory 40Dh	7h	Data Memory 40Dh	7h
TREG0	6h	TREG0	6h
P	36h	P	2Ah

Example 2

```
MPY   *,AR2
```

	Before Instruction		After Instruction
ARP	1	ARP	2
AR1	40Dh	AR1	40Dh
Data Memory 40Dh	7h	Data Memory 40Dh	7h
TREG0	6h	TREG0	6h
P	36h	P	2Ah

Example 3

```
MPY   #031h
```

	Before Instruction		After Instruction
TREG0	2h	TREG0	2h
P	36h	P	62h

Example 4

```
MPY   #01234h
```

	Before Instruction		After Instruction
TREG0	2h	TREG0	2h
P	36h	P	2468h

Syntax

Direct: [*label*] **MPYA** *dma*
Indirect: [*label*] **MPYA** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	0	0	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	0	1	0	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) + (shifted P register) → ACC
(TREG0 register) × (dma) → P register

Affected by OVM and PM; affects C and OV.

Description

The contents of TREG0 are multiplied by the contents of the addressed data memory location. The result is placed in the P register. The previous product, shifted as defined by the PM status bits, is also added to the accumulator.

Words

1

Cycles

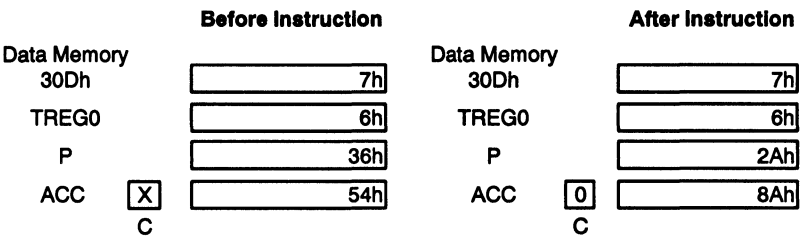
Direct: [*label*] **MPYA** *dma*
Indirect: [*label*] **MPYA** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

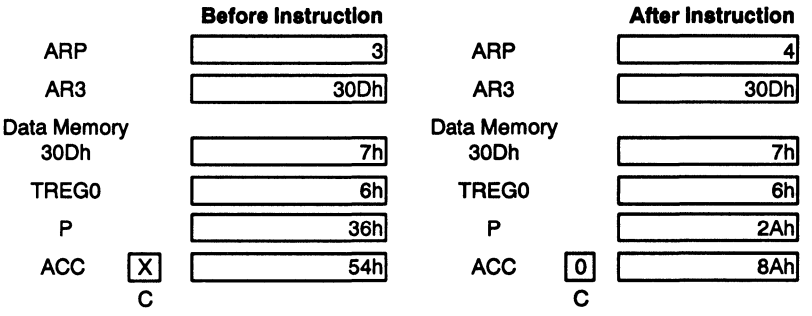
Example 1

```
MPYA   DAT13  ;(DP = 6, PM = 0)
```



Example 2

```
MPYA   *,AR4  ;(PM = 0)
```



Syntax

Direct: [*label*] **MPYS** *dma*
Indirect: [*label*] **MPYS** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	0	0	1	0	Data Memory Address						
Indirect:	0	1	0	1	0	0	0	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) – (shifted P register) → ACC
(TREG0) × (dma) → P register

Affected by OVM and PM; affects C and OV.

Description

The contents of TREG0 are multiplied by the contents of the addressed data memory location. The result is placed in the P register. The previous product, shifted as defined by the PM status bits, is also subtracted from the accumulator, and the result is placed in the accumulator.

Words

1

Cycles

Direct: [*label*] **MPYS** *dma*
Indirect: [*label*] **MPYS** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
MPYS   DAT13  ;(DP = 6, PM = 0)
```

	Before Instruction	After Instruction
Data Memory 30Dh	7h	7h
TREG0	6h	6h
P	36h	2Ah
ACC	54h	1Eh
C	X	1

Example 2

```
MPYS   *,AR5  ;(PM = 0)
```

	Before Instruction	After Instruction
ARP	4	5
AR4	30Dh	30Dh
Data Memory 30Dh	7h	7h
TREG0	6h	6h
P	36h	2Ah
ACC	54h	1Eh
C	X	1

Syntax

Direct: [*label*] **MPYU** *dma*
Indirect: [*label*] **MPYU** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	1	0	1	0	Data Memory Address						
Indirect:	0	1	0	1	0	1	0	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
Unsigned (TREG0) × unsigned (dma) → P register

Not affected by SXM.

Description

The unsigned contents of TREG0 are multiplied by the unsigned contents of the addressed data memory location. The result is placed in the P register. The multiplier acts as a signed 17 × 17-bit multiplier for this instruction, with the MSB of both operands forced to zero.

The shifter at the output of the P register will always invoke sign-extension on the P register when PM = 3 (right-shift by 6 mode). Therefore, this shift mode should not be used if unsigned products are desired.

The MPYU instruction is particularly useful for computing multiple-precision products, such as when multiplying two 32-bit numbers to yield a 64-bit product.

Words

1

Cycles

Direct: [*label*] **MPYU** *dma*
Indirect: [*label*] **MPYU** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
MPYU   DAT16  ;(DP = 4)
```

	Before Instruction		After Instruction
Data Memory 210h	0FFFFh	Data Memory 210h	0FFFFh
TREG0	0FFFFh	TREG0	0FFFFh
P	1h	P	0FFFE0001h

Example 2

```
MPYU   *,AR6
```

	Before Instruction		After Instruction
ARP	5	ARP	6
AR5	210h	AR5	210h
Data Memory 210h	0FFFFh	Data Memory 210h	0FFFFh
TREG0	0FFFFh	TREG0	0FFFFh
P	1h	P	0FFFE0001h

Syntax [*label*] **NEG**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	0	1	0

Execution

(PC) + 1 → PC
(ACC) × −1 → ACC

Affected by OVM; affects OV and C.

Description The contents of the accumulator are replaced with its arithmetic complement (twos complement). The OV bit is set when taking the NEG of 80000000h. If OVM = 1, the accumulator contents are replaced with 7FFFFFFFh. If OVM = 0, the result is 80000000h. The carry bit C on the 'C5x is reset to zero by this instruction for all nonzero values of the accumulator, and is set to one if the accumulator equals zero.

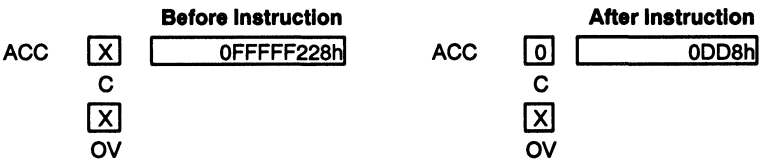
Words 1

Cycles [*label*] **NEG**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

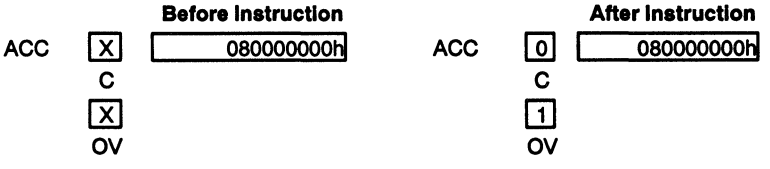
Example 1

```
NEG     ;(OVM = X)
```



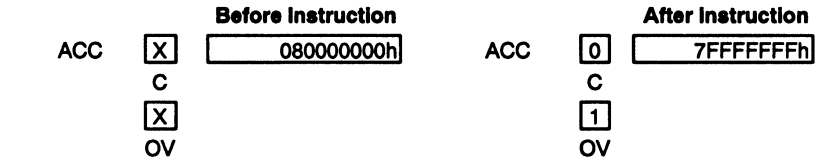
Example 2

```
NEG     ;(OVM = 0)
```



Example 3

```
NEG     ;(OVM = 1)
```



Syntax [*label*] **NMI**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	0	0	1	0

Execution

(PC) + 1 → stack
24h → PC
1 → INTM

Not affected by INTM.

Description This instruction forces the program counter to the nonmaskable interrupt vector located at 24h. The instruction has the same affect as a hardware nonmaskable interrupt. Interrupts are globally disabled (INTM=1). Automatic context save is not performed.

Words 1

Cycles [*label*] **NMI**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example

```
NMI     ;Control is passed to program memory location 24h and
        ;PC+1 is pushed onto the stack
```

Syntax [*label*] **NOP**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0

Execution (PC) + 1 → PC

Description No operation is performed. The NOP instruction affects only the PC. The NOP instruction is useful to create pipeline and execution delays.

Words 1

Cycles [*label*] **NOP**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
NOP    ;No operation is performed.
```

Syntax

[label] **NORM** *{ind}*

Operands

None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC

If (ACC) = 0;
  Then TC → 1;
Else, if (ACC(31)) **XOR** (ACC(30)) = 0:
  Then TC → 0,
    (ACC) × 2 → ACC
    Modify AR(ARP) as specified;
  Else TC → 1.

Affects TC.

Description

The NORM instruction normalizes a signed number that is contained in the accumulator. Normalizing a fixed-point number separates it into a mantissa and an exponent. This is done by finding the magnitude of the sign-extended number. ACC bit 31 is exclusive-ORed with ACC bit 30 to determine if bit 30 is part of the magnitude or part of the sign extension. If they are the same, they are both sign bits, and the accumulator is left-shifted to eliminate the extra sign bit.

The AR(ARP) is modified as specified to generate the magnitude of the exponent. It is assumed that AR(ARP) is initialized before normalization begins. The default modification of the AR(ARP) is an increment.

Multiple executions of the NORM instruction may be required to completely normalize a 32-bit number in the accumulator. Although using NORM with RPT does not cause execution of NORM to fall out of the repeat loop automatically when the normalization is complete, no operation is performed for the remainder of the repeat loop. Note that NORM functions on both positive and negative 2s-complement numbers.

CAUTION

The NORM instruction executes the auxiliary register operation during the execution phase of the pipeline. Therefore, the auxiliary register used in the NORM instruction should not be used by an auxiliary register instruction in the next two instruction words immediately following the NORM instruction. The auxiliary register pointer (ARP) should not be modified by the next two words, as well.

Words

1

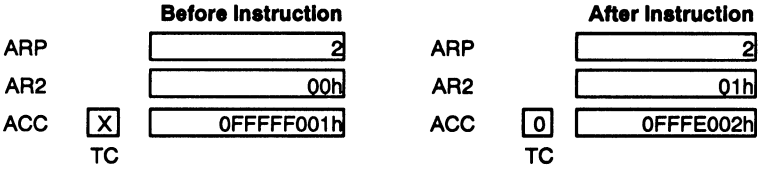
Cycles

[*label*] **NORM** {*ind*}

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
PR	**PDA**	**PSA**	**PE**
n	n	n	n+p

Example 1

```
NORM   *+
```



Example 2

31-Bit Normalization:

```
       MAR    *,AR1      ;Use AR1 to store the exponent.
       LAR    AR1,#0h    ;Clear out exponent counter.
LOOP   NORM   *+         ;One bit is normalized.
       BCND   LOOP,NTC   ;If TC = 0, magnitude not found yet.
```

Example 3

15-Bit Normalization:

```
MAR *,AR1          ;Use AR1 to store the exponent.
LAR    AR1,#0Fh    ;Initialize exponent counter.
RPT    #14         ;15-bit normalization specified (yielding
                   ;a 4-bit exponent and 16-bit mantissa).
NORM   *-          ;NORM automatically stops shifting when first
                   ;significant magnitude bit is found,
                   ;performing NOPs for the remainder of the
                   ;repeat loops
```

The method in Example 2 is used to normalize a 32-bit number and yields a 5-bit exponent magnitude. The method in Example 3 is used to normalize a 16-bit number and yields a 4-bit magnitude. If the number requires only a small amount of normalization, the Example 2 method may be preferable to the Example 3 method. This is because the loop in Example 2 runs only until normalization is complete. Example 3 always executes all 15 cycles of the repeat loop. Specifically, Example 2 is more efficient if the number requires three or less shifts. If the number requires six or more shifts, Example 3 is more efficient.

Note:

The NORM instruction may be used without a specified operand. In that case, any comments on the same line as the instruction are interpreted as the operand. If the first character is an asterisk *, then the instruction is assembled as NORM * with no auxiliary register modification taking place upon execution. Therefore, TI recommends that you replace the NORM instructions with NORM *+ when you want the default increment modification.

Syntax

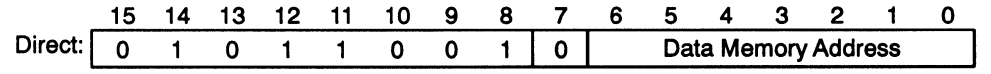
Direct: [*label*] **OPL** [*#lk,*] *dma*
Indirect: [*label*] **OPL** [*#lk,*] {*ind*} [*,next ARP*]

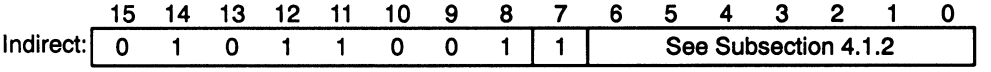
Operands

0 ≤ dma ≤ 127
lk: 16-bit constant
0 ≤ next ARP ≤ 7

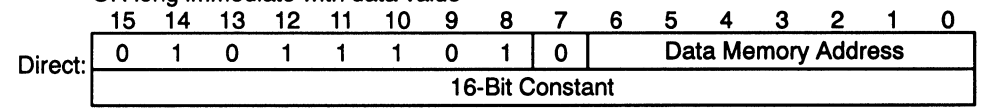
Opcode

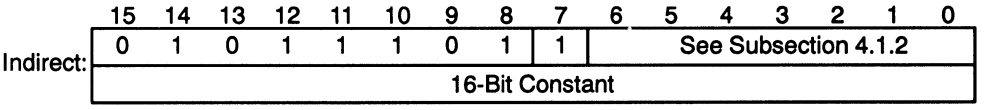
OR DBMR contents with data value





OR long immediate with data value





Execution

lk unspecified:
(PC) + 1 → PC
dma **OR** (DBMR) → dma

lk specified:

(PC) +2 → PC
dma **OR** lk → dma

Affects TC.

Description

If a long immediate constant is specified, it is ORed with the value at the specified data memory address. If the constant is not specified, the second operand to the OR operation is the contents of the dynamic bit manipulation register (DBMR). The result of the operation is always written back into the data memory location specified. The contents of the accumulator are not affected. If the result of the OR operation is 0, then the TC bit is set to 1. Otherwise, the TC bit is set to 0.

Words

1 (Long immediate value not specified)

2 (Long immediate value specified)

Cycles

Direct: [*label*] **OPL** [#*lk*,] *dma*
Indirect: [*label*] **OPL** [#*lk*,] {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 3†	1+p
Operand Ext	2+2d	2+2d	2+2d	5+2d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	2n–2	2n–2	2n–2 2n+1†	2n–2+p
Operand Ext	4n–2+2nd	4n–2+2nd	4n–2+2nd	4n+1+2nd+p

† If the operand and the code are in the same SARAM block.

Direct: [*label*] **OPL** [#*lk*,] *dma*
Indirect: [*label*] **OPL** [#*lk*,] {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2	2+2p
Operand Ext	3+2d	3+2d	3+2d	6+2d+2p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n+1	n+1	n+1	n+1+2p
Operand SARAM	2n–1	2n–1	2n–1 2n+2†	2n–1+2p
Operand Ext	4n–1+2nd	4n–1+2nd	4n–1+2nd	4n+2+2nd+2p

† If the operand and the code reside in same SARAM block.

Example 1

```
OPL   DAT10          ;(DP=6)
```

	Before Instruction		**After Instruction**
DBMR	0FFF0h	DBMR	0FFF0h
Data Memory 30Ah	0001h	Data Memory 30Ah	0FFF1h

Example 2

```
OPL #0FFFh,DAT10     ;(DP=6)
```

	Before Instruction		**After Instruction**
Data Memory 30Ah	0001h	Data Memory 30Ah	0FFFh

Example 3

```
OPL   *,AR6
```

	Before Instruction		After Instruction
ARP	3	ARP	6
AR3	300h	AR3	300h
DBMR	0F0h	DBMR	0F0h
Data Memory 300h	0Fh	Data Memory 300h	0FFh

Example 4

```
OPL   #1111h,*,AR3
```

	Before Instruction		After Instruction
ARP	6	ARP	3
AR6	306h	AR6	306h
Data Memory 306h	0Eh	Data Memory 306h	111Fh

Syntax

Direct:	[*label*] **OR** *dma*
Indirect:	[*label*] **OR** {*ind*} [,*next ARP*]
Long Immediate:	[*label*] **OR** #*lk* [,*shift*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
lk: 16-bit constant
0 ≤ shift ≤ 16

Opcode

OR accumulator with data value

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	1	0	1	0	Data Memory Address						
Indirect:	0	1	1	0	1	1	0	1	1	See Subsection 4.1.2						

OR with ACC long immediate with shift

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	1	1	1	0	0	SHFT †			
	16-Bit Constant															

OR with ACC long immediate with shift of 16

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0
	16-Bit Constant															

Execution

Direct or Indirect Addressing:

(PC) + 1 → PC
(ACC(15–0)) **OR** dma → ACC(15–0)
(ACC(31–16)) → ACC(31–16)

Immediate Addressing:

(PC) + 2 → PC
(ACC) **OR** lk × 2^{shift} → ACC
Not affected by SXM.

Description

The accumulator is ORed with the contents of the addressed data memory location or with a left-shifted long immediate value. The result remains in the accumulator. All bit positions unoccupied by the data operand are zero-filled, no matter what the value of the SXM status bit is. Thus, the high word of the accumulator is unaffected by this instruction if direct or indirect addressing is used, or if immediate addressing is used with a shift of zero. Zeros are shifted into the least significant bits of the operand if immediate addressing is used with a nonzero shift count.

Words

1 (Direct or indirect addressing)

2 (Long immediate addressing)

Cycles

Direct: [*label*] **OR** *dma*
Indirect: [*label*] **OR** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Long Immediate: [*label*] **OR** *#lk* [,*shift*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
OR      DAT8    ;(DP = 8)
```

	Before Instruction	After Instruction
Data Memory 408h	0F000h	0F000h
ACC	100002h	10F002h
C	X	X

Example 2

```
OR      *,AR0
```

	Before Instruction	After Instruction
ARP	1	0
AR1	300h	300h
Data Memory 300h	1111h	1111h
ACC	222h	1333h
C	X	X

Example 3

```
OR      #08111h,8
```

Syntax [*label*] **ORB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	0	1	1

Execution

(PC) + 1 → PC
(ACC) **OR** (ACCB) → ACC

Description The contents of the accumulator are ORed with the contents of the accumulator buffer (ACCB). The result is placed in the accumulator.

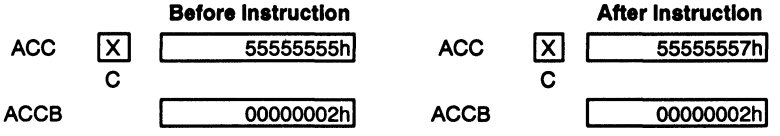
Words 1

Cycles [*label*] **ORB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ORB
```



Syntax

Direct: [*label*] **OUT** *dma* , *PA*
Indirect: [*label*] **OUT** {*ind*}, *PA* [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ *PA* ≤ 65535

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
While (repeat counter) ≠ 0
  Port address → address bus A15–A0
  (dma) → Data bus D15–D0
  Port address + 1 → Port address
  (repeat counter – 1) → (repeat counter)
  (dma) → (port address)

Description

The OUT instruction writes a 16-bit value from a data memory location to the specified I/O port. The $\overline{IS}$ line goes low to indicate an I/O access, and the $\overline{STRB}$, R/$\overline{W}$, and READY timings are the same as for an external data memory write. Note that port addresses 50h–5Fh are memory-mapped (see subsection 5.1.1); the other port addresses are not.

RPT can be used with the OUT instruction to write consecutive words from data memory to I/O space. In the repeat mode, the port address (PA) is incremented after each access.

Words

2

Cycles

Direct: [*label*] **OUT** *dma* , *PA*
Indirect: [*label*] **OUT** {*ind*}, *PA* [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	$3+io_{dst}$	$3+io_{dst}$	$3+io_{dst}$	$5+io_{dst}+2p_{code}$
Source SARAM	$3+io_{dst}$	$3+io_{dst}$	$3+io_{dst}$ $4+io_{dst}$†	$5+io_{dst}+2p_{code}$
Source Ext	$3+d_{src}+io_{dst}$	$3+d_{src}+io_{dst}$	$3+d_{src}+io_{dst}$	$6+d_{src}+io_{dst}+2p_{code}$

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+3+nio_{dst}+2p_{code}$
Source SARAM	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$ $3n+1+nio_{dst}$†	$3n+3+nio_{dst}+2p_{code}$
Source Ext	$5n-2+nd_{src}+nio_{dst}$	$5n-2+nd_{src}+nio_{dst}$	$5n-2+nd_{src}+nio_{dst}$	$5n+1+nd_{src}+nio_{dst}+2p_{code}$

† If the source operand and the code are in the same SARAM block.

Example 1

```
OUT     DAT0,PA7    ;(DP = 4) Output data word stored in data memory
                    ;location 200h to peripheral on port address 7.
```

Example 2

```
OUT     *,PA15      ;Output data word referenced by current auxiliary
                    ;register to peripheral on port address 15.
```

Syntax [*label*] **PAC**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	0	1	1

Execution (PC) + 1 → PC
(shifted P register) → ACC

Affected by PM.

Description The contents of the P register, shifted as specified by the PM status bits, are loaded into the accumulator.

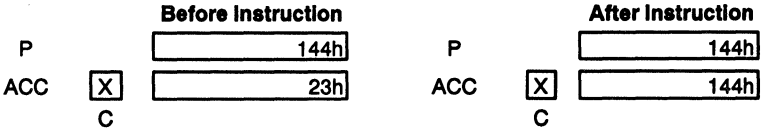
Words 1

Cycles [*label*] **PAC**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
PAC  ;(PM = 0)
```



Syntax [*label*] **POP**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0

Execution

(PC) + 1 → PC
(TOS) → ACC(15–0)
0 → ACC(31–16)
Pop stack one level

Description

The contents of the top of the stack (TOS) are copied to the low accumulator, and the stack is popped after the contents are copied. The upper half of the accumulator is set to all zeros.

The hardware stack is last-in, first-out with eight locations. Any time a pop occurs, every stack value is copied to the next higher stack location, and the top value is removed from the stack. After a pop, the bottom two stack words will have the same value. Because each stack value is copied, if more than seven stack pops (POP, POPD, RETC, RETE, RETI, or RET instructions) occur before any pushes occur, all levels of the stack contain the same value. No provision exists to check stack underflow.

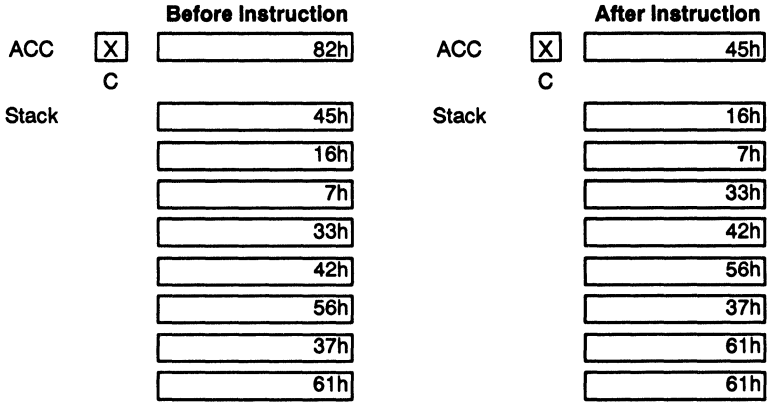
Words 1

Cycles [*label*] **POP**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
POP
```



Syntax

Direct: [*label*] **POPD** *dma*
Indirect: [*label*] **POPD** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	0	1	0	0	Data Memory Address						
Indirect:	1	0	0	0	1	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(TOS) → dma
POP stack one level

Description

The value from the top of the stack is transferred into the data memory location specified by the instruction. The values are also popped in the lower seven locations of the stack. The stack operation is described in the previous instruction, POP. The lowest stack location remains unaffected. No provision exists to check stack underflow.

Words

1

Cycles

Direct: [*label*] **POPD** *dma*
Indirect: [*label*] **POPD** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
POPD   DAT10  ;(DP = 8)
```

	Before Instruction		After Instruction
Data Memory 40Ah	55h	Data Memory 40Ah	92h
Stack	92h	Stack	72h
	72h		8h
	8h		44h
	44h		81h
	81h		75h
	75h		32h
	32h		0AAh
	0AAh		0AAh

Example 2

```
POPD   *+,AR1
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	300h	AR0	301h
Data Memory 300h	55h	Data Memory 300h	92h
Stack	92h	Stack	72h
	72h		8h
	8h		44h
	44h		81h
	81h		75h
	75h		32h
	32h		0AAh
	0AAh		0AAh

Syntax

Direct: [*label*] **PSHD** *dma*
Indirect: [*label*] **PSHD** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	1	0	1	1	0	0	Data Memory Address						
Indirect:	0	1	1	1	0	1	1	0	1	See Subsection 4.1.2						

Execution

(dma) → TOS
(PC) + 1 → PC
Push all stack locations down one level.

Description

The value from the data memory location specified by the instruction is transferred to the top of the stack. The values are also pushed down in the lower seven locations of the stack, as described in the PUSH instruction. The lowest stack location is lost.

Words

1

Cycles

Direct: [*label*] **PSHD** *dma*
Indirect: [*label*] **PSHD** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† if the operand and the code are in the same SARAM block.

Example 1

```
PSHD   DAT127 ;(DP = 3)
```

	Before Instruction		After Instruction
Data Memory 1FFh	65h	Data Memory 1FFh	65h
Stack	2h	Stack	65h
	33h		2h
	78h		33h
	99h		78h
	42h		99h
	50h		42h
	0h		50h
	0h		0h

Example 2

```
PSHD   *,AR1
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	1FFh	AR0	1FFh
Data Memory 1FFh	12h	Data Memory 1FFh	12h
Stack	2h	Stack	12h
	33h		2h
	78h		33h
	99h		78h
	42h		99h
	50h		42h
	0h		50h
	0h		0h

Syntax [*label*] **PUSH**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	1	1	0	0

Execution

(PC) + 1 → PC
Push all stack locations down one level
ACC(15–0) → TOS

Description

The contents of the lower half of the accumulator are copied onto the top of the hardware stack. The stack is pushed down before the accumulator value is copied.

The hardware stack is last-in,first-out with eight locations. If more than eight pushes (due to CALA, CALL, CC, PSHD, PUSH, TRAP, INTR, and NMI instructions) occur before a pop, the first data values written will be lost with each succeeding push.

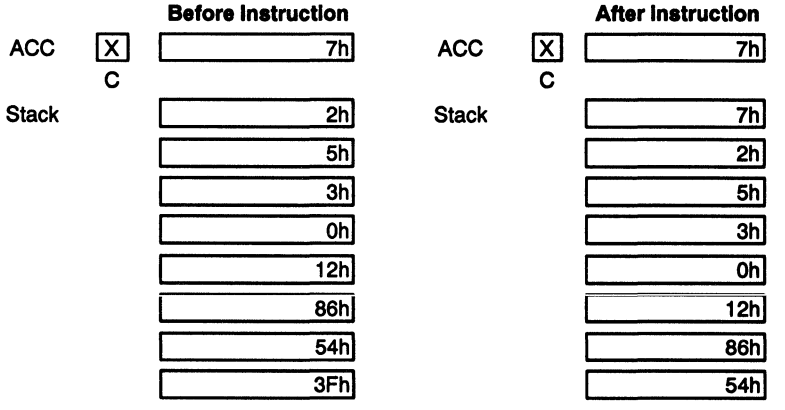
Words 1

Cycles [*label*] **PUSH**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
PUSH
```



Syntax [*label*] **RET**[*D*]

Operands None

Opcode

RET:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0

RETD:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Execution (TOS) → PC
Pop stack one level.

Description The contents of the top stack register are copied into the program counter. The stack is then popped one level. RET is used with CALA, CALL, and CC for subroutines. The two one-word instructions or one two-word instruction following the RET instruction are fetched and executed before the execution of the return, if the delayed version is specified with the "D" suffix.

Words 1

Cycles [*label*] **RET**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

[*label*] **RETD**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
RET
```

	Before Instruction		After Instruction
PC	96h	PC	37h
Stack	37h	Stack	45h
	45h		75h
	75h		21h
	21h		3Fh
	3Fh		45h
	45h		6Eh
	6Eh		6Eh
	6Eh		6Eh

Example 2

```
RETD
MAR    *,4
LACC   #1h
```

	Before Instruction		After Instruction
PC	96h	PC	37h
ARP	0	ARP	4
ACC	0h	ACC	01h
Stack	37h	Stack	45h
	45h		75h
	75h		21h
	21h		3Fh
	3Fh		45h
	45h		6Eh
	6Eh		6Eh
	6Eh		6Eh

Syntax [*label*] **RETC** [*D*] [*cond1*] [, *cond2*] [,...]

Operands

Conditions:

ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
$\overline{BIO}$ low	BIO
TC=0	NTC
TC=1	TC
Unconditional	UNC

Opcode

RETC:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	TP †		ZLVC †				ZLVC †			

RETCD:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	TP †		ZLVC †				ZLVC †			

Execution

If (condition(s)) then
(TOS) → PC
Pop stack one level.
Else, continue

Description A standard return, RET, is executed if the specified conditions are met. Note that not all combinations of conditions are meaningful. The two one-word instructions or one two-word instruction following the RETC are fetched and executed before the execution of the return, if the delayed version is specified with the "D" suffix. If the delayed instruction is specified, the two instruction words following the RETCD instruction have no effect on the conditions being tested.

Words 1

Cycles [*label*] **RETC** [*cond1*] [, *cond2*] [,...]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	2	2	2	2+p
Condition False	2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

[*label*] **RETCD** [*cond1*] [, *cond2*] [,...]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	4	4	4	4+3p†
Condition False	2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example 1

```
RETC   GEQ,NOV  ;A return, RET, is executed if the
                ;accummulator contents are positive and the
                ;OV bit is a zero.
```

Example 2

```
RETCD C         ;A return, RET, is executed if the carry
MAR*,4          ;bit is set. The two instructions following
LARAR3,#1h      ;the return instruction are executed
                ;before the return is taken.
```

Syntax [*label*] **RETE**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0

Execution

(TOS) → PC
Pop stack one level.
0 → global interrupt enable (INTM bit in ST0)

Description The contents of the top stack register are copied into the program counter. The stack is then popped one level. RETE automatically clears the global interrupt enable bit to 0 (INTM in ST0) and pops the shadow register values (see RETI description). RETE is the equivalent of setting the INTM bit to 0 and executing a RETI instruction.

Words 1

Cycles [*label*] **RETE**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example

```
RETE
```

	Before Instruction		After Instruction
PC	96h	PC	37h
ST0	xx6xh	ST0	xx4xh
Stack	37h	Stack	45h
	45h		75h
	75h		21h
	21h		3Fh
	3Fh		45h
	45h		6Eh
	6Eh		6Eh
	6Eh		6Eh

Syntax [*label*] **RETI**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	1	1	0	0	0

Execution (TOS) → PC
Pop stack one level.

Description The contents of the top stack register are copied into the program counter. The RETI instruction also pops the values in the shadow registers (stored when an interrupt was taken) back into their corresponding strategic registers. The following registers are shadowed: ACC, ACCB, PREG, ST0, ST1, PMST, ARCR, INDX, TREG0, TREG1, and TREG2. The XF bit in status register ST1 is not saved or restored to/from the shadow registers during interrupt service routines.

Words 1

Cycles [*label*] **RETI**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example `RETI`

	Before Instruction		After Instruction
PC	96h	PC	37h
Stack	37h	Stack	45h
	45h		75h
	75h		21h
	21h		3Fh
	3Fh		45h
	45h		6Eh
	6Eh		6Eh
	6Eh		6Eh

Syntax [*label*] **ROL**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	1	1	0	0

Execution

(PC) + 1 → PC
C → ACC(0)
(ACC(31)) → C
(ACC(30–0)) → ACC(31–1)

Affects C.
Not affected by SXM.

Description The ROL instruction rotates the accumulator left one bit. The MSB is shifted into the carry bit, and the value of the carry bit from before the execution of the instruction is shifted into the LSB.

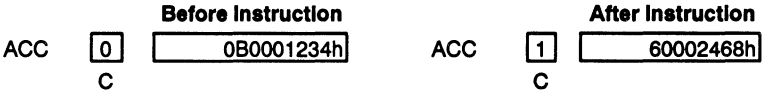
Words 1

Cycles [*label*] **ROL**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ROL
```



Syntax [*label*] **ROLB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	0	0

Execution

(PC) + 1 → PC
C → ACCB(0)
(ACCB(30–0)) → ACCB(31–1)
(ACCB(31)) → ACC(0)
(ACC(30–0)) → ACC(31–1)
(ACC(31)) → C

Affects C.
Not affected by SXM.

Description The ROLB instruction causes a 65-bit rotation. The contents of both the accumulator (ACC) and accumulator buffer (ACCB) are rotated to the left by one bit. The MSB of the original contents in the accumulator shifts into the carry position. The original value of the carry bit (C) shifts into the LSB position of the accumulator buffer, and the MSB of the original contents of the accumulator buffer shifts into the LSB position of the accumulator.

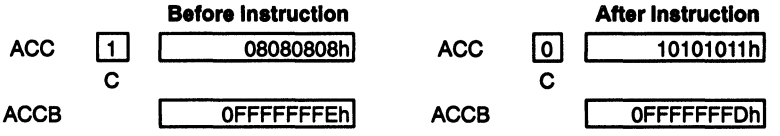
Words 1

Cycles [*label*] **ROLB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ROLB
```



Syntax [*label*] **ROR**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	1	1	0	1

Execution

(PC) + 1 → PC
C → ACC(31)
(ACC(0)) → C
(ACC(31–1)) → ACC(30–0)

Affects C.
Not affected by SXM.

Description The ROR instruction rotates the accumulator right one bit. The LSB is shifted into the carry bit, and the value of the carry bit from before the execution of the instruction is shifted into the MSB.

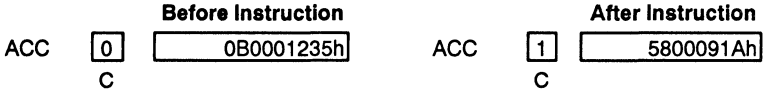
Words 1

Cycles [*label*] **ROR**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ROR
```



Syntax [*label*] **RORB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	0	1

Execution

(PC) + 1 → PC
C → ACC(31)
(ACC(31–1)) → ACC(30–0)
(ACC(0)) → ACCB(31)
(ACCB(31–1)) → ACCB(30–0)
(ACCB(0)) → C

Affects C.
Not affected by SXM.

Description The RORB instruction causes a 65-bit rotation. The contents of both the accumulator (ACC) and accumulator buffer (ACCB) are rotated to the right by one bit. The LSB of the original contents in the accumulator buffer shifts into the carry position. The original value of the carry bit (C) shifts into the MSB position of the accumulator, and the LSB of the original contents of the accumulator shifts into the MSB position of the accumulator buffer.

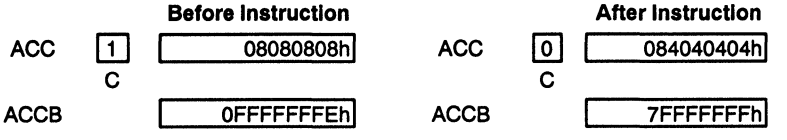
Words 1

Cycles [*label*] **RORB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
RORB
```



Syntax

Direct:	[*label*] **RPT** *dma*
Indirect:	[*label*] **RPT** {*ind*} [,*next ARP*]
Short Immediate:	[*label*] **RPT** #*k*
Long Immediate:	[*label*] **RPT** #*lk*

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
0 ≤ k ≤ 255
0 ≤ lk ≤ 65535

Opcode

Repeat next instruction

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	0	1	1	0	Data Memory Address						
Indirect:	0	0	0	0	1	0	1	1	1	See Subsection 4.1.2						

Repeat next instruction specified by long immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	0
	16-Bit Constant															

Repeat next instruction specified by short immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	1	0	1	1	1	0	1	1	8-Bit Constant							

Execution

Direct or Indirect Addressing:
(PC) + 1 → PC
(dma) → RPTC

Short Immediate Addressing:
(PC) + 1 → PC
k → RPTC

Long Immediate Addressing:
(PC) + 2 → PC
lk → RPTC

Description

The repeat counter (RPTC) is loaded with the addressed data memory location if direct or indirect addressing is used, an 8-bit immediate value if short immediate addressing is used, or a 16-bit immediate value if long immediate addressing is used. The instruction following the RPT is repeated *n* times, where *n* is one more than the initial value of the RPTC. Since the RPTC cannot be saved during a context switch, repeat loops are regarded as multicycle instructions and are not interruptible. However, the processor can halt a repeat loop in response to an external $\overline{\text{HOLD}}$ signal. The execution restarts when $\overline{\text{HOLD}}$/$\overline{\text{HOLDA}}$ are deasserted.The RPTC is set to zero on a device reset.

RPT is especially useful for block moves, multiply-accumulates, normalization, and other functions. The repeat instruction itself is not repeatable.

Words

1 (Direct, indirect, or short immediate addressing)

2 (Long immediate addressing)

Cycles

Direct: [*label*] **RPT** *dma*
Indirect: [*label*] **RPT** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† If the operand and the code are in the same SARAM block.

Short Immediate: [*label*] **RPT** #*k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Long Immediate: [*label*] **RPT** #*lk*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
RPT    DAT127 ;(DP = 31)
```

	Before Instruction		**After Instruction**
Data Memory 0FFFh	0Ch	Data Memory 0FFFh	0Ch
RPTC	0h	RPTC	0Ch

Example 2

```
RPT   *,AR1
```

	Before Instruction		After Instruction
ARP	0	ARP	1
AR0	300h	AR0	300h
Data Memory 300h	0FFFh	Data Memory 300h	0FFFh
RPTC	0h	RPTC	0FFFh

Example 3

```
RPT   #1 ;Repeat next instruction 2 times.
```

	Before Instruction		After Instruction
RPTC	0h	RPTC	1h

Example 4

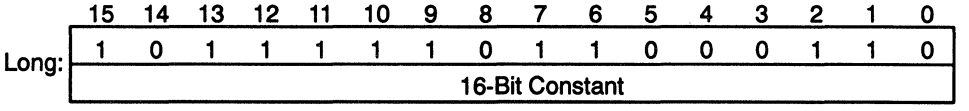
```
RPT   #1111h ;Repeat next instruction 4370 times.
```

	Before Instruction		After Instruction
RPTC	0h	RPTC	1111h

Syntax [*label*] **RPTB** *pma*

Operands $0 \le pma \le 65535$

Opcode



Execution

1 → BRAF
PC+2 → PASR
pma → PAER

Description

The RPTB instruction allows a block of instructions to be repeated a number of times specified by the memory-mapped block repeat count register (BRCR) without any penalty for looping. The BRCR must be loaded before execution of an RPTB instruction. When the RPTB is executed, the start and end address pointers PASR and PAER are loaded with PC+2 and pma, respectively. The block-repeat-active status bit (BRAF) is set to one. Block repeat can be deactivated by clearing the BRAF bit. The number of loop iterations is given by (BRCR) + 1.

The RPTB instruction is interruptible. However, RPTB instructions cannot be nested unless the BRCR, PAER, and PASR registers are appropriately saved and restored and the block repeat active flag (BRAF) is properly set. Single-instruction repeat loops (RPT, RPTZ) can be included as part of RPTB blocks.

CAUTION

The repeat block must contain at least three instruction words for proper operation.

Words 2

Cycles [*label*] **RPTB** *pma*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
        SPLK    #iterations_minus_1,BRCR ;initialize BRCR
        RPTB    end_block - 1
        LACC    DAT1
        ADD     DAT2
        SACL    DAT1
end_block
```

Syntax Long Immediate: [*label*] **RPTZ** #*lk*

Operands 0 ≤ lk ≤ 65535

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	1
16-Bit Constant															

Execution

0 → ACC
0 → PREG
(PC) + 1 → PC
lk → RPTC

Description The RPTZ instruction clears the accumulator and product register and repeats the instruction following the RPTZ *n* times, where *n* = *lk+1*. RPTZ is equivalent to the following instruction sequence:

```
MPY #0
PAC
RPT #<lk>
```

Words 2

Cycles Long Immediate: [*label*] **RPTZ** #*lk*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
RPTZ   #7FFh     ;Zero product register and accumulator.
MACD   pma,*+    ;Repeat MACD 2048 times.
```

Syntax [*label*] **SACB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	1	1	0

Execution

(PC) + 1 → PC
(ACC) → ACCB

Description The accumulator contents are copied to the accumulator buffer (ACCB).

Words 1

Cycles [*label*] **SACB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
SACB
```

	Before Instruction		After Instruction
ACC	7C638421h	ACC	7C638421h
ACCB	5h	ACCB	7C638421h

Syntax

Direct: [*label*] **SACH** *dma* [,*shift*]
Indirect: [*label*] **SACH** {*ind*} [,*shift*[,*next ARP*]]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7
$0 \le$ shift ≤ 7 (defaults to 0)

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	1	1	SHF †			0	Data Memory Address						
Indirect:	1	0	0	1	1	SHF †			1	See Subsection 4.1.2						

† See Section 4.5.

Execution

(PC) + 1 → PC
$[(ACC) \times 2^{shift}] \rightarrow$ dma

Not affected by SXM

Description

The SACH instruction copies the entire accumulator into a shifter, where it left-shifts the entire 32-bit number from 0 to 7 bits. It then copies the upper 16 bits of the shifted value into data memory. The accumulator itself remains unaffected.

Words

1

Cycles

Direct: [*label*] **SACH** *dma* [,*shift*]
Indirect: [*label*] **SACH** {*ind*} [,*shift*[,*next ARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
SACH   DAT10,1  ;(DP = 4)
```

	Before Instruction	After Instruction
ACC	4208001h	4208001h
C	X	X
Data Memory 20Ah	0h	0841h

Example 2

```
SACH   *+,0,AR2
```

	Before Instruction	After Instruction
ARP	1	2
AR1	300h	301h
ACC	4208001h	4208001h
C	X	X
Data Memory 300h	0h	0420h

Syntax

Direct: [*label*] **SACL** *dma* [,*shift*]
Indirect: [*label*] **SACL** {ind} [,*shift*[,*next ARP*]]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7
$0 \le$ shift ≤ 7 (defaults to 0)

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	1	0	SHF †			0	Data Memory Address						
Indirect:	1	0	0	1	0	SHF †			1	See Subsection 4.1.2						

† See Section 4.5.

Execution

(PC) + 1 → PC
16 LSBs of [(ACC) × 2^{shift}] → dma

Not affected by SXM.

Description

The low-order bits of the accumulator are shifted left from 0 to 7 bits, as specified by the shift code, and stored in data memory. The low-order bits are filled with zeros on the shift, and the high-order bits are lost. The accumulator itself remains unaffected.

Words

1

Cycles

Direct: [*label*] **SACL** *dma* [,*shift*]
Indirect: [*label*] **SACL** {ind} [,*shift*[,*next ARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
SACL   DAT11,1  ;(DP = 4)
```

	Before Instruction		After Instruction
ACC	7C63 8421	ACC	7C63 8421h
C	X	C	X
Data Memory 20Bh	05h	Data Memory 20Bh	0842h

Example 2

```
SACL   *,0,AR7
```

	Before Instruction		After Instruction
ARP	6	ARP	7
AR6	300h	AR6	300h
ACC	00FF 8421h	ACC	00FF 8421h
C	X	C	X
Data Memory 300h	05h	Data Memory 300h	8421h

Syntax

Direct: [*label*] **SAMM** *dma*
Indirect: [*label*] **SAMM** {*ind*} [, *next ARP*]

Operands

$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	0	0	0	0	Data Memory Address						
Indirect:	1	0	0	0	1	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) → dma(0–7)

Description

The low word of the accumulator is copied to the addressed memory-mapped register. The upper 9 bits of the data address are set to zero, regardless of the current value of DP or the upper 9 bits of AR(ARP). This instruction allows the accumulator to be stored to any memory location on data page 0 without modifying the DP field in status register ST0.

Words

1

Cycles

Direct: [*label*] **SAMM** *dma*
Indirect: [*label*] **SAMM** {*ind*} [, *next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand MMR†	1	1	1	1+p
Operand MMPORT	$2+io_{dst}$	$2+io_{dst}$	$2+io_{dst}$	$4+io_{dst}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand MMR‡	n	n	n	n+p
Operand MMPORT	$2+nio_{dst}$	$2+nio_{dst}$	$2+nio_{dst}$	$2n+2+p+ nio_{dst}$

† Add one more cycle if source is a peripheral memory mapped register.
‡ Add *n* more cycles if source is a peripheral memory mapped register.

Example 1

```
SAMM   PRD     ;(DP = 6)
```

	Before Instruction		After Instruction
ACC	80h	ACC	80h
PRD	05h	PRD	80h
Data Memory 325h	0Fh	Data Memory 325h	0Fh

Example 2

```
SAMM   *,AR2  ;(BMAR = 1Fh)
```

	Before Instruction		After Instruction
ARP	7	ARP	2
AR7	31Fh	AR7	31Fh
ACC	080h	ACC	080h
BMAR	0h	BMAR	080h
Data Memory 31Fh	11h	Data Memory 31Fh	11h

Syntax

Direct: [*label*] **SAR** *AR, dma*
Indirect: [*label*] **SAR** *AR*,{ind} [,*next ARP*]

Operands

$0 \le$ dma ≤ 127
$0 \le$ AR ≤ 7
$0 \le$ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	0	ARX †			0	Data Memory Address						
Indirect:	1	0	0	0	0	ARX †			1	See Subsection 4.1.2						

† See Section 4.5.

Execution

(PC) + 1 → PC
(AR) → dma

Description

The contents of the designated auxiliary register (ARx) are stored in the addressed data memory location. When the contents of the current auxiliary register are modified in the indirect addressing mode, SAR ARn (when n = ARP) stores the value of the auxiliary register contents before it is incremented, decremented, or indexed by INDX.

Words

1

Cycles

Direct: [*label*] **SAR** *AR, dma*
Indirect: [*label*] **SAR** *AR*,{ind} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
SAR    AR0,DAT30 ;(DP = 6)
```

	Before Instruction		After Instruction
AR0	37h	AR0	37h
Data Memory 31Eh	18h	Data Memory 31Eh	37h

Example 2

```
SAR   AR0,*+
```

	Before Instruction		After Instruction
AR0	401h	AR0	402h
Data Memory 401h	0h	Data Memory 401h	401h

Syntax **SATH**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	1	0

Execution

(PC) + 1 → PC
16 × (TREG1(4)) → count
(ACC) right-shifted by count → ACC

Affected by SXM.

Description The accumulator is barrel-shifted right by 16 bits if bit 4 of TREG1 is a one. If bit 4 of TREG1 is a zero, the accumulator is unaffected. Zeros are shifted in if SXM=0. Copies of ACC(31) are shifted in if SXM=1. The SATH instruction in conjunction with the SATL instruction allows a 2-cycle 0- to 31-bit right shift. The carry bit is unaffected.

Words 1

Cycles **SATH**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
SATH   ;(SXM = 0)
```

		Before Instruction			After Instruction
ACC	X C	0FFFF0000h	ACC	X C	0000FFFFh
TREG1		xx1xh	TREG1		xx1xh

Example 2

```
SATH   ;(SXM = 1)
```

		Before Instruction			After Instruction
ACC	X C	0FFFF0000h	ACC	X C	0FFFFFFFFh
TREG1		xx1xh	TREG1		xx1xh

Syntax **SATL**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	1	0

Execution

(PC) + 1 → PC
(TREG1(3–0)) → count
(ACC) right-shifted by count → ACC

Affected by SXM.

Description The accumulator is barrel-shifted right by the value specified in the 4 LSBs of TREG1. Zeros are shifted in if SXM=0. Copies of ACC(31) are shifted in if SXM=1. The SATL instruction in conjunction with the SATH instruction allows a 2-cycle 0- to 31-bit right shift. The carry bit is unaffected.

Words 1

Cycles **SATL**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
SATL   ;(SXM = 0)
```

	Before Instruction		After Instruction	
ACC	X: 0FFFF0000h		ACC	X: 3FFFC000h
C			C	
TREG1	x2h		TREG1	x2h

```
SATL   ;(SXM = 1)
```

	Before Instruction		After Instruction	
ACC	X: 0FFFF0000h		ACC	X: 0FFFC000h
C			C	
TREG1	x2h		TREG1	x2h

Syntax [*label*] **SBB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0

Execution

(PC) + 1 → PC
(ACC) – (ACCB) → ACC

Description The contents of the accumulator buffer (ACCB) are subtracted from the contents of the accumulator. The result is stored in the accumulator, and the accumulator buffer is not affected. The carry bit is reset to zero if the result of the subtraction generates a borrow.

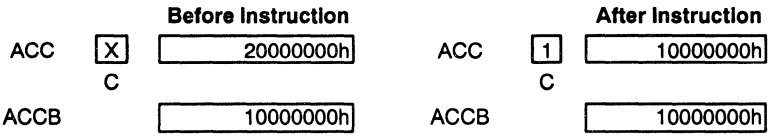
Words 1

Cycles [*label*] **SBB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
SBB
```



Syntax [*label*] **SBBB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	1

Execution

(PC) + 1 → PC
(ACC) – (ACCB) – (Logical inversion of C) → ACC

Description The contents of the accumulator buffer (ACCB) and the logical inversion of the carry bit are subtracted from the accumulator(ACC). The results are stored in the accumulator, and the accumulator buffer is not affected. The carry bit is set to zero if the result generates a borrow.

Words 1

Cycles [*label*] **SBBB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
SBBB
```

		Before Instruction			After Instruction
ACC	[1] C	20000000h	ACC	[1] C	10000000h
ACCB		10000000h	ACCB		10000000h

Example 2

```
SBBB
```

		Before Instruction			After Instruction
ACC	[0] C	098012h	ACC	[1] C	01h
ACCB		098010h	ACCB		098010h

Syntax [*label*] **SBRK** *#k*

Operands $0 \le k \le 255$

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	0	1	1	1	1	1	0	0	8-Bit Constant							

Execution

(PC) + 1 → PC
AR(ARP) – 8-bit positive constant → AR(ARP)

Description The 8-bit immediate value is subtracted, right-justified, from the currently selected auxiliary register with the result replacing the auxiliary register contents. The subtraction takes place in the ARAU, with the immediate value treated as a 8-bit positive integer.

Words 1

Cycles [*label*] **SBRK** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
SBRK  #0FFh
```

	Before Instruction		After Instruction
ARP	7	ARP	7
AR7	0h	AR7	0FF01h

Syntax [*label*] **SETC** *control bit*

Operands *control bit*: ST0 or ST1 bit (from : {C, CNF, HM, INTM, OVM, SXM, TC, XF})

Opcode

Set overflow mode (OVM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	0	1	1

Set sign extension mode (SXM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	1	1	1

Set hold mode (HM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	0	0	1

Set TC bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	0	1	1

Set carry (C)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	1	1	1

Set XF pin high

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	1	1	0	1

Set CNF bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	1	0	1

Set INTM bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	0	0	0	0	1

Execution

(PC) + 1 → PC
1 → control bit

Description The specified control bit is set to 1. Note that LST may also be used to load ST0 and ST1. See subsection 3.6.3 for more information on each control bit.

Words 1

Cycles [*label*] **SETC** *control bit*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
SETC   TC  ;TC is bit 11 of ST1
```

	Before Instruction		**After Instruction**
ST1	x1xxh	ST1	x9xxh

Syntax [*label*] **SFL**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	1

Execution

(PC) + 1 → PC
(ACC(31)) → C
(ACC(30–0)) → ACC(31–1)
0 → ACC(0)

Affects C.
Not affected by SXM bit.

Description The SFL instruction shifts the entire accumulator left one bit. The least significant bit is filled with a zero, and the most significant bit is shifted into the carry bit (C). Note that SFL, unlike SFR, is unaffected by SXM.

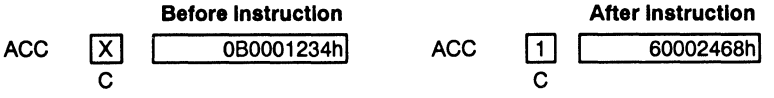
Words 1

Cycles [*label*] **SFL**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
SFL
```



Syntax [*label*] **SFLB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	1	0

Execution

(PC) + 1 → PC
0 → ACCB(0)
(ACCB(30–0)) → ACCB(31–1)
(ACCB(31)) → ACC(0)
(ACC(30–0)) → ACC(31–1)
(ACC(31) → C

Affects C.
Not affected by SXM bit.

Description The SFLB instruction shifts the concatenation of the accumulator (ACC) and accumulator buffer (ACCB) left by one bit position. The least significant bit of the accumulator buffer is filled with a zero, and the most significant bit of the accumulator buffer is shifted into the least significant bit of the accumulator. The most significant bit of the accumulator is shifted into the carry bit (C). The SFLB instruction is unaffected by SXM.

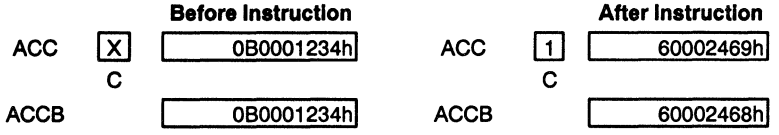
Words 1

Cycles [*label*] **SFLB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
SFLB
```



Syntax [*label*] **SFR**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	1	0	1	0

Execution

(PC) + 1 → PC

If SXM = 0:
Then 0 → ACC(31).
If SXM = 1
Then (ACC(31)) → ACC(31).

(ACC(31–1)) → ACC(30–0)
(ACC(0)) → C

Affects C.
Affected by SXM bit.

Description The SFR instruction shifts the accumulator right one bit.

If SXM = 1, the instruction produces an arithmetic right shift. The sign bit (MSB) is unchanged and is also copied into bit 30. Bit 0 is shifted into the carry bit (C).

If SXM = 0, the instruction produces a logic right shift. All of the accumulator bits are shifted right by one bit. The least significant bit is shifted into the carry bit, and the most significant bit is filled with a zero.

Words 1

Cycles [*label*] **SFR**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example 1

```
SFR     ;(SXM = 0)
```

	Before Instruction		After Instruction	
ACC	[X]	0B0001234h	ACC	[0] 5800091Ah
	C			C

Example 2

```
SFR     ;(SXM = 1)
```

	Before Instruction		After Instruction	
ACC	[X]	0B0001234h	ACC	[0] 0D800091Ah
	C			C

Syntax [*label*] **SFRB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	0	1	1	1

Execution

(PC) + 1 → PC

If SXM=0:
  Then 0 → ACC(31)
If SXM=1:
  Then (ACC(31)) → ACC(31)

(ACC(31–1)) → ACC(30–0)
(ACC(0)) → ACCB (31)
(ACCB(31–1)) → ACCB(30–0)
(ACCB(0)) → C

Affects C.
Affected by SXM.

Description

The SFRB instruction shifts the concatenation of the accumulator (ACC) and accumulator buffer (ACCB) right by one bit position. The LSB of the ACCB is shifted into the carry bit.

If SXM=1, the instruction produces an arithmetic right shift. The sign bit (MSB) of the accumulator is unchanged and is also copied into bit 30. Bit 0 of the accumulator buffer is shifted into the carry bit (C).

If SXM=0, the instruction produces a logic right shift. All of the accumulator and accumulator buffer bits are shifted right by one bit. The least significant bit of the accumulator buffer is shifted into the carry bit, and the most significant bit of the accumulator becomes zero.

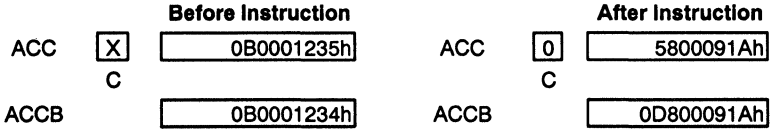
Words 1

Cycles [*label*] **SFRB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

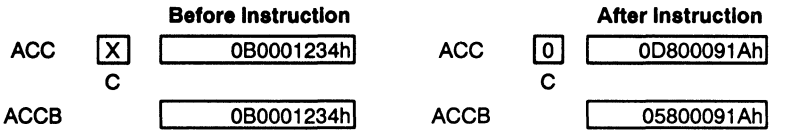
Example 1

```
SFRB   ;(SXM = 0)
```



Example 2

```
SFRB    ;(SXM = 1)
```



Syntax

Direct: [*label*] **SMMR** *dma, #addr*
Indirect: [*label*] **SMMR** {*ind*}, *#addr* [, *next ARP*]

Operands

$0 \le$ addr ≤ 65535
$0 \le$ dma ≤ 127
$0 \le$ next ARP ≤ 7

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

PFC → MCS
(PC) + 2 → PC
1K → PFC
While (repeat counter ≠ 0):
    (src, specified by lower 7 bits of dma) → (dst, addressed by PFC)
    (PFC) + 1 → PFC
    (repeat counter) – 1 → repeat counter
MCS → PFC

Description

The memory-mapped register value pointed at by the lower 7 bits of the data memory address is stored to the data memory location addressed by the 16-bit address, *addr*. The 9 MSBs of the data memory address of the memory-mapped register are set to zero, regardless of the current value of DP or the upper 9 bits of AR(ARP). This instruction allows any memory location on data page 0 to be stored anywhere in data memory without modifying the DP field in status register ST0. When using the SMMR instruction with the RPT instruction, the destination address, *#addr*, is incremented after every memory-mapped store operation.

Words

2

Cycles

Direct: [*label*] **SMMR** *dma, #addr*
Indirect: [*label*] **SMMR** {*ind*}, *#addr* [, *next ARP*]

Cycle Timings for a Single Instruction

	PR	PDA	PSA	PE
Destination DARAM Source MMR‡	2	2	2	$2+2p_{code}$
Destination SARAM Source MMR‡	2	2	2 3†	$2+2p_{code}$
Destination Ext Source MMR‡	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+2p_{code}$

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Destination DARAM Source MMPORT	$3+io_{src}$	$3+io_{src}$	$3+io_{src}$	$4+io_{src}+2p_{code}$
Destination SARAM Source MMPORT	$3+io_{src}$	$3+io_{src}$	$3+io_{src}$ $4+io_{src}$†	$3+io_{src}+2p_{code}$
Destination Ext Source MMPORT	$4+io_{src}+d_{dst}$	$4+io_{src}+d_{dst}$	$4+io_{src}+d_{dst}$	$6+io_{src}+d_{dst}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Destination DARAM Source MMR§	2n	2n	2n	$2n+2p_{code}$
Destination SARAM Source MMR§	2n	2n	2n 2n+2†	$2n+2p_{code}$
Destination Ext Source MMR§	$3n+nd_{dst}$	$3n+nd_{dst}$	$3n+nd_{dst}$	$3n+3+nd_{dst}+2p_{code}$
Destination DARAM Source MMPORT	$2n+nio_{src}$	$2n+nio_{src}$	$2n+nio_{src}$	$2n+1+nio_{src}+2p_{code}$
Destination SARAM Source MMPORT	$2n+nio_{src}$	$2n+nio_{src}$	$2n+nio_{src}$ $2n+2+nio_{src}$†	$2n+1+nio_{src}+2p_{code}$
Destination Ext Source MMPORT	$5n-2+nd_{dst}+nio_{src}$	$5n-2+nd_{dst}+nio_{src}$	$5n-2+nd_{dst}+nio_{src}$	$5n+1+nd_{dst}+nio_{src}+2p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ Add one more cycle if source is a peripheral memory-mapped register.
§ Add *n* more cycles if source is a peripheral memory-mapped register.

Example 1

```
SMMR   CBCR,#307h    ;(DP = 6, CBCR = 1Eh)
```

	Before Instruction		After Instruction
Data Memory 307h	1376h	Data Memory 307h	5555h
CBCR	5555h	CBCR	5555h

Example 2

```
SMMR   *,#307h,AR6   ;(CBCR = 1Eh)
```

	Before Instruction		After Instruction
ARP	6	ARP	6
AR6	0F01Eh	AR6	0F01Eh
Data Memory 307h	1376h	Data Memory 307h	5555h
CBCR	5555h	CBCR	5555h

Syntax [*label*] **SPAC**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	0	0	1	0	1

Execution

(PC) + 1 → PC
(ACC) – (shifted P register) → ACC

Affects OV and C; affected by PM and OVM.
Not affected by SXM.

Description The contents of the P register, shifted as defined by the PM status bits, are subtracted from the contents of the accumulator. The result is stored in the accumulator. Note that SPAC is not affected by the SXM, and the P register is always sign-extended.

The SPAC instruction is a subset of LTS, MPYS, and SQRS.

Words 1

Cycles [*label*] **SPAC**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
SPAC ;(PM = 0).
```

Before Instruction			After Instruction		
P		10000000h	P		10000000h
ACC	X	70000000h	ACC	1	60000000h
	C			C	

Syntax

Direct: [*label*] **SPH** *dma*
Indirect: [*label*] **SPH** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	0	1	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	0	0	1	1	0	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(P register shifter output (31–16)) → dma

Affected by PM.

Description

The high-order bits of the P register, shifted as specified by the PM bits, are stored in data memory. Neither the P register nor the accumulator is affected by this instruction. High-order bits are sign-extended when the right-shift-by-6 mode is selected. Low-order bits are taken from the low P register when left shifts are selected.

Words

1

Cycles

Direct: [*label*] **SPH** *dma*
Indirect: [*label*] **SPH** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
SPH    DAT3   ;(DP = 4, PM = 0).
```

	Before Instruction		After Instruction
P	0FE079844h	P	0FE079844h
203h	4567h	203h	0FE07h

Example 2

```
SPH    *,AR7  ;(PM = 2)
```

	Before Instruction		After Instruction
ARP	6	ARP	7
AR6	203h	AR6	203h
P	0FE079844h	P	0FE079844h
Data Memory 203h	4567h	Data Memory 203h	0E079h

Syntax

Direct: [*label*] **SPL** *dma*
Indirect: [*label*] **SPL** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	0	0	0	Data Memory Address						
Indirect:	1	0	0	0	1	1	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(P register shifter output (15–0)) → dma

Affected by PM.

Description

The low-order bits of the P register, shifted as specified by the PM bits, are stored in data memory. Neither the P register nor the accumulator is affected by this instruction. High-order bits are taken from the high P register when the right-shift-by-6 mode is selected. Low-order bits are zero-filled when left shifts are selected.

Words

1

Cycles

Direct: [*label*] **SPL** *dma*
Indirect: [*label*] **SPL** {*ind*} [,*next ARP*]

	PR	PDA	PSA	PE
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
SPL    DAT5   ;(DP = 1, PM = 2).
```

	Before Instruction		After Instruction
P	0FE079844h	P	0FE079844h
Data Memory 205h	4567h	Data Memory 205h	08440h

Example 2

```
SPL    *,AR3  ;(PM = 0).
```

	Before Instruction		After Instruction
ARP	2	ARP	3
AR2	205h	AR2	205h
P	0FE079844h	P	0FE079844h
Data Memory 205h	4567h	Data Memory 205h	09844h

Syntax

Direct: [*label*] **SPLK** *#lk,dma*
Indirect: [*label*] **SPLK** *#lk, {ind} [,next ARP]*

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
lk: 16-bit constant

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	0	Data Memory Address						
16-Bit Constant															

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	0	1	See Subsection 4.1.2						
16-Bit Constant															

Execution

(PC) + 2 → PC
lk → dma

Description

The SPLK instruction allows a full 16-bit pattern to be written into any memory location. The parallel logic unit (PLU) supports this bit manipulation independently of the ALU so that the ACC is unaffected.

Words

2

Cycles

Direct: [*label*] **SPLK** *#lk,dma*
Indirect: [*label*] **SPLK** *#lk, {ind} [,next ARP]*

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2 3†	2+2p
Operand Ext	3+d	3+d	3+d	5+d+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

Example 1

```
SPLK   #7FFFh,DAT3   ;(DP = 6)
```

	Before Instruction		**After Instruction**
Data Memory 303h	0FE07h	Data Memory 303h	7FFFh

Example 2

```
SPLK   #1111h,*+,AR4
```

	Before Instruction		**After Instruction**
ARP	0	ARP	4
AR4	300h	AR4	301h
Data Memory 300h	07h	Data Memory 300h	1111h

Syntax

[label] **SPM** *constant*

Operands

0 ≤ constant ≤ 3

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	PM †	

† See Section 4.5.

Execution

(PC) + 1 → PC
Constant → product register shift mode (PM) status bits

Affects PM.
Unaffected by SXM.

Description

The two low-order bits of the instruction word are copied into the PM field of status register ST1. The PM status bits control the P register output shifter. This shifter has the ability to shift the P register output either one or four bits to the left or six bits to the right. The bit combinations and their meanings are shown below:

PM	Action
00	No shift of multiplier output
01	Output left-shifted 1 place and zero-filled
10	Output left-shifted 4 places and zero-filled
11	Output right-shifted 6 places, sign-extended; LSB bits lost.

The left-shifts allow the product to be justified for fractional arithmetic. The right shift by six bits has been incorporated to implement up to 128 multiply-accumulate processes without the possibility of overflow occurring. PM may also be loaded by an LST #1 instruction.

Words

1

Cycles

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† Note that ADD, ADRK, LACL, MPY, SBRK, SPM, SUB, XC, and RPT are nonrepeatable.

Example

```
SPM 3   ;Product register shift mode 3 is selected, causing
        ;all subsequent transfers from the product register
        ;to the ALU to be shifted to the right six places.
```

Syntax

Direct: [*label*] **SQRA** *dma*
Indirect: [*label*] **SQRA** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	0	1	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	0	1	0	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) + (shifted P register) → ACC
(dma) → TREG0
(dma) × (dma) → P register

Affects OV and C.
Affected by PM and OVM.

Description

The contents of the P register, shifted as defined by the PM status bits, are added to the accumulator. The addressed data memory value is then loaded into TREG0, squared, and stored in the P register.

Words

1

Cycles

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
SQRA   DAT30  ;(DP = 6, PM = 0).
```

	Before Instruction	After Instruction
Data Memory 31Eh	0Fh	0Fh
TREG0	3h	0Fh
P	12Ch	0E1h
ACC	1F4h	320h
C	X	0

Example 2

```
SQRA   *,AR4  ;(PM = 0).
```

	Before Instruction	After Instruction
ARP	3	4
AR3	31Eh	31Eh
Data Memory 31Eh	0Fh	0Fh
TREG0	3h	0Fh
P	12Ch	0E1h
ACC	1F4h	320h
C	X	0

Syntax

Direct: [*label*] **SQRS** *dma*
Indirect: [*label*] **SQRS** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	0	0	1	1	0	Data Memory Address						
Indirect:	0	1	0	1	0	0	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) – (shifted P register) → ACC
(dma) → TREG0
(dma) × (dma) → P register

Affects OV and C.
Affected by PM and OVM.

Description

The contents of the P register, shifted as defined by the PM status bits, are subtracted from the accumulator. The addressed data memory value is then loaded into TREG0, squared, and stored in the P register.

Words

1

Cycles

Direct: [*label*] **SQRS** *dma*
Indirect: [*label*] **SQRS** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Example 1

```
SQRS   DAT9   ;(DP = 6, PM = 0).
```

	Before Instruction	After Instruction
Data Memory 309h	08h	08h
TREG0	1124h	08h
P	190h	40h
ACC	1450h	12C0h
X		1
C		C

Example 2

```
SQRS   *,AR5   ;(PM = 0)
```

	Before Instruction	After Instruction
ARP	3	5
AR3	309h	309h
Data Memory 309h	08h	08h
TREG0	1124h	08h
P	190h	40h
ACC	1450h	12C0h
X		1
C		C

Syntax

Direct: [*label*] **SST** #*n*, *dma*
Indirect: [*label*] **SST** #*n*, {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
n = 0,1
0 ≤ next ARP ≤ 7

Opcode

Store Status Register 0 SST#0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	1	0	0	Data Memory Address						
Indirect:	1	0	0	0	1	1	1	0	1	See Subsection 4.1.2						

Store Status Register 1 SST#1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	0	0	1	1	1	1	0	Data Memory Address						
Indirect:	1	0	0	0	1	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(status register STn) → dma

Description

Status register STn is stored in data memory. In the direct addressing mode, status register STn is always stored in page 0, regardless of the value of the DP register. The processor automatically forces the page to be 0, and the specified location within that page is defined in the instruction. Note that the DP register is not physically modified. This allows storage of the DP register in the data memory on interrupts, etc., in the direct addressing mode without having to change the DP. In the indirect addressing mode, the data memory address is obtained from the auxiliary register selected (see the LST instruction for more information). In the indirect addressing mode, any page in data memory may be accessed.

Status registers ST0 and ST1 are defined in subsection 3.6.3, *Status and Control Registers.*

Words

1

Cycles

Direct: [*label*] **SST** #*n*, *dma*
Indirect: [*label*] **SST** #*n*, {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Example 1

```
SST   #0,DAT96    ;(DP = 6)
```

	Before Instruction		After Instruction
ST0	0A408h	ST0	0A408h
Data Memory 60h	0Ah	Data Memory 60h	0A408h

Example 2

```
SST   #1,*,AR7
```

	Before Instruction		After Instruction
ARP	0	ARP	7
AR0	300h	AR0	300h
ST1	2580h	ST1	2580h
Data Memory 300h	0h	Data Memory 300h	2580h

Syntax

Direct:	[*label*] **SUB** *dma* [,*shift1*]
Indirect:	[*label*] **SUB** {*ind*} [,*shift1* [,*next ARP*]]
Short Immediate:	[*label*] **SUB** *#k*
Long Immediate:	[*label*] **SUB** *#lk* [,*shift2*]

Operands

$0 \le dma \le 127$
$0 \le shift1 \le 16$ (defaults to 0)
$0 \le next\ ARP \le 7$
$0 \le k \le 255$
$-32768 \le lk \le 32767$
$0 \le shift2 \le 15$ (defaults to 0)

Opcode

Subtract from accumulator with shift

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	1	1	SHFT †				0	Data Memory Address						
Indirect:	0	0	1	1	SHFT †				1	See Subsection 4.1.2						

Subtract from accumulator with shift of 16

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	0	1	0	Data Memory Address						
Indirect:	0	1	1	0	0	1	0	1	1	See Subsection 4.1.2						

Subtract from ACC short immediate

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Short:	1	0	1	1	1	0	1	0	8-Bit Constant							

Subtract from ACC long immediate with shift

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Long:	1	0	1	1	1	1	1	1	1	0	1	0	SHFT †			
	16-Bit Constant															

† See Section 4.5.

Execution

Direct or Indirect Addressing:

(PC) + 1 → PC
(ACC) – [(dma) × 2^{shift1}] → ACC
Affects C and OV.
Affected by SXM and OVM.

Short Immediate Addressing:

(PC) + 1 → PC
(ACC) – k → ACC
Affects C and OV.
Affected by OVM.

Long Immediate Addressing:

(PC) + 2 → PC
(ACC) − lk × 2^{shift2} → ACC
Affects C and OV.
Affected by SXM and OVM.

Description

The contents of the addressed data memory location or a 16-bit constant are left-shifted and subtracted from the accumulator if direct, indirect, or long immediate addressing is used. During shifting, low-order bits are zero-filled. High-order bits are sign-extended if SXM = 1 and zero-filled if SXM = 0. The result is then stored in the accumulator.

When short immediate addressing is used, an 8-bit positive constant is subtracted from the accumulator. In this case, no shift value may be specified, the subtraction is unaffected by SXM, and the instruction is not repeatable.

The carry bit is reset to zero if the result of a subtraction generates a borrow; otherwise, it is set to 1. If a 16-bit shift is specified with the subtraction, the instruction may reset the carry bit to 0 only if the result of the subtraction generates a borrow; otherwise, C is unaffected.

Words

1 (Direct, indirect, or short immediate)

2 (Long immediate)

Cycles

Direct: [*label*] **SUB** *dma* [,*shift1*]
Indirect: [*label*] **SUB** {*ind*} [,*shift1* [,*next ARP*]]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Short Immediate: [*label*] **SUB** *#k*

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p

Cycle Timings for a Repeat (RPT) Execution			
PR	**PDA**	**PSA**	**PE**
Not Repeatable			

Long Immediate: [*label*] **SUB** *#lk* [,*shift2*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
SUB    DAT80  ;(DP = 8, SXM=0)
```

	Before Instruction	After Instruction
Data Memory 450h	11h	11h
ACC	24h	13h
C	X	1

Example 2

```
SUB    *-,1,AR0 ;(SXM = 0)
```

	Before Instruction	After Instruction
ARP	7	0
Data Memory AR7	301h	300h
301h	04h	04h
ACC	09h	01h
C	X	1

Example 3

```
SUB    #8h    ;(SXM = 1)
```

	Before Instruction	After Instruction
ACC	07h	0FFFFFFFFh
C	X	0

Example 4

```
SUB    #0FFFh,4  ;(SXM = 0)
```

	Before Instruction	After Instruction
ACC	0FFFFh	0Fh
C	X	1

Syntax

Direct: [*label*] **SUBB** *dma*
Indirect: [*label*] **SUBB** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	0	0	0	Data Memory Address						
Indirect:	0	1	1	0	0	1	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) – (dma) – (logical inversion of C) → ACC

Affects OV and C.
Affected by OVM.
Not affected by SXM.

Description

The contents of the addressed data memory location and the logical inversion of the carry bit are subtracted from the accumulator with sign extension suppressed. The carry bit is then affected in the normal manner.

The SUBB instruction can be used in performing multiple-precision arithmetic.

Words

1

Cycles

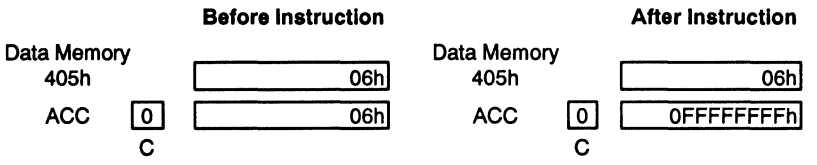
Direct: [*label*] **SUBB** *dma*
Indirect: [*label*] **SUBB** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

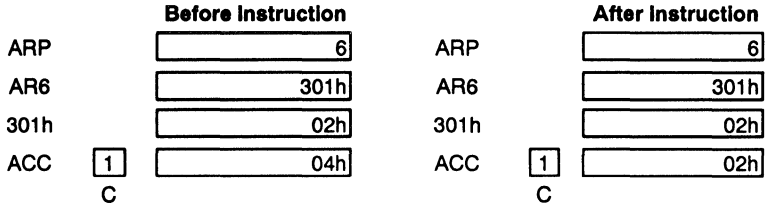
Example 1

```
SUBB   DAT5   ;(DP = 8)
```



Example 2

```
SUBB   *
```



In the first example, C is originally zeroed, presumably from the result of a previous subtract instruction that performed a borrow. The effective operation performed was 6 – 6 – (0–) = –1, generating another borrow (resetting carry) in the process. In the second example, no borrow was previously generated (C=1), and the result from the subtract instruction does not generate a borrow.

Syntax

Direct: [*label*] **SUBC** *dma*
Indirect: [*label*] **SUBC** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	0	0	0	1	0	1	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	0	0	0	1	0	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) – [(dma) × 2^{15}] → ALU output

If ALU output ≥ 0:
Then (ALU output) × 2 + 1 → ACC;
Else (ACC) × 2 → ACC.

Affects OV and C.
Affected by SXM.
Not affected by SXM, and OVM (no saturation).

Description

The SUBC instruction performs conditional subtraction, which may be used for division. The 16-bit dividend is placed in the low accumulator, and the high accumulator is zeroed. The divisor is in data memory. SUBC is executed **16** times for 16-bit division. After completion of the last SUBC, the quotient of the division is in the lower-order 16-bit field of the accumulator, and the remainder is in the higher-order 16-bits of the accumulator. SUBC assumes that the divisor and the dividend are both positive. The divisor is not sign extended. The dividend, which is in the accumulator, must initially be positive (that is, bit 31 must be **0**) and must remain positive following the accumulator shift, which occurs in the first portion of the SUBC execution.

If the 16-bit dividend contains fewer than 16 significant bits, the dividend may be placed in the accumulator and left-shifted by the number of leading nonsignificant zeroes. The number of executions of SUBC is reduced from 16 by that number. One leading zero is always significant.

Note that SUBC affects OV but is not affected by OVM, and therefore the accumulator does not saturate upon positive or negative overflows when executing this instruction. The carry bit is affected in the normal manner during this instruction.

Words

1

Cycles

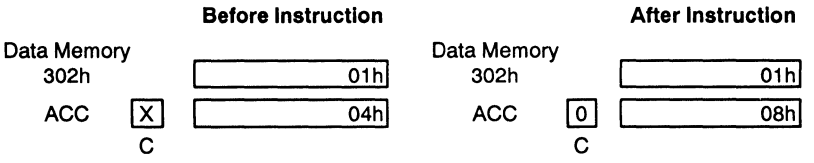
Direct: [*label*] **SUBC** *dma*
Indirect: [*label*] **SUBC** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

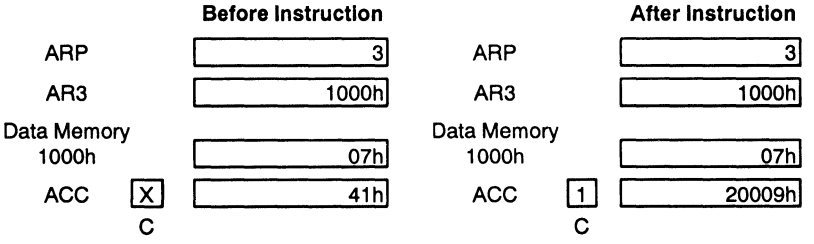
Example 1

```
SUBC   DAT2   ;(DP = 6)
```



Example 2

```
RPT    #15
SUBC   *
```



Syntax

Direct: [*label*] **SUBS** *dma*
Indirect: [*label*] **SUBS** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	1	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	0	1	1	0	0	1	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) – (dma) → ACC

Affects OV and C; affected by OVM.
Not affected by SXM.

Description

The contents of the specified data memory location are subtracted from the accumulator with sign extension suppressed. The data is treated as a 16-bit unsigned number, regardless of SXM. The accumulator behaves as a signed number. SUBS produces the same results as a SUB instruction with SXM = 0 and a shift count of 0.

Words

1

Cycles

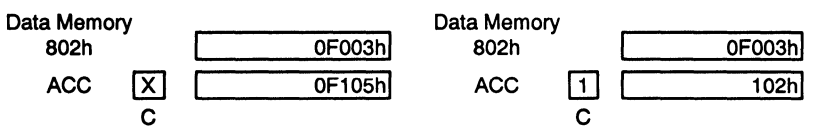
Direct: [*label*] **SUBS** *dma*
Indirect: [*label*] **SUBS** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

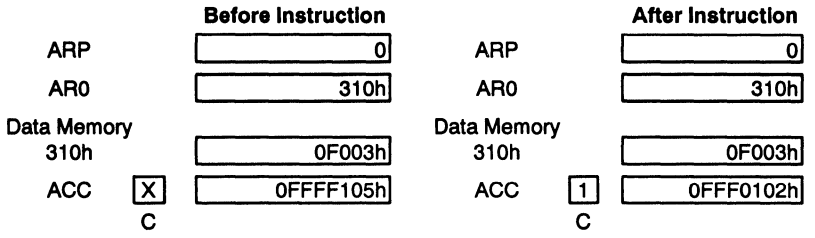
Example 1

```
SUBS   DAT2   ;(DP = 16, SXM = 1).
```



Example 2

```
SUBS   *   ;(SXM = 1)
```



Syntax

Direct: [*label*] **SUBT** *dma*
Indirect: [*label*] **SUBT** {ind} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	0	1	1	1	0	Data Memory Address						
Indirect:	0	1	1	0	0	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(ACC) − [(dma) × $2^{TREG1(3-0)}$] → (ACC)

If SXM = 1:
Then (dma) is sign-extended.
If SXM = 0:
Then (dma) is not sign-extended.

Affects OV and C; affected by SXM and OVM.

Description

The data memory value is left-shifted and subtracted from the accumulator. The left-shift is defined by the four LSBs of TREG1, resulting in shift options from 0 to 15 bits. The result replaces the accumulator contents. Sign extension on the data memory value is controlled by the SXM status bit.

Software compatibility with the 'C25 can be maintained by setting the TRM bit of the PMST status register to zero. This causes any 'C25 instruction that loads TREG0 to write to all three TREGs.

Words

1

Cycles

Direct: [*label*] **SUBT** *dma*
Indirect: [*label*] **SUBT** {ind} [,*next ARP*]

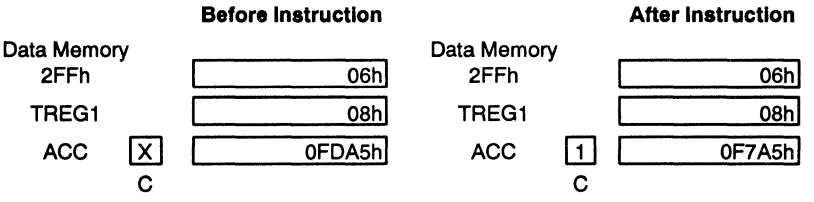
Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

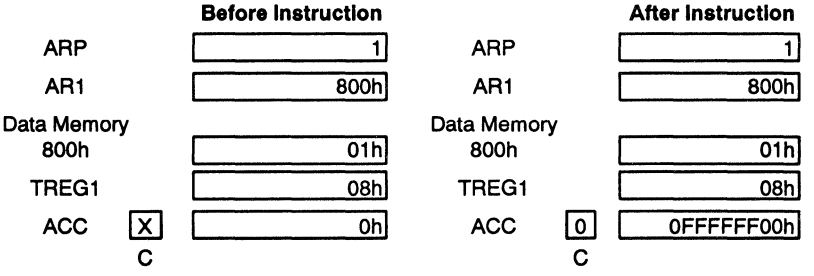
Example 1

```
SUBT   DAT127  ;(DP = 4)
```



Example 2

```
SUBT   *
```



Syntax

Direct: [*label*] **TBLR** *dma*
Indirect: [*label*] **TBLR** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	0	1	1	0	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	0	1	1	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(PFC) → MCS
(ACC(15–0)) → PFC

If (repeat counter) ≠ 0:
  Then (pma, addressed by PFC) → dma,
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (repeat counter) –1 → repeat counter.
  Else (pma, addressed by PFC) → dma,
    Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Description

The TBLR instruction transfers a word from a location in program memory to a data memory location specified by the instruction. The program memory address is defined by the low-order 16 bits of the accumulator. For this operation, a read from program memory is performed, followed by a write to data memory. When the repeat mode is used, TBLR effectively becomes a single-cycle instruction, and the program counter that contains the ACCL is incremented once each cycle.

Words

1

Cycles

Direct: [*label*] **TBLR** *dma*
Indirect: [*label*] **TBLR** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction

	PR	PDA	PSA	PE
Source DARAM/ROM Destination DARAM	3	3	3	$3+p_{code}$
Source SARAM Destination DARAM	3	3	3	$3+p_{code}$
Source Ext Destination DARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$	$3+p_{src}+p_{code}$
Source DARAM/ROM Destination SARAM	3	3	3 4†	$3+p_{code}$

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination SARAM	3	3	3 4†	3+p_{code}
Source Ext Destination SARAM	3+p_{src}	3+p_{src}	3+p_{src} 4+p_{src}†	3+p_{src}+p_{code}
Source DARAM/ROM Destination Ext	4+d_{dst}	4+d_{dst}	4+d_{dst}	6+d_{dst}+p_{code}
Source SARAM Destination Ext	4+d_{dst}	4+d_{dst}	4+d_{dst}	6+d_{dst}+p_{code}
Source Ext Destination Ext	4+p_{src}+d_{dst}	4+p_{src}+d_{dst}	4+p_{src}+d_{dst}	6+p_{src}+d_{dst}+p_{code}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	n+2	n+2	n+2	n+2+p_{code}
Source SARAM Destination DARAM	n+2	n+2	n+2	n+2+p_{code}
Source Ext Destination DARAM	n+2+np_{src}	n+2+np_{src}	n+2+np_{src}	n+2+np_{src}+p_{code}
Source DARAM/ROM Destination SARAM	n+2	n+2	n+2 n+4†	n+2+p_{code}
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ 2n+2§	n+2+p_{code} 2n‡
Source Ext Destination SARAM	n+2+np_{src}	n+2+np_{src}	n+2+np_{src} n+4+np_{src}†	n+2+np_{src}+p_{code}
Source DARAM/ROM Destination Ext	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+4+nd_{dst}+p_{code}
Source SARAM Destination Ext	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+4+nd_{dst}+p_{code}
Source Ext Destination Ext	4n+np_{src}+nd_{dst}	4n+np_{src}+nd_{dst}	4n+np_{src}+nd_{dst}	4n+2+np_{src}+nd_{dst}+p_{code}

† If the destination operand and the code are in the same SARAM block.

‡ If both the source and the destination operands are in the same SARAM block.

§ If both operands and the code are in the same SARAM block.

Example 1

```
TBLR   DAT6   ;(DP = 4)
```

	Before Instruction		After Instruction
ACC	23h	ACC	23h
Program Memory 23h	306h	Program Memory 23h	306h
Data Memory 206h	75h	Data Memory 206h	306h

Example 2

```
TBLR   *,AR7
```

	Before Instruction		After Instruction
ARP	0	ARP	7
AR0	300h	AR0	300h
ACC	24h	ACC	24h
Program Memory 24h	307h	Program Memory 24h	307h
Data Memory 300h	75h	Data Memory 300h	307h

Syntax

Direct: [*label*] **TBLW** *dma*
Indirect: [*label*] **TBLW** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	1	0	1	0	0	1	1	1	0	Data Memory Address						
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Indirect:	1	0	1	0	0	1	1	1	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
(PFC) → MCS
(ACC(15–0)) → PFC

If (repeat counter) ≠ 0:
  Then (dma, addressed by PFC) → pma,
    Modify AR(ARP) and ARP as specified,
    (PFC) + 1 → PFC
    (repeat counter) –1 → repeat counter.
  Else (dma, addressed by PFC) → pma,
    Modify AR(ARP) and ARP as specified.
(MCS) → PFC

Description

The TBLW instruction transfers a word in data memory to program memory. The data memory address is specified by the instruction, and the program memory address is specified by the lower 16 bits of the accumulator. A read from data memory is followed by a write to program memory to complete the instruction. When the repeat mode is used, TBLW effectively becomes a single-cycle instruction, and the program counter that contains the ACCL is incremented once each cycle.

Words

1

Cycles

Direct: [*label*] **TBLW** *dma*
Indirect: [*label*] **TBLW** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction

	PR	PDA	PSA	PE
Source DARAM Destination DARAM	3	3	3	$3+p_{code}$
Source SARAM Destination DARAM	3	3	3	$3+p_{code}$
Source Ext Destination DARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$	$3+d_{src}+p_{code}$
Destination SARAM Source DARAM	3	3	3 4†	$3+p_{code}$

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination SARAM	3	3	3 4†	3+p_{code}
Source Ext Destination SARAM	3+d_{src}	3+d_{src}	3+d_{src} 4+d_{src}†	3+d_{src}+p_{code}
Source DARAM Destination Ext	4+p_{dst}	4+p_{dst}	4+p_{dst}	5+p_{dst}+p_{code}
Source SARAM Destination Ext	4+p_{dst}	4+p_{dst}	4+p_{dst}	5+p_{dst}+p_{code}
Source Ext Destination Ext	4+d_{src}+p_{dst}	4+d_{src}+p_{dst}	4+d_{src}+p_{dst}	5+d_{src}+p_{dst}+p_{code}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+2	n+2	n+2	n+2+p_{code}
Source SARAM Destination DARAM	n+2	n+2	n+2	n+2+p_{code}
Source Ext Destination DARAM	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src}+p_{code}
Source DARAM Destination SARAM	n+2	n+2	n+2 n+3†	n+2+p_{code}
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ 2n+1§	n+2+p_{code} 2n‡
Source Ext Destination SARAM	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src} n+3+nd_{src}†	n+2+nd_{src}+p_{code}
Source DARAM Destination Ext	2n+2+np_{dst}	2n+2+np_{dst}	2n+2+np_{dst}	2n+3+np_{dst}+p_{code}
Source SARAM Destination Ext	2n+2+np_{dst}	2n+2+np_{dst}	2n+2+np_{dst}	2n+3+np_{dst}+p_{code}
Source Ext Destination Ext	4n+nd_{src}+np_{dst}	4n+nd_{src}+np_{dst}	4n+nd_{src}+np_{dst}	4n+1+nd_{src}+np_{dst}+ p_{code}

† If the destination operand and the code are in the same SARAM block.

‡ If both the source and the destination operands are in the same SARAM block.

§ If both operands and the code are in the same SARAM block.

Example 1

```
TBLW   DAT5   ;(DP = 32)
```

	Before Instruction		After Instruction
ACC	257h	ACC	257h
Data Memory 1905h	4339h	Data Memory 1905h	4339h
Program Memory 257h	306h	Program Memory 257h	4399h

Example 2

```
TBLW   *
```

	Before Instruction		After Instruction
ARP	6	ARP	6
AR6	1006h	AR6	1006h
ACC	258h	ACC	258h
Data Memory 1006h	4340h	Data Memory 1006h	4340h
Program Memory 258h	307h	Program Memory 258h	4340h

Syntax [*label*] **TRAP**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	0	0	0	1

Execution

(PC) + 1 → stack
22h → PC

Not affected by INTM; does not affect INTM.

Description The TRAP instruction is a software interrupt that transfers program control to program memory location 22h and pushes the program counter plus one onto the hardware stack. The instruction at location 22h may contain a branch instruction to transfer control to the TRAP routine. Putting the PC + 1 onto the stack enables a return instruction to pop the return PC (points to the instruction after the TRAP) from the stack. The TRAP instruction is not maskable.

Words 1

Cycles [*label*] **TRAP**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
4	4	4	4+3p†
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Example

```
TRAP   ;Control is passed to program memory location 22h and
       ;PC + 1 is pushed onto the stack.
```

Syntax

[*label*] **XC** *k* [,*cond1*] [,*cond2*] [,...]

Operands

k = 1 or 2

Conditions:

ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
$\overline{\text{BIO}}$ low	BIO
TC=0	NTC
TC=1	TC
Unconditional	UNC

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	N †	0	1	TP †		ZLVC †				ZLVC †			

† See Section 4.5.

Execution

If (condition(s))
Then next **k** instructions executed
Else execute NOP's for next **k** instructions

Description

If k = 2 and conditions are met, the one two-word instruction or two one-word instructions following the XC instruction execute. If k = 1 and conditions are met, the one-word instruction following the XC instruction executes. If the conditions are not met, one or two NOPs are executed. Note that not all combinations of conditions are meaningful. The XC instruction and two-instruction words following the XC are uninterruptible.

Conditions tested are sampled one full cycle before the XC is executed. Therefore, if the instruction before the XC is a single-cycle instruction, its execution will not affect the condition of the XC. If the instruction prior to the XC does affect the condition being tested, interrupt operation with the XC can cause undesired results.

Words

1

Cycles

[*label*] **XC** *k* [,*cond1*] [,*cond2*] [,...]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example

```
XC  1,LEQ,C
MAR  *+
ADD DAT100
```

If the accumulator contents are less than or equal to zero and the carry bit is set, the ARP is modified prior to the execution of the ADD instruction.

Syntax

Direct:	[*label*] **XOR** *dma*
Indirect:	[*label*] **XOR** {*ind*} [,*next ARP*]
Long Immediate:	[*label*] **XOR** #*lk*, [,*shift*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7
lk: 16-bit constant
0 ≤ shift ≤ 16

Opcode

Direct:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	Data Memory Address						

Indirect:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	1	See Subsection 4.1.2						

Long:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	0	1	SHFT †			
16-Bit Constant															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	1	0	0	0	0	0	1	1
16-Bit Constant															

XOR with ACC long immediate with shift of 16

† See Section 4.5.

Execution

Direct or Indirect Addressing:
(PC) + 1 → PC
(ACC(15–0)) **XOR** dma → ACC(15–0)
(ACC(31–16)) → ACC(31–16)

Long Immediate Addressing:
(PC) + 2 → PC
(ACC(31–0)) **XOR** (lk × 2^{shift}) → ACC(31–0)

Description

With direct or indirect addressing, the low half of the accumulator is XORed with the contents of the addressed data memory location; the upper half of the accumulator is unaffected. With immediate addressing, the long immediate constant is shifted and zero-extended on both ends and XORed with the high- and low-order bits of the accumulator. The carry bit (C) is unaffected by XOR.

Words

1 (Direct or indirect addressing)

2 (Long immediate addressing)

Cycles

Direct: [*label*] **XOR** *dma*
Indirect: [*label*] **XOR** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

Long Immediate: [*label*] **XOR** *#lk*, [,*shift*]

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Example 1

```
XOR    DAT127 ;(DP = 511)
```

	Before Instruction	After Instruction
Data Memory 0FFFFh	0F0F0h	0F0F0h
ACC	12345678h	1234A688h
C	X	X

Example 2

```
XOR    *+,AR0
```

	Before Instruction	After Instruction
ARP	7	0
AR7	300h	301h
Data Memory 300h	0FFFFh	0FFFFh
ACC	1234F0F0h	12340F0Fh
C	X	X

Example 3

```
XOR    #0F0F0h,4
```

	Before Instruction	After Instruction
ACC	11111010h	111E1F10h
C	X	X

Syntax [*label*] **XORB**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	0	1	1	0	1	0

Execution

(PC) + 1 → PC
(ACC) **XOR** (ACCB) → ACC

Description The contents of the accumulator buffer (ACCB) are exclusive-ORed with the contents of the accumulator. The results are placed in the accumulator, and the accumulator buffer is unaffected.

Words 1

Cycles [*label*] **XORB**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
XORB
```

	Before Instruction		After Instruction
ACCB	0F0F0F0F0h	ACCB	0F0F0F0F0h
ACC	0FFFF0000h	ACC	0F0FF0F0h

Syntax

Direct: [*label*] **XPL** [#*lk*,] *dma*
Indirect: [*label*] **XPL** [#*lk*,] {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
lk: 16-bit constant
0 ≤ next ARP ≤ 7

Opcode

XOR DBMR with data value

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	1	0	0	0	0	Data Memory Address						
Indirect:	0	1	0	1	1	0	0	0	1	See Subsection 4.1.2						

XOR long immediate with data value

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	0	1	1	1	0	0	0	Data Memory Address						
	16-Bit Constant															
Indirect:	0	1	0	1	1	1	0	0	1	See Subsection 4.1.2						
	16-Bit Constant															

Execution

lk unspecified:
(PC) + 1 → PC
(dma) **XOR** (DBMR) → dma

lk specified:
(PC) + 2 → PC
(dma) **XOR** lk → dma
Affects TC.

Description

If a long immediate constant is specified, it is XORed with the addressed data memory value. If it is not specified, the addressed data memory value is XORed with the contents of the dynamic bit manipulation register (DBMR). In either case, the result is written back into the specified data memory location, and the accumulator contents are not disturbed. If the result of the XOR operation is 0, then the TC bit is set to 1. Otherwise, the TC bit is set to 0.

Words

1 (Long immediate value not specified)

2 (Long immediate value specified)

Cycles

Direct: [*label*] **XPL** [#*lk*,] *dma*
Indirect: [*label*] **XPL** [#*lk*,] {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 3†	1+p
Operand Ext	2+2d	2+2d	2+2d	5+2d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	2n–2	2n–2	2n–2 2n+1†	2n–2+p
Operand Ext	4n–2+2nd	4n–2+2nd	4n–2+2nd	4n+1+2nd+p

† If the operand and the code are in the same SARAM block.

Direct: [*label*] **XPL** [*#lk,*] *dma*
Indirect: [*label*] **XPL** [*#lk,*] {*ind*} [*,next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2	2+2p
Operand Ext	3+2d	3+2d	3+2d	6+2d+2p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n+1	n+1	n+1	n+1+2p
Operand SARAM	2n–1	2n–1	2n–1 2n+2†	2n–1+2p
Operand Ext	4n–1+2nd	4n–1+2nd	4n–1+2nd	4n+2+2nd+2p

† If the operand and the code reside in same SARAM block.

Example 1

```
XPL    #100h,DAT60 ;(DP = 0)
```

	Before Instruction		**After Instruction**
Data Memory 60h	01h	Data Memory 60h	101h

Example 2

```
XPL    DAT60  ;(DP=0)
```

	Before Instruction		**After Instruction**
DBMR	0FFFFh	DBMR	0FFFFh
Data Memory 60h	101h	Data Memory 60h	0FEFEh

Example 3

```
XPL   #1000h,*,AR6
```

	Before Instruction		After Instruction
ARP	0	ARP	6
AR0	300h	AR0	300h
Data Memory 300h	0FF00h	Data Memory 300h	0EF00h

Example 4

```
XPL   *-,AR0
```

	Before Instruction		After Instruction
ARP	6	ARP	0
AR6	301h	AR6	300h
DBMR	0FF00h	DBMR	0FF00h
Data Memory 301h	0EF00h	Data Memory 301h	1000h

Syntax

Direct: [*label*] **ZALR** *dma*
Indirect: [*label*] **ZALR** {*ind*} [,*next ARP*]

Operands

0 ≤ dma ≤ 127
0 ≤ next ARP ≤ 7

Opcode

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Direct:	0	1	1	0	1	0	0	0	0	Data Memory Address						
Indirect:	0	1	1	0	1	0	0	0	1	See Subsection 4.1.2						

Execution

(PC) + 1 → PC
8000h → ACC(15–0)
(dma) → ACC(31–16)

Description

To load a data memory value into the high-order half of the accumulator, the ZALR instruction rounds the value by adding 1/2 LSB; that is, the 15 low bits (bits 0–14) of the accumulator are set to zero, and bit 15 of the accumulator is set to one.

Words

1

Cycles

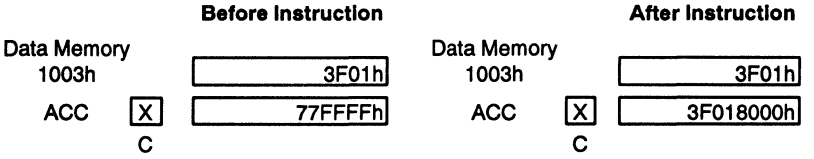
Direct: [*label*] **ZALR** *dma*
Indirect: [*label*] **ZALR** {*ind*} [,*next ARP*]

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.

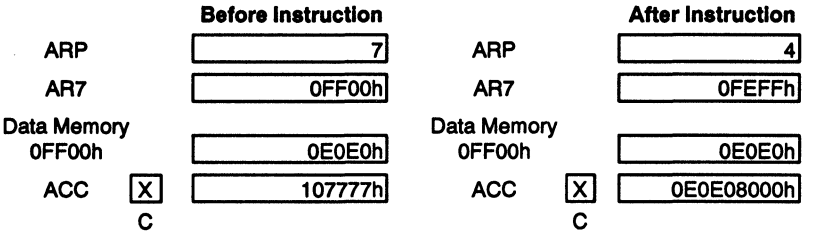
Example 1

```
ZALR   DAT3   ;(DP = 32)
```



Example 2

```
ZALR   *-,AR4
```



Syntax [*label*] **ZAP**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	0	1

Execution

(PC) + 1 → PC
0 → ACC
0 → PREG

Description The accumulator and product register are zeroed. The ZAP instruction speeds up the preparation for a repeat multiply/accumulate.

Words 1

Cycles [*label*] **ZAP**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ZAP
```

	Before Instruction		After Instruction
PREG	3F011111h	PREG	00000000h
ACC	77FFFF77h	ACC	00000000h

Syntax [*label*] **ZPR**

Operands None

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	0	0

Execution

(PC) + 1 → PC
0 → PREG

Description The product register is set to zero.

Words 1

Cycles [*label*] **ZPR**

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution			
n	n	n	n+p

Example

```
ZPR
```

	Before Instruction		After Instruction
PREG	3F011111h	PREG	00000000h

4.4 'C2x-to-'C5x Instruction Set Mapping

Table 4–5 provides a map between the 'C2x and 'C5x instruction sets. The Texas Instruments 'C5x assembler accepts instruction mnemonics from either instruction set. Because the 'C5x instruction set is a superset of the 'C2x instruction set, there are some 'C5x instructions that do not appear in the table.

Table 4–6. Mapping Summary

Accumulator Memory Reference Instructions	
'C2x Mnemonic	**'C5x Mnemonic**
ABS	ABS
ADD	ADD
ADDC	ADDC
ADDH	ADD
ADDK	ADD
ADDS	ADDS
ADDT	ADDT
ADLK	ADD
AND	AND
ANDK	AND
CMPL	CMPL
LAC	LACC
LACK	LACL
LACT	LACT
LALK	LACC
NEG	NEG
NORM	NORM†
OR	OR
ORK	OR
ROL	ROL
ROR	ROR
SACH	SACH
SACL	SACL
SBLK	SUBB
SFL	SFL
SFR	SFR
SUB	SUB
SUBB	SUBB

† There is a potential pipeline conflict with the NORM instruction. See the NORM instruction summary for details.

Table 4–6. Mapping Summary (Continued)

Accumulator Memory Reference Instructions (Concluded)	
'C2x Mnemonic	**'C5x Mnemonic**
SUBC	SUBC
SUBH	SUB
SUBK	SUB
SUBS	SUBS
SUBT	SUBT
XOR	XOR
XORK	XOR
ZAC	LACL
ZALH	LACC
ZALR	ZALR
ZALS	LACL
Auxiliary Registers and Data Page Pointer Instructions	
'C2x Mnemonic	**'C5x Mnemonic**
ADRK	ADRK
CMPR	CMPR
LAR	LAR
LARK	LAR
LARP	MAR
LDP	LDP
LDPK	LDP
LRLK	LAR
MAR	MAR
SAR	SAR
SBRK	SBRK

Table 4–6. Mapping Summary (Continued)

T Register, P Register, and Multiply Instructions	
'C2x Mnemonic	**'C5x Mnemonic**
APAC	APAC
LPH	LPH
LT	LT
LTA	LTA
LTD	LTD
LTP	LTP
LTS	LTS
MAC	MAC
MACD	MACD
MPY	MPY
MPYA	MPYA
MPYK	MPY
MPYS	MPYS
MPYU	MPYU
PAC	PAC
SPAC	SPAC
SPH	SPH
SPL	SPL
SPM	SPM
SQRA	SQRA
SQRS	SQRS
Branch/Call Instructions	
'C2x Mnemonic	**'C5x Mnemonic**
B	B
BACC	BACC
BANZ	BANZ
BBNZ	BCND
BBZ	BCND
BC	BCND
BGEZ	BCND

Table 4–6. Mapping Summary (Continued)

Branch/Call Instructions (Concluded)	
'C2x Mnemonic	**'C5x Mnemonic**
BGZ	BCND
BIOZ	BCND
BLEZ	BCND
BLZ	BCND
BNC	BCND
BNV	BCND
BNZ	BCND
BV	BCND
BZ	BCND
CALA	CALA
CALL	CALL
RET	RET
TRAP	TRAP
I/O and Data Memory Operations	
'C2x Mnemonic	**'C5x Mnemonic**
BLKD	BLDD
BLKP	BLPD
DMOV	DMOV
FORT†	OPL APL
IN	IN
OUT	OUT
RFSM†	APL
RTXM†	APL
RXF	CLRC
SFSM†	OPL
STXM	OPL
SXF	SETC
TBLR	TBLR
TBLW	TBLW

† The suggested mapping requires that the data page pointer be set to 0.

Table 4–6. Mapping Summary (Concluded)

Control Instructions	
'C2x Mnemonic	**'C5x Mnemonic**
BIT	BIT
BITT	BITT
CNFD	CLRC
CNFP	SETC
DINT	SETC
EINT	CLRC
IDLE	IDLE
LST	LST
LST1	LST
NOP	NOP
POP	POP
POPD	POPD
PSHD	PSHD
PUSH	PUSH
RC	CLRC
RHM	CLRC
ROVM	CLRC
RPT	RPT
RPTK	RPT
RSXM	CLRC
RTC	CLRC
SC	SETC
SHM	SETC
SOVM	SETC
SST	SST
SST1	SST
SSXM	SETC
STC	SETC

4.5 Instruction Set Opcode

This section summarizes the opcodes of the instruction set for the 'C5x digital signal processors. This instruction set is a superset of the 'C1x and 'C2x instruction sets. The instructions are arranged according to function and are alphabetized within each category.

The following symbols are used in the opcode table:

Symbol	Meaning
A	Data memory address bit.
A R X	Three-bit field containing the auxiliary register value (0 – 7).
B I T X	Four-bit field specifies which bit to test for the BIT instruction.
C M	See CMPR instruction.
I	Addressing mode bit. 0 = direct addressing mode 1 = indirect addressing mode
I I I I I I I I	Short Immediate value.
I N T R #	Interrupt vector number.
P M	Constant copied into PM bits in status register ST1. See SPM instruction.
S H F	Three-bit shift value.
S H F T	Four-bit shift value.
N	Field for the XC instruction indicating the number of instructions (one or two) to conditionally execute. N=1 One instruction to execute. N=2 Two instruction to execute.
T P	Two bits used by the conditional execution instructions to represent the conditions TC, NTC, and BIO. **TP Meaning** 0 0 $\overline{\text{BIO}}$ low 0 1 TC=1 1 0 TC=0 1 1 None of the above condition.
Z L V C	Four-bit field representing the following conditions: Z: ACC = 0 L: ACC < 0 V: Overflow C: Carry A conditional instruction contains two of these four-bit fields. The four-LSB field of the instruction is a four-bit mask field. A one in the corresponding mask bit indicates that condition is being tested. The second four-bit field (bits 4 – 7) indicates the state of the conditions designated by the mask bits as being tested. For example, to test for ACC ≥ 0, the Z and L fields are set, while the V and C fields are not set. The next four-bit field contains the state of the conditions to test. The Z field is set to indicate to test the condition ACC = 0, and the L field is reset to indicate to test the condition ACC ≥ 0. The conditions possible with these 8 bits are shown in the BCND, CC, and XC instructions. To determine if the conditions are met, the four LSB bit mask is ANDed with the conditions. If any bits are set, the conditions are met.
+ 1 word	Indicates the instruction is a two-word instruction. The second word is a 16-bit long immediate value or a 16-bit program memory address for immediate addressing.

Table 4–7. Opcode Summary

Accumulator Memory Reference Instructions		
Instruction	**Mnemonic**	**Opcode**
Absolute value of accumulator	ABS	1011 1110 0000 0000
Add ACCB to accumulator with carry	ADCB	1011 1110 0001 0001
Add to accumulator with shift	ADD	0010 SHFT IAAA AAAA
Add to low ACC short immediate	ADD	1011 1000 IIII IIII
Add to ACC long immediate with shift	ADD	1011 1111 1001 SHFT + 1 word
Add to accumulator with shift of 16	ADD	0110 0001 IAAA AAAA
Add to accumulator with carry	ADDC	0110 0000 IAAA AAAA
Add ACCB to accumulator	ADDB	1011 1110 0001 0000
Add to low accumulator with sign suppressed	ADDS	0110 0010 IAAA AAAA
Add to ACC with shift specified by TREG1	ADDT	0110 0011 IAAA AAAA
AND accumulator with data value	AND	0110 1110 IAAA AAAA
AND with ACC long immediate with shift	AND	1011 1111 1011 SHFT + 1 word
AND with ACC long immediate with shift of 16	AND	1011 1110 1000 0001 + 1 word
AND ACCB with accumulator	ANDB	1011 1110 0001 0010
Barrel shift accumulator right	BSAR	1011 1111 1110 SHFT
Complement accumulator	CMPL	1011 1110 0000 0001
Store ACC in ACCB if ACC > ACCB	CRGT	1011 1110 0001 1011
Store ACC in ACCB if ACC< ACCB	CRLT	1011 1110 0001 1100
Exchange ACCB with accumulator	EXAR	1011 1110 0001 1101
Load accumulator with ACCB	LACB	1011 1110 0001 1111
Load accumulator with shift	LACC	0001 SHFT IAAA AAAA
Load ACC long immediate with shift	LACC	1011 1111 1000 SHFT + 1 word
Load ACC with shift of 16	LACC	0110 1010 IAAA AAAA
Load low word of ACC with immediate	LACL	1011 1001 IIII IIII
Load low word of accumulator	LACL	0110 1001 IAAA AAAA
Load ACC with shift specified by TREG1	LACT	0110 1011 IAAA AAAA
Load ACCL with memory-mapped register	LAMM	0000 1000 IAAA AAAA
Negate accumulator	NEG	1011 1110 0000 0010
Normalize accumulator	NORM	1010 0000 IAAA AAAA
OR accumulator with data value	OR	0110 1101 IAAA AAAA
OR with ACC long immediate with shift	OR	1011 1111 1100 SHFT + 1 word
OR with ACC long immediate with shift of 16	OR	1011 1110 1000 0010 + 1 word
OR ACCB with accumulator	ORB	1011 1110 0001 0011
Rotate accumulator 1 bit left	ROL	1011 1110 0000 1100
Rotate ACCB and accumulator left	ROLB	1011 1110 0001 0100
Rotate accumulator 1 bit right	ROR	1011 1110 0000 1101
Rotate ACCB and accumulator right	RORB	1011 1110 0001 0101
Store accumulator in ACCB	SACB	1011 1110 0001 1110
Store high accumulator with shift	SACH	1001 1SHF IAAA AAAA
Store low accumulator with shift	SACL	1001 0SHF IAAA AAAA
Store ACCL to memory-mapped register	SAMM	1000 1000 IAAA AAAA
Shift ACC 16 specified by TREG1 [4]	SATH	1011 1110 0101 1010
Shift ACC 0–15 specified by TREG1 [3,0]	SATL	1011 1110 0101 1011
Subtract ACCB from accumulator	SBB	1011 1110 0001 1000
Subtract ACCB from accumulator with carry	SBBB	1011 1110 0001 1001
Shift accumulator 1 bit left	SFL	1011 1110 0000 1001
Shift ACCB and accumulator left	SFLB	1011 1110 0001 0110
Shift accumulator 1 bit right	SFR	1011 1110 0000 1010
Shift ACCB and accumulator right	SFRB	1011 1110 0001 0111

Table 4–7. Opcode Summary (Continued)

Accumulator Memory Reference Instructions (Concluded)		
Instruction	**Mnemonic**	**Opcode**
Subtract from accumulator with shift	SUB	0011 SHFT IAAA AAAA
Subtract from accumulator with shift of 16	SUB	0110 0101 IAAA AAAA
Subtract from ACC short immediate	SUB	1011 1010 IIII IIII
Subtract from ACC long immediate with shift	SUB	1011 1111 1010 SHFT + 1 word
Subtract from accumulator with borrow	SUBB	0110 0100 IAAA AAAA
Conditional subtract	SUBC	0000 1010 IAAA AAAA
Subtract from ACC with sign suppressed	SUBS	0110 0110 IAAA AAAA
Subtract from ACC, shift specified by TREG1	SUBT	0110 0111 IAAA AAAA
XOR accumulator with data value	XOR	0110 1100 IAAA AAAA
XOR with ACC long immediate with shift	XOR	1011 1111 1101 SHFT + 1 word
XOR with ACC long immediate with shift of 16	XOR	1011 1110 1000 0011 + 1 word
XOR ACCB with accumulator	XORB	1011 1110 0001 1010
Zero ACC, load high ACC with rounding	ZALR	0110 1000 IAAA AAAA
Zero accumulator and product register	ZAP	1011 1110 0101 1001
Auxiliary Registers and Data Page Pointer Instructions		
Instruction	**Mnemonic**	**Opcode**
Add to AR short immediate	ADRK	0111 1000 IIII IIII
Compare AR with CMPR	CMPR	1011 1111 0100 01CM
Load AR from addressed data	LAR	0000 0ARX IAAA AAAA
Load AR short immediate	LAR	1011 0ARX IIII IIII
Load AR long immediate	LAR	1011 1111 0000 1ARX + 1 word
Load data page pointer with addressed data	LDP	0000 1101 IAAA AAAA
Load data page immediate	LDP	1011 110I IIII IIII
Modify auxiliary register	MAR	1000 1011 IAAA AAAA
Store AR to addressed data	SAR	1000 0ARX IAAA AAAA
Subtract from AR short immediate	SBRK	0111 1100 IIII IIII
Parallel Logic Unit Instructions		
Instruction	**Mnemonic**	**Opcode**
AND DBMR with data value	APL	0101 1010 IAAA AAAA
AND long immediate with data value	APL	0101 1110 IAAA AAAA + 1 word
Compare DBMR to data value	CPL	0101 1011 IAAA AAAA
Compare data with long immediate	CPL	0101 1111 IAAA AAAA + 1 word
OR DBMR to data value	OPL	0101 1001 IAAA AAAA
OR long immediate with data value	OPL	0101 1101 IAAA AAAA + 1 word
Store long immediate to data	SPLK	1010 1110 IAAA AAAA + 1 word
XOR DBMR to data value	XPL	0101 1000 IAAA AAAA
XOR long immediate with data value	XPL	0101 1100 IAAA AAAA + 1 word

Table 4–7. Opcode Summary (Continued)

T Register, P Register, and Multiply Instructions		
Instruction	**Mnemonic**	**Opcode**
Add product to accumulator	APAC	1011 1110 0000 0100
Load high product register	LPH	0111 0101 IAAA AAAA
Load TREG0	LT	0111 0011 IAAA AAAA
Load TREG0 and accumulate previous product	LTA	0111 0000 IAAA AAAA
Load TREG0, accumulate previous product, and move data	LTD	0111 0010 IAAA AAAA
Load TREG0 and load ACC with PREG	LTP	0111 0001 IAAA AAAA
Load TREG0 and subtract previous product	LTS	0111 0100 IAAA AAAA
Multiply/accumulate	MAC	1010 0010 IAAA AAAA + 1 word
Multiply/accumulate with data shift	MACD	1010 0011 IAAA AAAA + 1 word
Mult/ACC with source ADRS in BMAR and DMOV	MADD	1010 1011 IAAA AAAA
Mult/ACC with source address in BMAR	MADS	1010 1010 IAAA AAAA
Multiply data value times TREG0	MPY	0101 0100 IAAA AAAA
Multiply TREG0 by 13-bit immediate	MPY	110I IIII IIII IIII
Multiply TREG0 by long immediate	MPY	1011 1110 1000 0000 + 1 word
Multiply TREG0 by data, add previous product	MPYA	0101 0000 IAAA AAAA
Multiply TREG0 by data, ACC – PREG	MPYS	0101 0001 IAAA AAAA
Multiply unsigned data value times TREG0	MPYU	0101 0101 IAAA AAAA
Load accumulator with product register	PAC	1011 1110 0000 0011
Subtract product from accumulator	SPAC	1011 1110 0000 0101
Store high product register	SPH	1000 1101 IAAA AAAA
Store low product register	SPL	1000 1100 IAAA AAAA
Set PREG shift count	SPM	1011 1111 0000 00PM
Data to TREG0, square it, add PREG to ACC	SQRA	0101 0010 IAAA AAAA
Data to TREG0, square it, ACC – PREG	SQRS	0101 0011 IAAA AAAA
Zero product register	ZPR	1011 1110 0101 1000

Table 4–7. Opcode Summary (Continued)

Branch Instructions		
Instruction	**Mnemonic**	**Opcode**
Branch unconditional with AR update	B	0111 1001 1AAA AAAA + 1 word
Branch unconditional with AR update delayed	BD	0111 1101 1AAA AAAA + 1 word
Branch addressed by ACC	BACC	1011 1110 0010 0000
Branch addressed by ACC delayed	BACCD	1011 1110 0010 0001
Branch AR = 0 with AR update	BANZ	0111 1011 1AAA AAAA + 1 word
Branch AR = 0 with AR update delayed	BANZD	0111 1111 1AAA AAAA + 1 word
Branch conditional	BCND	1110 00TP ZLVC ZLVC + 1 word
Branch conditional delayed	BCNDD	1111 00TP ZLVC ZLVC + 1 word
Call subroutine addressed by ACC	CALA	1011 1110 0011 0000
Call subroutine addressed by ACC delayed	CALAD	1011 1110 0011 1101
Call unconditional with AR update	CALL	0111 1010 1AAA AAAA + 1 word
Call unconditional with AR update delayed	CALLD	0111 1110 1AAA AAAA + 1 word
Call conditional	CC	1110 10TP ZLVC ZLVC + 1 word
Call conditional delayed	CCD	1111 10TP ZLVC ZLVC + 1 word
Software interrupt	INTR	1011 1110 011I NTR#
Nonmaskable interrupt	NMI	1011 1110 0101 0010
Return	RET	1110 1111 0000 0000
Return conditional	RETC	1110 11TP ZLVC ZLVC
Return conditionally, delayed	RETCD	1111 11TP ZLVC ZLVC
Return, delayed	RETD	1111 1111 0000 0000
Return from interrupt with enable	RETE	1011 1110 0011 1010
Return from interrupt	RETI	1011 1110 0011 1000
Trap	TRAP	1011 1110 0101 0001
Execute next one or two INST on condition	XC	111N 01TP ZLVC ZLVC
I/O and Data Memory Operations		
Instruction	**Mnemonic**	**Opcode**
Block move from data to data memory	BLDD	1010 1000 IAAA AAAA + 1 word
Block move data to data DEST long immediate	BLDD	1010 1001 IAAA AAAA + 1 word
Block move data to data with source in BMAR	BLDD	1010 1100 IAAA AAAA
Block move data to data with DEST in BMAR	BLDD	1010 1101 IAAA AAAA
Block move data to PROG with DEST in BMAR	BLDP	0101 0111 IAAA AAAA
Block move from program to data memory	BLPD	1010 0101 IAAA AAAA + 1 word
Block move Prog to data with source in BMAR	BLPD	1010 0100 IAAA AAAA
Data move in data memory	DMOV	0111 0111 IAAA AAAA
Input external access	IN	1010 1111 IAAA AAAA + 1 word
Load memory mapped register	LMMR	1000 1001 IAAA AAAA + 1 word
Out external access	OUT	0000 1100 IAAA AAAA + 1 word
Store memory mapped register	SMMR	0000 1001 IAAA AAAA + 1 word
Table read	TBLR	1010 0110 IAAA AAAA
Table write	TBLW	1010 0111 IAAA AAAA

Table 4–7. Opcode Summary (Concluded)

Control Instructions		
Instruction	**Mnemonic**	**Opcode**
Test bit specified immediate	BIT	0100 BITX IAAA AAAA
Test bit in data value as specified by TREG2	BITT	0110 1111 IAAA AAAA
Reset overflow mode	CLRC	1011 1110 0100 0010
Reset sign extension mode	CLRC	1011 1110 0100 0110
Reset hold mode	CLRC	1011 1110 0100 1000
Reset TC bit	CLRC	1011 1110 0100 1010
Reset carry	CLRC	1011 1110 0100 1110
Reset CNF bit	CLRC	1011 1110 0100 0100
Reset INTM bit	CLRC	1011 1110 0100 0000
Reset XF pin	CLRC	1011 1110 0100 1100
Idle	IDLE	1011 1110 0010 0010
Load status register 0	LST	0000 1110 IAAA AAAA
Load status register 1	LST	0000 1111 IAAA AAAA
No operation	NOP	1000 1011 0000 0000
Pop PC stack to low accumulator	POP	1011 1110 0011 0010
Pop stack to data memory	POPD	1000 1010 IAAA AAAA
Push data memory value onto PC stack	PSHD	0111 0110 IAAA AAAA
Push low accumulator to PC stack	PUSH	1011 1110 0011 1100
Repeat instruction as specified by data	RPT	0000 1011 IAAA AAAA
Repeat next INST specified by long immediate	RPT	1011 1110 1100 0100 + 1 word
Repeat INST specified by short immediate	RPT	1011 1011 IIII IIII
Block repeat	RPTB	1011 1110 1100 0110 + 1 word
Clear ACC/PREG and repeat next INST long immediate	RPTZ	1011 1110 1100 0101 + 1 word
Set overflow mode	SETC	1011 1110 0100 0011
Set sign extension mode	SETC	1011 1110 0100 0111
Set hold mode	SETC	1011 1110 0100 1001
Set TC bit	SETC	1011 1110 0100 1011
Set carry	SETC	1011 1110 0100 1111
Set XF pin high	SETC	1011 1110 0100 1101
Set CNF bit	SETC	1011 1110 0100 0101
Set INTM bit	SETC	1011 1110 0100 0001
Store status register 0	SST	1000 1110 IAAA AAAA
Store status register 1	SST	1000 1111 IAAA AAAA
Idle until interrupt — low power mode	IDLE2	1011 1110 0010 0011

Chapter 5

Peripherals

The seven peripheral interfaces connected to the 'C50, 'C51 and 'C53 core CPU are the serial port, TDM serial port, timer, software-programmable wait-state generators, I/O ports, divide-by-one clock, and XF and $\overline{BIO}$ pins. These peripherals are controlled through registers that reside in the memory map. The serial ports and timer are synchronized to the core CPU via interrupts. Peripherals and peripheral control are discussed in this chapter as shown below.

Topic **Page**

5.1 Peripheral Control

Peripheral circuits are operated and controlled through access of memory-mapped control and data registers. The operation of the serial ports and timer is synchronized to the processor via interrupts or through interrupt polling. Setting and clearing bits can enable, disable, initialize, and dynamically reconfigure the peripherals. Data is transferred to and from the peripherals through memory-mapped data registers. When a peripheral is not in use, the internal clocks are shut off from that peripheral, allowing for lower power consumption when the device is in normal run mode or idle mode.

5.1.1 Memory-Mapped Registers and I/O Ports

Twenty-eight core processor registers are mapped into the data memory space, they are listed in subsection 3.4.1. In addition to these core registers, 15 peripheral registers and 16 I/O ports are mapped into the data memory space. Table 5–1 lists the memory-mapped registers and I/O ports of the 'C5x. Note that all writes to memory-mapped peripheral registers require one additional machine cycle.

Table 5–1. Memory-Mapped Registers and I/O Ports

Memory-Mapped Core Processor Registers			
Name	**Address**		**Description**
	Dec	**Hex**	
—	0–3	0–3	Reserved
IMR	4	4	Interrupt Mask Register
GREG	5	5	Global Memory Allocation Register
IFR	6	6	Interrupt Flag Register
PMST	7	7	Processor Mode Status Register
RPTC	8	8	Repeat Counter Register
BRCR	9	9	Block Repeat Counter Register
PASR	10	A	Block Repeat Program Address Start Register
PAER	11	B	Block Repeat Program Address End Register
TREG0	12	C	Temporary Register Used for Multiplicand
TREG1	13	D	Temporary Register Used for Dynamic Shift Count (5 bits only)
TREG2	14	E	Temporary Register Used as Bit Pointer in Dynamic Bit Test (4 bits only)
DBMR	15	F	Dynamic Bit Manipulation Register
AR0	16	10	Auxiliary Register Zero
AR1	17	11	Auxiliary Register One
AR2	18	12	Auxiliary Register Two

Table 5–1. Memory-Mapped Registers and I/O Ports (Continued)

Memory-Mapped Core Processor Registers (Concluded)			
Name	**Address**		**Description**
	Dec	**Hex**	
AR3	19	13	Auxiliary Register Three
AR4	20	14	Auxiliary Register Four
AR5	21	15	Auxiliary Register Five
AR6	22	16	Auxiliary Register Six
AR7	23	17	Auxiliary Register Seven
INDX	24	18	Index Register
ARCR	25	19	Auxiliary Register Compare Register
CBSR1	26	1A	Circular Buffer 1 Start Register
CBER1	27	1B	Circular Buffer 1 End Register
CBSR2	28	1C	Circular Buffer 2 Start Register
CBER2	29	1D	Circular Buffer 2 End Register
CBCR	30	1E	Circular Buffer Control Register
BMAR	31	1F	Block Move Address Register
Memory-Mapped Peripheral Registers			
DRR	32	20	Data Receive Register
DXR	33	21	Data Transmit Register
SPC	34	22	Serial Port Control Register
—	35	23	Reserved
TIM	36	24	Timer Register
PRD	37	25	Period Register
TCR	38	26	Timer Control Register
—	39	27	Reserved
PDWSR	40	28	Program/Data S/W Wait-State Register
IOWSR	41	29	I/O S/W Wait-State Register
CWSR	42	2A	S/W Wait-State Control Register
—	43–47	2B–2F	Reserved
TRCV	48	30	TDM Data Receive Register
TDXR	49	31	TDM Transmit Data Register
TSPC	50	32	TDM Serial Port Control Register
TCSR	51	33	TDM Channel Select Register
TRTA	52	34	TDM Receive/Transmit Address Register
TRAD	53	35	TDM Received Address Register

Table 5–1. Memory-Mapped Registers and I/O Ports (Concluded)

Name	Address		Description
	Dec	Hex	
—	54–79	36–4F	Reserved
Memory-Mapped I/O Ports†			
PA0	80	50	I/O Port 50h
PA1	81	51	I/O Port 51h
PA2	82	52	I/O Port 52h
PA3	83	53	I/O Port 53h
PA4	84	54	I/O Port 54h
PA5	85	55	I/O Port 55h
PA6	86	56	I/O Port 56h
PA7	87	57	I/O Port 57h
PA8	88	58	I/O Port 58h
PA9	89	59	I/O Port 59h
PA10	90	5A	I/O Port 5Ah
PA11	91	5B	I/O Port 5Bh
PA12	92	5C	I/O Port 5Ch
PA13	93	5D	I/O Port 5Dh
PA14	94	5E	I/O Port 5Eh
PA15	95	5F	I/O Port 5Fh

† See Section 6.2 for memory-mapped I/O ports.

5.1.2 Interrupts

The 'C5x devices have four external, maskable user interrupts ($\overline{INT4}$–$\overline{INT1}$) that external devices can use to interrupt the processor; there is one external nonmaskable interrupt ($\overline{NMI}$). Internal interrupts are generated by the serial port (RINT and XINT), the timer (TINT), the TDM port (TRNT and TXNT), and the software interrupt instructions (TRAP, NMI, and INTR). Interrupt priorities are set so that reset ($\overline{RS}$) has the highest priority and $\overline{INT4}$ has the lowest priority. The $\overline{NMI}$ has the second highest priority.

This subsection explains interrupt organization and management. Vector-relative locations and priorities for all internal and external interrupts are shown in Table 5–2. No priority is set for the TRAP instruction (used for software interrupts), but it is included here because it has its own vector location. Each interrupt address has been spaced apart by two locations so that branch instructions can be accommodated in those locations.

The interrupt vectors reside at locations determined by the five-bit IPTR field of the PMST and the address values shown in Table 5–2. The IPTR field is set

to zero upon device reset, resulting in the interrupt vectors mapping to 0000h in the program memory space. The vectors' program address can be remapped to the beginning of any of the 32 2K-word blocks composing program memory space. This is done by loading a five-bit block address (5 MSBs of a full 16-bit address) into the IPTR. For example, the vectors can be moved to the beginning of the on-chip program RAM of the 'C50 by loading IPTR with 1. When an interrupt trap occurs, the value in the IPTR is loaded into the most significant five bits of the vector address, and the relative address of the interrupt causing the trap constitutes the 6 LSBs of the vector address. This relative addressing scheme applies to all interrupts as well as to the software trap. It does not apply to the reset vector, because the reset signal forces the IPTR to be set to zero.

Table 5–2. Interrupt Locations and Priorities

Name	Location		Priority	Function
	Dec	Hex		
$\overline{RS}$	0	0	1 (highest)	External reset signal
$\overline{NMI}$	36	24	2	Nonmaskable interrupt
$\overline{INT1}$	2	2	3	External user interrupt #1
$\overline{INT2}$	4	4	4	External user interrupt #2
$\overline{INT3}$	6	6	5	External user interrupt #3
TINT	8	8	6	Internal timer interrupt
RINT	10	A	7	Serial port receive interrupt
XINT	12	C	8	Serial port transmit interrupt
TRNT	14	E	9	TDM port receive interrupt
TXNT	16	10	10	TDM port transmit interrupt
$\overline{INT4}$	18	12	11	External user interrupt #4
——	20–33	14–21	N/A	Reserved
TRAP	34	22	N/A	Trap instruction vector
——	38–39	26–27	N/A	Reserved
——	40–63	28–3F	N/A	Software interrupts

When an interrupt occurs, it is stored in the 16-bit interrupt flag register (IFR). Note that this happens regardless of whether that interrupt is currently enabled or disabled. Each interrupt sets a flag in IFR. The flag can be cleared in any of the following three ways:

1) Device reset ($\overline{RS}$ active low),
2) The program takes the interrupt trap, or
3) The program writes a one to the appropriate bit in the IFR.

The IFR is located at address 6 in the data memory space and can be read to identify active interrupts and written to clear interrupts. The IFR register is laid out as follows:

15 9	8	7	6	5	4	3	2	1	0
Reserved	$\overline{INT4}$	TXNT	TRNT	XINT	RINT	TINT	$\overline{INT3}$	$\overline{INT2}$	$\overline{INT1}$

Note that the 'C5x uses only ten of the sixteen generic interrupt lines to the core CPU shown in Section 3.8.

A one in a specific bit, when read, indicates an active interrupt. For example, if the IFR is read to be 0005h, then $\overline{INT3}$ and $\overline{INT1}$ are active. A one can be written to a specific bit to clear the corresponding interrupt. In the example, if a one is written to bit zero (0001h to IFR), then the $\overline{INT1}$ interrupt would be cleared. In the above example, the value 0005h could be written back into the IFR to clear both pending interrupts.

A corresponding interrupt flag is automatically cleared when the interrupt trap is taken. When the CPU accepts the interrupt and fetches the instruction at the interrupt vector location, it generates an interrupt acknowledge ($\overline{IACK}$) signal that clears the appropriate interrupt flag bit. A hardware reset ($\overline{RS}$ active low) clears all pending interrupt flags.

The 'C5x devices have a memory-mapped interrupt mask register (IMR) for masking external and internal interrupts. The layout of the register is as follows:

15 9	8	7	6	5	4	3	2	1	0
Reserved	$\overline{INT4}$	TXNT	TRNT	XINT	RINT	TINT	$\overline{INT3}$	$\overline{INT2}$	$\overline{INT1}$

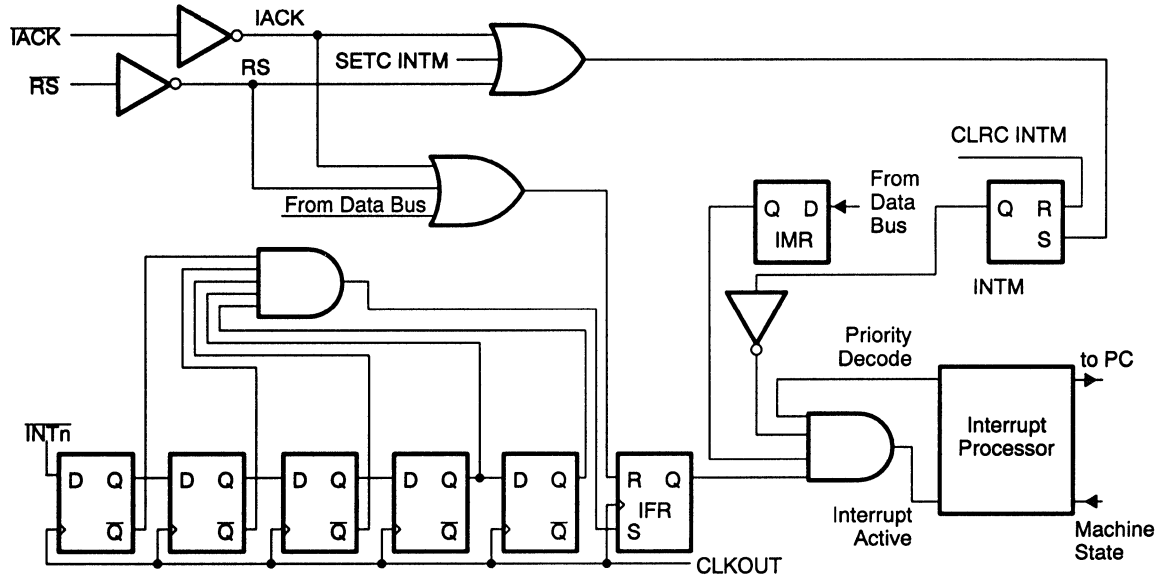
A 1 in bit positions 8 through 0 of the IMR enables the corresponding interrupt, provided that INTM = 0. The IMR is accessible with both read and write operations. Note that $\overline{RS}$ and $\overline{NMI}$ are not included in the IMR; the IMR has no effect on reset or a nonmaskable interrupt.

Interrupts may be asynchronously triggered. In the functional logic organization for $\overline{INT4}$–$\overline{INT1}$, shown in Figure 5–1, the external interrupt $\overline{INTn}$ is synchronized to the core via a five flip-flop synchronizer. The actual implementation of the interrupt circuits is similar to this logic implementation. A one is loaded into the IFR if a 1-1-0-0-0 sequence on five consecutive CLKOUT1 cycles is detected.

The 'C5x devices sample the external interrupt pins multiple times to avoid noise-generated interrupts. To detect an active interrupt, these devices must sample the signal low on at least three consecutive machine cycles. Once an

interrupt is detected, the devices must sample the signal high on at least two consecutive machine cycles to be able to detect another interrupt. The external interrupt pins are sampled on the rising edge of CLKOUT1. If the external interrupts are running asynchronously, the pulses should be stretched to guarantee three consecutive low samples.

Figure 5–1. External Interrupt Logic Diagram



If the INTM bit and mask registers have been properly enabled, the interrupt signal is accepted by the processor. An $\overline{\text{IACK}}$ signal is then generated. The $\overline{\text{IACK}}$ clears the appropriate interrupt edge flip-flop and sets the INTM =1. The logic is the same for $\overline{\text{INT1}}$–$\overline{\text{INT4}}$. $\overline{\text{NMI}}$ uses the same logic, except that it is not affected by IMR or INTM status.

The context of the interrupted code segment is automatically saved by the processor. When the processor takes the interrupt trap ($\overline{\text{IACK}}$ goes active low), the accumulator, accumulator buffer, product register, index register, auxiliary compare register, ST0, ST1 (except for the XF bit), PMST, and all three temporary registers are pushed onto corresponding one-deep stacks. At the completion of the ISR, the RETI (return from interrupt) or RETE instruction causes the stacks to be popped automatically to restore the interrupted code segment's context. Because these stacks are one deep, nesting of interrupts requires a software context save. However, the overhead is lowered considerably by the automatic context save. Therefore, it is usually more code efficient to serially execute multiple ISRs.

Interrupt service routines can be invoked in software via the INTR instruction (see page 4–76 for details).

5.1.3 Peripheral Reset

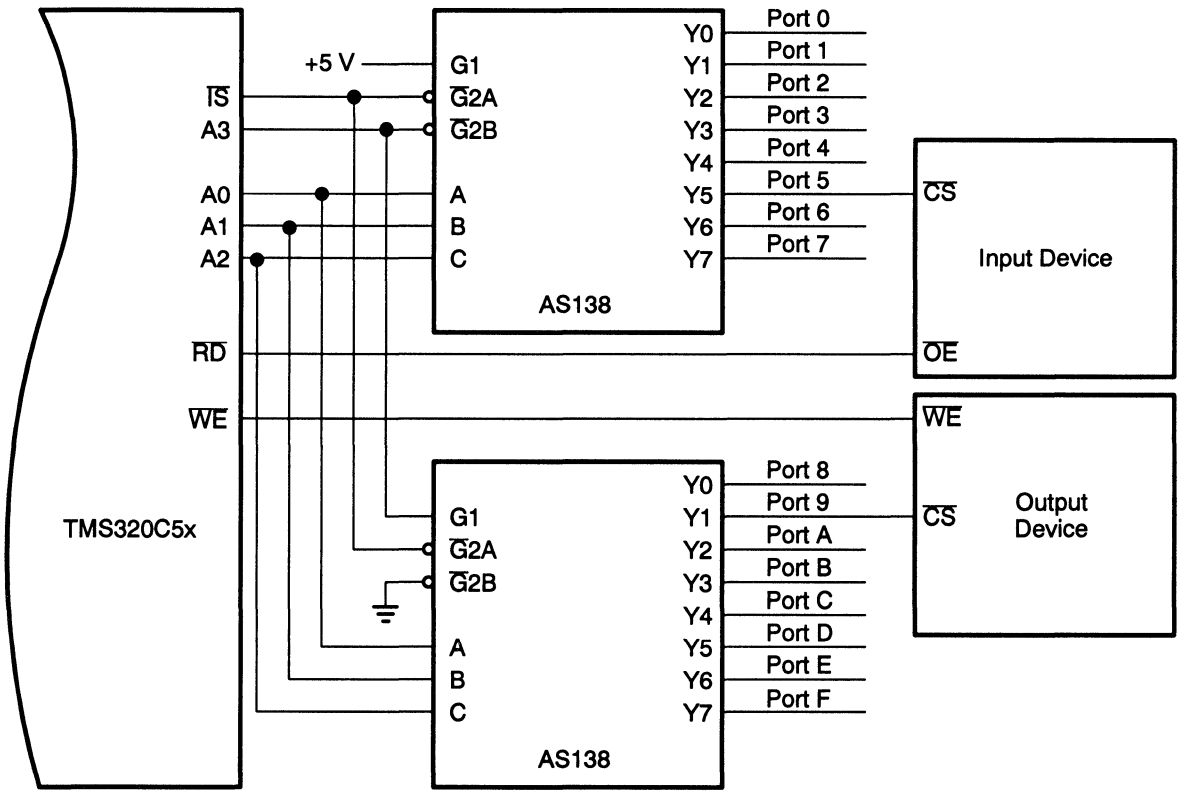
A number of actions occur when the 'C5x is reset. Subsection 3.8.1 describes what happens in the 'C5x core when reset is activated. On a device reset, the core CPU sends an $\overline{\text{SRESET}}$ signal to the peripheral circuits. The $\overline{\text{SRESET}}$ signal has the following consequences in the peripheral circuits:

1) The two software wait-state registers are set to 0FFFFh, causing all external accesses to occur with 7 wait states. The CWSR is loaded with 0Fh.
2) The FO bits of the SPC and TSPC registers are set to zero, selecting a word length of 16 bits for each serial port.
3) The FSM bits of the SPC and TSPC registers are set to zero. FSM must be set to one for operation with frame sync pulses.
4) The TXM bits of the SPC and TSPC are set to zero, configuring the FSX and TFSX pins as inputs.
5) The SPC and TSPC registers are loaded with 0y00h, where the 2 MSBs of y are 10 (binary) and the 2 LSBs of y reflect the current levels on the transmit and receive clock pins of the respective port.
6) The TIM and PRD registers are loaded with 0FFFFh. The TDDR field of the TCR is set to zero. The timer is started.

5.2 Parallel Input/Output Ports

The 'C5x devices have 64K parallel input/output ports. I/O port accesses are defined as accesses during which the I/O space select signal ($\overline{IS}$) is active. Sixteen of the 64K ports are mapped in data memory space as shown in Table 5–1. All 64K I/O ports can be accessed with the IN and OUT instructions. The 16 memory-mapped I/O ports (50h–5Fh) can also be accessed via any instruction that reads or writes a location in data space. $\overline{RD}$ can be used in conjunction with chip-select logic to generate an output enable signal for an external peripheral. The $\overline{WE}$ signal can be used in conjunction with chip-select logic to generate a write enable signal for an external peripheral. Figure 5–2 shows typical I/O port interface circuitry. Note that the decode section can be simplified if fewer I/O ports are used.

Figure 5–2. I/O Port Interface Circuitry



5.3 Software-Programmable Wait-State Generators

Software-programmable wait-state generators can be used to extend external bus cycles by up to 7 machine cycles. This provides a convenient means for interfacing external devices that do not satisfy the full-speed access-time requirements of the 'C5x. Devices requiring more than 7 wait states can be interfaced with the hardware READY line. When all external accesses are configured for zero wait states, the internal clocks to the wait-state generator are shut off, allowing the device to run in a lower power mode of operation.

The software-programmable wait-state generators are controlled by two 16-bit wait-state registers (PDWSR and IOWSR) and a 5-bit control register (CWSR). Each of the three external spaces (program, data, and I/O spaces) has an assigned field in a software wait-state register. Wait states for the program and data spaces are specified in the lower and upper halves of PDWSR, respectively. Wait states for I/O space are specified in IOWSR. The bits of CWSR control the mapping between wait-state register contents and the number of wait states.

The program and data spaces each consist of 64K addresses. Each 64K space can be viewed as being composed of four 16K-word blocks. Each 16K address segment in program and data space is associated with 2 bits in PDWSR, as shown in Table 5–3. The value of a 2-bit field in PDWSR specifies the number of wait states to be inserted for each access in the given space and address range.

Table 5–3. Software Wait-State Registers

Register	Bits	Space	Address Range	
PDWSR	0–1	Program	0000h–3FFFh	
	2–3		4000h–7FFFh	
	4–5		8000h–0BFFFh	
	6–7		0C000h–0FFFFh	
	8–9	Data	0000h–3FFFh	
	10–11		4000h–7FFFh	
	12–13		8000h–0BFFFh	
	14–15		0C000h–0FFFFh	
IOWSR			BIG = 0	BIG = 1
	0–1	I/O	Port 0/1, Port 10/11, etc.	0000h–1FFFh
	2–3		Port 2/3, Port 12/13, etc.	2000h–3FFFh
	4–5		Port 4/5, Port 14/15, etc.	4000h–5FFFh
	6–7		Port 6/7, Port 16/17, etc.	6000h–7FFFh
	8–9		Port 8/9, Port 18/19, etc.	8000h–9FFFh
	10–11		Port 0A/0B, Port 1A/1B, etc.	0A000h–0BFFFh
	12–13		Port 0C/0D, Port 1C/1D, etc.	0C000h–0DFFFh
	14–15		Port 0E/0F, Port 1E/1F, etc.	0E000h–0FFFFh

The I/O space wait-state register (IOWSR) can be mapped in either of two ways, as specified by the BIG bit in the CWSR register. If BIG=0, each of 8 pairs of memory-mapped I/O ports has its own 2-bit field in IOWSR. Note that even when BIG=0, the entire I/O space is configured with wait states on two-word boundaries (i.e., port 0/1, port 10/11, and port 20/21 all have the same number of wait states). This configuration provides maximum flexibility when I/O bus-cycles access peripherals such as D/A and A/D devices. However, if I/O accesses read and/or write devices that are addressable (e.g., external RAM), BIG can be set to 1. In this case, the 64K I/O space is divided into eight 8K-word address blocks, with each block having an independently programmable number of wait states.

Note that the wait-state generators affect external accesses only; internal accesses always have zero wait states.

The four bits in CWSR allow the user to select one of two mappings between 2-bit wait-state fields and the number of wait states for the corresponding space. As shown in Table 5–4, if a particular bit of CWSR is a zero, the mapping between wait-state field values and the resulting number of wait states is direct: the number of wait states for external accesses in the space associated with that control bit is equal to the wait-state field value. If the control bit

of CWSR is a one, the number of wait states is determined by the mapping shown in Table 5–4. Table 5–5 shows the layout of the CWSR register in PDWSR and IOWSR registers. You should always program the CWSR register prior to configuring the PDWSR and IOWSR registers to avoid configuring memory with too few wait states during the set-up of wait-state registers.

Table 5–4. Wait-State Field Values and Wait States as a Function of CWSR Bit n

Wait-State Field† of PDWSR or IOWSR (Binary Value)	No. of Wait States (CWSR Bit n = 0)	No. of Wait States (CWSR Bit n = 1)
00	0	0
01	1	1
10	2	3
11	3	7

† This bit field corresponds to the bit field defined in the second column of Table 5–3.

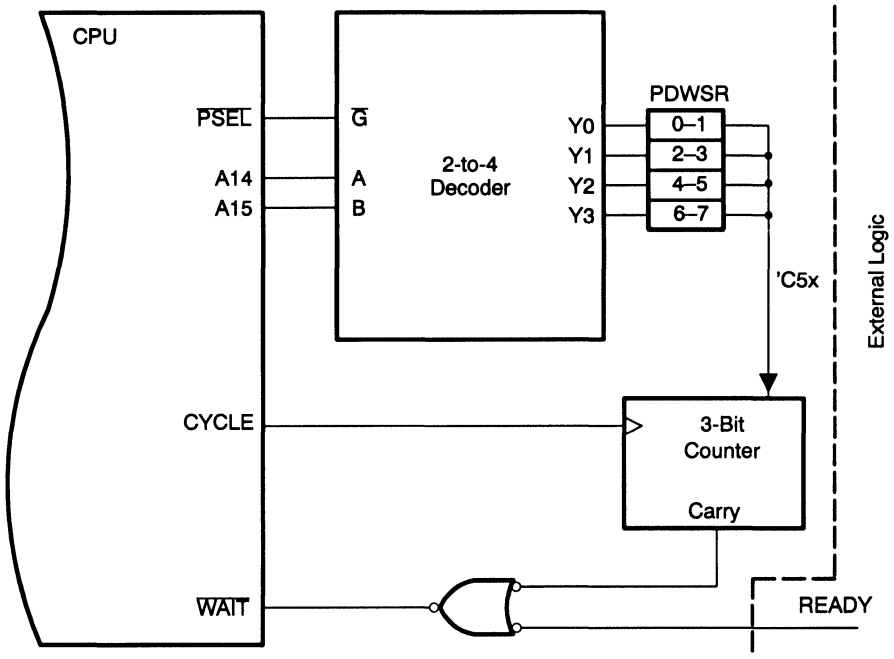
Table 5–5. Space Controlled by CWSR Bit n

n (Bit Position In CWSR)	Space
0	Program
1	Data
2	I/O (lower-half: Port 0–Port 7 if BIG=0, 0000h–7FFFh if BIG=1)
3	I/O (upper-half: Port 8–Port F if BIG=0, 8000h–0FFFFh if BIG=1)
4	BIG mode bit

Figure 5–3 shows a block diagram of the wait-state generator logic for external program space. When an external program access is decoded, the appropriate field of the PDWSR wait-state register is loaded into the counter. If the field is not 000, a not-ready signal is sent to the CPU. The not-ready condition is maintained until the counter decrements to zero and the external READY line is high. The external READY and the wait-state register READY are ORed together to generate the CPU $\overline{\text{WAIT}}$ signal. Also, the READY line is sampled at the falling edge of CLKOUT. (Note that the external READY line is machine-sampled only at the last cycle of an external access if the on-chip wait-state generator is used to insert software wait states).

Upon reset, all the software wait-state control register fields are set to 7. CWSR is set to 0Fh. Device reset also sets the BIG bit of the CWSR register to zero.

Figure 5–3. Software Wait-State Generator Block Diagram

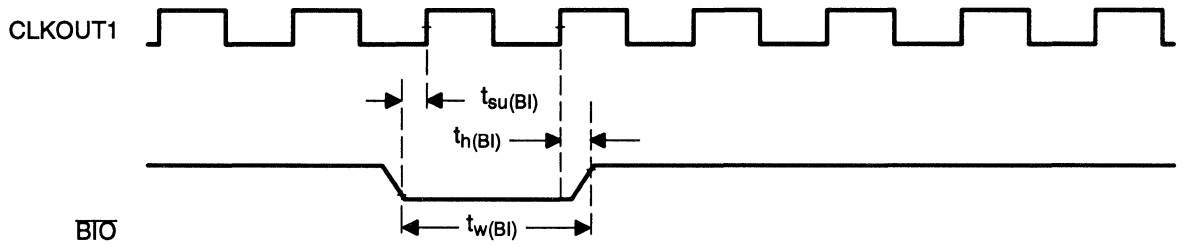


5.4 General-Purpose I/O Pins

The 'C5x devices have two general-purpose pins that are software controlled. The $\overline{BIO}$ pin is a branch control input pin, and the XF pin is an external flag output pin. For detailed timing specifications of $\overline{BIO}$ and XF signals, refer to Appendix A.

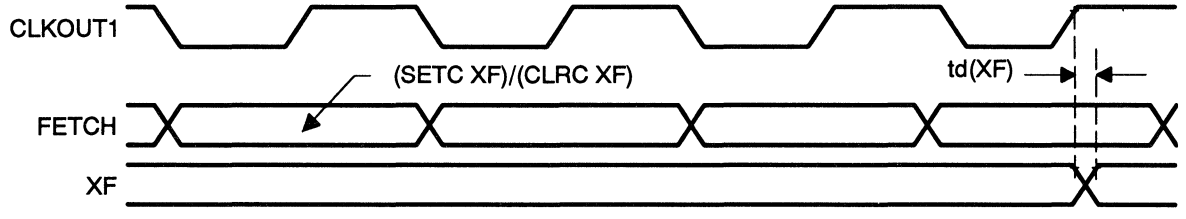
The $\overline{BIO}$ pin monitos peripheral device status—especially as an alternative to an interrupt when time-critical loops must not be disturbed. A branch can be conditionally executed when the $\overline{BIO}$ input is active (low). The timing diagram, shown in Figure 5–4, is an example of the $\overline{BIO}$ operation. This timing diagram is for a sequence of single-cycle, signal-word instructions located in external memory. The $\overline{BIO}$ condition is sampled during the decode phase of the pipeline for the XC instruction. All other instructions sample the $\overline{BIO}$ pin during the execute phase of the pipeline.

Figure 5–4. $\overline{BIO}$ Timing Diagram



The XF (external flag) pin signals to external devices via software. It is set high by the SETC XF (set external flag) instruction and reset to a low level by the CLRC XF (reset external flag) instruction. XF is set high upon device reset. The relationship between the time SETC/CLRC instruction is fetched, and the time the XF pin is set or reset as shown in Figure 5–5. As with $\overline{BIO}$, the timing shown for XF is for a sequence of single-cycle, single-word instructions located in external memory. Actual timing may vary with different instruction sequences.

Figure 5–5. External Flag Timing Diagram



5.5 Serial Port

A full duplex (bidirectional) on-chip serial port provides direct communication with serial devices such as codecs, serial A/D (analog to digital) converters, and other serial systems. The interface signals are compatible with codecs and many other serial devices. The serial port may also be used for intercommunication between processors in multiprocessing applications (the TDM port is further optimized for such an application).

Both receive and transmit operations are double-buffered on the 'C5x, thus allowing a continuous communications stream (either 8- or 16-bit data packets. The continuous mode provides operation that once initiated requires no further frame synchronization pulses when transmitting at maximum packet frequency. The serial port is fully static and thus will function at arbitrarily low clocking frequencies. The maximum operating frequency of the serial port while using internal clocks is CLKOUT1/4 (5 Mbit/s at 50 ns, 7.14 Mbit/s at 35 ns). When the serial ports are in reset the device may be configured to shut off the serial port internal clocks, allowing the device to run in a lower power mode of operation.

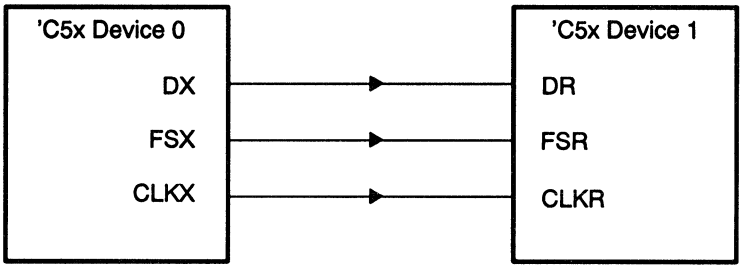
5.5.1 Serial Port Operation

Table 5–6 lists the pins used in serial port operation. Three signals are necessary to connect the transmit pins of the transmitting device with the receive pins of the receiving device for data transmission. The transmitted serial data signal (DX) sends the actual data. The transmit frame synchronization signal (FSX) initiates the transfer (at the beginning of the packet), and the transmit clock signal (CLKX) clocks the bit transfer. The corresponding pins on the receive device are DR, FSR and CLKR, respectively. Figure 5–6 shows these pins for two 'C5x serial ports connected for a one-way transfer from device 0 to device 1.

Table 5–6. Serial Port Pins

Pins	Description
CLKX	Transmit clock signal
CLKR	Receive clock signal
DX	Transmitted serial data signal
DR	Received serial data signal
FSX	Transmit frame synchronization signal
FSR	Receive framing synchronization signal

Figure 5–6. One-Way Serial Port Transfer



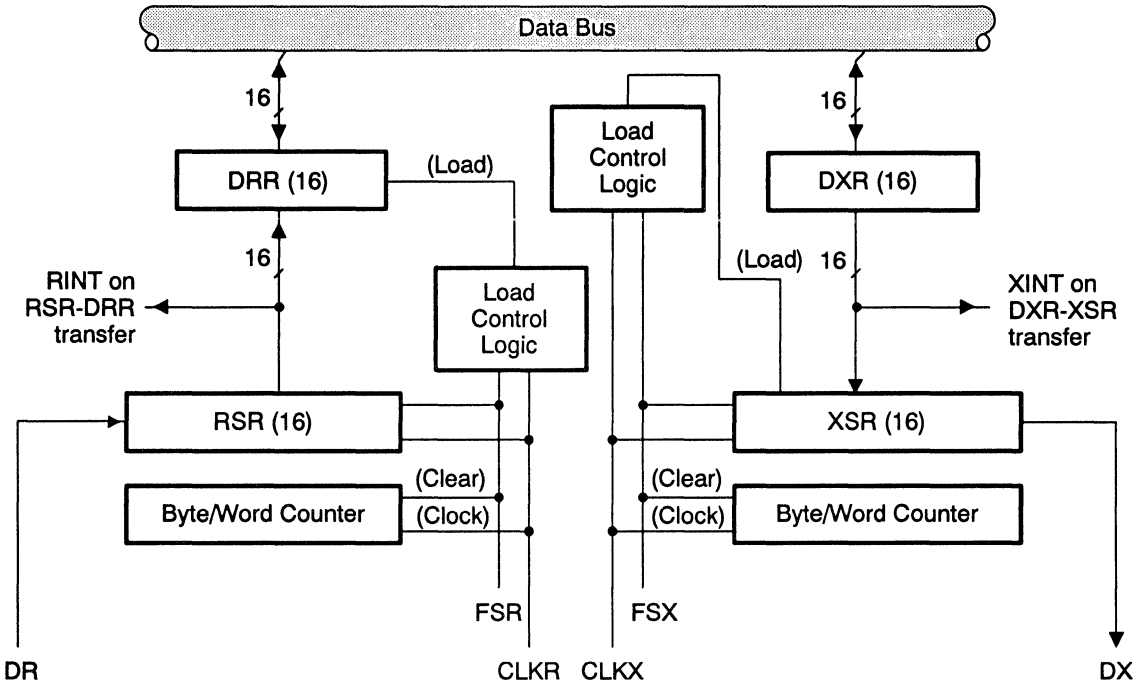
The serial port operates through the three memory-mapped registers (SPC, DXR, and DRR) and two other registers (XSR and RSR) that are not accessible but permit double-buffering capability. These five registers are listed in Table 5–7.

Table 5–7. Serial Port Registers

Registers	Description
SPC	Serial port control register
DXR	Data transmit register
DRR	Data receive register
XSR	Transmit shift register
RSR	Receive shift register

Figure 5–7 shows how the pins and registers are configured on the serial port and how the double-buffering is implemented.

Figure 5–7. Serial Port Block Diagram



The SPC controls serial port operation; the functions of SPC bit fields are described in Table 5–8. Transmit data is written to the DXR, while received data is read from the DRR. A transmit is executed by writing data to the DXR, which copies the data to the XSR when the XSR is empty (the last word has been serially transmitted, that is, driven on the DX pin). The XSR manages the shifting of the data to the DX pin, thus allowing another write to DXR as soon as the DXR-to-XSR copy is completed.

Upon completion of the DXR-to-XSR copy, a 0-to-1 transition occurs on the transmit ready XRDY bit in the SPC and generates a serial port transmit interrupt (XINT — see subsection 5.1.2 for more information on 'C5x interrupts) that signals that DXR is ready for a new word. The process is similar on the receive side. Data from the DR pin is shifted into the RSR, which copies it to the data receive register (DRR) from which it may be read. Upon completion of the RSR-to-DRR copy, a 0-to-1 transition occurs on the receive ready (RRDY) bit in the SPC and generates a serial port receive interrupt (RINT). Thus, the serial port is double-buffered because data can be transferred to or from DXR or DRR while another transmit or receive is being performed. Note that the transfer timing is synchronized by the frame sync pulse in burst mode and is discussed in more detail in subsection 5.5.2.

Figure 5–8 shows the 16-bit memory-mapped register that configures the serial port. Some of the bits are read-only while others are read/write.

Figure 5–8. Serial Port Control Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FREE	SOFT	RSRFULL	XSREMPTY	XRDY	RRDY	IN1	IN0	RRST	XRST	TXM	MCM	FSM	FO	DLB	RES
R/W	R/W	R	R	R	R	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R

Note: R = Read, W = Write

Table 5–8. Serial Port Control Register Bits Summary

Bit	Name	Function
0	Reserved	Always read as zero.
1	DLB	The Digital Loopback Mode Bit can be used to put the serial port in digital loopback mode. When DLB=1, DR and FSR are connected to DX and FSX, respectively, through multiplexers, as shown in Figure 5–9(a) and Figure 5–9(b). Additionally, CLKR is driven by CLKX if MCM=1. If DLB=1 and MCM=0, CLKR is taken from the CLKR pin of the device. This configuration allows CLKX and CLKR to be tied together externally and supplied by a common external clock source. The logic diagram for CLKR is shown in Figure 5–9(c). If DLB=0, DR, FSR, and CLKR are taken from the respective device pins. Note that TXM must be set to one for proper operation in DLB mode. Note also that the FSX and DX signals appear on the device pins when DLB=1, but FSR and DR do not.
2	FO	The Format Bit specifies the word length of the serial port transmitter and receiver. If FO=0, data is transmitted and/or received as 16-bit words. If FO=1, data is transferred as 8-bit bytes. The data is transferred with the MSB first.
3	FSM	The Frame Synch Mode Bit specifies whether frame synchronization pulses are required for serial port operation. If FSM=1, a frame sync pulse is required on FSX/FSR for the transmission/reception of each word. When the serial port is operated in the continuous mode, FSM=0. Refer to subsection 5.5.1 for more details on the frame sync signals.
4	MCM	The Clock Mode Bit specifies the clock source for CLKX. If MCM=0, CLKX is taken from the CLKX pin. If MCM=1, CLKX is driven by an on-chip clock source having a frequency equal to one-fourth of CLKOUT1. Note that if MCM=1 and DLB=1, a CLKR signal is also supplied by the internal source.
5	TXM	The Transmit Mode Bit configures the FSX pin as an input (TXM = 0) or as an output (TXM = 1). When TXM = 1, frame sync pulses are generated internally when data is transferred from the DXR to DSR to initiate data transfers. The internally generated framing signal is synchronous with respect to CLKX. When TXM = 0, the transmitter idles until a frame synch pulse is supplied on the FSX pin.
6 7	XRST RRST	The Transmit Reset and Receive Reset signals reset the transmitter and receiver, respectively. If the SPC is to be modified to reconfigure the serial port, a total of two writes should be made to the SPC. The first write should write zeroes to XRST and RRST and the desired configuration to bits 1–5. The second write should write ones to XRST and RRST, taking the serial port out of reset. When a zero is written to either of these bits, activity in the corresponding section of the serial port halts. Note that when XRDY=0, writing a zero to XRST generates a transmit interrupt. When XRST=0, RRST=0, and MCM=0, the internal clocks to the serial ports are shut off, allowing the device to run in a lower power mode of operation.

Table 5–8. Serial Port Control Register Bits Summary (Continued)

Bit	Name	Function
8 9	IN0 IN1	The Input 0 Bit and Input 1 Bit allow the CLKR and CLKX pins to be used as bit inputs. IN0 and IN1 reflect the current levels of the CLKR and CLKX pins, respectively, of the device. The levels on these pins can be read by reading the SPC. They can be tested by using the PLU or the BIT or BITT instruction. Note that there is a latency of between 0.5 and 1.5 CLKOUT1 cycles in length from CLKR/CLKX switching to the new CLKR/CLKX value being represented in the SPC.
10 11	RRDY XRDY	Receive Ready and Transmit Ready Bits. A transition from 0 to 1 of the RRDY bit indicates that the receive shift register (RSR) has been copied to the DRR and that the data can be read. A receive interrupt is generated upon the transition. A transition from 0 to 1 of the XRDY bit indicates that the DXR contents have been copied to the XSR and that data is ready to be loaded with a new data word. A transmit interrupt is generated upon the transition. These bits can be polled in software in lieu of using serial port interrupts.
12	$\overline{\text{XSREMPTY}}$	The Transmit Shift Register Empty Flag. This bit indicates whether the transmitter has experienced underflow. Underflow occurs when two conditions are satisfied: 1) the XSR empties, and 2) the DXR has not been reloaded since the last DXR-to-XSR transfer. Note that underflow does not constitute an error condition in burst mode. If another frame synch pulse occurs prior to writing the DXR while in burst mode, the previous data in the XSR is shifted out the DX pin. Writing to DXR inactivates the XSREMPTY bit. $\overline{\text{XSREMPTY}}$=0 indicates underflow.
13	RSRFULL	The Receive Shift Register Full Flag. This bit indicates whether the receiver has experienced overrun. Overrun occurs when three conditions are satisfied: 1) RSR is full, 2) the DRR has not been read since the last RSR-to-DRR transfer, and 3) a frame sync pulse appears on FSR. Note that condition 3 applies only when FSM=1. When FSM=0, only the first two conditions apply. When RSRFULL=1, the receiver halts and waits for the DRR to be read. The data in the RSR is preserved, but any data sent on DR while the receiver is halted is lost. Reading DRR, device reset, and serial port reset each clear the RSRFULL bit. RSRFULL=1 indicates overflow.
14	SOFT	The SOFT bit. This bit is enabled when the FREE bit is 0. If FREE=0, the SOFT bit selects immediate stop if 0, stop after word completion if 1. See page 5-23.
15	FREE	The FREE bit. If FREE=1, free run is selected, regardless of the value of the SOFT bit. If FREE=0, the SOFT bit selects the emulation mode as described above. See page 5-23.

Bit 0 is reserved and is read as 0 (although it performs a function in the TDM serial port, explained in Section 5.6). The format bit FO, bit 1 of the SPC, specifies whether data is transmitted as 16-bit words (FO=0) or 8-bit bytes (F0=1). Note that in the latter case, only the lower byte of whatever is written to DXR on the transmitter is transmitted and the lower byte of whatever is read from DRR on the receiver is received. To transmit a whole 16-bit word in 8-bit byte mode on the transmitter, two writes to DXR are necessary, with the appropriate shifts of the value because the upper 8 bits written to DXR are ignored. Similarly, to receive a whole 16-bit word in 8-bit mode on the receiver, two reads from DRR are necessary, with the appropriate shifts of the value, because the upper 8 bits in DRR are random values.

The source device for the clock for serial port transfers is set by bit 4 (MCM) of the SPC register. If MCM=1, then the CLKX is configured as an output and is driven by an internal clock source with a frequency equal to 1/4 of CLKOUT1. If MCM=0, CLKX is configured as an input and thus accepts an external clock. Note that the CLKR pin is always configured as an input.

The source device for the frame synchronization pulse is set with the TXM bit, (bit 3). Like MCM, if TXM=1, the FSX pin is configured as an output and drives a pulse at the beginning of every transmit. If TXM=0, FSX is configured as an input and accepts an external frame sync signal. Note that the FSR pin is always configured as an input.

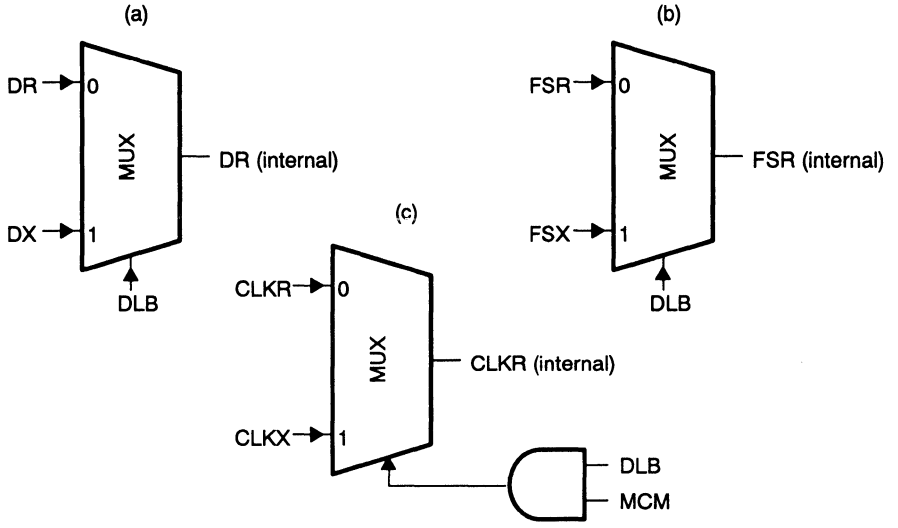
The reset of the serial port for both transmitter and receiver is done by the XRST bit and the RRST bit, bits 6 and 7, respectively. These signals are active low, so that if XRST=RRST=0, the serial port is in reset. To modify SPC to configure the serial port, a total of two writes to the SPC are necessary. The first write should write zeros to the XRST and RRST and the desired configuration bits 1–5. While maintaining the desired configuration bits, the second write should write ones to XRST, and RRST, bits, taking the serial port out of reset. Note that these bits can be reset individually if desired. When a zero is written to either of these bits, activity in the corresponding section of the serial port stops. When XRST=0 and RRST=0, the particular internal clocks to the serial port are shut off. This minimizes the switching and allows the device to operate on lower power consumption (as long as the CLKX bit is configured as an input — that is, with MCM=0).

The FSM bit (bit 3) specifies whether frame syncs are needed in consecutive serial port transmits. If FSM=1, a frame sync is required for every transfer and the mode is referred to as **burst mode**, because there may be periods of inactivity on the serial port between transmits. The frequency of packet writes to DXR is called packet frequency. The packets can be 8 or 16 bits long, depending on FO.

As the packet frequency increases, it reaches a maximum that is equivalent to 8 or 16 clock cycles, depending on FO. Note that this cycle count corresponds to 32 or 64 instruction cycles on the CPU, again depending on FO if internal 'C5x clocks are used. Thus, if transmitting at maximum rate for more than one transmission, the frame sync signal becomes extraneous. The continuous mode of operation (FSM=1) is then the mode that requires only an initial frame sync pulse, as long as a write to DXR for transmit, or a read from DRR for receive, is executed during each transmission. The timing of both modes is dicussed in detail in subsections 5.5.2 and 5.5.3.

The DLB bit, (bit 1) is a digital loop back mode that allows testing of the serial port code with just one device. When DLB=1, DR and FSR are connected to DX and FSX, respectively, through multiplexers, as shown in Figure 5–9.

Figure 5–9. Receiver Signal MUXes



CLKR is driven by CLKX if MCM=1. But if MCM=0 while DLB=1, then CLKR is taken from the CLKR pin. This allows for external clock generation of these signals during digital loopback mode. If DLB=0, then normal operation occurs where DR, FSR, and CLKR are all taken from their respective pins.

Bits 10–13 in the SPC are read-only status bits that indicate various states in serial port operation. Writes and reads to the serial port may be synchronized by polling RRDY and XRDY, (bits 10 and 11, respectively) or by using the interrupts that they generate. A transition from 0 to 1 of the RRDY bit indicates that the RSR has been copied to the DRR and that the received data may be read. A receive interrupt (RINT) is generated upon this transition. A transition from 0 to 1 of the XRDY bit indicates that the DXR contents have been copied to the XSR and that DXR is ready to be loaded with a new data word. A transmit interrupt (XINT) is generated upon this transition. Polling these bits in software may either substitute for or complement the use of serial port interrupts. In other words, both polling and interrupts can be used together if so desired. The $\overline{\text{XSREMPTY}}$ bit (bit 12) indicates whether the transmitter has experienced underflow. (When $\overline{\text{XSREMPTY}}$=0, it is active).

The following three situations cause the $\overline{\text{XSREMPTY}}$ flag to become active:

DXR has not been loaded since the last DXR–XSR transfer

AND XSR empties (The actual transition of $\overline{\text{XSREMPTY}}$ occurs after the last bit has been shifted out of XSR)

OR serial port reset (XRST=0)

OR device reset

When $\overline{\text{XSREMPTY}}$ is active, the transmit side of the serial port halts, thus driving no value (the DX pin is in a high-impedance state). An exception occurs in burst mode with external frame syncs, which is explained in subsection 5.5.4. Note that underflow does not constitute an error condition in the burst mode, although it does in the continuous mode (error conditions are further discussed in subsection 5.5.4). The $\overline{\text{XSREMPTY}}$ flag becomes inactive ($\overline{\text{XSREMPTY}}$=1) when:

A write to DXR occurs. Note that more information on the transmit timing is explained in subsection 5.5.2.

The RSRFULL bit, (bit 13) indicates whether the receiver has experienced overrun (When RSRFULL=1, it is active).

Overrun occurs when:

The DRR has not been read since the last RSR-to-DRR transfer.

AND RSR is full.

AND a frame sync pulse appears on FSR.

Note that in continuous mode (FSM=0), only the first two conditions apply; therefore, RSRFULL transitions after the last bit has been shifted out. When RSRFULL=1, the receiver halts and waits for DRR to be read. The data in RSR is preserved, but any new data driven on the DR pin while the receiver is halted is lost.

The RSRFULL flag becomes inactive (RSRFULL=0) under the following three conditions:

DRR is read

OR serial port is reset (RRST=0)

OR device is reset

IN0 and IN1 (bits 8 and 9) in the SPC allow the CLKR and CLKX pins to be used as bit inputs. IN0 and IN1 reflect the current levels of the CLKR and CLKX pins. The levels on the pins can be read by reading the SPC. They can be tested by using the PLU or BIT or BITT instructions. Note that there is a latency of between 0.5 and 1.5 CLKOUT1 cycles in length from CLKR/CLKX switching to the new CLKR/CLKX value being represented in the SPC. Note that if the serial port is put into reset, IN0 and IN1 can be used as bit inputs and DRR and DXR as general-purpose registers. SOFT and FREE (bits 14 and 15) are special emulation bits that determine the state of the serial port clock when a breakpoint is encountered in the high-level language debugger. If the FREE bit (bit 15) is set to one, then upon a software breakpoint, the clock continues to run (that is, free runs) and data is shifted out. In this case, SOFT (bit 14) is a *don't care*. But if FREE is 0, then SOFT takes effect. If SOFT=0, then the

clock immediately stops, thus aborting any transmission. If the SOFT bit is 1, the particular transmission continues until completion of the word, and then the clock halts. The options are as follows:

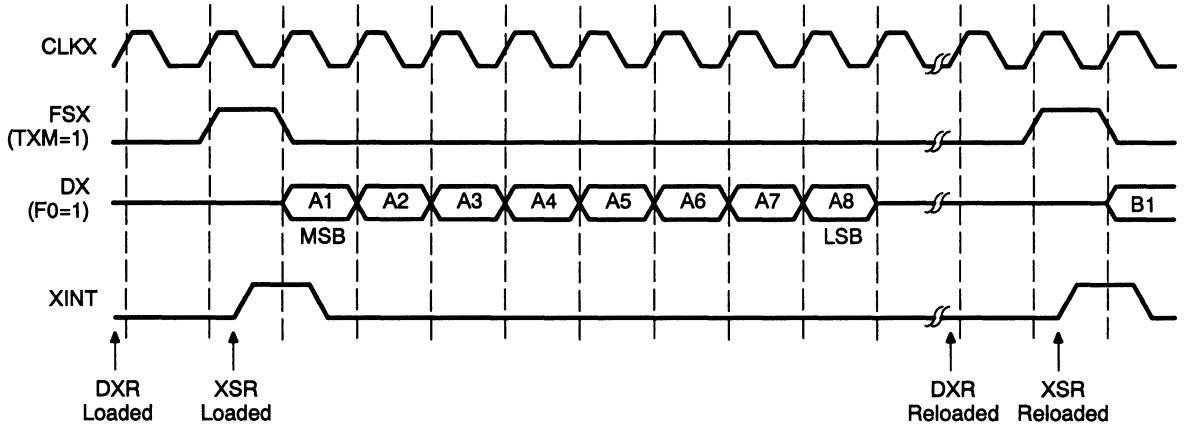
FREE	SOFT	
1	X	Free run
0	0	Immediate stop
0	1	Stop after completion of word

The receive side functions in a similar fashion. Note that if an option besides *immediate stop* is chosen, the receiver continues running and an overflow error is possible. The default value for these bits is *immediate stop*.

5.5.2 Transmit and Receive Operations (Burst Mode)

In burst mode operation, there are periods of serial port inactivity between packet transmits. The data packet is marked by the frame sync pulse on FSX. On the transmit device, the transmission is initiated by a write to DXR. The value in DXR is shifted to XSR; upon a frame sync pulse on FSX (generated internally or externally depending on TXM), the value in XSR is shifted out and driven on the DX pin. If DXR is reloaded before the old DXR contents have been transferred to XSR, the old DXR contents are overwritten. The DXR is copied to the XSR only if the XSR is empty and the DXR has been loaded since the last DXR to XSR transfer. The DXR should be written to only if XRDY=1, which is guaranteed if the DXR write is made in response to a transmit interrupt or polling XRDY. The timing for the serial port transmit is shown in Figure 5–10.

Figure 5–10. Burst-Mode Serial Port Transmit Operation



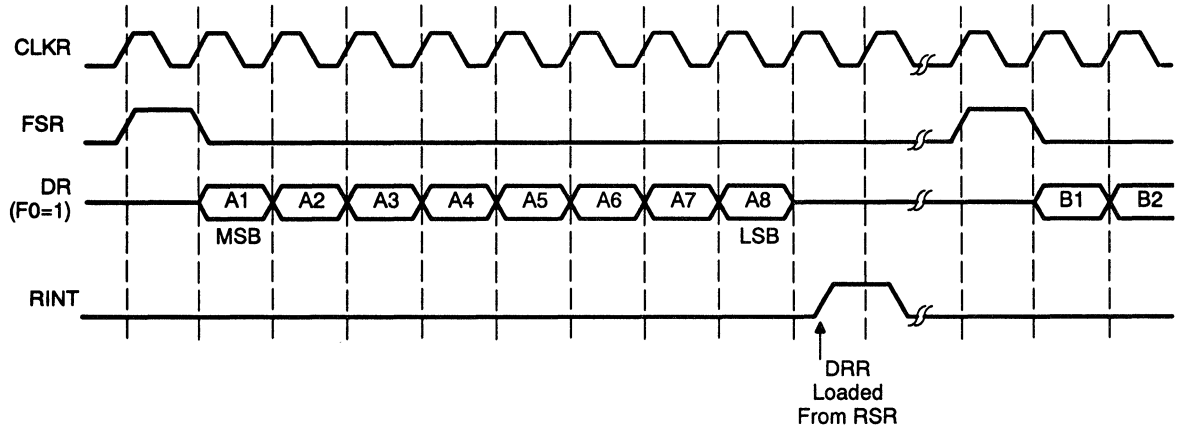
Note in the following discussion that the timings are slightly different for internally (TXM=1, FSX is an output) and externally (TXM=0, FSX is an input) gen-

erated frame syncs. This distinction is made because in the former case, the frame sync pulse is generated by the transmitting device as a direct result of a write to DXR. In the latter case, there is no such direct effect. Instead, the transmitting device must write to DXR and wait for an externally generated frame sync.

If frame sync pulses are internally generated (TXM=1), then after a write to DXR, a frame sync pulse is generated on the next rising edge of CLKX (For externally generated frame syncs the following events will occur whenever the frame sync pulse appears by the rising edge of CLKX after a write to DXR). Then on the next falling edge of CLKX, XSR is loaded with the value from DXR, and XRDY goes high, generating a transmit interrupt (XINT). On the next rising edge of the CLKX cycle, the first data bit (MSB first) is driven on the DX pin. With the fall of the frame sync pulse, the rest of the bits will be shifted out. (Therefore, the first bit could have variable length if the frame sync is generated externally and does not fall within one CLKX cycle. Internally generated frame syncs are guaranteed by 'C5x timings).

When all the bits are transferred, the DX pin enters the high-impedance state. Note that if DXR had not been loaded when XINT was generated, the $\overline{\text{XSREMPTY}}$ flag would become active (go low), indicating underflow. Thus, there is a 2-CLKX cycle latency (approximately) after DXR is loaded, before the data is driven on the line, assuming that the frame sync pulse is generated internally (TXM=1). If the pulse is externally generated, this latency does not exist, and the timing specifications are relaxed. With externally generated frame sync, if the $\overline{\text{XSREMPTY}}$ flag is active and a frame sync pulse is generated, any old data in the DXR is transmitted. This is explained in detail in subsection 5.5.4.

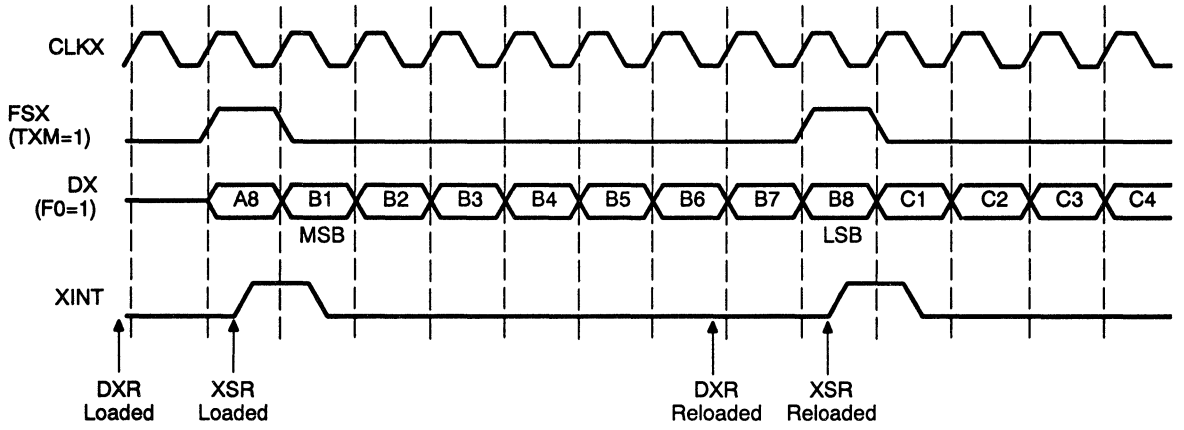
Figure 5–11. Burst-Mode Serial Port Receive Operation



The shifting into RSR begins on the falling edge of the CLKR cycle after the frame sync has gone low. After all the bits have been received, the contents of the RSR are transferred to the DRR on the falling edge of CLKR and RRDY goes high, generating a receive interrupt (RINT), as shown in Figure 5–11. Note that if the DRR from the previous receive had not been read and a frame sync appears, the RSRFULL flag would go high. This condition is an actual error and introduces questions of the serial port's behavior under various error situations: for example, the appearance of frame sync during a receive. Various error situations are discussed in subsection 5.5.4.

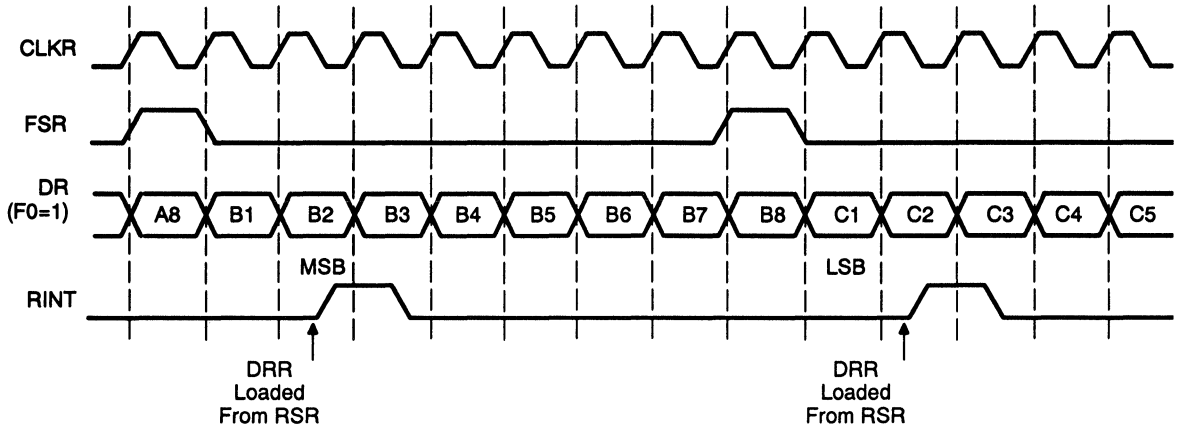
Note that if the packet frequency is increased, the inactivity period between the data packets for adjacent transfers decreases to zero. This corresponds to a minimum period between frame sync pulses (equivalent to 8 or 16 CLKX/R cycles, depending on FO) that corresponds to a maximum packet frequency at which the serial port may operate. At maximum packet frequency in Figure 5–12, the timing looks like a compressed version of Figure 5–10.

Figure 5–12. Burst-Mode Serial Port Transmit at Maximum Packet Frequency



The data bits in consecutive packets are transmitted continuously with no inactivity in between the bits. The frame sync pulse overlaps the last bit transmitted in the previous packet. The receive side in Figure 5–13 looks similar.

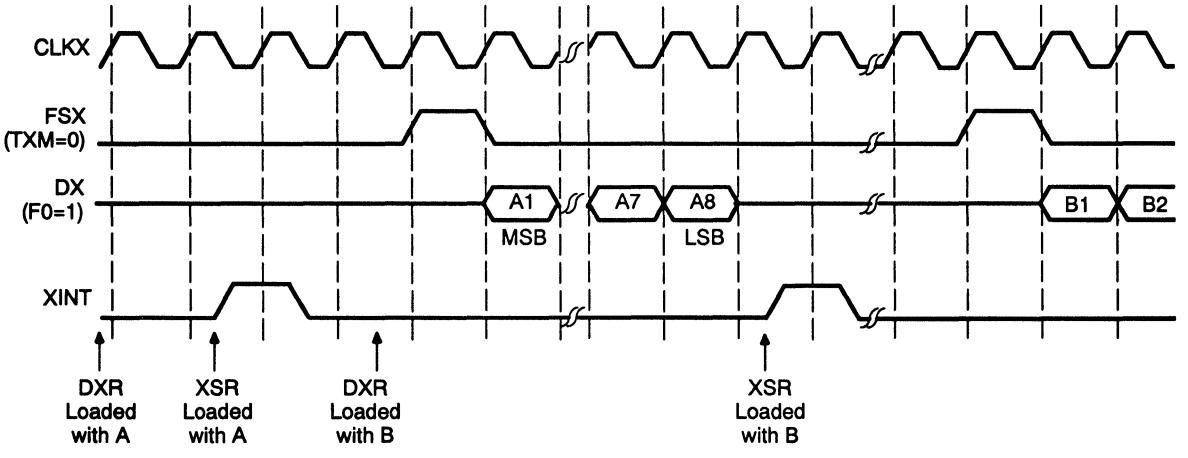
Figure 5–13. Burst-Mode Serial Port Receive at Maximum Packet-Frequency



The maximum packet frequency transfer looks like a compressed version of burst mode with no periods of inactivity. The frame sync pulse overlaps the first bit transmitted.

Figure 5–12 and Figure 5–13 show the transfer of multiple data packets at maximum packet frequency; the frame sync appears to be extraneous information. Since the data packets are transmitted at a constant rate, the CLK provides enough timing information for the transfer and permits a continuous stream of data. Theoretically, only an initial frame sync signal is needed to initiate the multipacket transfer. This continuous mode is supported by the 'C5x serial port and is discussed in subsection 5.5.3.

Figure 5–14. Burst-Mode Serial Transmit Operation With Delayed Frame Sync in External Frame Sync Mode



The operation of the serial port with external frame sync is similar to that with internal frame sync. Events occur when the external frame sync appears. When the external frame sync is delayed, however, the double buffer is filled and frozen until the delayed frame sync appears, as shown in Figure 5–14. When the delayed frame sync occurs, A is transmitted on DX; after the transmit, a DXR-to-XSR copy of B occurs, and XINT is generated. The next frame sync after the delayed frame sync causes B to be transmitted on DX. Note than when the loading of B into DXR occurs, a DXR-to-XSR copy of B does not occur, and XINT is not generated because A has not been transmitted on DX. Any subsequent writes to DXR before the delayed frame sync occurs would overwrite DXR.

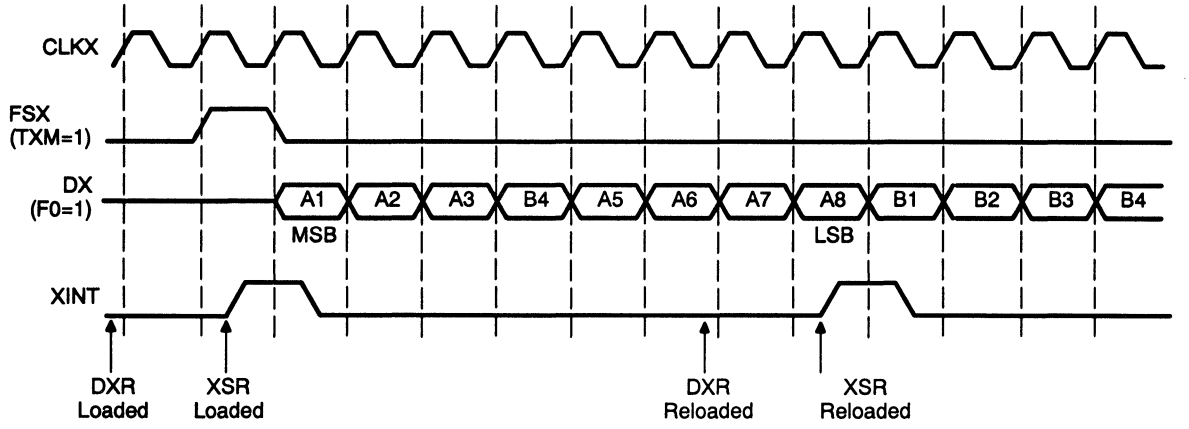
5.5.3 Transmit and Receive Operations (Continuous Mode)

In the continuous mode, the frame sync signal on FSX/FSR is not necessary for consecutive packet transfers at maximum packet frequency after the initial pulse. Continuous mode is selected by setting FSM=0. Upon the first store to DXR in continuous mode, a frame sync is generated for the first transmission and then no more. As long as DXR is updated once every transmission, the continuous mode continues. Failing to update causes the serial port to halt, as in the burst mode case (The $\overline{\text{XSREMPTY}}$ flag becomes asserted etc.). If DXR is written to after the halt, the device restarts the continuous mode transmit and generates an FSX, assuming that the frame sync is internally generated. This distinction that occurs between transmits using internal and external frame syncs is similar to the one discussed in subsection 5.5.2.

If the frame syncs are externally generated (TXM=0), then DXR should be loaded, and the appearance of an external frame sync on the FSX pin restarts a new continuous mode transmit. If the DXR has not been updated with external frame sync, the DX pin remains in the high-impedance state. This is different from the burst mode operation and is covered in detail in subsection 5.5.4. The continuous mode may be discontinued — in other words changed to burst mode — only by a serial port or device reset. Changing the FSM bit during transmit or halt is not guaranteed to switch to burst mode.

The transmit timing in continuous mode is shown in Figure 5–15.

Figure 5–15. Serial Port Transmit Continuous Operation



Transmit timing in continuous mode is similar to the continuous stream in Figure 5–12. The major difference is the lack of a frame sync pulse after the initial one. As long as DXR is updated once per transmission, this mode will continue. Overwrites to DXR behave just as in burst mode. The data written last will be transmitted. XSR operation is not disturbed. An external FSX pulse on the line will abort the present transmission, cause one data packet to be lost, and initiate a new continuous mode transmit. This is explained in more detail in subsection 5.5.4.

The receive operation is similar to the transmit operation. After the initial frame sync pulse on FSR, no more frame syncs are needed. This mode will continue as long as DRR is read every transmission. If it is not read, the serial port receive will halt (RSRFULL flag becomes active). Reading DRR will restart the continuous mode as soon as a frame sync is received. The continuous mode must be discontinued with a serial port or device reset. The receive timing can be seen in Figure 5–16.

Figure 5–16. Serial Port Receive Continuous Operation

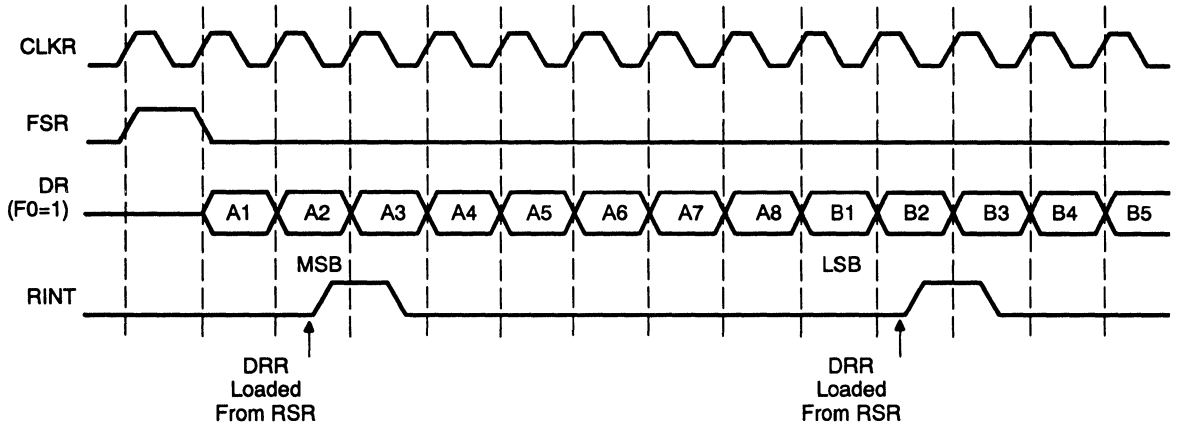


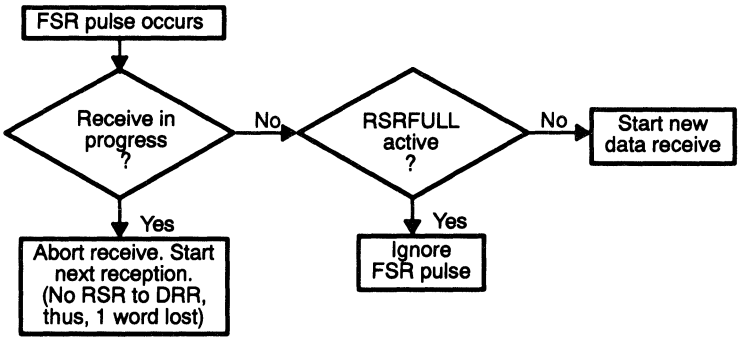
Figure 5–16 shows no frame signals; otherwise, it is similar to Figure 5–13. If a pulse occurs on FSR during transmission (an error), then the receive operation is aborted, one packet is lost, and a new receive cycle is begun. This is discussed in more detail on page 5-22.

5.5.4 Error Conditions

Error conditions result from an unprogrammed event occurring to the serial port. These conditions are operational aberrations such as overrun, underflow, or a frame sync pulse during a transmission. You may need to understand how the serial port handles these errors and the state it acquires during these error conditions. Because they differ slightly in burst and continuous modes, the error conditions are discussed separately.

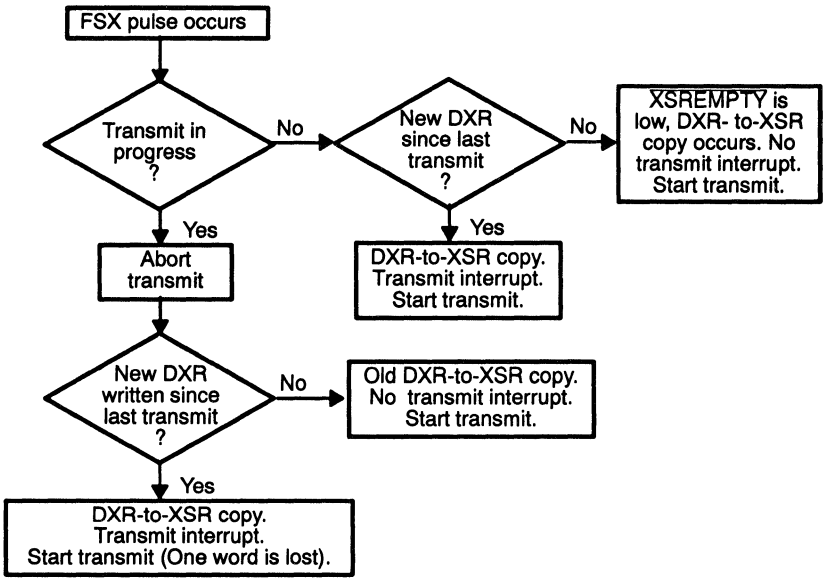
In burst mode, the first error condition (discussed in subsection 5.5.1) is the RSRFULL flag. Basically, this flag occurs when the device has not read incoming data and more data is being sent, which is indicated by a frame sync pulse on FSR. The processor halts serial port receives until DRR is read. Thus, any further data sent is lost. If receive errors continue, and the frame sync occurs during a receive (that is, data is being shifted into RSR from DR pin), then the present receive is aborted and a new one begins. Thus, the data that was being loaded into RSR is lost, but the data in DRR is not. No RSR-to-DRR copy occurs. Figure 5–17 shows the serial port receive side behavior for a frame sync pulse during a receive and includes nonerror situations.

Figure 5–17. Receive Error (Normal or Burst Mode)



Transmit errors in burst mode result when a frame sync occurs during various conditions. Underrun in burst mode is not considered an error but is explained in subsection 5.5.1. If a transmit is in progress (that is, XSR data is being driven on the DX pin) when the frame sync pulse occurs, then the present transmit is aborted, and data in the XSR is lost. Then, whatever data is in the DXR at the time of the frame sync pulse is transferred to XSR (DXR-to-XSR copy) for transmitting. However, a transmit interrupt XINT is generated only if the DXR has been written to after the last transmit. Also, if $\overline{\text{XSREMPTY}}$ is active and a frame sync pulse appears, the old data in DXR is shifted out. Figure 5–18 summarizes serial port transmit behavior with error (and nonerror) conditions.

Figure 5–18. Transmit Error (Normal or Burst Mode)



In continuous mode, errors take on a broader meaning. Data transfer is supposed to be occuring at all times in continuous mode. Thus, underflow

($\overline{\text{XSREMPTY}}$=0) is considered an error in continuous mode because data is not being transmitted. As in burst mode, overrun is an error, and both of these cause the serial port receive or transmit sections to halt. The operation of both these flags is explained in subsection 5.5.1 in the $\overline{\text{XSREMPTY}}$ and RSRFULL flags description. Underflow and overrun errors are not fatal; they can be corrected by reading DRR or writing to DXR. In a write to DXR to deactivate $\overline{\text{XSREMPTY}}$, either a frame sync pulse is generated (if FSM=1) or required (if FSM=0). On the receive side, however, after DRR is read to deactivate RSRFULL, a frame sync pulse is not required. The receive side of the serial port keeps track of the word (either 8- or 16-bit) boundary, even though it is not receiving data. When the RSRFULL flag is deactivated by a read from DRR, the receiver begins the read from the correct bit.

Another cause for error is the appearance of frame syncs during a transmission. After the initial frame sync in continuous mode, no others should occur. When a frame sync pulse occurs during a transmit, the current transmit operation (that is, serially driving XSR data onto DX pin) is aborted, and data in XSR is lost. A new transmit cycle is initiated, as long as the DXR is updated once per transmission afterward. During a receive in continuous mode, the situation is similar: if a frame sync pulse occurs, one packet of data (8-bit byte or 16-bit word, depending on FO) is lost. The RSR bit counter is reset, so the data that was being shifted into RSR from the DR pin is lost. Data then driven on DR is shifted into RSR. Therefore, the frame sync during transmission chart for continuous mode looks like the left half of the burst mode charts in Figure 5–17 and Figure 5–18 because a receive or transmit is always in progress.

Figure 5–19 and Figure 5–20 show receive and transmit errors for continuous mode. Note that if a frame sync occurs after deactivating the RSRFULL flag by reading DRR but before the beginning of the next word (either 8- or 16-bit) boundary, a receive abort condition occurs. Also, note a major difference in the transmit continuous mode error compared with transmit burst mode error. If $\overline{\text{XSREMPTY}}$ is active in continuous mode and an external frame sync occurs, no old data is transmitted. Instead, since underflow in continuous mode is considered an error, the frame sync pulse is ignored, and the DX pin remains in the high-impedance state.

Figure 5–19. Receive Error (Continuous Mode)

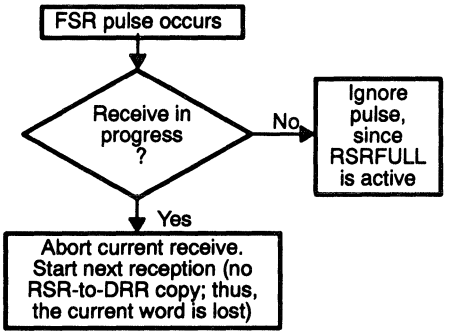
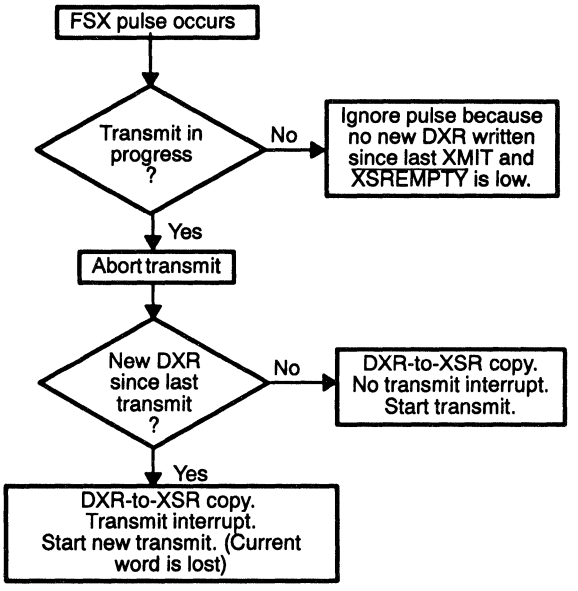


Figure 5–20. Transmit Error (Continuous Mode)



5.5.5 Example

The code example that follows shows a one-way transmit from device 0 to device 1 of an arithmetic sequence of numbers. The numbers are written in each device in a block from 9000h to b000h in data memory. Device 0 waits in a BIO loop for a ready to receive signal (XF) from device 1 and initializes the transfer with a value of zero. Only its transmit interrupt is enabled; its transmit ISR writes the value it will send into its own memory.

```
* Device 0 – Transmit side
:         :                    :
                               ;Setup SPC as CLK source
                               ;and internal frame sync
          SPLK   #0038h, SPC   ;Set TXM=MCM=FSM=1,
                               ;TDM=DLB=FO=0.
                               ;And put SP into reset
                               ;(XRST=RRST=0)
          SPLK   #00F8h, SPC   ;Take SP out of reset
                               ;Setup interrupts
          SPLK   #0ffffh, IFR ;clear IFR
          SPLK   #020h, IMR    ;Turn on XINT
          CLRC   INTM          ;enable interrupts
ILOOP     BCND   SENDZ, BIO    ;Wait to for ready–to–
          B      ILOOP         ;receive from other device
SENDZ     LACL   #0            ;First transmit/write
                               ;value is 0
          LAR    AR7, #9000h   ;Setup where to write
          SACL   *             ;Write first value
          SACL   DXR           ;Transmit first value
SELF1     B      SELF1         ;Wait for interrupts
XMT_ISR   LACC   AR7           ;Check if past 0x0b000
          SUB    #0b000h       ;i.e. end of block
          BCND   END_SERP,GEQ  ;Go to tight loop if so
                               ;Add one and transmit
          LACL   *+            ;Load value
          ADD    #1            ;Add one
          SACL   *             ;Write value
          SACL   DXR           ;Transmit value
          RETE
END_SERP  B      END_SERP      ;Sit in tight loop after
                               ;block is complete.
:         :                    :
```

The code in device 1 follows. It sends a ready-to-recieve signal (XF) to device 0. Only its receive interrupt is masked and its receive ISR reads from the DRR, writes to the block, and checks to see if it has reached the end of the block.

```
          :
          :
          :
Device 1 – Receive
                                ;Set SP as CLK, frame
                                ;sync receive
        SPLK #0008h, SPC        ;Set TXM=MCM=DLB=FO=0,
                                ;FSM=1.
                                ;And put SP into reset
                                ;(XRST=RRST=0)
        SPLK #00C8h, SPC        ;Take SP out of reset
                                ;Setup interrupts
        SPLK #0ffffh, IFR       ;clear IFR
        SPLK #010h, IMR         ;Turn on RINT
```

```
          CLRC INTM              ;Enable interrupts
          LAR  AR7, #9000h       ;Setup where to write
                                 ;received data
          CLRC XF                ;Signal ready to receive
SELF1     B    SELF1             ;Wait for interrupts
RCV_ISR
          LACL DRR               ;Load received value
          SACL *+                ;Write to memory block
          LACC AR7               ;Check if past 0x9000
          SUB  #0b000h           ;i.e. end of block
          BCND END_SERP, GEQ     ;Go to tight loop if so
END_SERP  B    END_SERP          ;Sit in tight loop after
                                 ;block is complete.
```

5.6 TDM Serial Port

The 'C5x devices have a TDM (time-division-multiplexed) serial port that allows the device to communicate serially with up to seven other 'C5x devices. The TDM port provides a simple and efficient interface for multiprocessing applications.

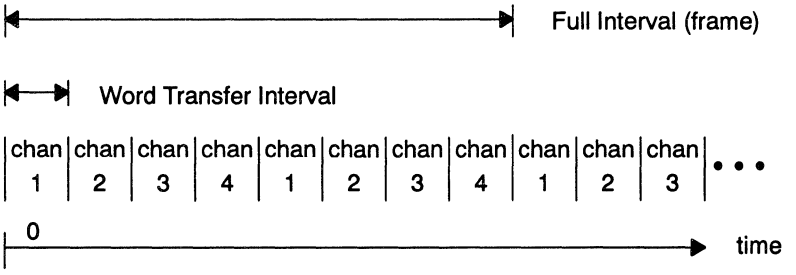
The TDM serial port is a superset of the serial port described in Section 5.5. By means of the TDM bit in the TSPC control register, the port can be configured in multiprocessing mode (TDM=1) or stand-alone mode (TDM=0). When in stand-alone mode, the port operates as described in Section 5.5. When in multiprocessing mode, the port behaves as described in this section. The port can be shut down for low power consumption via the XRST and RRST bits as described in Section 5.5.

5.6.1 Time-Division Multiplexing

Time-division multiplexing is the division of time intervals into a number of subintervals, with each subinterval representing a communications channel according to a prespecified arrangement. Figure 5–21 shows a 4-channel TDM scheme. Note that the first time slot is labeled chan 1 (channel 1), the next chan 2 (channel 2), etc. Channel 1 is active during the first communications period and during every fourth period thereafter. The remaining 3 channels are interleaved in time with channel 1, as shown in the figure.

The 'C5x TDM port supports eight TDM channels. You can independently specify which device is to transmit and which device or devices are to receive for each channel. This results in a high degree of flexibility in interprocessor communications.

Figure 5–21. Time-Division Multiplexing



5.6.2 TDM Port Operation

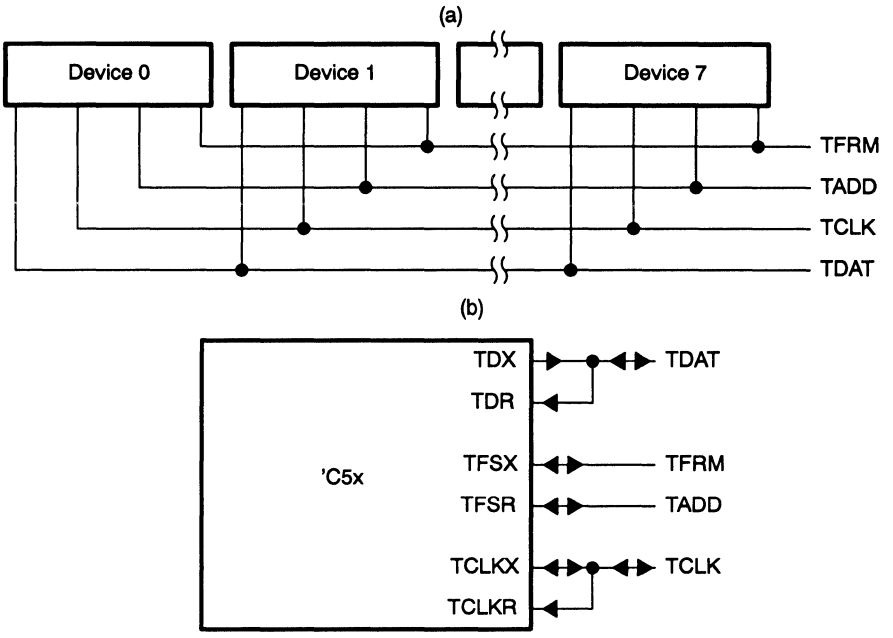
Figure 5–22(a) shows the 'C5x TDM port architecture. Up to eight devices can be placed on the four-wire serial bus. This four-wire bus consists of a conven-

tional serial port's bus of clock, frame, and data (TCLK, TFRM, and TDAT) wires plus an additional wire (TADD) that carries the device addressing information. The TADD line, which is driven by a particular device for a particular time slot, determines which devices in the TDM configuration can execute a valid TDM receive on that time slot. This is similar to a valid serial port read operation described in Section 5.5, except that the corresponding TDM registers are named differently. The TDM receive register is TRCV, and the TDM receive shift register is TRSR. The actual data is transmitted on the bidirectional TDAT line.

Note in Figure 5–22(b) that the device TDX and TDR pins are tied together externally to form the TDAT line. Also note that only one device can drive the data and address line (TDAT and TADD) in a particular slot. Meanwhile, in that particular slot, all the devices (including the one driving that slot) sample the TDAT and TADD lines to see if the data is a TDM valid read. This is discussed in detail later in this section. In a valid TDM read, the value is transferred from the TRSR register to the TRCV register, and a receive interrupt is generated, indicating that the TRCV has valid receive data and can be read.

All TDM port operations are synchronized by the TCLK and TFRM lines, which are generated by one device each (typically the same device), referred to as the TCLK and TFRM sources. The word master is not used here because it implies that one device controls the other. This is not the case, and you must set TCSR to prevent slot contention. Consequently, the remaining devices in the TDM configuration use these lines as inputs. Figure 5–22(b) shows TCLKX and TCLKR are externally tied together to form the TCLK line. Also, TFRM and TADD originate from the TFSX and TFSR pins respectively. The reason for this is to make the TDM serial port easy to use in standalone mode. The TDM port operation is controlled by several memory-mapped registers.

Figure 5–22. TDM Four-Wire Bus



Each device has six memory-mapped registers associated with the TDM serial port. The layout of these registers is shown in Figure 5–23. The TRCV and TDXR registers have the same functions as the DRR and DXR registers respectively, described in Section 5.5. The TSPC register is identical to the SPC register except that bit 0 is not reserved in TSPC. See subsection 5.5.1 for its operation. This bit (TDM) configures the port in stand-alone mode (TDM=0 – In this mode the TDM serial port operates like the standard serial port described in Section 5.5) or in multiprocessor mode (TDM=1).

Bits DLB and FO in the TSPC are hard-configured when the port is in multiprocessor mode. These bits are set to zero when TDM=1, resulting in no access to the digital loopback mode and in a fixed word length of 16 bits (A different type of loopback is covered in the example in subsection 5.6.5). The value of FSM does not affect the port when TDM=1. Also, when TDM=1 the underflow and overrun flags are not operational (subection 5.6.4 explains how these errors are treated in TDM mode). If TDM=1, changes made to the contents of the TSPC become effective upon completion of channel 7 of the current frame. Thus the TSPC value cannot be changed for a the current frame. Any changes take effect on the next frame.

Figure 5–23. TDM Port Registers

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRCV	Receive Data															
TDXR	Transmit Data															
TSPC	FREE	SOFT	X	X	XRDY	RRDY	IN1	IN0	RRST	XRST	TXM	MCM	FSM	FO	DLB	TDM
TCSR	X	X	X	X	X	X	X	X	CH7	CH6	CH5	CH4	CH3	CH2	CH1	CH0
TRTA	TA7	TA6	TA5	TA4	TA3	TA2	TA1	RA0	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
TRAD	X	X	X2	X1	X0	S2	S1	S0	A7	A6	A5	A4	A3	A2	A1	A0

The source device for the timing signals TFRM and TCLK is set by MCM and TXM, respectively. The TCLK source device is identified by setting the TXM bit of its TSPC register to one. Typically, this device is the same one that supplies the TDM port clock signal TCLK. TCLKX pin is configured as an input if MCM=0 and an output if MCM=1. In the latter (internal 'C5x clock) case, the device whose MCM=1 supplies the clock (TCLK frequency=one fourth of CLKOUT1 frequency) for all devices on the TDM bus. The clock can be supplied by an external source if MCM=0 for all devices. TFRM can also be supplied externally if TXM=0. An external TFRM must meet TDM receive timing specifications with repect to TCLK for proper operation. No more than one device should have MCM or TXM set to one at any given time. The specification of which device is to supply clock and framing signals is typically made only once, during system initialization.

The TDM channel select register (TCSR) of a given device specifies in which time slot(s) that device is to transmit. A 1 in bits 0–7 of the TCSR sets the transmitter active during the corresponding time slot. A key system-level constraint repeated here is that no more that one device can transmit during the same time slot. The devices do ***not*** check for bus contention. You must assign the slots consistently. As in TSPC operation, a write to TCSR during a particular frame is valid only during the next frame. However, a given device can transmit in more than one slot. This is discussed in more detail in subsection 5.6.3, with an emphasis on the utilization of TRTA, TDXR, and TCSR in this respect.

The TDM receive/transmit address register (TRTA) of a given device specifies two key pieces of information. The lower half specifies the receive address of the device, while the upper half of TRTA specifies the transmit address. The receive address is the 8-bit value that a device compares to the 8-bit value it samples on the TADD line in a particular slot to determine whether it should execute a valid TDM receive. The receive address establishes the slots in which that device may receive. This process occurs on each device during every slot. The transmit address corresponds to what a device drives on the TADD line during a transmit operation on an assigned slot. The transmit ad-

dress establishes which receiving devices may execute a valid TDM receive on the driven data.

Only one device at a time can drive a transmit address on TADD. Each processor bitwise-logical-ANDs the value it samples on the TADD line with its receive address. If this operation results in a nonzero value, then a valid TDM receive is executed. Thus, for one device to transmit to another, there must be at least one bit in the upper half of the first device's TRTA (the transmit address) with a value of 1 that matches one bit with a value of 1 in the lower half of TRTA (the receive address) of the second device. This method of configuration of TRTA allows the transmitting device to control which devices receive, without having to change the receive address on any of the devices.

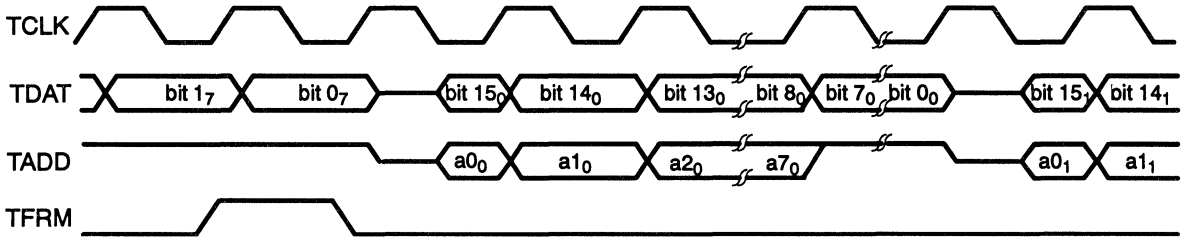
The TDM receive address register (TRAD) holds various information on the status of the TADD line, which can be polled to verify the integrity of this line and to verify the relationship between instruction cycle and TDM port timing. Bits 13–11 (x_2–x_0) hold the current slot number value, whether a valid data receive was executed or not. This value is latched at the begininng of the slot and latched only until the end of the slot. Bits 10–8 (s_2–s_0) hold the number of the last slot plus one (modulo 8) in which data was received. This value is latched at the end of the slot in which a valid data receive occurred during the TDM receive interrupt (TRNT), and maintained until the end of the next slot that is a valid receive. Bits 7–0 (a_7–a_0) hold the last value sampled on the TADD line, whether a valid data receive was executed or not. This value is latched half-way through the slot (so the value on the TADD may be shifted in) and maintained until half-way through the next slot, whether a valid receive is executed or not.

5.6.3 Transmit and Receive Operations (TDM Mode)

Figure 5–16 shows the timing for the TDM port transfers. The TCLK and TFRM signals are generated by the timing source device. The TCLK frequency is one fourth the frequency of CLKOUT1 if generated by a 'C5x device. The TFRM pulse occurs every 128 TCLK cycles. This allows 16 data bits for each of 8 time slots to be driven on the TDAT line. This also permits the processor to execute a maximum of 64 instructions between each slot, assuming that a 'C5x internal clock is used. Beginning with slot 0 and with the MSB first, the transmitter drives 16 data bits for each slot, with each bit having a duration of 1 TCLK cycle (the exception is the first bit of each slot, as noted below). The data is driven onto the TDAT line on the rising edge of TCLK and read on the falling edge. Meanwhile, the transmitter also drives the TADD line with its transmit address. This information, unlike that on TDAT, is only one byte long and is transmitted with the LSB first for the first half of the slot. During the second half of the slot (that is, the last eight TCLK periods) the TADD line is driven high. The TDM

receive logic samples the TADD line only for the first eight TCLK periods, ignoring it during the second half of the slot. Therefore, the transmitting device (if not a 'C5x) may choose to drive TADD high or low during that time period.

Figure 5–24. Serial Port Timing in TDM Mode



If none of the devices on the TDM bus are configured to transmit in a slot (that is, none of the devices have a 1 for the corresponding slot in their TCSR register), that slot qualifies as an empty slot. In an empty slot, both TADD and TDAT will be high impedance. This has the potential for spurious receives because the device actually samples TDAT and TADD for every slot and determines a valid TDM receive if its receive address matches the receive address on the TADD line. To avoid spurious reads, a pull-down 1-kΩ resistor ***must*** be tied to the TADD line. This causes the TADD line to read low on empty slots. Otherwise, any noise on the TADD line that happens to match a particular receive address would result in a spurious read. If power dissipation is a concern and the resistor is not desired, then an arbitrary processor with transmit address equal to 0h can drive empty slots by writing to TDXR in those slots. Slot manipulation is explained later in this section. The 1-kΩ resistor is not needed in the TDAT line.

An empty slot is defined by the following two cases: the first obvious case occurs when no device has its TCSR configured to transmit in that slot. A second more subtle case occurs when TDXR has not been written to before a slot. This may happen when TCSR contents are changed because they are not sampled until the TFRM pulse occurs. Therefore, any subsequent change takes effect only on the next frame. The same is true for the receive address (the lower half of TRTA). But the transmit address (upper half of TRTA) and the TDXR (obviously) may be changed for the current frame for a particular slot, assuming that slot has not yet been reached when the instruction is executed.

Note that the transmit address does not need to be written every time a write to TDXR is executed. During a write to TDXR, whatever value is in the TRTA is transmitted. You can test the current slot by examining TRAD while using the XRDY flag or transmit interrupt. This flexibility affords TDM slot manipula-

tion and even slot sharing if you so desire. The key is to understand the timing relationship between the instructions being executed and the frame/slots of the TDM port. Simply stated, the TCSR and the receive address (lower half of TRTA) take effect only at the start of a new frame, while the transmit address (upper half of TRTA) and TDXR (transmit data) can take effect at the start of a new slot.

When changing a transmit address on the fly, be careful not to corrupt the receive address; both are located in the same register TRTA. Thus, this scheme follows the philosophy of allowing the transmitting device to set which devices can receive. Regarding empty slots, note that in a TDM port the frame sync on TFRM is being transmitted at all times, not just when there is a write to TDXR. Thus, if a device does not happen to write to TDXR during its selected slots (by TCSR), it will have an empty slot that shows up as high impedance on the TDAT and TADD lines.

As a final note on timing, the duration of the first bit (bit 15 TDAT and bit 0 of TADD) of each slot is only half the normal duration. Also, the TFRM overlaps bit 0 of time slot 7. Refer to the timing diagrams in Appendix A.

5.6.4 TDM Error Conditions

Due to time slots and the ability for one processor to transmit in multiple slots, the concept of overflow and underrun becomes unclear. Thus, the overrun and underflow flags are not enabled in the TDM port in TDM mode. On the receive side, if DRR has not been read and a valid receive operation is initiated (due to the value on TRTA and the device's receive address), the present value of DRR is overwritten. Thus, the TDM port is ***not*** halted. On the other hand, during a transmit if DXR has not been updated, nothing will be driven on the TADD or TDAT lines. The pins will be in high impedance. This mode of operation prevents spurious transmits from occurring.

If TFRM pulses occur during a nonregular time in transmission, the TDM port fails. In other words, only one TFRM should occur every 128 TCLK cycles. Unlike the serial port, the TDM port cannot be reinitialized with a frame sync pulse during transmission.

5.6.5 Example of TDM Operation

Table 5–9 shows the data represented by the TADD signal for each of the eight channels, given the transmitter and receiver designations shown. This example shows the configuration for eight devices to communicate with each other. In this example, device 0 broadcasts to all device addresses. In subsequent frames, devices 1–7 communicate to one other processor.

Table 5–9. Interprocessor Communications Scenario

Channel	TADD Data	Transmitter Device	Receiver Device(s)
0	0FEh	0	1–7
1	40h	7	6
2	20h	6	5
3	10h	5	4
4	08h	4	3
5	04h	3	2
6	02h	2	1
7	01h	1	0

Table 5–10 shows the TDM port register contents of each device that results in the scenario given in Table 5–9. Device 0 provides the clock and frame control signals for all channels and devices. The TCSR and TRTA register contents specify which device is to transmit on a given channel and which devices are to receive.

Table 5–10. TDM Register Contents

Device	TSPC	TRTA	TCSR
0	xxF9h	0FE01h	xx01h
1	xxC9h	0102h	xx80h
2	xxC9h	0204h	xx40h
3	xxC9h	0408h	xx20h
4	xxC9h	0810h	xx10h
5	xxC9h	1020h	xx08h
6	xxC9h	2040h	xx04h
7	xxC9h	4080h	xx02h

In Table 5–10, the transmit address of a particular device (the upper byte of TRTA) matches the receive address (the lower byte of TRTA) of the receiving device. But it is not necessary for the transmit and receive addresses to match exactly. Remember that the matching operation implemented on the receive side is a bitwise AND. Thus, only one bit must match. The advantage of this scheme is that a transmitting device can select the devices to receive its data by changing its transmit address only. The receive address of the receiving device does not need to be changed (assuming the receive address is unique). In the example, device 0 can transmit to any combination of the other devices by merely writing to the upper byte of TRTA. For example, if it changed its TRTA to 08001h on the fly, it would transmit only to device 7. A device can write to itself because the transmit is executed on the rising edge and the receive

on the falling edge of TCLK. To enable this sort of loop back, it is necessary to have the wired-OR pins connected (the TDAT and TCLK lines). In the example, if device 0 has a TRTA of 00101h, it would transmit to itself.

In the code example below, a one-way transmit from device 0 to device 1 of an arithmetic sequence of numbers is shown. The numbers are written in each device in a block from 4000h to 6000h in data memory. Device 0 transmits on slot 0 and has a transmit address of 01h. It waits in a $\overline{\text{BIO}}$ loop for a ready to receive signal (XF) from device 1 and initializes the transfer with a value of zero. Only its transmit interrupt is enabled, and its transmit ISR writes the value it will send into its own memory.

```
* Device 0 – Transmit side
                :
                :
                :
        SPLK #1h, TCSR          ;Setup TCSR to xmt on
                                ;slot 0
        SPLK #100h, TRTA        ;Setup transmit address
                                ;Set up TSPC as TCLK, TFRM
                                ;source
        SPLK #0039h, TSPC       ;Set TXM=MCM=FSM=TDM=1,
                                ;DLB=FO=0.
                                ;And put TDM into reset
                                ;(XRST=RRST=0)
        SPLK #00F9h, TSPC       ;Take TDM out of reset
                                ;Setup interrupts
        SPLK #0ffffh, IFR       ;clear IFR
        SPLK #080h, IMR         ;Turn on TXNT
        CLRC INTM               ;enable interrupts
TILOOP  BCND TSENDZ, BIO        ;Wait for ready-to-
        B    TILOOP             ;receive from other device
        TSENDZ  LACL #0         ;First transmission/write
                                ;value is 0.
        LAR  AR7, #4000h        ;Setup where to write
        SACL *                  ;Write first value
        SACL TDXR               ;Transmit first value
SELF2   B    SELF2              ;Wait for interrupts
TXMT_ISR
        LACC AR7                ;Check if past 0x6000
        SUB  #6000h             ;i.e. end of block
        BCND END_TDMP, GEQ      ;Go to tight loop if so.
                                ;Add one and transmit
        LACL *+                 ;Load value
        ADD  #1                 ;Add one
        SACL *                  ;Write value
        SACL TDXR               ;Transmit value
        RETE
```

```
END_TDMP    B  END_TDMP          ;Sit in tight loop after
                                 ;block is complete.
                    :
                    :
                    :
```

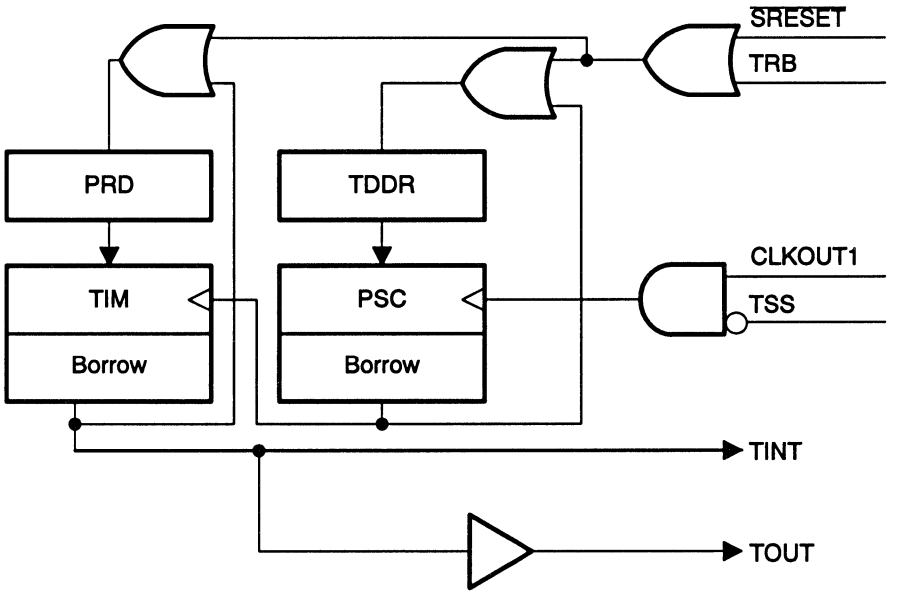
The code in device 1 follows. It has a receive address of 01h and sends a ready-to-receive signal (XF) to device 0. Only its receive interrupt is masked, and its receive ISR reads from the TDRR, writes to the block, and checks to see if it has reached the end of the block.

```
                    :
                    :
                    :
*Device 1 – receive side
        SPLK #0h, TCSR           ; Setup TCSR to xmt on
                                 ; no slots
        SPLK #001h, TRTA         ; Setup receive address

                                 ; Set TDM as TCLK, TFRM
                                 ; receive
        SPLK #0009h, TSPC        ; Set TXM=MCM=DLB=FO=0,
                                 ; FSM=TDM=1.
                                 ; And put TDM into reset
                                 ; (XRST=RRST=0)
        SPLK #00C9h, TSPC        ; Take TDM out of reset

                                 ; Setup interrupts
        SPLK #0ffffh, IFR        ; clear IFR
        SPLK #040h, IMR          ; Mask on TRNT

        CLRC INTM                ; enable interrupts
        LAR  AR7, #4000h         ; Setup where to write
                                 ; received data
        CLRC XF                  ; Signal ready to receive

SELF2   B    SELF2               ; Wait for interrupts

TRCV_ISR
        LACC TRCV                ; Load received value
        SACL *+                  ; Write to memory block
        LACC AR7                 ; Check if past 0x6000
        SUB  #6000h              ; i.e. end of block
        BCND END_TDMP, GEQ       ; Go to tight loop if so
        RETE

END_TDMP    B  END_TDMP          ; Sit in tight loop after
                                 ;  block is complete.
```

5.7 Timer

The timer is an on-chip down counter that can be used to periodically generate CPU interrupts. The timer is decremented by one at every CLKOUT1 cycle. A timer interrupt (TINT) is generated each time the counter decrements to zero. The timer thus provides a convenient means of performing periodic I/O or other functions. Figure 5–25 shows a logical block diagram of the timer. When the timer is stopped (TSS = 1), the internal clocks to the timer are shut off, allowing the device to run in a lower power mode of operation.

Figure 5–25. Timer Block Diagram



The timer interrupt rate is given by

$$\text{TINT rate} = \frac{1}{t_{c(C)} \times u \times v} = \frac{1}{t_{c(C)} \times (\text{TDDR} + 1) \times (\text{PRD} + 1)}$$

where $t_{c(C)}$ is the period of CLKOUT1, u is the sum of the TDDR contents (see Table 5–11) plus 1, and v is the sum of the PRD contents (see Figure 5–25) plus 1.

Therefore, the timer interrupt rate is equal to the CLKOUT1 frequency divided by two independent factors. Referring to Figure 5–25, each of the two divisors is implemented with a down counter and period register. The counter and period registers for the first stage are the PSC and TDDR fields of the TCR, respectively, and each is 4 bits wide. The counter and period registers for the second stage are the memory-mapped, 16-bit wide TIM and PRD registers. Each time

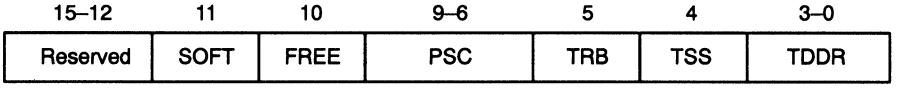
a counter decrements to zero, a borrow is generated on the next CLKOUT1 cycle, and the counter is reloaded with the contents of its corresponding period register. The output of the second stage is the timer interrupt signal sent to the CPU and to the timer output pin (TOUT). The width of the borrow pulse appearing on the output of stage 2 is equal to $t_{c(C)}$ (see Appendix A).

The timer operation is controlled via the timer control register (TCR). Bits 0–3 constitute the TDDR field of the TCR. Upon reset, TDDR is set to zero. The timer can be stopped and restarted with the TSS bit and can be reset with the TRB bit. The timer is stopped by setting the TSS bit to one and restarted by setting the TSS bit to zero. When the timer stopped, the internal clocks are shut off to the timer, allowing a lower power mode of operation. Upon reset, the TSS bit is zero, and the timer immediately starts timing. The timer period can be reloaded by setting the TRB bit to one. These bits are defined in the TCR as shown in Table 5–11. Bits 6–9 constitute the PSC field of the TCR. Figure 5–26 shows the bit layout of the timer control register.

Table 5–11. Timer Control Register

Bit	Name	Description
0–3	TDDR	Timer Divide-Down Ratio
4	TSS	Stop Timer = 1, Restart Timer = 0
5	TRB	Reload Timer with Period = 1
6–9	PSC	Prescaler Counter

Figure 5–26. Timer Control Register (TCR)



The contents of the PRD register are loaded into the timer counter register (TIM) when the timer counter register decrements to zero or when the timer is reset by setting the TRB bit to 1. The TRB bit is always read as zero. When a 1 is written to TRB, the timer is reset, but TRB is still read as zero. The TDDR (timer divide down register) is loaded by writing the appropriate divide-down value into the TCR. As with the TIM/PRD register pair, the value of TDDR is not immediately loaded into the prescaler counter (PSC). The prescaler counter is loaded with the value in TDDR when it decrements to zero or when the timer is reset by setting the TRB bit to 1. The PSC can be read by reading the TCR register, but cannot be written directly via software. Bits 10 and 11 are special emulation bits that determine the state of the serial port clock when a breakpoint is encountered in the high-level language debugger. Please see page 5-23 for their functional description. Bits 15–12 are always read as zero.

The current value in the timer can be read by reading the TIM register; the prescaler counter can be read by reading the TCR. Because it takes two instructions to read both registers, there may be a change between the two reads as the counter decrements. Therefore, where precise timing measurements are being made, it may be more accurate to stop the timer to read these two values. The timer can be stopped by setting the TSS bit to one and restarted by resetting this bit to zero.

The timer provides a convenient and efficient way to generate a sample clock for an analog interface. Consider the following example of using the timer to generate a sample rate of 50 kHz. The initialization for this example is as follows:

```
*Clkin frequency = 20 MHz,  timer is running at 10 MHz.
*
    LDP  #0
    SPLK #199,PRD    ;Load timer period for 20 us period.
    OPL  #8,IMR      ;Set timer interrupt mask bit
    SPLK #20h,TCR    ;reload and start timer.
    SPLK #10000b,IFR ;Clear any pending timer interrupts.
    CLRC INTM        ;global interrupt enable.
*
```

Consider an analog-to-digital converter operating at this sample rate. A typical interrupt service routine (ISR) would be as follows:

```
*50 kHz sample rate A/D interrupt service routine
*
TIMER_ISR MAR*,AR3  ;Use auxiliary register reserved for
                    ;timer ISR.
          IN *,14   ;Read A/D.
          RETE      ;Re-enable interrupts and return.
*
```

5.8 Divide-by-One Clock

The divide-by-one clock feature on the 'C5x consists of a phase lock loop (PLL) peripheral, which provides the capability to supply a clock cycling at the machine cycle rate of the CPU. This is a desirable feature because it reduces a system's high-frequency noise that is due to a high-speed switching clock. When this peripheral feature is implemented, the external frequency source can be used by injecting the clock directly into CLKIN2, with X1 left unconnected and X2 connected to V_{DD}. The divide-by-one option is used when the CLKMD1 pin is strapped high and CLKMD2 is strapped low. The PLL is not enabled in all other clock modes, and clocks are shut off to the module to allow a lower power mode of operation.

The processor generates two internal clocks, via the input clock, to the device. The CLKOUT1 signal indicating the CPU machine cycle rate equals the input clock. The PLL has a maximum operating frequency of 28.6 MHz (on a 35-ns 'C5x device). The PLL requires a transitory locking time of 256 cycles. See Appendix A for more information on the external input frequency specification.

Chapter 6

Memory

The total memory address range of the 'C5x devices is 224K 16-bit words. The memory space is divided into four specific memory segments: 64K program, 64K local data, 32K global data, and 64K I/O port. The parallel nature of the architecture of the 'C5x devices allows for the device to perform three concurrent memory operations in any given machine cycle: fetching an instruction, reading an operand, and writing an operand. The 'C5x memory configuration and operation are described in the following sections:

Topic **Page**

6.1 Memory Space

The 'C5x design is based on the enhanced Harvard architecture. This architecture has multiple memory spaces that can be accessed on three parallel buses; this makes it possible to access both program and data simultaneously. The three parallel buses are the program read/write bus (PAB), data read bus (DAB1), and data write bus (DAB2). Each bus accesses different memory spaces for different aspects of the device operation. The 'C5x memory is organized into four individually selectable spaces: program, local data, global data, and input/output ports (I/O). These spaces compose an address range of 224K words. Within any of these spaces RAM, ROM, EPROM, EEPROM, or memory-mapped peripherals can reside either on- or off-chip.

The program space contains the instructions to be executed as well as tables used in execution. The local data space stores data used by the instructions. The global data space can share data with other processors within the system or can serve as additional data space. The I/O space interfaces to external memory-mapped peripherals and can also serve as extra data storage space. Within a given machine cycle, the CALU can execute as many as three concurrent memory operations. This chapter describes each memory space and the 'C5x memory map.

The 'C5x devices include a considerable amount of on-chip memory to aid in system performance and integration. The 'C50 includes 2K words of boot ROM, 9K words program/data single-access RAM (SARAM) , and 1056 words of dual-access data RAM (DARAM). The boot ROM resides in program space at address 0 and includes a device test (for internal use) and boot code. The 9K block of single-access RAM can be mapped to program and/or data space and resides at address 0800h in either space. The single-access RAM requires a full machine cycle to perform a read or a write. The dual-access RAM can be read from and written to in the same cycle. The 1056 words of dual-access RAM are configured in three blocks: block 0 (B0) is 512 words at address 0100h–02FFh in local data memory, or 0FE00h–0FFFFh in program space; block 1 (B1) is 512 words at address 0300h–04FFh in local data memory; and block 2 (B2) is 32 words at address 060h in local data memory.

The 'C51 removes the 2K boot ROM from program memory space. It also replaces 8K words of single-access program/data RAM with an 8K-word block of maskable ROM. The ROM is located in the address range 0h–1FFFh in program space. The additional 1K word of single-access RAM is mapped to data space (800h–0BFFh), program space (2000h–23FFh), or both spaces. The dual-access blocks of RAM on the 'C51 are mapped at the same addresses as the 'C50.

The 'C53 has 16K words of on-chip maskable ROM and 3K words of single-access RAM. The ROM is located in the address range 0–3FFFh in program

space. The 3K words of single-access RAM are mapped into data space (800–13FFh), program space (4000–4BFFh), or both spaces. The dual-access RAM blocks on all 'C5x devices are mapped at the same addresses.

Figure 6–1. 'C50 Memory Map

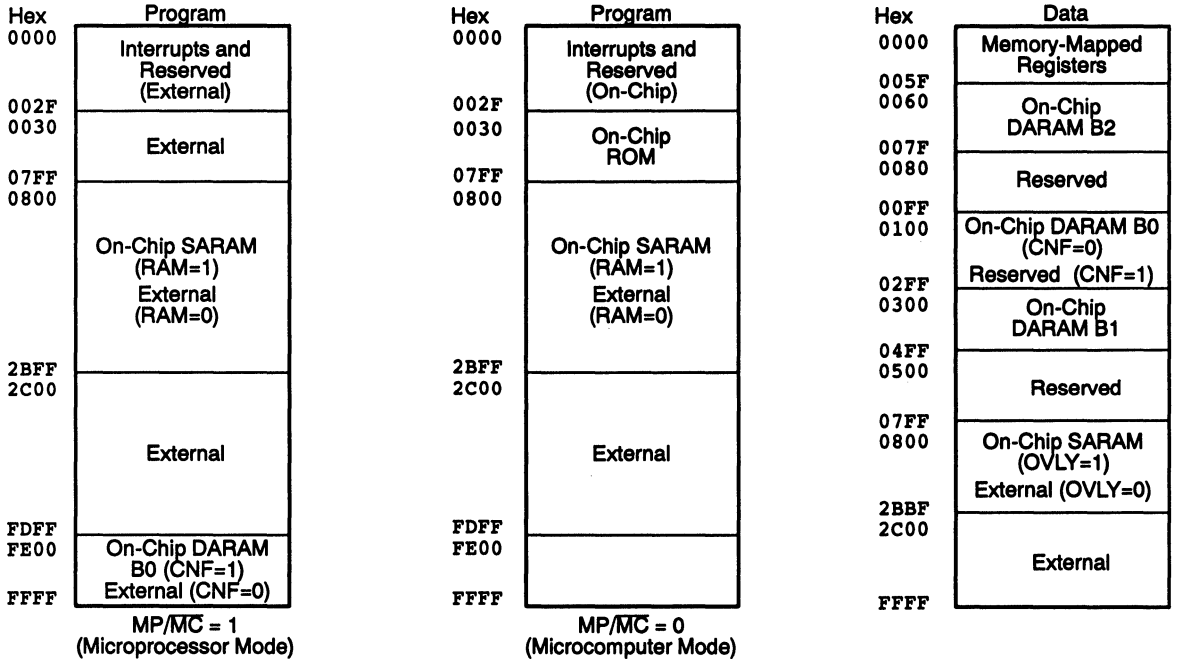


Figure 6–2. 'C51 Memory Map

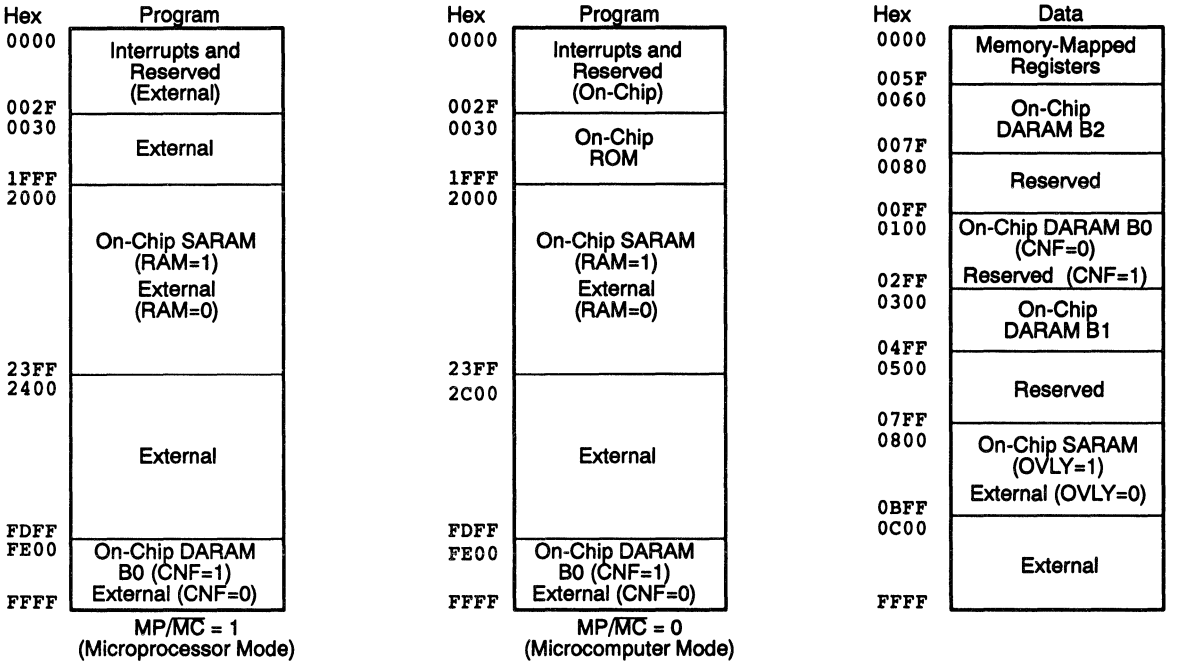
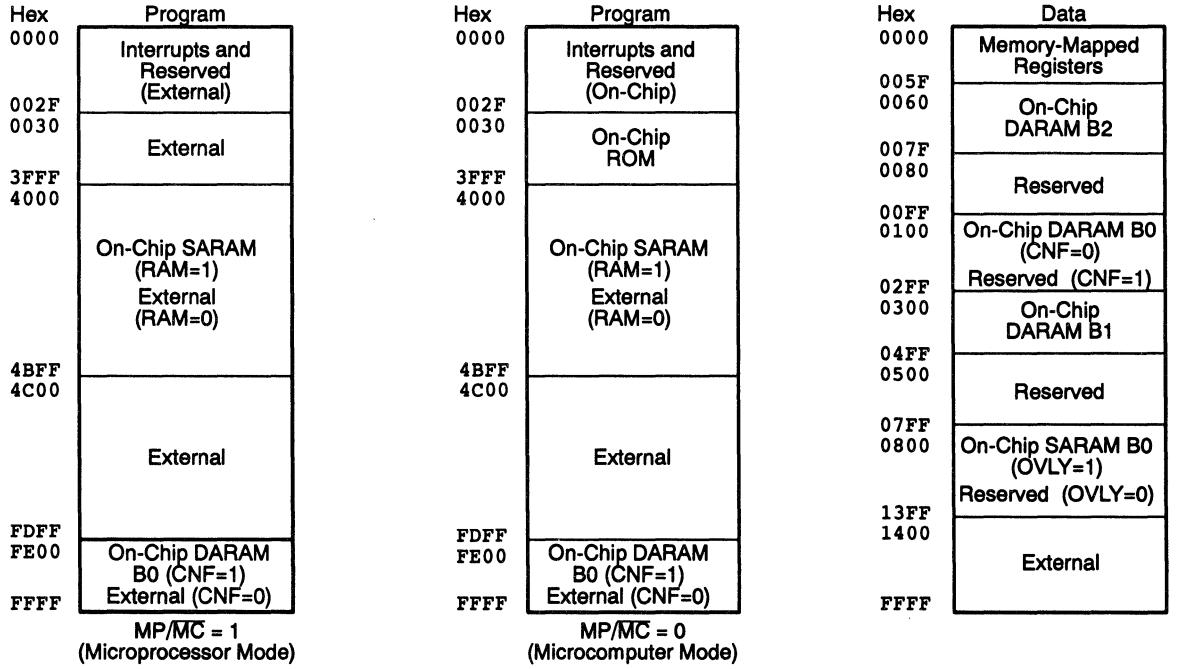


Figure 6–3. 'C53 Memory Map



6.2 Program Memory

The external program memory space on the 'C5x devices addresses up to 64K 16-bit words. In addition, 'C5x devices have on-chip ROM, single-access program/data RAM, and dual-access RAM. Software can configure these memory cells to reside inside or outside of the program address map. When they are mapped into program space, the device automatically accesses them when it addresses within their bounds. When the CALU generates an address outside these bounds, the device automatically generates an external access. The advantages of operating from on-chip memory are as follows:

1) Higher performance because no wait states are required for slower external memories.
2) Lower cost than external memory.
3) Lower power than external memory.

The advantage of operating from off-chip memory is the ability to access a larger address space.

6.2.1 Program Space Configurability

The program memory can reside both on- and off-chip. After reset, the configuration is set by the level on the MP/$\overline{\text{MC}}$ pin. If this pin is high, the device is configured as a microprocessor, and the on-chip ROM is not addressed. If this pin is low, the device is configured as a microcomputer, and the on-chip ROM is enabled. The 'C5x devices fetch their reset vector at location 0 of program memory; so, if the device is operating as a microcomputer, it starts running from on-chip ROM. Otherwise, it starts running from off-chip memory. Once the program is running, you can change the MP/$\overline{\text{MC}}$ configuration by setting or clearing the MP/$\overline{\text{MC}}$ bit in the PMST register. Note that the MP/$\overline{\text{MC}}$ pin is sampled only at reset. The following instruction removes the ROM from program space:

```
OPL#8,PMST   ;Remove boot ROM from program space.
```

You can submit code to be masked for the 'C51's 8K-word or for the 'C53's 16K-word on-chip ROM. This is a process-masked ROM cell, which requires ROM codes to be submitted to Texas Instruments for implementation in the device, as detailed in Appendix H.

At reset, the single-access RAM and the 512-word program/data (B0) RAM are not resident in program space. You can make the single-access RAM resident in program space by setting the RAM bit in the PMST register to 1. When the RAM bit is set, these RAM cells become addressable in program space. You can make the dual-access RAM block B0 resident in program space (0FE00h–0FFFFh) by setting the CNF bit to 1. The following code example maps these blocks into program space.

```
OPL    #010h,PMST    ;Map 'C5x single-access memory
                     ;in program space.
SETC   CNF           ;Map B0 to program space.
```

Table 6–1 through Table 6–3 show program memory configurations available on the 'C5x devices. Note that all addresses are specified in hexadecimal.

Table 6–1. 'C50 Program Memory Configuration Control

CNF	RAM	MP/$\overline{MC}$	ROM	SARAM	DARAM B0	Off-Chip
0	0	0	0000–07FF			0800–FFFF
0	0	1				0000–FFFF
0	1	0	0000–07FF	0800–2BFF		2C00–FFFF
0	1	1		0800–2BFF		0000–07FF
						2C00–FFFF
1	0	0	0000–07FF		FE00–FFFF	0800–FDFF
1	0	1			FE00–FFFF	0000–FDFF
1	1	0	0000–07FF	0800–2BFF	FE00–FFFF	2C00–FDFF
1	1	1		0800–2BFF	FE00–FFFF	0000–07FF
						2C00–FDFF

Table 6–2. 'C51 Program Memory Configuration Control

CNF	RAM	MP/$\overline{MC}$	ROM	SARAM	DARAM B0	Off-Chip
0	0	0	0000–1FFF			2000–FFFF
0	0	1				0000–FFFF
0	1	0	0000–1FFF	2000–23FF		2400–FFFF
0	1	1		2000–23FF		0000–1FFF
						2400–FFFF
1	0	0	0000–1FFF		FE00–FFFF	2000–FDFF
1	0	1			FE00–FFFF	0000–FDFF
1	1	0	0000–1FFF	2000–23FF	FE00–FFFF	2400–FDFF
1	1	1		2000–23FF	FE00–FFFF	0000–1FFF
						2400–FDFF

Table 6–3. 'C53 Program Memory Configuration Control

CNF	RAM	MP/$\overline{MC}$	ROM	SARAM	DARAM B0	Off-Chip
0	0	0	0000–3FFF			4000–FFFF
0	0	1				0000–FFFF
0	1	0	0000–3FFF	4000–4BFF		4C00–FFFF
0	1	1		4000–4BFF		0000–3FFF
						4000–FFFF
1	0	0	0000–3FFF		FE00–FFFF	2000–FDFF
1	0	1			FE00–FFFF	0000–FDFF
1	1	0	0000–3FFF	4000–4BFF	FE00–FFFF	4000–FDFF
1	1	1		4000–4BFF	FE00–FFFF	0000–1FFF
						2400–FDFF

6.2.2 Program Memory Address Map

The reset, interrupt, and trap vectors are addressed in program space. These vectors are soft—meaning that the processor, when taking the trap, loads the PC with the trap address and executes code at the vector location. Two words are reserved at each vector location for a branch instruction to the appropriate interrupt service routine. Table 6–4 shows the interrupt vector addresses after reset.

Table 6–4. 'C5x Interrupt Vector Addresses

Name	Location		Priority	Function
	Dec	Hex		
$\overline{RS}$	0	0	1 (highest)	External reset signal
$\overline{INT1}$	2	2	3	External user interrupt #1
$\overline{INT2}$	4	4	4	External user interrupt #2
$\overline{INT3}$	6	6	5	External user interrupt #3
TINT	8	8	6	Internal timer interrupt
RINT	10	A	7	Serial port receive interrupt
XINT	12	C	8	Serial port transmit interrupt
TRNT	14	E	9	TDM port receive interrupt
TXNT	16	10	10	TDM port transmit interrupt
$\overline{INT4}$	18	12	11	External user interrupt #4
—	20–33	14–21	N/A	Reserved
TRAP	34	22	N/A	Software trap instruction
NMI	36	24	2	Nonmaskable interrupt
—	38–41	26–29	N/A	Reserved for emulation and test
—	42–47	2A–2F	N/A	Software interrupts

At reset, these vectors are mapped absolutely to address 0h in program space. However, the vectors can be remapped to the beginning of any 2K-word page in program space after reset. This is done by loading the interrupt vector pointer (IPTR) bits in the PMST register with the appropriate 2K-word page boundary address. After loading IPTR, any user interrupt or trap vector is mapped to the new 2K-word page. For example:

```
OPL #05800h, PMST  ;Remap vectors to start at 5800h.
```

This example moves the interrupt vectors to off-chip program space at address 05800h. Any subsequent interrupt (except for a reset) will fetch its interrupt vector from that new location. For example, if, after loading the IPTR, an $\overline{INT2}$ occurs, the interrupt service routine vector will be fetched from location 5804h in program space as opposed to location 04h. This feature facilitates moving the desired vectors out of the boot ROM and then removing the ROM from the memory map. Once the system code is booted into the system from the boot-loader code resident in ROM, the application reloads the IPTR with a value pointing to the new vectors. In the above example, the OPL instruction is used to modify the PMST. This example assumes that the IPTR is currently set to 0s. If it is not,then it must be set to 0s before this instruction is executed; this assures that the correct value for IPTR is set.

The reset vector can not be remapped, because reset loads the IPTR with 0s. Therefore, the reset vector will always be fetched at location 0 in program memory. In addition, for the 'C51/'C53, 100 words are reserved in the on-chip ROM for device-testing purposes. Application code written to be implemented in on-chip ROM must reserve these 100 words at the top of the ROM addresses.

6.2.3 Program Memory Addressing

The program memory space contains the code for applications. It can also hold table information and immediate operands. The program memory is accessed only by the PAB address bus. The address for this bus is generated by the program counter (PC) when instructions and long immediate operands are accessed. The PAB address bus can also be loaded with long immediate, low accumulator, or registered addresses for block transfers, multiply/accumulates, and table read/writes.

The 'C5x devices fetch instructions by putting the PC on the PAB bus and reading the appropriate location in memory. While the read is executing, the PC is incremented for the next fetch. If there is a program address discontinuity (for example, branch, call, return, interrupt, or block repeat), the appropriate ad-

dress is loaded into the PC. The PC is also loaded when operands are fetched from program memory. Operands are fetched from program memory when the device reads or writes to tables (TBLR and TBLW), when it transfers data to/ from data space (BLPD and BLDP), or when it uses the program bus to fetch a second multiplicand (MAC, MACD, MADS, and MADD). The PC is loaded with a value other than PC + 1 in the following ways:

- Long immediate address with branch or call instructions.
- Long immediate address with MAC, MACD, BLDP or BLPD instructions.
- Low accumulator with BACC or CALA instructions.
- Low accumulator with TBLR or TBLW instruction.
- BMAR with MADS, MADD, BLDP or BLPD instructions.
- CALU with an interrupt vector address (INTR, TRAP, or NMI) instruction.
- CALU with PASR when at the end of a block repeat loop.
- Top of stack popped with a return instruction.

The address flow of a program can be traced externally through the address visibility feature. This feature can be used to debug during program development; it is enabled after reset and disabled/re-enabled by setting/clearing the AVIS bit in the PMST register. The address visibility mode sends the program address out to the address pins of the device, even when on-chip program memory is addressed. Note that the memory control signals ($\overline{PS}$, $\overline{RD}$, etc.) are not active in address visibility mode.

Instruction addresses can be externally clocked with the falling edge of the instruction acquisition ($\overline{IAQ}$) pin (see Appendix A for $\overline{IAQ}$ timings). These instruction addresses include both words of a two-word instruction but do not include block transfers, table reads, or multiply/accumulate operands. The address visibility mode also allows a specific interrupt trap to be decoded in conjunction with the interrupt acknowledge ($\overline{IACK}$) pin. While $\overline{IACK}$ is low, address pins A1–A4 can be decoded to identify which interrupt is being acknowledged (see Appendix A for $\overline{IACK}$ timings). Once the system is debugged, the address visibility mode can be disabled by setting the AVIS bit to one. Disabling the address visibility mode lowers the power consumption of the device and the RF noise of the system. Note that if the processor is running while $\overline{HOLDA}$ is active low (HM = 0), the address is not visible at the pins, regardless of the address visibility mode.

6.2.4 Program Memory Security Feature

The on-chip program memory can be secured on the 'C5x devices. This security feature does not allow an instruction fetched from off-chip memory to read or write on-chip program memory. The pipeline controller tracks instructions fetched from off-chip memory, and, if the operand address resides in on-chip

program space, the instruction reads invalid data off the bus. The limitations of the mode are as follows:

- Instructions fetched from off-chip memory cannot read or write on-chip single-access and read-only memory.
- Instructions fetched from B0 cannot read or write on-chip single-access and read-only program memory.
- Coefficients for off-chip multiply/accumulate instructions cannot reside in on-chip single-access and read-only program memory.
- The on-chip single-access memory cannot be mapped to data space.
- The emulator cannot work with on-chip program memory.
- The program memory address range that corresponds to the on-chip single-access RAM is not available for external memory.

This feature can be used with the on-chip ROM to secure program code that is stored in external memory. The ROM code can include a decryption algorithm that takes encrypted off-chip code, decrypts it, and stores the routine in on-chip single-access program RAM. This is a process-mask option and, like the ROM, must be submitted to Texas Instruments for implementation.

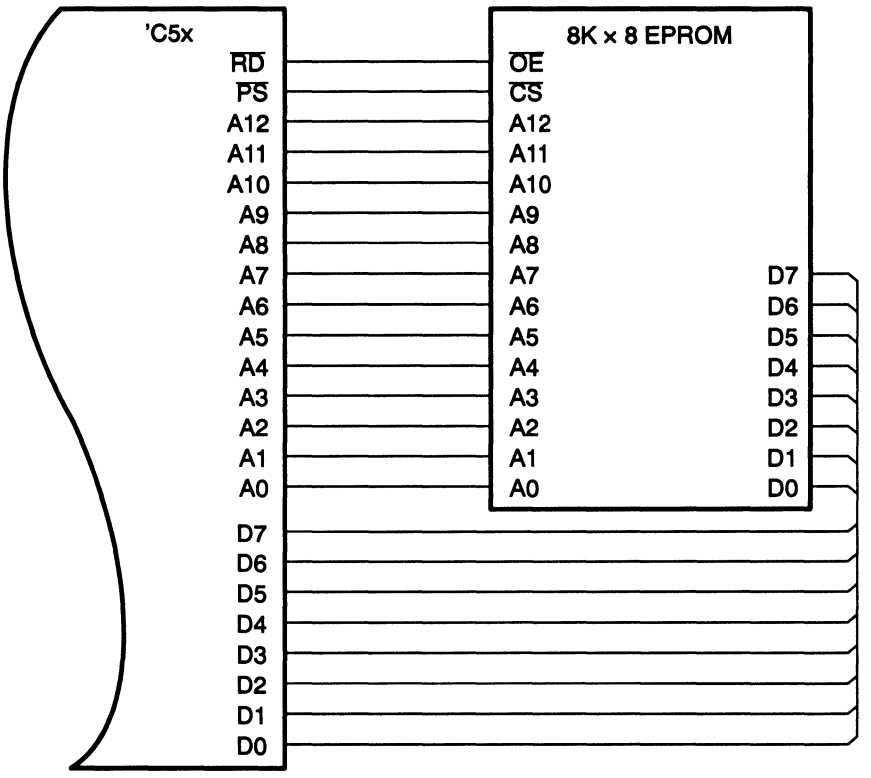
6.2.5 External Interfacing to Program Memory

The 'C5x devices can address up to 64K words of program memory off-chip. These are key signals for external memory interfacing:

A0–A15	16-Bit Bidirectional Address Bus
D0–D15	16-Bit Bidirectional Data Bus
$\overline{PS}$	Program Memory Select
$\overline{STRB}$	External Memory Access Active Strobe
$\overline{RD}$	Read Select (External Device Output Enable)
$\overline{WE}$	Write Enable
$\overline{IACK}$	Interrupt Acknowledge
READY	Memory Ready to Complete Cycle
$\overline{HOLD}$	Request for Control of Memory Interface
$\overline{HOLDA}$	Acknowledge $\overline{HOLD}$ Request
$\overline{BR}$	Bus Request
$\overline{IAQ}$	Acknowledge Bus Request (when $\overline{HOLDA}$ is low)

An example of a minimal external program memory interface is shown in Figure 6–4. In this figure, the 'C5x device interfaces to an 8K × 8 EPROM. The use of 8-bit-wide memories saves power, board space, and cost over 16-bit wide memory banks. The 16-bit-wide memory banks can be used with the same basic interface as the 8-bit-wide memories. Note that the 'C5x cannot directly execute code from 8-bit-wide memory. An on-chip program (such as a bootloader) is required to read 8-bit-wide memory to form 16-bit long instruction words and transfer them to on-chip RAM.

Figure 6–4. Interface to External EPROM



The program select ($\overline{PS}$) signal is connected directly to the chip select ($\overline{CS}$) to select the EPROM on any external program access. The EPROM is addressed in any 8K address block in program space. If multiple blocks of memory are to be interfaced in program space, a decode circuit that gates $\overline{PS}$ and the appropriate address bits can be used to drive the memory block chip selects.

The $\overline{RD}$ signal is tied directly to the output enable ($\overline{OE}$) pin of the EPROM. The $\overline{OE}$ signal enables the output drivers of the EPROM. The drivers are turned off in time to guarantee that no data bus conflicts occur with an external write by the 'C5x devices.

The device can be interfaced to external program RAM by connecting the $\overline{WE}$ signal to the write enable signal of the RAM device. The 'C5x devices take two cycles on all external writes, including a half cycle before the $\overline{WE}$ goes low and a half cycle after $\overline{WE}$ goes high; this prevents buffer conflicts on the external buses. Additional write cycles can be obtained by modifying the software wait-state generator registers. Subsection 6.3.4 includes an example of interfacing to external RAM.

6.3 Local Data Memory

The local data memory space on the 'C5x addresses up to 64K of 16-bit words. The 'C50, 'C51, and 'C53 have 9K, 1K and 3K words of on-chip single-access RAM (SARAM), respectively. All 'C5x devices have the same 1056 words of dual-access RAM (DARAM). These on-chip memory cells can be configured by software in or out of the local data address map. When these cells are mapped into data space, the device automatically accesses them when addressing within their bounds. When an address is generated outside these bounds, the device automatically generates an external access. The advantages of operating from on-chip memory are as follows:

1) Higher performance because no wait states are required.
2) Higher performance because of better flow within the pipeline of the CALU.
3) Lower cost than external memory.
4) Lower power than external memory.

The advantage of operating from off-chip memory is the ability to access a larger address space.

6.3.1 Local Data Space Configurability

The local data memory can reside both on and off chip. At reset, the configuration maps the 1056 words of dual-access RAM into local data space. Block B0 can be reconfigured into program space by setting the CNF bit in ST1 to 1. The single-access RAM can be mapped into data space by setting the OVLY bit to 1 in the PMST register. Table 6–5 the possible local data memory configurations available on the 'C50. Table 6–6 and Table 6–7 show the possible local data memory configurations available on the 'C51 and 'C53, respectively. Note that all locations in the address range, 0h–800h, that are not mapped into on-chip memory are on-chip reserved locations (80h–FFh and 500h–7FFh). Addresses 0–4Fh contain on-chip memory-mapped registers, and addresses 50–5Fh contain the memory-mapped I/O ports.

Table 6–5. 'C50 Local Data Memory Configuration Control

CNF	OVLY	DARAM B0	DARAM B1	DARAM B2	SARAM	Off-Chip
0	0	100h–2FFh	300h–4FFh	60h–7Fh		800h–FFFFh
0	1	100h–2FFh	300h–4FFh	60h–7Fh	800h–2BFFh	2C00h–FFFFh
1	0	–	300h–4FFh	60h–7Fh		800h–FFFFh
1	1	–	300h–4FFh	60h–7Fh	800h–2BFFh	2C00h–FFFFh

Table 6–6. 'C51 Local Data Memory Configuration Control

CNF	OVLY	DARAM B0	DARAM B1	DARAM B2	SARAM	Off-Chip
0	0	100h–2FFh	300h–4FFh	60h–7Fh		800h–FFFFh
0	1	100h–2FFh	300h–4FFh	60h–7Fh	800h–BFFh	C00h–FFFFh
1	0	–	300h–4FFh	60h–7Fh		800h–FFFFh
1	1	–	300h–4FFh	60h–7Fh	800h–BFFh	C00h–FFFFh

Table 6–7. 'C53 Local Data Memory Configuration Control

CNF	OVLY	DARAM B0	DARAM B1	DARAM B2	SARAM	Off-Chip
0	0	100h–2FFh	300h–4FFh	60h–7Fh		800h–FFFFh
0	1	100h–2FFh	300h–4FFh	60h–7Fh	800h–13FFh	1400h–FFFFh
1	0	–	300h–4FFh	60h–7Fh		800h–FFFFh
1	1	–	300h–4FFh	60h–7Fh	800h–13FFh	1400h–FFFFh

6.3.2 Local Data Memory Address Map

The 64K words of local data memory space include the memory-mapped registers for the device. The memory-mapped registers reside in data page 0. Data page 0 has five sections of register banks: core CPU registers, peripheral registers, test/emulation reserved area, I/O space port hole, and scratch-pad RAM.

- The 28 core CPU registers can be accessed with zero wait states. Some of these registers can be accessed through paths other than the data bus — for example, auxiliary registers can be loaded by the auxiliary register arithmetic unit (ARAU) by using the LAR instruction.
- The peripheral registers are the control and data registers used in the peripheral circuits. These registers reside on a dedicated peripheral bus structure called the TIBUS. They require one wait state when accessed.
- The test/emulation reserved area is used by the test and emulation systems for special information transfers. ***Writing to this area can cause the device to change its operational mode and, therefore, affect the operation of the application.***
- The I/O space port hole provides addressability to 16 words of I/O space within the data address space. This allows access to I/O space (other than IN and OUT instructions) via the more extensive addressing modes available within the data space. For example, the SAMM instruction can write to an I/O memory-mapped port as an OUT instruction does. The external interface looks as if an OUT instruction occurs ($\overline{IS}$ active). Port addresses reside off-chip and are subject to external wait states. They are also affected by the on-chip software wait-state generator, like any other nonmemory-mapped I/O port.
- The scratch-pad RAM block (B2) includes 32 words of dual-access RAM for variable storage without fragmenting the larger RAM blocks, both on

the device and external to the device. Table 6–8 shows the address map of data page 0.

Table 6–8. Data Page 0 Address Map

Name	Address		Description
	Dec	Hex	
Core Processor Memory-Mapped Registers			
——	0–3	0–3	Reserved
IMR	4	4	Interrupt Mask Register
GREG	5	5	Global Memory Allocation Register
IFR	6	6	Interrupt Flag Register
PMST	7	7	Processor Mode Status Register
RPTC	8	8	Repeat Counter Register
BRCR	9	9	Block Repeat Counter Register
PASR	10	A	Block Repeat Program Address Start Register
PAER	11	B	Block Repeat Program Address End Register
TREG0	12	C	Temporary Register Used for Multiplicand
TREG1	13	D	Temporary Register Used for Dynamic Shift Count (5 bits only)
TREG2	14	E	Temporary Register Used as Bit Pointer In Dynamic Bit Test (4 bits only)
DBMR	15	F	Dynamic Bit Manipulation Register
AR0	16	10	Auxiliary Register Zero
AR1	17	11	Auxiliary Register One
AR2	18	12	Auxiliary Register Two
AR3	19	13	Auxiliary Register Three
AR4	20	14	Auxiliary Register Four
AR5	21	15	Auxiliary Register Five
AR6	22	16	Auxiliary Register Six
AR7	23	17	Auxiliary Register Seven
INDX	24	18	Index Register
ARCR	25	19	Auxiliary Register Compare Register
CBSR1	26	1A	Circular Buffer 1 Start Register
CBER1	27	1B	Circular Buffer 1 End Register
CBSR2	28	1C	Circular Buffer 2 Start Register
CBER2	29	1D	Circular Buffer 2 End Register
CBCR	30	1E	Circular Buffer Control Register
BMAR	31	1F	Block Move Address Register
Peripheral Memory-Mapped Registers			
DRR	32	20	Data Receive Register
DXR	33	21	Data Transmit Register
SPC	34	22	Serial Port Control Register
——	35	23	Reserved

Table 6–8. Data Page 0 Address Map (Continued)

Name	Address		Description
	Dec	Hex	
Peripheral Memory-Mapped Registers (Continued)			
TIM	36	24	Timer Register
PRD	37	25	Period Register
TCR	38	26	Timer Control Register
—	39	27	Reserved
PDWSR	40	28	Program/Data S/W Wait-State Register
IOWSR	41	29	I/O Port S/W Wait-State Register
CWSR	42	2A	Control S/W Wait-State Register
—	43–47	2B–2F	Reserved for Test/Emulation
TRCV	48	30	TDM Data Receive Register
TDXR	49	31	TDM Data Transmit Register
TSPC	50	32	TDM Serial Port Control Register
TCSR	51	33	TDM Channel Select Register
TRTA	52	34	Receive/Transmit Address Register
TRAD	53	35	Received Address Register
—	54–79	36–4F	Reserved
Memory-Mapped I/O Ports			
PA0	80	50	I/O Port 80
PA1	81	51	I/O Port 81
PA2	82	52	I/O Port 82
PA3	83	53	I/O Port 83
PA4	84	54	I/O Port 84
PA5	85	55	I/O Port 85
PA6	86	56	I/O Port 86
PA7	87	57	I/O Port 87
PA8	88	58	I/O Port 88
PA9	89	59	I/O Port 89
PA10	90	5A	I/O Port 90
PA11	91	5B	I/O Port 91
PA12	92	5C	I/O Port 92
PA13	93	5D	I/O Port 93
PA14	94	5E	I/O Port 94
PA15	95	5F	I/O Port 95
B2	96–127	60–7F	Scratch Pad RAM

6.3.2.1 Auxiliary Register (AR0–AR7)

The eight 16-bit auxiliary registers (AR0–AR7) can be accessed by the CALU and modified by the ARAU or the PLU. The primary function of the auxiliary

registers is generating 16-bit addresses to data space. However, these registers can also act as general-purpose registers or counters. Subsection 6.3.3 describes how these registers are used in indirect addressing.

6.3.2.2 Auxiliary Register Compare Register (ARCR)

The auxiliary register compare register (ARCR) is a 16-bit register for address boundary comparison. The ARCR is compared to the selected AR by the CMPR instruction, and the result of the compare is placed in the TC bit of ST1. Subsection 6.3.3 describes how the ARCR can be used in memory management.

6.3.2.3 Index Register (INDX)

The index register (INDX) is used by the ARAU as a step value for indirect addressing modifications to auxiliary registers (i.e., addition or subtraction by more than 1). For example, when the ARAU steps across a row of a matrix, the indirect address is incremented by 1. However, when the ARAU steps down a column, the address is incremented by the dimension of the matrix. The ARAU can add or subtract the value stored in INDX from AR(ARP) as part of the indirect address operation. The INDX register is also used to map the dimension of the address block used for bit-reversal addressing. Subsection 6.3.3 describes how INDX can be used in memory management.

6.3.2.4 Circular Buffer Registers (CBSR1, CBER1, CBSR2, CBER2, CBCR)

The 'C5x devices support two concurrent circular buffers operating in conjunction with user-specified auxiliary registers. Two circular buffer start registers (CBSR1 and CBSR2) indicate the 16-bit address where the circular buffer starts. Two circular buffer end registers (CBER1 and CBER2) indicate the end of the circular buffers. The circular buffer control register (CBCR) controls the operation of these circular buffers. Subsection 6.3.3 describes how circular buffers can be used in memory management.

6.3.2.5 Block Move Address Register (BMAR)

The 16-bit block move address register (BMAR) holds an address value for use with block moves and multiple/accumulate operations. This register provides 16-bit address to a second indirect-addressed operand for these operations. The use of the BMAR is described further in subsection 6.3.3.

6.3.2.6 Repeat Registers (RPTC, BRCR, PASR, and PAER)

The repeat counter (RPTC) holds the repeat count in a repeat single-instruction operation. This register is loaded by the RPT and RPTZ instructions.

CAUTION

The RPTC register is a memory-mapped register. However, you should avoid writing to this register. Writing to this register can cause undesired results.

The block repeat counter register (BRCR) holds the count value for the block repeat feature. This value is loaded before a block repeat operation is initiated. It can be changed while a block repeat is in progress; however, take caution in this case to avoid infinite loops. The program address start register (PASR) holds the start address of the block of code to be repeated. The program address end register (PAER) holds the end address of the block of code to be repeated. Both these registers are loaded by the RPTB instruction. Block repeats are described in more detail in subsection 3.6.5.

6.3.2.7 Interrupt Registers (IMR,IFR)

The interrupt mask register (IMR) is used to individually mask off specific interrupts at required times. The interrupt flag register (IFR) indicates the current status of the interrupts. Interrupts are described in detail in Section 3.8.

6.3.2.8 Global Memory Allocation Register (GREG)

The global memory allocation register (GREG) is used to allocate parts of the data address space as global memory. This register defines what amount of the local data space will be overlayed by global data space. The operation of GREG is further discussed in Section 6.4.

6.3.2.9 Dynamic Bit Manipulation Register (DBMR)

The dynamic bit manipulation register (DBMR) is used in conjunction with the PLU to provide a dynamic (execution time programmable) mask register. The use of this register is described in Section 3.7.

6.3.2.10 Temporary Registers (TREG0, TREG1, TREG2)

TREG0 holds one of the multiplicands of the multiplier. It can also be loaded via the CALU with the following instructions: LT, LTA, LTD, LTP, LTS, SQRA, SQRS, MAC, MACD, MADS, and MADD. TREG1 holds a dynamic (execution-time programmable) shift count for the prescaling shifter. TREG2 holds a dynamic bit address for the BITT instruction.

6.3.2.11 Processor Mode Status Register (PMST)

The processor mode status register (PMST) controls memory configurations of the 'C5x devices (with exception of the CNF bit in ST1). The PMST register is described in more detail in subsection 3.6.3 and in the configurability sections of Chapter 6.

6.3.2.12 Serial Port Registers (DRR, DXR, SPC)

Three registers control and operate the serial port. The serial port control register (SPC) contains the mode control and status bits of the serial port. The data receive register (DRR) holds the incoming serial data, and the data transmit register (DXR) holds the outgoing serial data. The serial port is described in more detail in Section 5.4.

6.3.2.13 TDM Serial Port Registers (TRCV, TDXR, TSPC, TCSR, TRTA, TRAD)

The TDM serial port is a feature superset of the first serial port. The TDM serial port supports applications that require serial communication in a multiprocessing environment. The TDM serial port is described in more detail in Section 5.4.

6.3.2.14 Timer Registers (TIM, PRD, TCR)

The timer operates with three registers. The TIM register is the current count of the timer. The PRD register defines the period for the timer. The TCR (timer control register) controls the operations of the timer. Refer to Section 5.6 for more details on the timer.

6.3.2.15 Software Wait-State Registers (PDWSR, IOWSR, CWSR)

The software wait-state registers contain the wait-state counts for the different banks of off-chip memory address ranges. PDWSR contains the wait-state count for the four 16K blocks of program and data memory. IOWSR contains the wait-state counts for the 16 partitions of I/O space. The CWSR control register determines the range of wait states you may select—(0, 1, 2, or 3) or (0, 1, 3, 7). In addition, the BIG bit in the CWSR register determines how the I/O space is partitioned. If BIG is set to **0**, the I/O wait states apply to the pair of port addresses. If the BIG bit is set to **1**, the I/O wait states apply to 8K blocks of the I/O space. Refer to Section 5.3 for more details on software wait states.

6.3.2.16 I/O Space Port Hole (PA0–15)

The I/O space port hole allows the addressing of sixteen locations (50h–5Fh) of I/O space via the addressing modes of the local data space. This means that these locations can be read directly into the CALU or written from the ACC. It also means that these locations can be acted upon by the PLU or addressed via the memory-mapped addressing mode. The locations can also be addressed with the IN and OUT instructions.

6.3.2.17 Scratch Pad RAM

This 32-word block of RAM can be used to hold overhead variables so that the larger blocks of RAM are not fragmented. This RAM block supports dual-ac-

cess operations and can be addressed by using the memory-mapped addressing mode or any data memory addressing mode.

6.3.3 Local Data Memory Addressing

The local data space address generation is controlled by the decode of the current instruction. Local data memory is read via data address bus 1 (DAB1) on instructions with only one data memory operand and program address bus (PAB) on instructions with a second data memory operand. An instruction operand is provided to the CALU in eight ways, as described in subsection 3.4.2. However, data memory addresses are generated in one of the following five ways:

- By the direct address bus (DAB) using the direct addressing mode (for example, ADD 010h) relative to the data page pointer (DP),
- By the direct address bus (DAB) using the memory-mapped addressing mode (for example, LAMM PMST) within data page zero,
- By the auxiliary register file bus (AFB) using the indirect addressing mode (for example, ADD *),
- By the value pointed at by the PC in long immediate address mode (for example, BLDD TBL1,*+), and
- By the block memory address register (BMAR) in registered block memory addressing mode (for example, BLDD, BMAR*+).

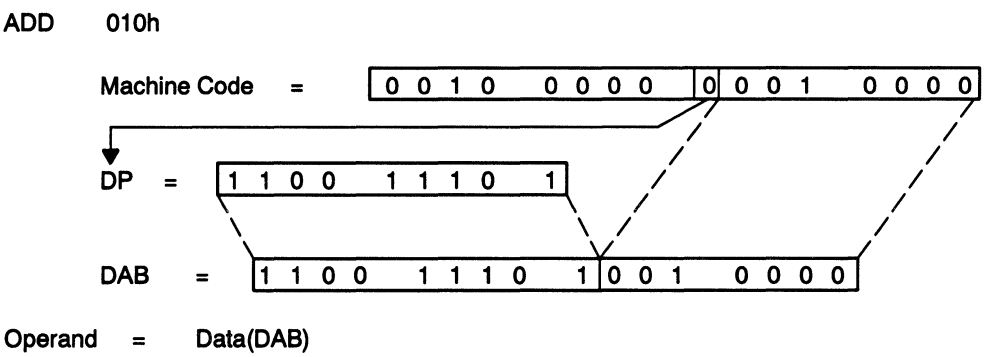
In the direct addressing mode, the 9-bit data memory page pointer (DP) points to one of 512 pages (1 page=128 words). The data memory address (dma), specified by the seven LSBs of the instruction, points to the desired word within the page. The address on the DAB is formed by concatenating the 9-bit DP with the 7-bit dma.

Figure 6–5 illustrates the direct addressing mode. In the illustration, the operand is fetched from data memory space via the data bus, and the address is the concatenated value of the DP and the seven LSBs of the instruction. For the following example, consider DP = 018Dh and TEMP1 = 010h:

```
LACC   TEMP1      ;ACC = TEMP1.
```

In the example, the accumulator is loaded with DATA(CE80).

Figure 6–5. Direct Addressing Mode



Note: DAB is the 16-bit internal address bus for data memory.

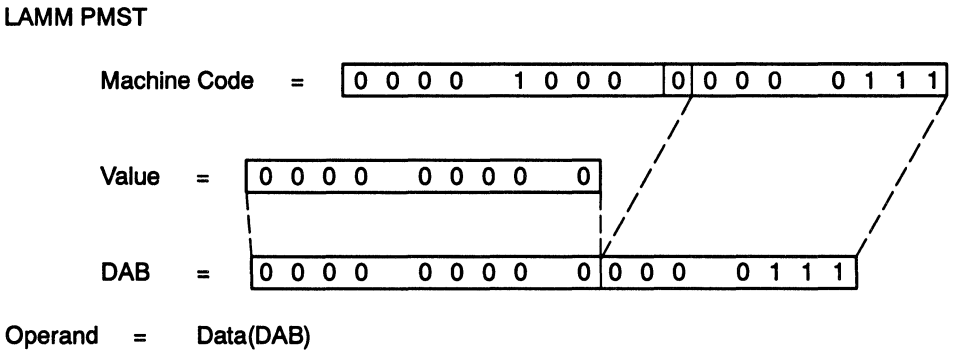
The memory-mapped addressing mode operates much like the direct addressing mode except that the most significant 9 bits of the address are forced to zero instead of being loaded with the contents of the DP. This makes it possible to address the memory-mapped registers of data page zero directly without the overhead of changing the DP or auxiliary register.

Figure 6–6 illustrates memory-mapped addressing mode. For the following example, consider DP = 0184h and TEMP1 = 08060h:

```
LAMM   07h          ;ACC = PMST
```

In this example, the contents of memory location 7h is loaded into the accumulator.

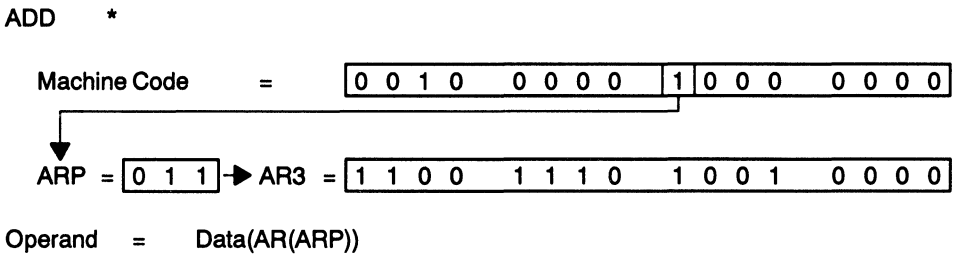
Figure 6–6. Memory-Mapped Addressing Mode



In the indirect addressing mode, the currently selected 16-bit auxiliary register AR(ARP) addresses the data memory through the AFB. While the selected auxiliary register provides the data memory address and the data is being manipulated by the CALU, the contents of the auxiliary register can be manipu-

lated through the ARAU. See Figure 6–7 for an example of indirect auxiliary register addressing. In this case, AR3 is the selected auxiliary register (ARP=3).

Figure 6–7. Indirect Addressing Mode

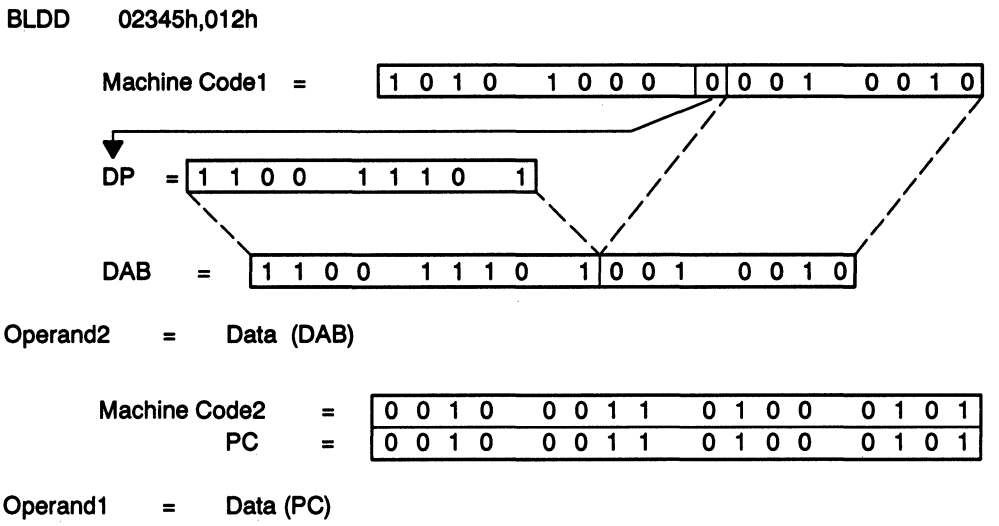


The following code illustrates the use of indirect addressing in a program:

```
* This routine uses indirect addressing to calculate the following equation:
*
*                  10
*                 ____
*                 \      X(I) x Y(I)
*                 /
*                 ____
*                 I = 1
*
* The routine assumes that the X values are located in on-chip RAM block B0,
* and the Y values in block B1. The efficiency of the routine is due to the
* use of indirect addressing and the repeat instruction.
*
SERIES MAR   *,AR4        ;ARP POINTS TO ADDRESS REGISTER 4.
       SETC  CNF          ;CONFIGURE BLOCK B0 AS PROGRAM MEMORY.
       LAR   AR4,#0300h   ;POINT AT BEGINNING OF DATA MEMORY.
       RPTZ  #9           ;CLEAR ACC AND P; REPEAT NEXT INST. 10 TIMES
       MAC   0FF00h,*+    ;MULTIPLY AND ACCUMULATE; INCREMENT AR4.
             APAC         ;ACCUMULATE LAST PRODUCT.
             RET          ;Accumulator contains result.
```

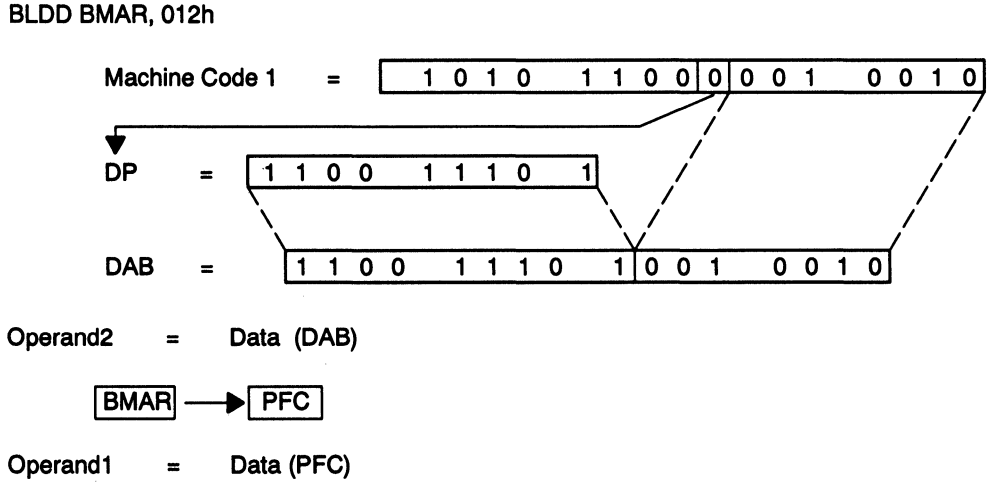
In the long immediate addressing mode, an operand is addressed by the second word of a two-word instruction. In this case, the program address/data bus (PAB) is used for the operand fetch. The prefetch counter (PFC) is pushed onto the microcall stack (MCS), and the long immediate value is loaded into the PFC. The PAB is then used for the operand fetch or write. At the completion of the instruction, the MCS is popped back to the PFC. The PC is incremented by two, and execution continues. This technique is used when two memory addresses are required for the execution of the instruction. The PFC is used so that when the instruction is repeated, the address generated can be autoincremented. Figure 6–8 illustrates this mode. In this illustration, the source address (OPERAND1) is fetched via PAB, and the destination address (OPERAND2) uses the direct addressing mode.

Figure 6–8. Long Immediate Addressing Mode



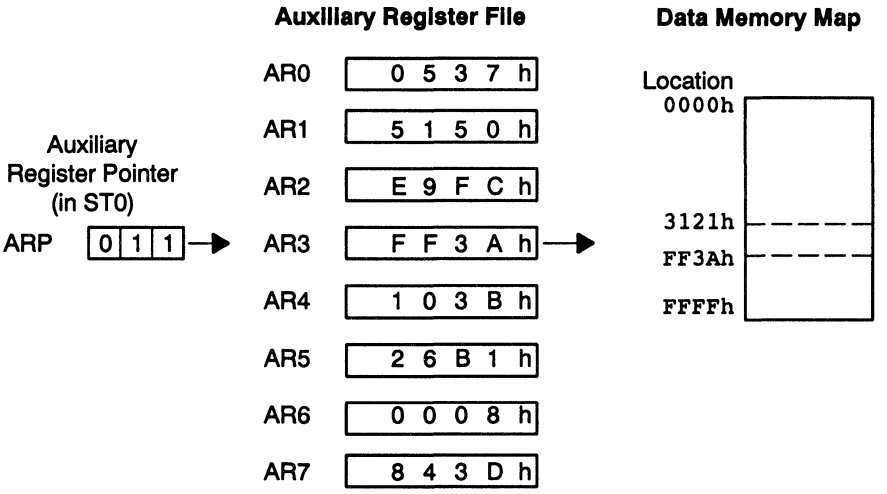
The registered block memory addressing mode operates like the long immediate addressing mode with the exception that the address comes from the BMAR register. The advantage of this technique over long immediate addressing is that it allows the address of the block of memory to be changed in runtime. On the other hand, the address in long immediate addressing mode resides in the program flow and cannot be easily changed. Figure 6–9 shows an example of registered block memory addressing mode.

Figure 6–9. Registered Block Memory Addressing Mode



'C5x devices provide a register file containing eight auxiliary registers (AR0–AR7). The auxiliary registers can be used for indirect addressing of the data memory or for temporary data storage. Indirect auxiliary register addressing (see Figure 6–10) allows placement of the data memory address of an instruction operand into one of the auxiliary registers. These registers are pointed to by a three-bit auxiliary register pointer (ARP) that is loaded with a value from 0 through 7, designating AR0 through AR7, respectively.

Figure 6–10. Indirect Auxiliary Register Addressing Example



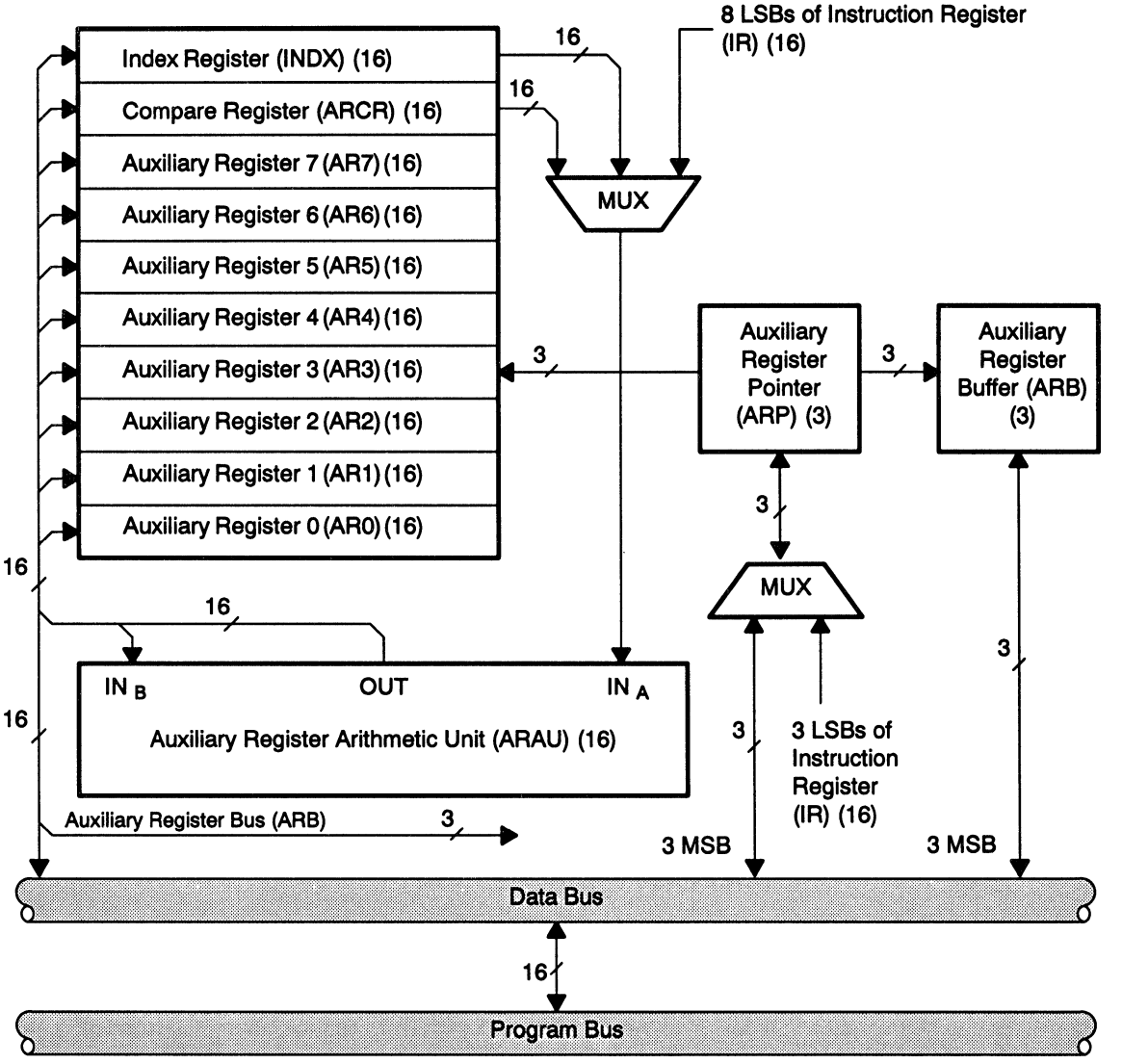
The auxiliary registers and the ARP can be updated directly from data memory, the accumulator, or the product register, or by an immediate operand defined in the instruction. The contents of these registers can also be stored in data memory or used as inputs to the CALU. These registers appear in the memory map as described in Table 6–8.

The auxiliary register file (AR0–AR7) is connected to the auxiliary register arithmetic unit (ARAU), shown in Figure 6–11. The ARAU can autoindex the current auxiliary register while the data memory location is being addressed. Indexing either by ± 1 or by the contents of the INDX register can be performed. As a result, accessing tables of information does not require the central arithmetic logic unit (CALU) for address manipulation. The CALU can perform other operations in parallel.

If more advanced address manipulation is required, such as multidimensional array addressing, the CALU can directly read from or write to the auxiliary registers. However, the ARAU updates of the ARs is done during the decode phase (second cycle) of the pipeline, whereas the CALU writes during the execution phase (fourth cycle) of the pipeline. Therefore, the two instructions di-

rectly following the CALU write to an auxiliary register should not use the same auxiliary register for address generation.

Figure 6–11. Auxiliary Register File



As shown in Figure 6–11, the index register, the compare register, or the eight LSBs of the instruction register can be connected to one of the inputs of the ARAU. The other input is fed by the current AR (being pointed to by ARP). AR(ARP) refers to the contents of the current AR pointed to by ARP. The ARAU performs the functions shown in Figure 6–12.

Figure 6–12. ARAU Functions

Function	Description
AR(ARP) + INDX → AR(ARP)	Index the current AR by adding a 16-bit unsigned integer contained in INDX. Example: ADD *0+.
AR(ARP) – INDX → AR(ARP)	Index the current AR by subtracting a 16-bit unsigned integer contained in INDX. Example: ADD *0–.
AR(ARP) + 1 → AR(ARP)	Increment the current AR by one. Example: ADD *+.
AR(ARP) – 1 → AR(ARP)	Decrement the current AR by one. Example: ADD *–.
AR(ARP) → AR(ARP)	Do not modify the current AR. Example: ADD *.
AR(ARP) + IR(7–0) → AR(ARP)	Add an 8-bit immediate value to current AR. Example: ADRK #055h.
AR(ARP) – IR(7–0) → AR(ARP)	Subtract an 8-bit immediate value from current AR. Example: SBRK #055h.
AR(ARP) + rc(INDX) → AR(ARP)	Bit-reverse indexing; add INDX with reverse-carry (rc) propagation. Example: ADD *BR0+.
AR(ARP) – rc(INDX) → AR(ARP)	Bit-reverse indexing; subtract INDX with reverse-carry (rc) propagation. Example: ADD *BR0–.

If (AR(ARP) == ARCR), then TC = 1
If (AR(ARP) < ARCR), then TC = 1
If(AR(ARP) > ARCR), then TC = 1
If(AR(ARP) ≠ ARCR), then TC = 1
	Compare current AR with ARCR and if condition is true, then set TC bit of the status register (ST1) to one. If false, then clear TC. Example: CMPR 3.

If (AR(ARP) = CBER), then AR(ARP) =CBSR	If at end of circular buffer, reload start address.

The index register (INDX) can be added to or subtracted from AR(ARP) on any AR update cycle. This 16-bit register is one of the memory-mapped registers and is used to increment or decrement the address in steps larger than one for operations such as addressing down a column of a matrix. The auxiliary register compare register (ARCR) is used as a limit to blocks of data and, in conjunction with the CMPR instruction, supports logical comparisons between AR(ARP) and ARCR. The 'C2x devices use AR0 for these two functions. Upon reset, a LAR load of AR0 also loads INDX and ARCR to maintain compatibility with the 'C2x devices. To avoid loading the INDX and ARCR registers on an AR0 load, the NDX bit of the PMST register is set to one. For the following example, assume INDX = 010h, ARP = 3, and AR3 = 0200h:

```
ADD*0+,4,AR5 ;ACC += addressed value shifted left 4.
```

In the example, DATA(200) is shifted left 4 bits and added to the ACC, AR3 is incremented by 10h, and ARP is changed to 5.

The 'C5x supports two circular buffers operating at a given time. These two circular buffers are controlled via the circular buffer control register (CBCR). The CBCR is defined in Table 6–9.

Table 6–9. Circular Buffer Control Register

Bit	Name	Function
0–2	CAR1	Identifies which auxiliary register is mapped to circular buffer 1
3	CENB1	Circular buffer 1 enable=1/disable=0. Set to 0 upon reset
4–6	CAR2	Identifies which auxiliary register is mapped to circular buffer 2
7	CENB2	Circular buffer 2 enable=1/disable=0. Set to 0 upon reset

Upon reset ($\overline{RS}$ rising edge), both circular buffers are disabled. To define a circular buffer, load the CBSR1/2 with the start address of the buffer and CBER1/2 with the end address. Load the auxiliary register to be used with the buffer with an address between the start and the end, load CBCR with the appropriate auxiliary register number, and set the enable bit. As the address is stepping through the circular buffer, the update is compared against the value contained in CBER1/2. When those values are equal and any AR modification occurs, the value contained in CBSR1/2 is automatically loaded into the AR. For the following example, assume CBSR1 = 0200h, CBER1 = 0203h, CBCR = 0Ch, AR4 = 0203h, and ARP = 4:

```
ADD*+ ;ACC += addressed value at 203h.
```

At the completion of the instruction, AR4 = 0200h.

Circular buffers can be used with either increment- or decrement-type updates. If increment updates are used, then the value in CBER must be greater than the value in CBSR. If decrement updates are used, the value in CBER must be less than the value in CBSR. The other indirect addressing modes may also be used; however, the ARAU tests only for the condition AR(ARP)=CBER. The ARAU will not wrap around if an AR update steps over the value contained in CBER. Note that the test in the ARAU is performed before the auxiliary register update. Refer to subsection 4.1.6 for details.

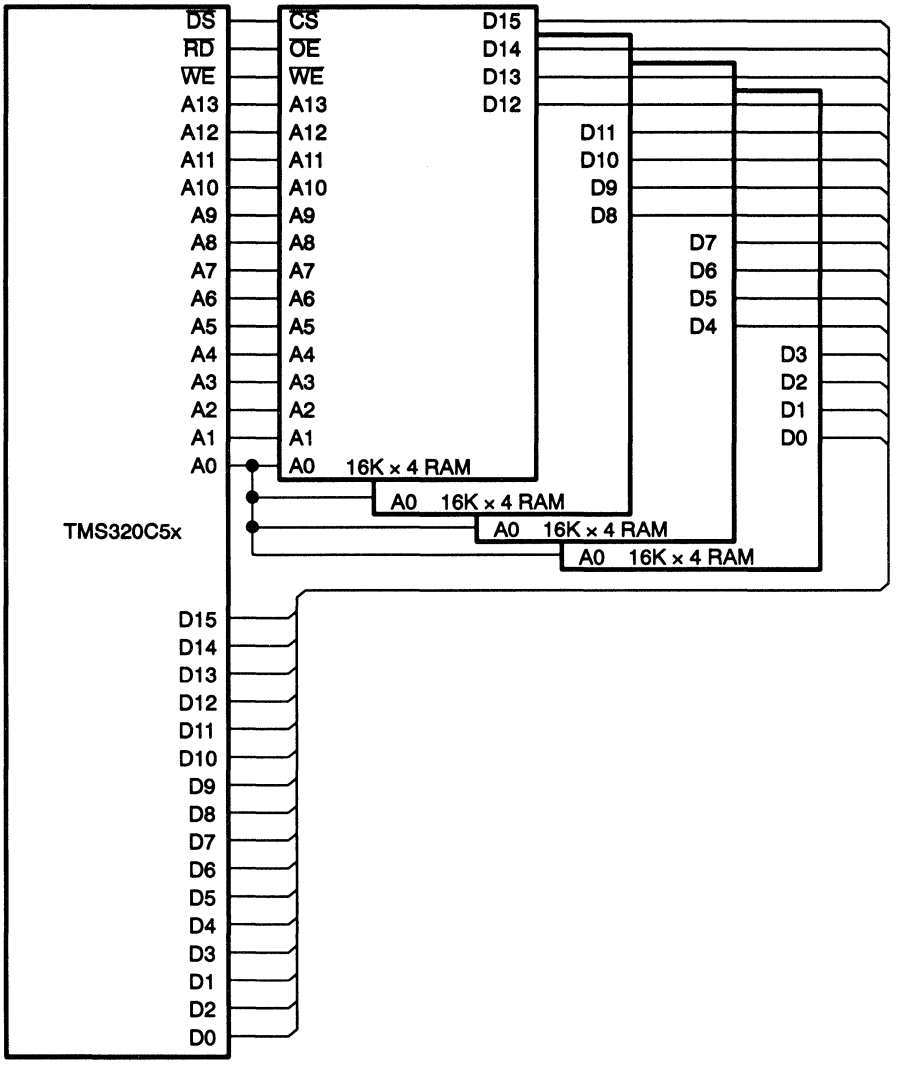
6.3.4 External Interfacing to Local Data Memory

The 'C5x devices can address up to 64K words of off-chip local data memory. These are the key signals for this interface:

A0–A15	16-Bit Bidirectional Address Bus
D0–D15	16-Bit Bidirectional Data Bus
$\overline{DS}$	Data Memory Select
$\overline{STRB}$	External Memory Access Active Strobe
$\overline{RD}$	Read Select (External Device Output Enable)
$\overline{WE}$	Write Enable
READY	Memory Ready to Complete Cycle
$\overline{HOLD}$	Request for Control of Memory Interface
$\overline{HOLDA}$	Acknowledge $\overline{HOLD}$ Request
$\overline{BR}$	Bus Request
$\overline{IAQ}$	Acknowledge Bus Request (when $\overline{HOLDA}$ is low)

An example of an external RAM interface is shown in Figure 6–13. In this figure, the 'C5x device interfaces to four 16K × 4-bit RAM devices. The data memory select ($\overline{DS}$) is directly connected to the chip select ($\overline{CS}$) of the devices. This means the external RAM block will be addressed in any of the four 16K banks of local data space. If there are additional banks of off-chip data memory, a decode circuit that gates $\overline{DS}$ with the appropriate address bits can be used to drive the memory block chip select.

Figure 6–13. Interface to External RAM



The $\overline{RD}$ signal is tied directly to the output enable ($\overline{OE}$) pin of the RAMs. This signal enables the output drivers of the RAM and turns them off in time to prevent data bus conflicts with an external write by the 'C5x device. If the RAM device does not have an $\overline{OE}$ pin, then $\overline{DS}$ should be gated with $\overline{STRB}$ and connected to the $\overline{CS}$ pin of the RAM to implement the same function. The $\overline{WE}$ signal of the 'C5x is tied to the $\overline{WE}$ signal of the RAM. The 'C5x takes at least two cycles on all external writes, including a half cycle before the $\overline{WE}$ goes low and a half cycle after $\overline{WE}$ goes high; this prevents buffer conflicts on the external buses. Additional wait states can be generated with the software wait-state generators.

6.4 Global Memory

For multiprocessing applications, the 'C5x devices are capable of allocating global data memory space and communicating with that space via the $\overline{BR}$ (bus request) and READY control signals. In addition, this capability can be used to extend the data memory address map by overlaying the address space.

Global memory is memory shared by more than one processor. Therefore, access to it must be arbitrated. When global memory is used, the processor's address space is divided into local and global sections. The local section is used by the processor to perform its individual function, and the global section is used to communicate with other processors. This implementation facilitates shared data multiprocessing in which data is transferred between two or more processors. Unlike a direct memory access (DMA) between two processors, reading or writing global memory does not require that one of the processors be halted.

6.4.1 Global Memory Configurability

A memory-mapped global memory allocation register (GREG) specifies part of the 'C5x data memory as global external memory. The register, GREG, memory-mapped to data memory address location 5h, is an eight-bit register connected to the eight LSBs of the internal data bus. The upper eight bits of location 5 are nonexistent and are read as ones.

The contents of GREG determine the size of the global memory space between 256 and 32K words. The legal values of GREG and corresponding global memory spaces are shown in Table 6–10. Note that values other than those listed in the table lead to fragmented memory maps and should be avoided.

Table 6–10. Global Data Memory Configurations

GREG Value	Local Memory		Global Memory	
	Range	# Words	Range	# Words
000000XX	0h–0FFFFh	65,536	–	0
10000000	0h–07FFFh	32,768	08000h–0FFFFh	32,768
11000000	0h–0BFFFh	49,152	0C000h–0FFFFh	16,384
11100000	0h–0DFFFh	57,344	0E000h–0FFFFh	8,192
11110000	0h–0EFFFh	61,440	0F000h–0FFFFh	4,096
11111000	0h–0F7FFh	63,488	0F800h–0FFFFh	2,048
11111100	0h–0FBFFh	64,512	0FC00h–0FFFFh	1,024
11111110	0h–0FDFFh	65,024	0FE00h–0FFFFh	512
11111111	0h–0FEFFh	65,280	0FF00h–0FFFFh	256

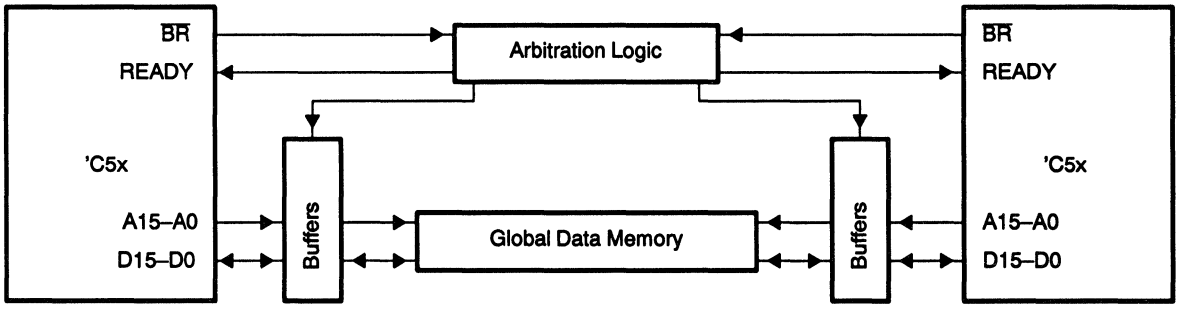
6.4.2 Global Memory Addressing

When a data memory address, either direct or indirect, corresponds to a global data memory address (as defined by GREG), $\overline{BR}$ is asserted low with $\overline{DS}$ to indicate that the processor wishes to make a global memory access. External logic then arbitrates for control of the global memory, asserting READY when the 'C5x device has control. The length of the memory cycle is controlled by the READY signal. In addition, the software wait-state generators can be used to extend the access times for slower, external memories. The wait-state generators corresponding to the overlapped memory address space in local data space will generate the wait states for the corresponding addresses in global data memory space.

6.4.3 External Interfacing of Global Memory

Global memory can be used in various digital signal processing tasks, such as filters or modems, where the algorithm being implemented may be divided into sections with a distinct processor dedicated to each section. With multiple processors dedicated to distinct sections of the algorithm, throughput may be increased via pipelined execution. Figure 6–14 illustrates an example of a global memory interface. Since the processors can be synchronized by using the $\overline{RS}$ pin, the arbitration logic may be simplified and the address and data bus transfers made more efficient.

Figure 6–14. Global Memory Interface



The global memory interface can also be used to extend the data memory address map beyond the reach of the 16-bit address bus by paging in an additional 32K words. Loading the GREG register with the appropriate value can overlay the local data memory with additional memory, starting at the highest memory address (0FFFFh) and moving down. This additional memory is differentiated from local memory accesses by the $\overline{BR}$ pin being active low. The rest of the memory interface control signals ($\overline{STRB}$, $\overline{DS}$, etc.) behave identically on a local or global data access.

6.5 Input/Output Space

The 'C5x devices support an I/O address space of 64K 16-bit parallel input and output ports. I/O ports allow access to peripherals typically used in DSP applications such as codecs, digital-to-analog (D/A) converters, and analog-to-digital (A/D) converters. This section discusses addressing I/O ports and interfacing I/O ports to external devices.

6.5.1 Addressing Input/Output Ports

Access to external parallel I/O ports is multiplexed over the same address and data bus for program/data memory accesses. I/O space access is distinguished from program/data memory accesses by the $\overline{IS}$ signal going active low. All 65,536 ports can be accessed via the IN and OUT instructions, as shown in the following example:

```
IN  DAT7,0FFFEh ;Read data to data memory from external
                ;device on port 65534.
OUT DAT7,0FFFFh ;Write data from data memory to external
                ;device on port 65535.
```

Sixteen of the 64K I/O ports are mapped in data memory space as shown in Table 6–4. The I/O ports may be accessed with the IN and OUT instructions along with any instruction that reads or writes a location in data space. In this way, I/O is treated the same way as memory. The following example illustrates the use of direct addressing to access an I/O device on port 51h:

```
SACL  51h    ;(DP = 0) Store accumulator to external
             ;device on port 81.
```

Accesses to memory-mapped I/O space are also distinguished from program/data accesses by the $\overline{IS}$ signal. $\overline{DS}$ is not active, even though the user is writing to data space.

6.5.2 Interfacing to I/O Ports

The $\overline{RD}$ and $\overline{WE}$ signals can be used along with chip-select logic to output data to an external device. The port address can be decoded and used as a chip select for the input or output device. The access times to I/O ports can be modified through the CWSR and IOWSR software wait-state registers. The BIG bit in the CWSR register determines how the I/O space is mapped to the software control registers. If the BIG bit is set to 0 in the CWSR register, the first sixteen ports are assigned in pairs to a software wait-state generator. Each following set of 16 registers maps accordingly to the first 16 ports when BIG = 0. For example, the 16 ports that correspond to the addresses in the data space port hole (ports 50h–5Fh) have the same wait states as ports 0–Fh. If the BIG

bit is set to 1, the wait states are mapped to program space in eight 8K blocks of memory. The following table shows how the software wait states are assigned to I/O ports according to the BIG bit:

I/O Ports When IWSR Bits	I/O Ports When BIG=0	BIG=1
0–1	Port 0/Port 1	Ports 0000h–1FFFh
2–3	Port 2/Port 3	Ports 2000h–3FFFh
4–5	Port 4/Port 5	Ports 4000h–5FFFh
6–7	Port 6/Port 7	Ports 6000h–7FFFh
8–9	Port 8/Port 9	Ports 8000h–9FFFh
10–11	Port 10/Port 11	Ports A000h–BFFFh
12–13	Port 12/Port 13	Ports C000h–DFFFh
14–15	Port 14/Port 15	Ports E000h–FFFFh

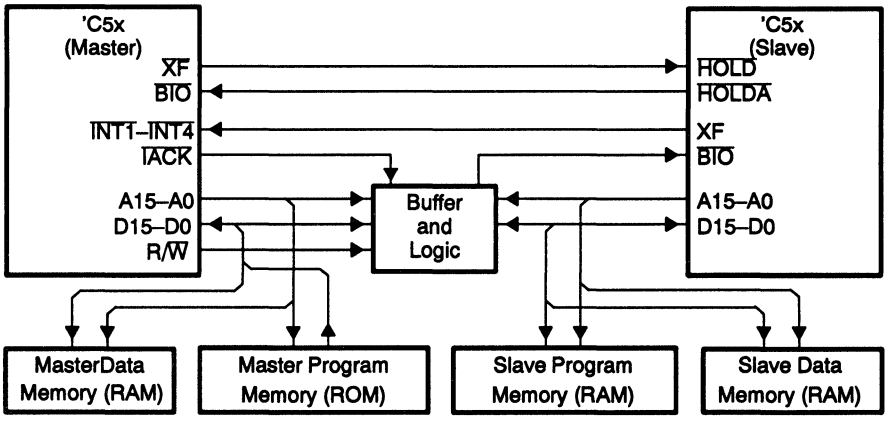
See Section 5.3 for details.

6.6 Direct Memory Access (DMA)

The 'C5x supports multiprocessing designs using direct memory access (DMA) of external memory or the 'C5x on-chip single access RAM. The DMA feature can be used for multiprocessing by temporarily halting the execution of one or more processors to allow another processor to read from or write to the 'C5x's local off-chip memory or on-chip single-access RAM. You can control the external memory access via the HOLD/HOLDA signals. The DMA access of internal RAM on the 'C5x is controlled by the HOLD, HOLDA, R/W, STRB, BR, and IAQ lines.

The multiprocessing is typically a master-slave configuration. The master may initialize a slave by downloading a program into its program memory space and/or may provide the slave with the necessary data by using external memory to complete a task. In a typical 'C5x direct memory access scheme, the master may be a general-purpose CPU, another 'C5x, or even an analog-to-digital converter. A simple 'C5x master-slave configuration is shown in Figure 6–15.

Figure 6–15. Direct Memory Access Using a Master-Slave Configuration



The master 'C5x device takes complete control of the slave's external memory by asserting HOLD low via its external flag (XF). This causes the slave to place its address, data , and control lines in a high-impedance state.

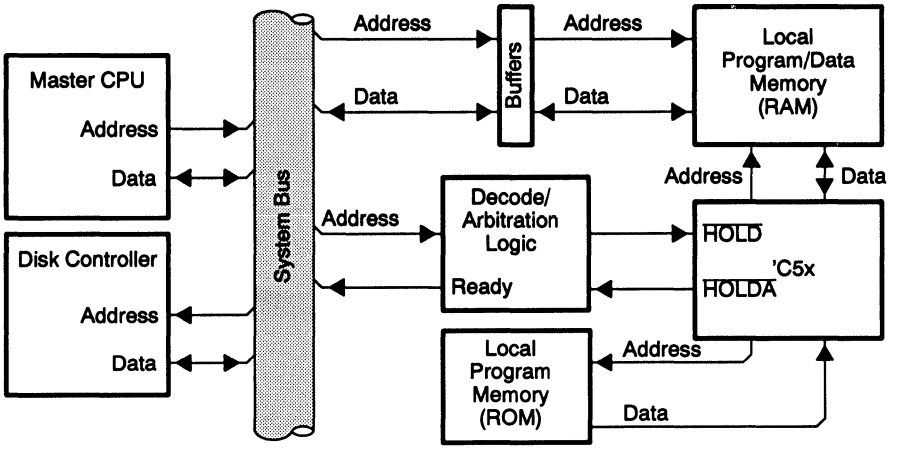
After control of the slave's buses is given up to the master processor, the slave alerts the master of the fact by asserting HOLDA. This signal may be tied to the master 'C5x BIO pin. The slave's XF pin may be used to indicate to the master when it has finished performing its task and needs to be reprogrammed or requires additional data to continue processing. In a multiple-slave configuration, priority of each slave's task may be determined by tying the slave's XF

signals to the appropriate $\overline{\text{INT}}$(4, 3, 2, or 1) pin on the master 'C5x device. The external bus interface of the slave 'C5x device is put in high-impedance mode when its $\overline{\text{HOLDA}}$ signal is asserted. While the $\overline{\text{HOLDA}}$ is active, the processor can continue running code out of its internal memory (internal ROM or single/ dual access RAM) if it is in concurrent hold mode (status bit HM is 0). However, $\overline{\text{IAQ}}$ pin does not indicate instruction acquisition, once $\overline{\text{HOLDA}}$ goes active. Otherwise, the processor will halt internal execution (status bit HM is 1). See Section 3.8 for interaction between $\overline{\text{HOLD}}$ $\overline{\text{RS}}$, and external interrupts.

A PC environment presents another example of a potential direct memory access scheme in which a system bus (the PC bus) is used for data transfer to external 'C5x memory. In this configuration, either the master CPU or a disk controller may place data onto the system bus, which can be downloaded into the local memory of the 'C5x device. In this case, the 'C5x acts more like a peripheral processor with multifunction capability. In a speech application, for example, the master can load the 'C5x program memory with algorithms to perform such tasks as speech analysis, synthesis, or recognition, and can fill the 'C5x data memory with the required speech templates. In another application example, the 'C5x can serve as a dedicated graphics engine. Programs can be downloaded via the system bus into program RAM. Data can come from PC disk storage or can be provided directly by the master CPU.

Figure 6–16 depicts a direct memory access using a PC environment. In this configuration, decode and arbitration logic are used to control the direct memory access. When the address on the system bus resides in the local memory of the peripheral 'C5x, this logic asserts the $\overline{\text{HOLD}}$ signal of the 'C5x while sending the master a not-ready indication to allow wait states. After the 'C5x acknowledges the direct memory access by asserting $\overline{\text{HOLDA}}$, READY is asserted and the information is transferred.

Figure 6–16. Direct Memory Access in a PC Environment



The 'C5x also provides direct access of the on-chip single-access RAM for external devices. DMA of the on-chip single-access RAM requires the following signals:

$\overline{\text{HOLD}}$	External request for control of address, data, and control lines.
$\overline{\text{HOLDA}}$	Indicates to external circuitry that the memory address, data, and control lines are in high impedance, allowing external access of on-chip single-access RAM.
$\overline{\text{BR}}$	Bus request signal. Externally driven low in hold mode to indicate a request for access to on-chip single-access RAM.
$\overline{\text{IAQ}}$	Acknowledge $\overline{\text{BR}}$ request for access to on-chip single-access RAM while $\overline{\text{HOLDA}}$ is low.
R/$\overline{\text{W}}$	Read/write signal indicates the data bus direction for DMA reads (high) and DMA writes (low).
$\overline{\text{STRB}}$	When active low and $\overline{\text{IAQ}}$ and $\overline{\text{HOLDA}}$ are low, this input signal is used to select the memory access. $\overline{\text{STRB}}$ determines the duration of the memory access.
A(15–0)	Address inputs during $\overline{\text{HOLDA}}$ and $\overline{\text{BR}}$ active low.
D(15–0)	DMA data.

To access the 'C5x device's on-chip single-access RAM, a master processor must control the device. The master processor initiates a DMA transfer by placing the 'C5x device in $\overline{\text{HOLD}}$. Once the device responds with a $\overline{\text{HOLDA}}$, the master can select access to the internal on-chip single-access RAM by lowering the $\overline{\text{BR}}$ input. The device responds with an $\overline{\text{IAQ}}$ to acknowledge access to the on-chip memory. Once access is granted, the master drives the R/$\overline{\text{W}}$ signal to indicate the direction of the transfer. On a DMA write, the master must drive the address and data lines for a write. On a DMA read, the master

must drive the address lines and latch the data. Each memory access (read or write) must be selected by the $\overline{STRB}$ signal. External access wait states are added by extending the $\overline{STRB}$ signal. The address decode of the DMA access includes only A13–A0, (A14 and A15 ignored). The ranges shown in Table 6–11 respond during DMA access, effectively overlaying A13–A0.

DMA access to on-chip single access RAM is not supported if the device is in concurrent hold mode (that is, HM=0).

Table 6–11. Address Ranges for On-Chip Single-Access RAM DMA

Device	Address Bus	Hex Address Range
'C50	A13–A0 used A15, A14 ignored	0000–23FF 4000–63FF 8000–A3FF C000–E3FF
'C51	A9–A0 used A13–A10 must be 0 A15–A14 ignored	0000–03FF 4000–43FF 8000–83FF C000–C3FF
'C53	A11–A0 used A13–A12 must be 0 A15–A14 ignored	0000–0BFF 4000–4BFF 8000–8BFF C000–CBFF

Note that the above address ranges correspond to 9K/1K/3K words of on-chip single-access RAM of the 'C50/51/53, respectively. For example, writing to the address 01h (using DMA) on a 'C50 affects the second memory location of the on-chip single-access RAM. Furthermore, writing to the address 4001h on 'C50 is equivalent to writing to the address 01h, as shown in Table 6–11.

6.7 Memory Management

The 'C5x devices have a programmable memory map, which can vary for each application. Instructions are provided for integrating the device memory into the system memory map. The 'C50 device includes 2K words of boot ROM, 9K words of single-access RAM, and 1056 words of dual-access RAM. The 'C51 device includes 8K words of program ROM, 1K words of single-access RAM, and 1056 words of dual-access RAM, wheras the 'C53 has 16K words of on-chip ROM, 3K words of single-access and 1056 words of dual-access RAM. Examples of moving and configuring memory are provided in this section.

6.7.1 Block Moves

The 'C5x devices address a large amount of memory but are limited in the amount of on-chip memory. Several instructions are available for moving blocks of data from off-chip slower memories to on-chip memory for faster program execution. In addition, data can be transferred from on-chip to off-chip for storage or multiprocessor applications.

The BLDD instruction facilitates the transfer of data from external or internal data memory to internal or external data memory. Example 6–1 illustrates the use of the BLDD command to move data (for example, a table of coefficients) from external memory to internal data RAM.

Example 6–1. Moving External Data to Internal Data Memory With BLDD

```
*
* This routine uses the BLDD instruction to move external data memory to
* internal data memory.
*
MOVED LACC   #8000h
      SAMM   BMAR      ;BMAR contains source address in data memory.
      LAR    AR7,#300h ;AR7 contains destination address in data memory.
      MAR    *,AR7     ;LARP = AR7.
      RPT    #511      ;Move 512 values to data memory block B1.
      BLDD   BMAR,*+
      RET
```

For systems with external data memory but no external program memory, the BLDP instruction can be used to move additional blocks of code into internal program memory. Example 6–2 illustrates the use of the BLDP instruction.

Example 6–2. Moving Data Memory to Program Memory With BLDP

```
*
* This routine uses the BLDP instruction to move external data memory to
* internal program memory. This instruction could be used to boot load a
* program to the on chip program RAM from external data memory.
*
MOVEDP LACC  #2000h
       SAMM  BMAR        ;BMAR contains dest. address in program memory ('C51)
       LAR   AR7,#0F000h ;AR7 contains source address in data memory
       MAR   *,AR7       ;ARP=AR7
       RPT   #1023       ;Move 1k of data to program memory space
       BLDP  *+
       RET
```

When no external data memory is available, program memory may contain necessary coefficient tables that should be loaded into internal data memory. The routine in Example 6–3 illustrates the use of the BLPD instruction to perform this function.

Example 6–3. Moving Program Memory to Data Memory With BLPD

```
*
* This routine uses the BLPD instruction to move external program memory to
* internal data memory. This routine is useful for loading a coefficient
* table stored in external program memory to data memory when no external
* data memory is available.
*
MOVEPD LAR   AR7,#300h   ;AR7 points to destination in data memory
       MAR   *,AR7       ;ARP=AR7
       RPT   #127        ;Move 128 values from external program to
       BLPD  #0FD00h,*+  ;internal data memory.
       RET
```

Another method of transferring data between memory spaces uses the TBLR and TBLW instructions. These instructions can specify a calculated, rather than predetermined, location of a block of data in program or data memory for transfer. The following examples illustrate the use of the TBLR and TBLW instructions.

Example 6–4. Moving Program Memory to Data Memory With TBLR

```
*
* This routine uses instruction TBLR to move program memory to data memory
* space. It differs from the BLPD instruction in that the accumulator
* contains the address in program memory from which to transfer. This allows
* for a calculated, rather than predetermined, location in program memory to
* be specified. The calling routine must load accumulator with the source
* address.
*
TABLER MAR   *,AR3      ;ARP=AR3
       LAR   AR3,#300h ;AR3 contains destination in data memory
       RPT   #127       ;Move 128 items to data memory block B2
       TBLR  *+         ;Accumulator contains external program
       RET              ;memory address.
```

Example 6–5. Moving Data Memory to Program Memory With TBLW

```
*
* This routine uses the TBLW instruction to move data memory to
* program memory. The calling routine must contain the destination program
* memory address in the accumulator.
*
TABLEW MAR   *,AR4        ;LARP = AR4.
       LAR   AR4,#300h    ;AR4 contains source address in data memory
       RPT   #511         ;Move 512 items from data memory to program
       TBLW  *+           ;memory.
       RET                ;Accumulator contains address of program RAM.
```

The IN and OUT instructions move data from data memory to an external port. The use of these instructions is shown in Example 6–6 and Example 6–7.

Example 6–6. Moving Data From I/O Space to Data Memory With SMMR

```
*
* This routine uses the SMMR instruction to move data from a memory-mapped
* I/O port to local data memory. Note that 16 I/O ports are mapped in data
* page 0 of the 'C5x memory map.
*
INPUT:
    LDP     #0
    RPT     #511        ;Input 512 values from port 51h to
    SMMR    51h,800h    ;table at 800h in data memory.
    RET
```

Example 6–7. Moving Data From Data Memory to I/O Space With LMMR

```
*
* This routine uses the LMMR instruction to move data from local data
* space to a memory-mapped I/O port. Note that 16 I/O ports are mapped
* in data page 0 of 'C5x memory map.
*
OUTPUT:
    LDP     #0          ;data page 0
    RPT     #63         ;Output 64 values from a table at 800h
    LMMR    50h,800h    ;in data memory to port 50h.
    RET
```

6.7.2 Boot Loader ('C50)

The main function of the boot loader is to transfer user code from an external source to the program memory at power-up. The 'C50 provides different ways to download the code to accommodate various system requirements. For some applications, a serial interface is appropriate. For others, a parallel interface is appropriate if the code is already stored in external ROM.

If the MP/$\overline{MC}$ pin of the 'C50 is sampled low during a hardware reset, execution begins at location zero of the on-chip ROM. The on-chip ROM is factory programmed with a boot-load program.

The boot-load program sets up the CPU status registers before initiating the boot load. Interrupts are globally disabled (INTM=0), internal dual-access RAM block B0 is mapped in program space (CNF=1), and the on-chip single-access RAM block is enabled in program space (RAM=1, OVLY=0). Seven wait states are selected for the entire program and data spaces. Initially, the 32K words of global data memory are enabled in data space 08000h to 0FFFFh. After the code transfer is complete, the global memory is disabled before control is transferred to the destination address.

The boot routine reads the global data memory location 0FFFFh by driving the bus request ($\overline{BR}$) and data strobe ($\overline{DS}$) pins low. The lower eight bits of the word read from global memory location 0FFFFh specify the mode of transfer. The rest of the bits are ignored by the boot loader.

Figure 6–17 lists available boot options and corresponding configuration byte patterns.

Figure 6–17. Boot Routine Selection Word

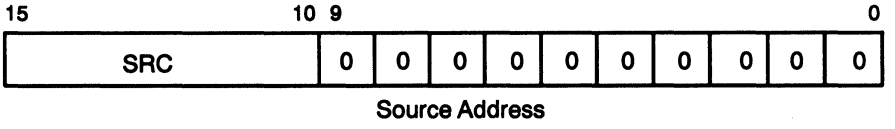
15 8	7 4	3 0	At Address FFFFh
XXXXXXXX	XXXX	0000	8-Bit Serial Mode
XXXXXXXX	XXXX	0100	16-Bit Serial Mode
XXXXXXXX	XXXX	1000	8-Bit Parallel I/O Mode
XXXXXXXX	XXXX	1100	16-Bit Parallel I/O Mode
XXXXXXXX	SRC	01	8-Bit Parallel EPROM Mode
XXXXXXXX	SRC	10	16-Bit Parallel EPROM Mode
XXXXXXXX	ADDR	10	Warm Boot

Note: X = Don't care condition
SRC = 6-bit page address for parallel modes
ADDR = 6-bit page address for warm boot

Parallel Boot

The parallel boot option is used if the code is stored in EPROMs (8-bit or 16-bit wide in global data space). The code is transferred from global data memory to program memory. The six MSBs of the source address are specified by the SRC field of the boot routine selection (BRS) word as shown in Figure 6–17. A 16-bit EPROM address is defined by this SRC field as shown in Figure 6–18.

Figure 6–18. 16-Bit EPROM Address

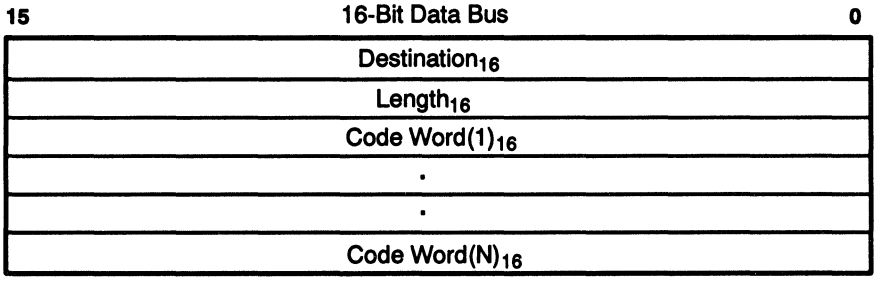


If the16-bit parallel mode is selected, data is read in 16-bit words from the source address, incrementing the address by one after every read operation. The destination address $destination_{16}$ and the length $length_{16}$ of the code are specified by the first two 16-bit words. The *length N* is defined as:

length N = number of 16-bit words to be transferred – 1

The number of 16-bit words specified by the parameter *N* do not include the first two words read, starting from the source address — that is, the destination and length parameters. This is shown in Figure 6–19. The code is transferred from the global data memory to the program memory. There is at least a four-instruction cycle delay between a read from EPROM and a write to the destination address. This ensures that if the destination is external memory (such as fast SRAM), there is enough time to turn off the source memory (EPROM) before the write operation is performed.

Figure 6–19. 16-Bit Parallel Boot



$Destination_{16}$	16-bit destination address.
$Length_{16}$	16-bit word that specifies the length of the code (N) that follows it.
Code $Word(N)_{16}$	N 16-bit words to be transferred.

After the specified length of code words are transferred to the program memory, the 'C50 branches to the destination address.

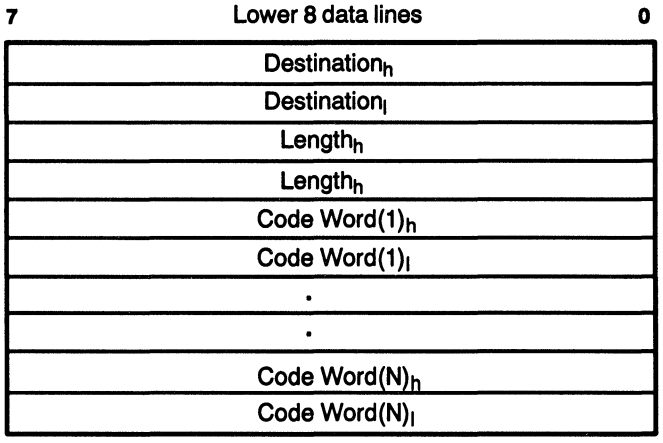
If the 8-bit parallel boot option is selected, two consecutive memory locations (starting at source address) are read to make one 16-bit word. The high-order byte should be followed by the low-order byte. Data is read from the lower eight data lines, ignoring the upper byte on the data bus. The destination address is a 16-bit word that constitutes address in program space where the boot code is transferred. The length *N* is defined as:

length N = number of 16-bit words to be transferred – 1

length N = (number of bytes to be transferred ÷ 2) – 1

The number of 16-bit words specified by the parameter *N* do not include the first four bytes (or first two words) read, starting from the source address —that is, the destination and length parameters. This is shown in Figure 6–20. The code is transferred from the global data memory to the program memory.There is at least a four-instruction cycle delay between a read from source memory (such as EPROM) and a write to the destination address. This ensures that if the destination is external memory (such as fast SRAM), there is enough time to turn off the source memory (EPROM) before the write operation is performed.

Figure 6–20. 8-Bit Parallel Boot



Destination$_h$ Destination$_l$	High and low bytes of destination address
Length$_h$ Length$_l$	High and low bytes of a 16-bit word that specifies the length N of the code that follows it
Code Word(1)$_h$ Code Word(1)$_l$ . . Code Word(N)$_h$ Code Word(N)$_l$	N high and N low bytes that constitute N words to be transferred

Serial Boot

In the serial boot option, the serial port control register (SPC) is set to 0xxF8h or 0xxFCh for 16-bit and 8-bit modes, respectively. The RRST and XRST bits are each set to 1, taking the serial port out of reset. FSM is set to 1, configuring the serial port in frame sync mode — that is, frame sync pulses are required to be supplied externally on the FSR pin. The value of the FO bit is set according to the mode selected (8- or 16-bit modes). The external flag XF signals that the 'C50 is ready to respond to the serial port receive section. XF is set to high at reset and is driven low to initiate reception. No frame sync pulses should appear on the FSR before XF going low. The receive clock must be supplied by a device external to the 'C50.

In the case of 16-bit serial mode, the first 16-bit word received by the device from the serial port specifies the destination address of boot code in program memory. The next 16-bit word specifies the length of the actual code that follows it. The length N is defined as:

length N = number of 16-bit words – 1

Note that the number of 16-bit words specified by the parameter *N* do not include the first two words read, starting from the source address — that is, the destination and length parameters. In the case of an 8-bit serial transfer, a higher-order byte followed by a low-order byte constitute a 16-bit word. The first 16-bit word received by the device from the serial port specifies the destination address of boot code in program space. The following 16-bit word specifies the length of the actual code that follows it. The length N is defined as:

length N = number of 16-bit words – 1

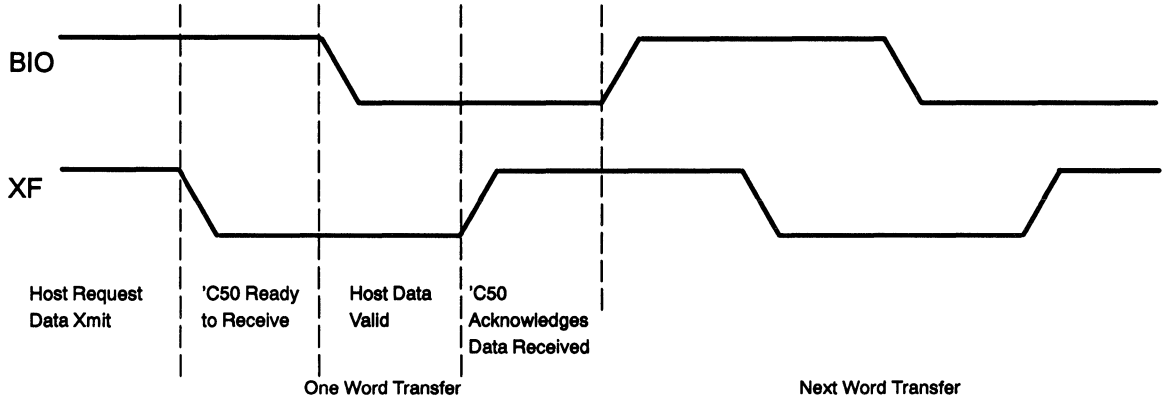
length N = (number of bytes to be transferred ÷ 2) – 1

After the specified length of code words is transferred to program memory, the 'C50 branches to the destination address.

I/O Boot

The I/O boot mode is provided to asynchronously transfer code from I/O port 50h to internal/external program memory. Each word may be either 16 bits or 8 bits long. The 'C50 communicates with the external device by using $\overline{BIO}$ and XF handshake lines. The handshake protocol shown in Figure 6–21 is required to succesfully transfer each word from port 50h:

Figure 6–21. Handshake Protocol



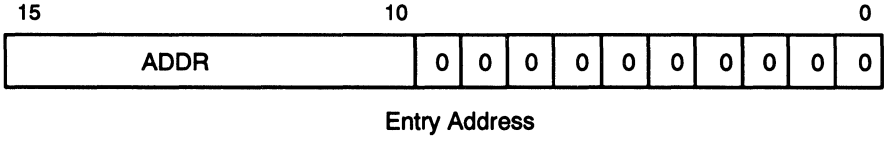
If the 8-bit transfer mode is selected, the lower eight data lines are read from port 50h. The upper byte on the data bus is ignored. The 'C50 reads two 8-bit words to form a 16-bit word. The low byte of a 16-bit word should follow the high byte.

For both 8-bit and 16-bit I/O modes, the first two 16-bit words received by the 'C50 must be the destination and the length of the code, respectively. See the parallel boot description for destination and length code words. A minimum delay of four clock cycles is provided between the XF rising edge and the write operation to the destination address. This allows the host processor sufficient time to turn off its data buffers before the 'C50 initiates write operation (if destination is external memory). Note that the 'C50 accesses the external bus only when XF is high.

Warm Boot

The warm boot option can be used if the program has already been transferred to internal (or external) memory by other means (for example, DMA), or if it is only a warm device reset. In this case, six MSBs of the 8-bit long BRS word specify the entry point of the code as shown in Figure 6–22.

Figure 6–22. Warm Boot



The 'C50 transfers control to the entry address if a warm boot is specified.

Chapter 7

Software Applications

The'C5x digital signal processors maintain source code compatibility with 'C1x and 'C2x generations and have architectural enhancements that improve performance and versatility. An orthogonal instruction set is augmented by new instructions that support additional hardware and handle data movement and memory-mapped registers. Other features include an independent parallel logic unit (PLU) for performing Boolean operations, a 32-bit accumulator buffer, and a set of registers that provide zero-latency context-switching capabilities to interrupt service routines. The on-chip dual-access RAM and memory-mapped register set are enhanced.

This chapter explains the use of 'C5x instruction set with particular emphasis on its new features and special applications. For a complete discussion of the assembler directives used in this chapter's examples, please consult the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide*, literature number SPRU018B. Major topics discussed in this chapter are listed below.

Topic **Page**

7.1 Processor Initialization

Before executing a digital signal processing algorithm, it is necessary to initialize the processor. Generally, initialization takes place anytime the processor is reset.

The processor is reset by applying a low level to $\overline{RS}$ input; the IPTR bits of PMST register are all cleared, thus mapping the vectors to page zero in program memory space. This means that the reset vector always resides at program memory location 0. This location normally contains a branch instruction to direct program execution to the system initialization routine. A hardware reset clears all pending interrupt flags and sets the INTM (global enable interrupts) bit to 1, thereby disabling all interrupts. It also initializes various status bits and peripheral registers. Refer to subsection 3.8.1 for details.

To configure the processor after the reset, the following internal functions should be initialized.

- ❑ Memory-mapped core processor and peripheral control registers
- ❑ Interrupt structure (INTM)
- ❑ Mode control (OVM, SXM, PM, AVIS, NDX, TRM)
- ❑ Memory control (RAM, OVLY, CNF)
- ❑ Auxiliary registers and the auxiliary register pointer (ARP)
- ❑ Data memory page pointer (DP)

The OVM (overflow mode), TC (test/control flag), IMR (interrupt mask register), auxiliary register pointer (ARP), auxiliary register pointer buffer (ARB), and data memory page pointer (DP) are not initialized by reset.

Example 7–1 shows coding for initializing the 'C5x to the following machine state, and for the initialization performed during hardware reset:

- ❑ Internal single-access RAM configured as program memory
- ❑ Interrupt vector table loaded in internal program memory
- ❑ Interrupt vector table pointer (IPTR)
- ❑ Internal dual-access RAM blocks filled with zero
- ❑ Interrupts enabled

Example 7–1. Initialization of 'C5x

```
        .title  'PROCESSOR INITIALIZATION'
        .mmregs
        .ref    ISR0,ISR1,ISR2,ISR3,ISR4,TIME
        .ref    RCV,XMT,TRCV,TXMT,TRP,NMISR

MAIN_PRG    .set    04000h      ;program space address of main
                                ;foreground routine
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
* Processor initialization for TMS320C50.
*
* For the TMS320C51, the memory mapping of S/A RAM in program
* space and data space is not identical. Therefore, memory location
* pointed to by address 0800h in data space is mapped to address
* 02000h in program space. Hence, the vector table must be loaded
* at data memory 0800h in order to keep the vector table address
* 02000h in program space.
*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

V_TBL   .sect   "vectors"

RESET   B   INIT                ;This section will be loaded in program
                                ;memory address 0h.
INT1    B   ISR1                ;INT1– begins processing here
INT2    B   ISR2                ;INT2– begins processing here
INT3    B   ISR3                ;INT3– begins processing here
TINT    B   TIME                ;Timer interrupt processing
RINT    B   RCV                 ;Serial port receive interrupt
XINT    B   XMT                 ;Serial port transmit interrupt
TRNT    B   TRCV                ;TDM port receive interrupt
TXNT    B   TXMT                ;TDM port transmit interrupt
INT4    B   ISR4                ;INT4– begins processing here

        .space  14*16           ;14 words

TRAP    B   TRP
NMI     B   NMISR

        .text

INIT    LDP     #0              ;Initialize data pointer
        OPL     #20h,PMST       ;Configure S/A RAM in data memory
        LAR     AR7,#02000h     ;data space address for vector table
;;;;    LAR     AR7,#0800h      ;for TMS320C51
        MAR     *,AR7           ;ARP <– AR7
        RPT     #39             ;for I=0,I<=39,I++
        BLPD    #V_TBL,*+       ;Load vector table at 2000h
        SPLK    #201Eh,PMST     ;Now configure S/A RAM in program space
                                ;and initialize vector table pointer
        SPLK    #01FFh,IMR      ;Clear interrupt mask register
        CLRC    OVM             ;Disable overflow saturation mode

        LAR     AR7,#60h        ;Initialize B2 block
        RPTZ    #31             ;for I=0,I<=31,I++
        SACL    *+              ;B2[I] = 0

        LAR     AR7,#100h       ;Initialize B0 and B1 blocks
        RPTZ    #1023           ;for I=0,I<=1023,I++
        SACL    *+              ;B0/B1[I] = 0

        CLRC    INTM            ;Globally enable interrupts
        B       MAIN_PRG        ;Return to foreground program
```

7.2 Interrupts

The 'C5x devices have four external maskable user interrupts ($\overline{INT1}$–$\overline{INT4}$) and one nonmaskable interrupt ($\overline{NMI}$) available for external devices. Internal interrupts are generated by the serial ports, the timer, and by the software interrupt instructions (INTR, TRAP, and NMI). The interrupt structure is described in subsection 5.1.2, *Interrupts*.

The 'C5x devices are capable of generating software interrupts using INTR instruction. This allows any of the 32 interrupt service routines to be executed from your software. The first 20 ISRs are reserved for external interrupts, peripheral interrupts, and future implementations. The other 12 locations in the interrupt vector table are user-definable. The INTR instruction can invoke any of the 32 interrupts available on the 'C5x devices.

The context saving and restoring function is done in hardware when an interrupt trap is executed. An 8-deep hardware stack is available for saving return addresses of the subroutines and the interrupt service routines. Also, there is a one-deep stack (or shadow registers) for each of the following registers:

ACC	accumulator
ACCB	accumulator buffer
PREG	product register
ST0	status register 0 (INTM not restored)
ST1	status register 1 (XF not restored)
PMST	processor mode status register
TREG0	temporary register for multiplier
TREG1	temporary register for shift count
TREG2	temporary register for bit test
INDX	indirect address index register
ARCR	auxiliary register compare register

When the interrupt trap is taken, all these registers are pushed onto the one-deep stack. These shadow registers are popped when the return-from-interrupt (RETI or RETE) is executed. Detailed discussion of interrupts are given in Section 3.8, *Interrupts*.

Example 7–2 illustrates the use of INTR instruction. The foreground program sets up auxiliary registers and invokes user-defined interrupt number 20. Since the context is saved automatically, the interrupt service routine is free to use any of the saved registers without destroying the calling program's variables. The routine shown here uses the CRGT instruction to find the maximum value of 16 executions of the equation $Y=aX^2+bX+c$. The X values are pointed at by AR1. AR2 and AR3 point to the coefficients and Y results, respectively. To return the result to the calling routine, all the registers are restored by executing an RETI instruction. The computed value is placed in the accumulator, and a standard return is executed because the stack is already popped.

Example 7–2. Use of INTR Instruction

```
* Foreground Program

            .mmregs

TEMP        .set    63h         ;Temporary storage.

            .
            .
            .
            LAR     AR1,#X      ;AR1 points to X values
            LAR     AR2,#COEFF  ;AR2 points to coefficients b,a,c in that order
            LAR     AR3,#Y      ;AR3 points to Y results
            INTR    20          ;Invoke software interrupt #20
            .
            .
*
* This routine uses the block repeat feature of the 'C50 to find the maximum
* value of 16 executions of the equation Y=aX^2+bX+c. The X values are pointed
* at by AR1. The Y results are pointed at by AR3. The coefficients are pointed
* at by AR2. At the completion of the routine, ACC contains the maximum value.
* AR1, AR2, and AR3 are modified. All other registers are unaffected. Note that
* this routine should not be called from within a repeat block.
*
ISR20       LDP     #0          ;Use page 0 of data memory.
            LACC    #08000h
            SACB                ;Initialize AccB with min. possible value
            MAR     *,AR1       ;ARP <- AR1
*
* Load Block repeat count register with 15.
*           SPLK    #0Fh,BRCR
*
* Repeat Block.
*


            RPTB  END_LOOP-1    ;For i=0; i<=15; i++.
             ZAP                ;ACC = PREG = X^2
             SQRA *+,AR2        ;TREG0 = X   PREG = X^2
             SPL  TEMP          ;Save X^2.
             MPY  *+            ;PREG = b*X
             LTA  TEMP          ;TREG = X^2   ACC = b*X
             MPY  *+            ;PREG = a*X^2
             APAC               ;ACC = a*X^2 + b*X
             ADD  *,0,AR3       ;ACC = A*X^2 + b*X + c
             SACL *+,0,AR1      ;Save Y.
             CRGT               ;Save maximum Y.
END_LOOP
            SACL  TEMP          ;Save the result temporarily
            LACC  #RE_ENTER     ;
            PUSH                ;Push re-entry address onto stack
            RETI                ;Pop all registers
RE_ENTER    LAMM  TEMP          ;Load ACC with the max. value
            RET                 ;Return to interrupted code
```

7.3 Software Stack

The 'C5x has an internal 8-deep hardware stack that is used to save and restore return addresses for subroutines and interrupts. See subsection 3.6.1 for further details. Provisions have been made on the 'C5x to extend the hardware stack into the data memory.

The PUSH and POP instructions can access the hardware stack via the accumulator. Two additional instructions, PSHD and POPD, are included in the instruction set so that the stack may be directly stored to and recovered from the data memory.

A software stack can be implemented by using POPD instruction at the beginning of each subroutine to save the PC in data memory. Then, before returning, a PSHD is used to put the proper value back onto the top of the stack.

When the stack has seven values stored on it, and two or more values are to be put on the stack before any other values are popped off, a subroutine that expands the stack is needed, such as the one shown in Example 7–3. In this example, the main program stores the stack, starting location in memory in AR2 and indicates to the subroutine whether to push the data from memory onto the stack or pop data from the stack to memory. If a zero is loaded into the accumulator before calling the subroutine, the subroutine pushes data from memory to the stack. If the accumulator contains a nonzero value, the subroutine pops data from the stack to memory.

Because the CALL instruction uses the stack to save the program counter, the subroutine pops this value into the accumulator and utilizes the BACC instruction to return to the main program. This prevents the program counter from being stored into a memory location. The subroutine in Example 7–3 uses the BCNDD (delayed conditional branch) instruction to determine whether a save or restore operation is to be performed.

Example 7–3. Software Stack Operation

```
*
* This routine expands the stack while letting the
* main program determine where to store the stack
* contents, or from where to restore them.
* Entry Conditions:
* ACC = 0 (restore stack); 1 (save stack)
* AR2 -> Top of software stack in data memory
*
STACK: BCNDD  POP,NEQ       ;Delayed branch if POPD required
        MAR   *,AR2         ;Use AR2 as stack pointer
        POP                 ;Get return address
       RPT    #6            ;repeat 7 times
        PSHD  *+            ;Put memory in stack
       BACC                 ;Return to main program
POP:   MAR    *-            ;Align AR2
       RPT    #6            ;Repeat 7 times
        POPD  *-            ;Put stack in memory
       MAR    *+            ;Realign stack pointer
       BACC                 ;Return to main program
```

7.4 Logical and Arithmetic Operations

7.4.1 Parallel Logic Unit (PLU)

The PLU provides direct logical path to data memory values without affecting the contents of the accumulator or product register. It allows direct manipulation of bits in any location in data memory space. Source operand can be either a long immediate value or the dynamic bit manipulation register (DBMR). The use of a long immediate value is particularly effective in initializing data memory locations, including the memory-mapped registers. The use of DBMR as source operand allows run-time computation of operands. It also reduces instruction execution time to one cycle, which may be important for time-critical routines.

Example 7–4 and Example 7–5 illustrate the use of PLU for initialization and logical operation. The UNPACK subroutine extracts individual bits from a single word and stores them separately in an array. The PACK subroutine does the opposite of UNPACK by getting each bit from a different location and packing them in a single word. In Example 7–5, notice that a NOP instruction is inserted in the repeat-block loop to make it three words long. A repeat-block must be at least three words long on 'C5x devices.

Example 7–4. Using PLU to Do Unpacking

```
        .title      'Routine to extract bits from a single word'
*       PCKD
*       ____________________
*       |Bn  ———————   B0|
*       ____________________
*
*       UNPCKD
*       ____________________
*       |0   —    0 |Bn|
*       ____________________
*       |0   —   0|Bn-1|
*       ____________________
*             . .
*       ____________________
*       |0   —    0|B0|
*       ____________________
        .mmregs
NO_BITS .set    16                      ;number of packed bits in the word
PCKD    .set    60h                     ;Input word
UNPCKD  .set    61h                     ;Output buffer. Each word will have
                                        ;one bit in LSB location.
        .text
UNPACK  LDP     #0                      ;DP=0
        MAR     *,AR0
        LAR     AR0,#UNPCKD+NO_BITS-1   ;End of table address
        SPLK    #NO_BITS-1,BRCR         ;Initialize the count register
        SPLK    #1,DBMR                 ;Load mask in DBMR register
        LACC    PCKD                    ;Packed bits -> Acc
        RPTB    LOOP-1                  ;Begin looping
         SACL   *                       ;Save remaining packed bits
         APL    *-                      ;Keep the LSB only
         SFR                            ;Shift right to eliminate unpacked bit
LOOP    RET                             ;Return back
```

Example 7–5. Using PLU to Do Packing

```
    .title      'Routine to pack input bits in a single word'
*
*   PCKD
*   ------------------
*   |Bn  -------  B0|
*   ------------------
*
*   UNPCKD
*   ------------------
*   |0   —    0 |Bn|
*   ------------------
*   |0   —   0|Bn-1|
*   ------------------
*         . .
*   ------------------
*   |0   —     0|B0|
*   ------------------
*
        .data
NO_BITS .set    16                  ;Number of bits to be packed
PCKD    .set    60h                 ;Packed word
UNPCKD  .set    61h                 ;Array of unpacked bits
        .text
PACK    LAR         AR0,#UNPCKD     ;AR0 points to start of UNPACKED array
        MAR         *,AR0           ;ARP <- AR0
        LDP         #0              ;DP=0
        SPLK    #NO_BITS-2,BRCR     ;Loop NO_BITS-1 times
        LACC    *+                  ;Get the MSB
        RPTB    LOOP-1              ;Begin looping
         SFL                        ;Make space for next bit
         ADD    *+                  ;Put next bit
         NOP
LOOP
        SACL    PCKD                ;Store the result
        RET                         ;Return back
```

7.4.2 Multiconditional Branch Instruction

The 'C5x allows multiple conditions to be tested before passing control to another section of program. Any of the following 13 conditions can be tested individually or in combination with others by CC, RETC, XC, and BCND instructions:

ACC=0	EQ
ACC≠0	NEQ
ACC<0	LT
ACC≤0	LEQ
ACC>0	GT
ACC≥0	GEQ
C=0	NC
C=1	C
OV=0	NOV
OV=1	OV
$\overline{BIO}$ low	BIO
TC=0	NTC
TC=1	TC

Testing the status of TC flag is mutually exclusive to testing the BIO pin. The code in Example 7–6 tests the carry flag and the sign bit of the accumulator simultaneously to locate a zero bit (beginning from MSB) in a 64-bit word consisting of ACC and ACCB with ACC having the higher part. This 64-bit word could be the serial port output where the first zero indicates the start bit.

Example 7–6. Using Multiple Conditions With BCND

```
        LDP     #0
        SPLK    #63,BRCR        ;no. of iterations - 1
        .
        .                       ;code to get 64-bit input word and load
                                ;it in ACC and ACCB
        .
        LAR     AR0,#0          ;initialize the bit counter
        RPTB    ENDLOOP-1       ;for I=0,I<=63,I++
         SFLB                   ;shift left ACC+ACCB, MSB is shifted
*                               ;out in Carry flag
         MAR    *+              ;increment bit counter
         BCND   ENDLOOP,NC,LT   ;exit if carry=0 and current MSB=1
ENDLOOP:                        ;ACC+ACCB contains aligned data now
        APL     #0fffeh,PMST    ;clear BRAF flag
```

7.4.3 Search Algorithm Using CRGT

The following example shows how the CRGT and RPTB instructions find the maximum value and its location by searching through a block of data. Loop overhead is minimized by using the block-repeat function. The accumulator is initialized with the minimum possible value (08000h) before the main search loop is entered.

To find the minimum value, CRGT instruction may be replaced by CRLT, and the accumulator is loaded with the maximum possible value (07FFFh) instead of the smallest. The rest of the code remains the same.

Example 7–7. Using CRGT and CRLT

```
*
* This routine searches through a block of data in the data memory
* to store the maximum value and the address of that value in memory
* locations MAXVAL and MAXADR, respectively. The data block could be
* of any size defined by the Block Repeat Counter Register (BRCR).
*
* KEY C5X instructions:
*
* RPTB  Repeat a block of code as defined by repeat counter BRCR
* CRGT  Compare ACC to ACCB. Store larger value in both ACC, ACCB.
*       Set CARRY bit if value larger than previously larger is found
* XC    Execute conditionally (1 or 2 words) if flag (Carry) is set.
*
*
MAXADR  .set    60h
MAXVAL  .set    61h
        .mmregs
        .text
        LDP     #0                  ;point to data page 0
        LAR     AR0, #0300h         ;AR= data memory addr
        SETC    SXM                 ;set sign extension mode
        LACC    #08000h             ;load minimum value
* Use #07FFFh (largest possible) to check for minimum value
        SACB                        ;into ACCB
        SPLK    #9,BRCR             ;rpt cont = 9 for 10 data values
        RPTB    endb -1             ;repeat block from here to endb-1
startb:
         LACC   *                   ;load data from <(AR0)> into ACC
         CRGT                       ;set carry if ACC > previous largest
* Use CRLT to find minimum value
         SACL   MAXVAL              ;save new largest which is in ACC & ACCB
         XC     #1,C                ;save addr if current value > previous largest
         SAR    AR0,MAXADR
         MAR    *+
endb:   RET
* At the end of routine, following
* registers contain:
*   ACC         = 32050
*   ACCB        = 32050
*   (MAXVAL)    = 32050
*   (MAXADR)    = 0307h
      .data                         ;data is expected to be in data RAM
      .word   5000                  ;start address = 0300h
      .word   10000
      .word   320
      .word   3200
      .word   -5600
      .word   -2105
      .word   2100
      .word   32050
      .word   1000
      .word   -1
      .end
```

7.4.4 Matrix Multiplication Using Nested Loops

The 'C5x provides three different types of instructions to implement code loops. The RPT (single-instruction repeat) instruction allows the following instruction to be executed N times. The RPTB (repeat block) instruction repeatedly executes a block of instructions with the loop count determined by the BRCR count register. The BANZ (branch if AR not zero) instruction is another way of implementing for-next loops with the count specified by an auxiliary register.

Three-level-deep nested loops can be efficiently implemented by these three instructions with each instruction controlling one loop. The following example implements this nested code structure to do N-by-N matrix multiplication. Note the use of BANZD (delayed BANZ) instruction to avoid flushing the instruction pipeline. Also, note the use of MADS (multiply-accumulate using BMAR) instruction to dynamically switch between the rows of matrix A to compute the elements of the product matrix C.

Example 7–8. Using Nested Loops

```
        .title "NxN Matrix Multiply Routine"
        .mmregs

*
* This routine performs multiplication of two NxN matrices.
* A x B = C where A,B, and C are NxN in size.
* Entry Conditions:
*     AR1 -> element (0,0) of A (in program space)
*     AR2 -> element (0,0) of B (in data space)
*     AR3 -> element (0,0) of C (in data space)
*     DP  =  0,     NDX = 1
*     ARP =  2
* Storage of matrix elements in memory (beginning from low
* memory):
*     M(0,0),...,M(0,N-1),M(1,0),...,M(N-1,N-1)
*
MTRX_MPY:
        LAR     AR0,#(N-1)      ;set up loop count
        SPLK    #N,INDX         ;row size
        SAR     AR2,AR4         ;Save addr of B
*                               ;for i=0,i<N,++i
LOOP1:  SMMR    AR1,BMAR        ;BMAR -> A(i,0)
        SPLK    #(N-1),BRCR     ;setup loop2 count
        SAR     AR4,AR5         ;AR5 -> B(0,0)
LOOP2:  RPTB    ELOOP2          ;for j=0,j<N,++j
        SAR     AR5,AR2         ;AR2 -> B(0,j)
LOOP3:  RPTZ    #(N-1)          ;for k=0,k<N,++k
ELOOP3:MADS     *0+             ;Acc=A(i,k)xB(k,j)
        APAC                    ;Final accumulation
        MAR     *,AR5           ;ARp = AR5
        MAR     *+,AR3          ;AR5 -> B(0,j+1)
ELOOP2:SACL     *+,0,AR2        ;Save C(i,j)
        MAR     *,AR0           ;loop back if
        BANZD   LOOP1,*-,AR1    ;count != N
        ADRK    N               ;AR1 -> A(i+1,0)
ELOOP1: MAR     *,AR2           ;ARp = AR2
```

7.5 Circular Buffers

Circular addressing is an important feature of the 'C5x instruction set. Algorithms like convolution, correlation, and FIR filters can make use of circular buffers in memory. The 'C5x supports two concurrent buffers operating via the auxiliary registers. These five memory-mapped registers control the circular buffer operation: CBSR1, CBSR2, CBER1, CBER2, CBCR. See subsection 4.1.6 for details.

The start and end addresses must be loaded in the corresponding buffer registers before the circular buffer is enabled. Also, the auxiliary register that acts as a pointer to the buffer must be initialized with the proper value.

Example 7–9 illustrates the use of a circular buffer to generate a digital sine wave. A 256-word sine-wave table is loaded in the B1 block of dual-access internal data memory from external program memory. Accessing the internal dual-access memory requires only one machine cycle. The block move address register (BMAR) is loaded with the ROM address of the table. The block-move instruction moves 256 samples of sine wave to internal data memory, which is then set up as a circular buffer.

The start and end addresses of this circular buffer are loaded into the corresponding registers. The auxiliary register AR7 is also initialized to the beginning of the sine-wave table. Note the use of SAMM instruction to update AR7. This is possible because all auxiliary registers are memory-mapped at page 0. Finally, the circular buffer #1 is enabled, and AR7 is mapped to that buffer. The other circular buffer is disabled.

Whenever the next sample is to be pulled off from the table, postincrement indirect addressing may be used with AR7 as the pointer. This ensures that the pointer will wrap around to the beginning of the table if the previous sample was the last one on the table.

Example 7–9. Use of Circular Addressing

```
    .title  'Digital Sine–Wave Generator'
    .mmregs
*****************************************************************
*  This routine illustrates the circular addressing capability of
*  TMS320C5x devices. A digital sine wave generator is implemented
*  as a circular buffer #1 with AR7 as its pointer. XSINTBL is the
*  location in external program memory where this table is stored.
*  It is moved to internal data memory block B1 where it is setup
*  as a circular buffer.
*****************************************************************

XSINTBL .set 03000h       ;program space address of sine table
        .text

SINTBL  LDP  #0
        LAR  AR0,#0300h   ;address of B1 block
        MAR  *,AR0
        LACC #XSINTBL     ;get sine table address in
*                         ;external program memory
        SAMM BMAR         ;load source register
*
        RPT  #255         ;move 256–word
         BLPD BMAR,*+     ;load table from external program
*                         ;memory to internal data memory
        SAMM CBSR1        ;start address of buffer=300h
        SAMM AR7          ;AR7 points to start of buffer
        ADD  #255
        SAMM CBER1        ;end address of buffer=3ffh
        SPLK #0Fh,CBCR    ;enable CB#1, disable CB#2
          .               ;pointer for CB#1 is AR7
          .
          .
NXTSMP  MAR  *,AR7
        LACC *+           ;get next sample from table
          .               ;AR7 is updated to next valid sample
          .
          .
DISBLE  APL  #0FFF7h,CBCR ;Disable CB#1
          .
          .
          .
        RET
```

If the step size must be greater than one, check to see if an update to the auxiliary register generates an address outside the range of the circular buffer. This may happen if the same sine table is used to generate sine waves of different frequencies by changing the step size. Modulo addressing can avoid such problems. A simple way to perform modulo addressing on 'C5x devices is to use the APL and OPL instructions. For example, to implement the modulo-256 counter, first load the DBMR (dynamic bit manipulation register) with 255 (the maximum value allowed); when the auxiliary register is updated (by any amount), it is ANDed with the DBMR register and ORed with the start address of the buffer. The start address of the modulo-2^k buffer must have zeros in the k LSBs. Hence, for modulo-256 addressing, the first 8 LSBs of the start register must be zero.

The following code does modulo-256 addressing:

```
START   .set    04000h      ;start address of the buffer

        LDP     #0
        LACL    #0FFh
        SAMM    DBMR        ;max value = 255
          .
          .
          .
        MAR     *0+         ;increment AR7 by some amount
        APL     AR7         ;extract lower 8 bits
        OPL     #START,AR7  ;add the start address
          .
          .
```

7.6 Single-Instruction Repeat (RPT) Loops

The 'C5x provides two different types of repeat instructions. The repeat block RPTB instruction implements code loops that can be 3 to 65536 words in size. These loops do not require any additional cycles to jump from the end-of-block to the start-of-block address at the end of each iteration. In addition, these zero-overhead loops are interruptible so that they can be used in background processing without affecting the latency of time-critical tasks.

On the other hand, the single-instruction repeat RPT pipelines the execution of the next instruction to provide a high-speed repeat mode. A 16-bit repeat counter RPTC allows execution of a single instruction 65536 times. When this repeat feature is used, the instruction being repeated is fetched only once. As a result, many multicycle instructions, such as MAC/MACD, BLDD/BLDP, or TBLR/TBLW, become single-cycle when repeated.

Some of 'C5x instructions behave differently in the single-instruction repeat mode to efficiently utilize the 'C5x multiple-bus architecture. The following instructions fall in this category:

BLDD, BLDP, BLPD, IN, OUT, MAC, MACD, MADS, MADD, TBLR, TBLW, LMMR, SMMR

Because the instruction is fetched and internally latched when in single-instruction repeat mode, the program bus is used by these instructions to read or write a second operand in parallel to the operations being done using the data bus. With the instruction latched for repeated execution, the program counter is loaded with the second operand address (which may be in data, program, or I/O space) and incremented on succeeding executions to read/write in successive memory locations. As an example, the MAC instruction fetches the multiplicand from the program memory via the program bus. Simultaneously with the program bus fetch, the second multiplicand is fetched from data memory via the data bus. In addition to these data fetches, preparation is made for accesses in the following cycle by incrementing the program counter and by indexing the auxiliary register. IN instruction is another example of an instruction that benefits from simultaneous transfers of data on both the program and data buses. In this case, data values from successive locations in I/O space may be read and transferred to data memory. For complete details of how the above-listed instructions behave in repeat mode, see the individual description of each instruction in Chapter 4.

The following example demonstrates the implementation of memory-to-memory block moves on the 'C5x using single-instruction repeat (RPT) loops.

Example 7–10. Memory-to-Memory Block Moves Using RPT

```
    .mmregs
    .text
*
* This routine uses the BLDD instruction to move external
* data memory to internal data memory.
*
MOVEDD:
    LACC #4000h
    SAMM BMAR              ;BMAR -> source in data memory.
    LAR  AR7,#100h         ;AR7 -> destination in data memory
    MAR  *,AR7             ;LARP = AR7.
    RPT  #1023             ;Move 1024 value to blocks B0 and B1
     BLDD BMAR,*+
    RET
*
* This routine uses the BLDP instruction to move external
* data memory to internal program memory. This instruction could be
* used to boot load a program to the 8K on chip program memory from
* external data memory.
*
MOVEDP:
    LACC #800h
    SAMM,BMAR              ;BMAR -> destination in program memory
    LAR  AR7,#0E000h       ;AR7 -> source in data memory.
    RPT  #8191             ;Move 8K to program memory space.
     BLDP *+
    RET
*
* This routine uses the BLPD instruction to move external
* program memory to internal data memory.  This routine
* is useful for loading a coefficient table stored in
* external program memory to data memory when no external
* data memory is available.
*

MOVEPD:
    LAR     AR7,#100h      ;AR7 -> destination in data memory.
    RPT     #127           ;Move 128 values from external program
       BLPD #3800h,*+      ;to internal data memory B0.
    RET
*
* This routine uses the TBLR instruction to move program
* memory to data memory space.  This differs from the BLPD
* instruction in that the accumulator contains the address
* in program memory from which to transfer. This allows
* for a calculated, rather than pre–determined, location in
* program memory to be specified.
*

TABLER:
    MAR     *,AR3          ;AR3 -> destination in data memory.
    LAR     AR3,#300h
    RPT     #127           ;Move 128 items to data memory block B1
       TBLR *+
    RET

*
* This routine uses the TBLW instruction to move data memory
* to program memory. The calling routine must contain the destination
* program memory address in the accumulator.
*
TABLEW:
    MAR     *,AR4          ;ARP = AR4.
    LAR     AR4,#380h      ;AR4 -> source address in data memory.
    RPT     #127           ;Move 128 items from data memory to
       TBLW *+             ;program memory.
  RET
```

```
*
* This routine uses the SMMR instruction to move data
* from a memory-mapped I/O port to local data memory.
* Note that 16 I/O ports are mapped in data page 0 of
* the 'C5x memory map.
*
INPUT:
      LDP     #0
      RPT     #511        ;Input 512 values from port 51h to
      SMMR 51h,800h       ;table at 800h in data memory.
      RET

*
* This routine uses the LMMR instruction to move data from
* local data space to a memory-mapped I/O port. Note that
* 16 I/O ports are mapped in data page 0 of TMS320C5x
* memory map.
*
OUTPUT:
      LDP     #0          ;data page 0
      RPT     #63         ;Output 64 values from a table at 800h
       LMMR   50h,800h    ;in data memory to port 50h.
      RET
```

7.7 Subroutines

Example 7–11 illustrates the use of a subroutine to determine the square root of a 16-bit number. The main routine executes to the point where the square root of a number should be taken. At this point, a delayed call (CALLD) is made to the subroutine, transferring control to that section of the program memory for execution and then returning to the calling routine via the delayed return (RETD) instruction when execution has completed.

This example shows several features of 'C5x instruction set. In particular, note the use of delayed-call (CALLD), delayed-return (RETD), and conditional-execute (XC) instructions. Due to the four-level-deep pipeline on 'C5x devices, normal branch instructions require 4 cycles to execute. Using delayed branches, only two cycles are required for execution. The XC instruction is useful where only one or two instructions are to be executed conditionally. In this example, notice how XC is used to avoid extra cycles due to branch instruction. Use of the XC instruction also helps in keeping the execution time of a routine constant, regardless of input conditions. This is because XC executes NOPs in place of instructions if conditions are not met.

Example 7–11. Square Root Computation Using XC

```
*  Autocorrelation
*  This routine performs a correlation of two vectors and then
*  calls a Square Root subroutine that will determine the RMS
*  amplitude of the wave form.
*
AUTOC
       .
       .
       .
       CALLD    SQRT        ;Call square root subroutine after
        MAR     *,AR0       ;executing next two instructions
        LACC    *           ;Get the value to be passed to SQRT
       .                    ;subroutine
       .
       .
*
*  Square Root Computation
*
*  This routine computes the square root of a number that is located
*  in the lower half of accumulator. The number is in Q15 format.
*
BRCR   .set 09h             ;DP=0
ST0    .set 60h             ;Internal RAM block B2
ST1    .set 61h
NUMBER.set 62h
TEMPR  .set 63h
GUESS  .set 64h
       .text
SQRT   SST  #0,ST0
       SST  #1,ST1          ;Save context
       LDP  #0
       SETC SXM             ;Set SXM=1
       SPM      1           ;Set PM mode for fractional arithmetic
       SACL NUMBER          ;Save the number
       LACL #0
       SACB                 ;Clear accumulator buffer
       SPLK #11,BRCR        ;initialize for 12 iterations
       SPLK #800h,GUESS     ;Set initial guess
```

```
        LACC   NUMBER
        SUB    #200h
        BCNDD  LOOP,LT    ;If NUMBER<200h then begin looping
         SPLK  #800h,TEMPR
        LACC   #4000h     ;Otherwise set initial guess
        SACL   GUESS      ;and temporary root to 4000h
        SACL   TEMPR
        SPLK   #14,BRCR   ;and increase iterations to 15
LOOP    RPTB   ENDLP-1    ;Repeat block
         SQRA  TEMPR      ;Square temporary root
         LACC  NUMBER,16
         SPAC              ;Acc=NUMBER-TEMPR**2
         NOP               ;Dead cycle for XC
         XC    2,GT        ;If NUMBER>TEMPR**2 skip next 2 instr.
         LACC  TEMPR,16
         SACB              ;Otherwise ROOT <- TEMPR
         LACC  GUESS,15
         SACH  GUESS       ;GUESS <- GUESS/2
         ADDB
         SACH  TEMPR       ;TEMPR <- GUESS+ROOT
ENDLP   LACB               ;High Acc contains square root of NUMBER
        RETD
         LST   #1,ST1
         LST   #0,ST0      ;Restore context
```

Note that the restore is done with the LST instruction to prevent ARP from being overwritten. If indirect addressing is used, the order is reversed.

7.8 Extended-Precision Arithmetic

Numerical analysis, floating-point computations, or other operations may require arithmetic to be executed with more than 32 bits of precision. Since the 'C5x devices are 16/32-bit fixed-point processors, software is required for the extended precision of arithmetic operations. Subroutines that perform the extended-arithmetic functions for 'C5x are provided in the examples of this section. The technique consists of performing the arithmetic by parts, similar to the way in which longhand arithmetic is done.

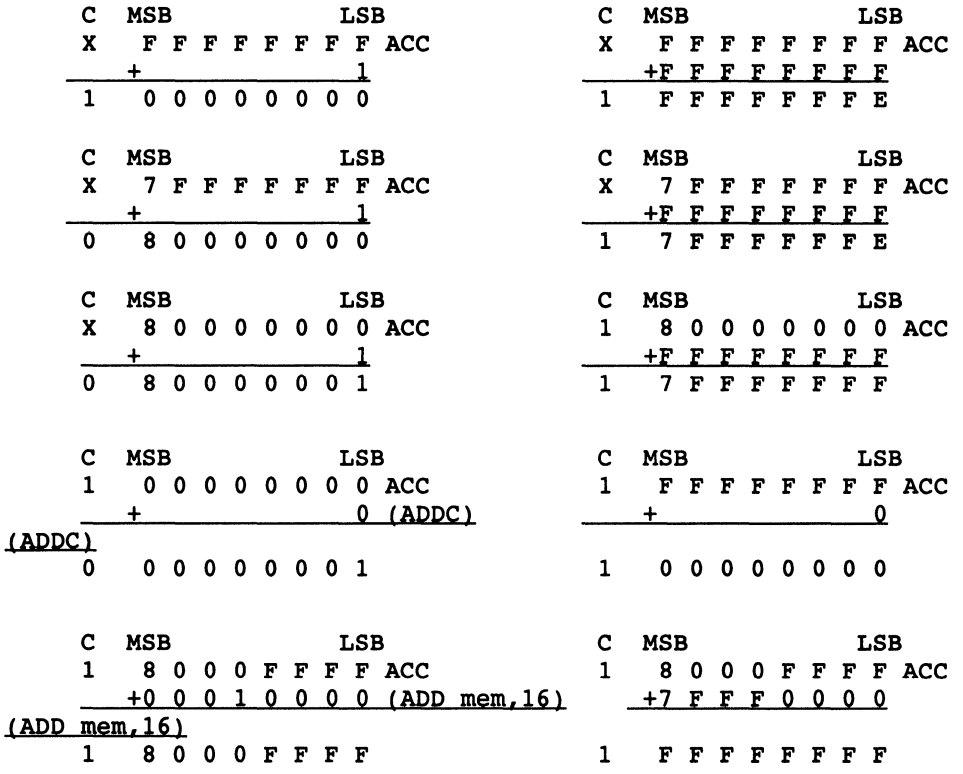
The 'C5x has several features that help make extended-precision calculations more efficient. One of the features is the carry bit. This bit is affected by all arithmetic operations of the accumulator, including addition and subtraction with the accumulator buffer. This allows 32-bit-long arithmetic operations using the accumulator buffer as the second operand.

The carry bit is also affected by the rotate and shift accumulator instructions. It may also be explicitly modified by the load status register ST1 and the set/reset control bit instructions. For proper operation, the overflow mode bit should be reset (OVM = 0) so that the accumulator results is not loaded with the saturation value.

7.8.1 Addition and Subtraction

The carry bit is set whenever the addition of a value from the input scaling shifter, the P register, or the accumulator buffer to the accumulator contents generates a carry out of bit 31. Otherwise, the carry bit is reset because the carry out of bit 31 is a zero. One exception to this case is the addition to the accumulator with a shift of 16 instruction (ADD mem,16), which can only set the carry bit. This allows the ALU to generate a proper single carry when the addition either to the lower or the upper half of the accumulator actually causes the carry. The following examples help to demonstrate the significance of the carry bit of the 'C5x for additions:

Figure 7–1. 32-Bit Addition



Example 7–12 shows an implementation of two 64-bit numbers added to each other to obtain a 64-bit result.

Example 7–12. 64-Bit Addition

```
*
* Two 64-bit numbers are added to each other producing a
* 64-bit result. The number X (X3,X2,X1,X0) and Y
* (Y3,Y2,Y1,Y0) are added resulting in W (W3,W2,W1,W0).
* If the result is required in 64-bit ACC/ACCB pair,
* replace the instructions as indicated in the comments
* below.
*
*   X3 X2 X1 X0
* + Y3 Y2 Y1 Y0
*   -----------
*   W3 W2 W1 W0 -OR- ACC ACCB
*

ADD64   LACC    X1,16   ;ACC = X1 00
        ADDS    X0      ;ACC = X1 X0
        ADDS    Y0      ;ACC = X1 X0 + 00 Y0
        ADD     Y1,16   ;ACC = X1 X0 + Y1 Y0
        SACL    W0      ;THESE 2 INSTR ARE REPLACED BY
        SACH    W1      ;"SACB" IF RESULT IS DESIRED IN (ACC ACCB)
        LACC    X3,16   ;ACC = X3 00
        ADDC    X2      ;ACC = X3 X2 + C
```

```
        ADDS    Y2       ;ACC = X3 X2 + 00 Y2 + C
        ADD     Y3,16    ;ACC = X3 X2 + Y3 Y2 + C
        SACL    W2       ;THESE 2 INSTR ARE NOT REQUIRED IF
        SACH    W3       ;THE RESULT IS DESIRED IN (ACC ACCB)
        RET
```

As in addition, the carry bit on the 'C5x is reset whenever the input scaling shifter, the P register, or the accumulator buffer value subtracted from the accumulator contents generates a borrow into bit 31. Otherwise, the carry bit is set because no borrow into bit 31 is required. One exception to this case is the SUB mem,16 instruction, which can only reset the carry bit. This allows the generation of the proper single carry when the subtraction from either the lower or the upper half of the accumulator actually causes the borrow. The examples in Figure 7–2 demonstrate the significance of the carry bit for subtraction.

Figure 7–2. 32-Bit Subtraction

```
      C  MSB               LSB                  C  MSB               LSB
      X   0 0 0 0 0 0 0 0 ACC                   X   0 0 0 0 0 0 0 0
ACC
     _____–________________1                  ______–F F F F F F F F
      0   F F F F F F F F                       0   0 0 0 0 0 0 0 1

      C  MSB               LSB                  C  MSB               LSB
      X   7 F F F F F F F ACC                   X   7 F F F F F F F
ACC
     _____–________________1                  ______–F F F F F F F F
      1   7 F F F F F F E                       C   8 0 0 0 0 0 0 0

      C  MSB               LSB                  C  MSB               LSB
      X   8 0 0 0 0 0 0 0 ACC                   X   8 0 0 0 0 0 0 0
ACC
     _____–________________1                  ______–F F F F F F F F
      1   7 F F F F F F F                       0   8 0 0 0 0 0 0 1

      C  MSB               LSB                  C  MSB               LSB
      0   0 0 0 0 0 0 0 0 ACC                   0   F F F F F F F F
ACC
     _____–________________0 (SUBB)           ______–________________0
(SUBB)
      0   F F F F F F F F                       1   F F F F F F F E

      C  MSB               LSB                  C  MSB               LSB
      0   8 0 0 0 F F F F ACC                   0   8 0 0 0 F F F F
ACC
     _____–0 0 0 1 0 0 0 0 (SUB mem,16)       ______–F F F F 0 0 0 0
(SUB mem,16)
      0   7 F F F F F F F                       0   8 0 0 1 F F F F
```

Example 7–13 implements the subtraction of two 64-bit numbers on the 'C5x. A borrow is generated within the accumulator for each of the 16-bit parts of the subtraction operation.

Example 7–13. 64-Bit Subtraction

```
*
 Two 64-bit numbers are subtracted, producing a 64-bit
* result. The number Y (Y3,Y2,Y1,Y0) is subtracted from
* X (X3,X2,X1,X0) resulting in W (W3,W2,W1,W0).
* If the result is required in 64-bit ACC/ACCB pair,
* replace the instructions as indicated in the comments
* below.
*
*   X3 X2 X1 X0
* - Y3 Y2 Y1 Y0
*  ------------
*   W3 W2 W1 W0 -OR- ACC ACCB
*

SUB64   LACC    X1,16   ; ACC = X1 00
        ADDS    X0      ; ACC = X1 X0
        SUBS    Y0      ; ACC = X1 X0 - 00 Y0
        SUB     Y1,16   ; ACC = X1 X0 - Y1 Y0
        SACL    W0      ; THESE 2 INSTR ARE REPLACED BY
        SACH    W1      ; "SACB" IF RESULT IS DESIRED IN (ACC ACCB)
        LACL    X2      ; ACC = 00 X2
        SUBB    Y2      ; ACC = 00 X2 - 00 Y2 - C
        ADD     X3,16   ; ACC = X3 X2 - 00 Y2 - C
        SUB     Y3,16   ; ACC = X3 X2 - Y3 Y2 - C
        SACL    W2      ; THESE 2 INSTR ARE NOT REQUIRED IF
        SACH    W3      ; THE RESULT IS DESIRED IN (ACC ACCB)
        RET
```

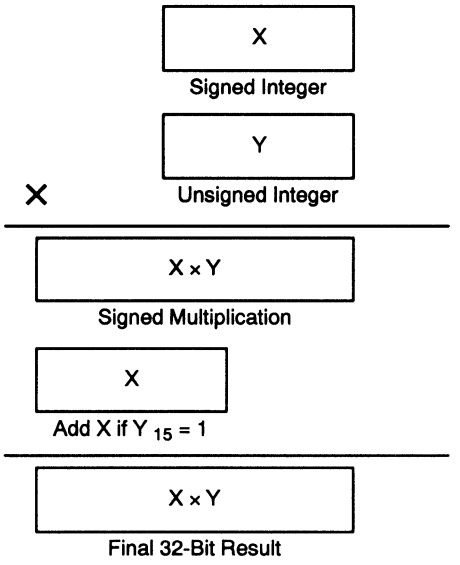
7.8.2 Multiplication

Another important feature that aids in extended-precision calculations is the MPYU (unsigned multiply) instruction. The MPYU instruction allows two unsigned 16-bit numbers to be multiplied and the 32-bit result placed in the product register in a single cycle. Efficiency is gained by generating partial products from the 16-bit portions of a 32-bit or larger value instead of having to split the value into 15-bit or smaller parts.

Further efficiency is gained by using the accumulator buffer to hold partial results instead of using a temporary location in data memory. The ability of 'C5x devices to barrel-shift the accumulator by 1 to 16 bits in only one cycle is also useful for scaling and justifying operands.

For 16-bit integer multiplication, in which one operand is a 2s-complement signed integer and the other one is an unsigned integer, you can use the algorithm shown in Figure 7–3.

Figure 7–3. 16-Bit Integer Multiplication



Steps required:

1) Multiply two operands X and Y as if they are signed integers.

2) If MSB of the unsigned integer Y is 1, add X to the upper half of the 32-bit signed product.

The correction factor must be added to the signed multiplication result because the bit weight of the MSB of any 16-bit unsigned integer is 2^{15}.

Consider following representation of a signed integer X and an unsigned integer Y:

$$X = -2^{15}x_{15} + 2^{14}x_{14} + 2^{13}x_{13} + \ldots + 2^{1}x_{1} + 2^{0}x_{0}$$

$$Y = 2^{15}y_{15} + 2^{14}y_{14} + 2^{13}y_{13} + \ldots + 2^{1}y_{1} + 2^{0}y_{0}$$

Multiplication of X and Y yields:

$$X \times Y = X \times (2^{15}y_{15} + 2^{14}y_{14} + 2^{13}y_{13} + \ldots + 2^{1}y_{1} + 2^{0}y_{0})$$

$$= 2^{15}y_{15}X + 2^{14}y_{14}X + 2^{13}y_{13}X + \ldots + 2^{1}y_{1}X + 2^{0}y_{0}X \quad (1)$$

However, if X and Y are considered signed integers, their multiplication yields:

$$X \times Y = X \times (-2^{15}y_{15} + 2^{14}y_{14} + 2^{13}y_{13} + \ldots + 2^{1}y_{1} + 2^{0}y_{0})$$

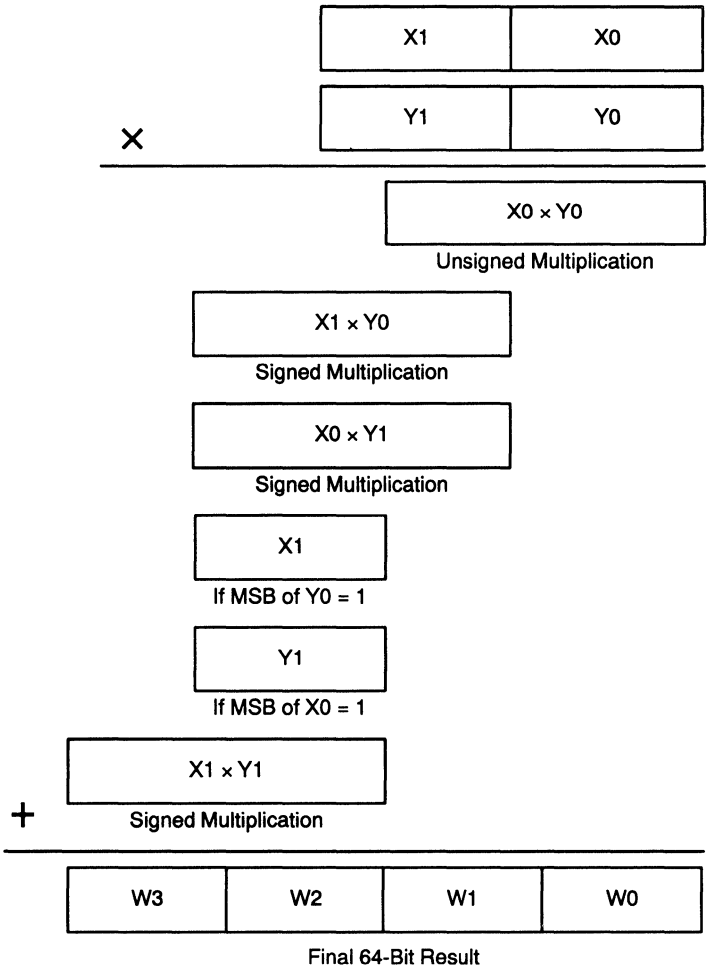
$$= -2^{15}y_{15}X + 2^{14}y_{14}X + 2^{13}y_{13}X + \ldots + 2^{1}y_{1}X + 2^{0}y_{0}X \quad (2)$$

The difference between (1) and (2) is in the first term on the right-hand side of the two equations.

Hence, if we add the correction term, $2^{16}y_{15}X$, to equation (2), the result would be identical to that of equation (1) and is the correct result.

This method of multiplying a signed integer with an unsigned integer can be used to implement extended-precision multiplication on 'C5x. The following description of a 32-bit multiplication algorithm is based on this method:

Figure 7–4. 32-Bit Multiplication Algorithm



The following example implements this algorithm. The product is a 64-bit integer number. Note in particular, the use of BSAR and XC instructions.

Example 7–14. 32-Bit Integer Multiplication

```
   .title "32–bit Optimized Integer Multiplication"
   .def    MPY32

*
* This routine multiplies two 32–bit signed integers result–
* ing in a 64–bit product. The operands are fetched from
* data memory and the result is written back to data memory.
* Data Storage:
*    X1,X0          32–bit operand
*    Y1,Y0          32–bit operand
*    W3,W2,W1,W0    64–bit product
* Entry Conditions:
*    DP  = 6, SXM = 1
*    OVM = 0
*

X1      .set    300h    ;DP=6
X0      .set    301h    ;DP=6
Y1      .set    302h    ;DP=6
Y0      .set    303h    ;DP=6
W3      .set    304h    ;DP=6
W2      .set    305h    ;DP=6
W1      .set    306h    ;DP=6
W0      .set    307h    ;DP=6

        .text

MPY32:
     BIT    X0,0  ;TC = X0 bit#15
     LT     X0    ;T = X0
     MPYU   Y0    ;P = X0Y0
     SPL    W0    ;Save W0
     SPH    W1    ;Save partial W1
     MPY    Y1    ;P = X0Y1
     LTP    X1    ;Acc = X0Y1, T = X1
     MPY    Y0    ;P = X1Y0
     MPYA   Y1    ;Acc = X0Y1+X1Y0, P=X1Y1
     ADDS   W1    ;Acc = X0Y1+X1Y0+X0Y02^–16
     SACL   W1    ;Save final W1
     BSAR   16    ;Shift Acc right by 16
     XC     1,TC  ;If MSB of X0 is 1
      ADD   Y1    ;Add Y1
     BIT    Y0,0  ;TC = Y0 bit#15
     APAC         ;ACC = X1Y1 + (X0Y1+X1Y0)2^–16
     XC     1,TC  ;IF MSB of Y0 is 1
      ADD   X1    ;Add X1
     SACL   W2    ;Save W2
     SACH   W3    ;Save W3
```

The next example performs fractional multiplication. The operands are in Q31 format, while the product is in Q30 format.

Example 7–15. 32-Bit Fractional Multiplication

```
   .title "32–bit Fractional Multiplication"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine multiplies two Q31 signed integers resulting
; in a Q30 product. The operands are fetched from data
; memory and the result is written back to data memory.
; Data Storage:
;    X1,X0          Q31 operand
;    Y1,Y0          Q31 operand
;    W1,W0          Q30 product
; Entry Conditions:
;    DP  = 6, SXM = 1
;    OVM = 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

X1      .set    300h    ;DP=6
X0      .set    301h    ;DP=6
Y1      .set    302h    ;DP=6
Y0      .set    303h    ;DP=6
W1      .set    304h    ;DP=6
W0      .set    305h    ;DP=6

        .text

      BIT   X0,0  ; TC = X0 bit#15
      LT    X0    ; TREG0 = X0
      MPY   Y1    ; P = X0*Y0
      LTP   X1    ; Acc = X0*Y0
      MPY   Y0    ; P = X1*Y0
      MPYA  Y1    ; Acc = X0*Y0 + X1*Y0
      BSAR  16    ; Throw away low 16 bits
      XC    1,TC  ; If MSB of X0 is 1
       ADD  Y1    ; then add Y1
      BIT   Y0,0  ; TC = Y0 bit#15
      APAC        ; Acc = Acc + X1*Y1
      XC    1,TC  ; If MSB of Y0 is 1
       ADD  X1    ; then add X1
      SACL  W0    ; Save lower product
      SACH  W1    ; Save upper product
```

7.8.3 Division

Integer and fractional division is implemented on the 'C5x by repeated subtractions executed with SUBC, a special conditional subtract instruction. Given a 16-bit positive dividend and divisor, the repetition of the SUBC command 16 times produces a 16-bit quotient in the low accumulator and a 16-bit remainder in the high accumulator.

SUBC implements binary division in the same manner as long division is done. The dividend is shifted until subtracting the divisor no longer produces a negative result. For each subtract that does not produce a negative answer, a one is put in the LSB of the quotient and then shifted. The shifting of the remainder and quotient after each subtract produces the separation of the quotient and remainder in the low and high halves of the accumulator.

Both the dividend and the divisor must be positive when using the SUBC command. Thus, the sign of the quotient must be determined and the quotient computed by using the absolute value of the dividend and divisor.

Integer and fractional division can be implemented with the SUBC instruction as shown in Example 7–16 and Example 7–17, respectively. When implementing a divide algorithm, it is important to know if the quotient can be represented as a fraction and the degree of accuracy to which the quotient is to be computed. For integer division, the absolute value of the numerator must be greater than the absolute value of the denominator. For fractional division, the absolute value of the numerator must be less than the absolute value of the denominator.

Long Division:

```
                     0000000000000110    Quotient
0000000000000101    )0000000000010001
                                 -101
                                  110
                                 -101
                                   11    Remainder
```

SUBC Method:

32 HIGH ACC	LOW ACC 0	Comment
0000000000000000	0000000000100001	(1) Dividend is loaded into ACC. The divisor is left-shifted 15 and subtracted from ACC. The subtraction is negative, so discard the result and shift left the ACC one bit.
−10	1000000000000000	
−10	0111111111011111	
0000000000000000	0000000001000001	(2) 2nd subtract produces negative answer, so discard result and shift ACC (dividend) left.
−10	1000000000000000	
−10	0111111110111110	
•		•
•		•
•		•
0000000000000100	0010000000000000	(14) 14th SUBC command. The result is positive. Shift result left and replace LSB with 1.
−10	1000000000000000	
0000000000000001	1010000000000000	
0000000000000011	0100000000000001	(15) Result is again positive. Shift result left and replace LSB with 1.
−10	1000000000000000	
0000000000000000	1100000000000001	
0000000000000001	1000000000000011	(16) Last subtract. Negative answer, so discard result and shift ACC left.
−10	1000000000000000	
	− 1111111111111101	
0000000000000011	0000000000000110	Answer reached after 16 SUBC instructions.
Remainder	Quotient	

Example 7–16. Integer Division Using SUBC

```
*
* This routine implements integer division with the SUBC instruction. For this
* integer division routine, the absolute value of the numerator must be greater
* than the absolute value of the denominator.  In addition, the calling routine
* must check to verify that the divisor does not equal 0.
*
* The 16–bit dividend is placed in the low accumulator, and the high accumulator
* is zeroed.  The divisor is in data memory.  At the completion of the last
* SUBC, the quotient of the division is in the lower–order 16–bits of the
* accumulator. The remainder is in the higher–order 16–bits.
*
* Key C5x Instruction:
* RETCD  return if conditions true – after executing next 2–word instruction or
* two single–word instructions
*

DENOM   .set    60h
NUMERA  .set    61h
QUOT    .set    62h
REM     .set    63h
TEMSGN  .set    64h
*
INTDIV  LDP     #0
        LT      NUMERA          ;Determine sign of quotient.
        MPY     DENOM
*
        SPH     TEMSGN          ;Save the sign
        LACL    DENOM
        ABS                     ;Make denominator and numerator positive.
        SACL    DENOM           ;Save absolute value of denominator
        LACL    NUMERA
        ABS
*
* If divisor and dividend are aligned, division can start here.
*
        RPT     #15             ;16 cycle division. Low accumulator contains
         SUBC   DENOM           ;the quotient and high accumulator contains the
*                               ;remainder at the end of the loop.
        BIT     TEMSGN,0        ;Test sign of quotient.
        RETCD   NTC             ;Return if sign positive, else continue.
         SACL   QUOT            ;Store quotient and remainder during delayed
         SACH   REM             ;return.
*
        LACL    #0              ;If sign negative, negate quotient
        RETD                    ;and return
         SUB    QUOT
         SACL   QUOT
```

Example 7–17. Fractional Division Using SUBC

```
*
* This routine implements fractional division with the SUBC instruction. For
* this division routine, the absolute value of the denominator must be
* greater than the absolute value of the numerator. In addition, the
* calling routine must check to verify that the divisor does not equal 0.
*
* The 16-bit dividend is placed in the high accumulator, and the low accumulator
* is zeroed. The divisor is in data memory.
*

DENOM       .set    60h
NUMERA      .set    61h
QUOT        .set    62h
REM         .set    63h
TEMSGN      .set    64h
*
FRACDIV     LDP     #0
            LT      NUMERA      ;Determine sign of quotient.
*
            MPY     DENOM
            SPH     TEMSGN
            LACL    DENOM
            ABS                 ;Make denominator and numerator positive.
            SACL    DENOM
            LACC    NUMERA,16   ;Load high accumulator, zero low accumulator.
            ABS
*
* If divisor and dividend are aligned, division can start here.
*
            RPT     #15         ;16-cycle division. Low accumulator contains
             SUBC   DENOM       ;the quotient and high accumulator contains the
                                ;remainder at the end of the loop.
*
            BIT     TEMSGN,0    ;Test sign of quotient.
            RETCD   NTC         ;Return if sign positive, else continue.
             SACL   QUOT        ;Store quotient and remainder during delayed
             SACH   REM         ;return.
*
            LACL    #0          ;If sign negative, negate quotient
            RETD                ;and return
             SUB    QUOT
             SACL   QUOT
```

7.9 Floating-Point Arithmetic

To implement floating-point arithmetic on the 'C5x, operands must be converted to fixed point for arithmetic operations and then converted back to floating point. Conversion to floating-point notation is performed by normalizing the input data.

To multiply two floating-point numbers, the mantissas are multiplied and the exponents added. The resulting mantissa must be renormalized. Floating-point addition or subtraction requires shifting the mantissa so that the exponents of the two operands match. The difference between the exponents is used to left-shift the lower power operand before adding. Then, the output of the add must be renormalized.

The 'C5x instructions used in floating-point operations are NORM, SATL, SATH, and XC. NORM may be used to convert fixed-point numbers to floating-point. SATL in combination with SATH provides a two-cycle 0–31-bit right shift. XC helps avoid extra cycles caused by branch instructions.

Example 7–18 and Example 7–19 show how to implement floating-point arithmetic on 'C5x devices. Floating-point numbers are generally represented by mantissa and exponent values. Single-precision IEEE floating-point numbers are represented by a 24-bit mantissa, an 8-bit exponent, and a sign bit. In order to simplify the routines, a format slightly different from the IEEE format is used. Four words are occupied by each floating-point number. One sign word, one word for exponent, and two words for mantissa are reserved in memory as described in the code below.

Example 7–18. Floating-Point Addition Using SATL and SATH

```
    .title  'Floating Point Addition Algorithm'
    .def    FL_ADD

*;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
*   THIS SUBROUTINE ADDS TWO FLOATING–POINT NUMBERS PRODUCING
*   A NORMALIZED FLOATING–POINT PRODUCT. THE FORMAT OF FLOATING–
*   POINT NUMBERS IS SPECIFIED BELOW.
*
*   INPUT / OUTPUT FORMAT
*   =====================
*   ________________
*   |  ALL 0 OR 1  |   SIGN WORD
*   ________________
*
*   ________________
*   |   16 BITS    |   EXPONENT
*   ________________
*
*   ________________
*   |0|   15 BITS  |   HIGH PART OF MANTISSA
*   ________________
*
*   ________________
*   |    16 BITS   |   LOW PART OF MANTISSA
*   ________________
*
*   Key C5x Instructions:
```

```
*
*   SAMM    save the accumulator contents in a memory-mapped
*       register
*   LACB    accumulator is loaded with contents of accumulator
*       buffer
*   SACB    contents of accumulator are copied in accumulator
*       buffer
*   SATL    accumulator is barrel-shifted right by the value
*       specified in the 4 LSBs of TREG1
*   SATH    accumulator is barrel-shifted right by 16 bits
*       if bit 4 of TREG1 is a one.
*   SPLK    store immediate long constant in data memory
*   CPL compare long immediate value (or DBMR) with data
*       memory
*       TC=1 if two values are same
*       TC=0 otherwise
*

TREG1   .set    0dh

ASIGN   .set    60h         ;Sign, exponent, high and low part of mantissa
AEXP    .set    61h         ;of input number A
AHI     .set    62h
ALO     .set    63h

BSIGN   .set    64h         ;Sign, exponent, high and low part of mantissa
BEXP    .set    65h         ;of input number B
BHI     .set    66h
BLO     .set    67h

CSIGN   .set    68h         ;Sign, exponent, high and low part of mantissa
CEXP    .set    69h         ;of the resulting floating point number C
CHI     .set    6Ah
CLO     .set    6Bh
DIFFEXP .set    6Ch

    .text

FL_ADD  LDP     #0          ;Initialization
        SETC    SXM         ;Set sign extension mode
        MAR     *,AR0       ;ARP <- AR0
        LAR     AR0,#0      ;AR0 is used by NORM instruction

CMPEXP  LACL    BLO         ;Load low Acc with BLO
        ADD     BHI,16      ;Add BHI to high Acc
        SACB                ;AccB = BHIBLO
        LACC    AEXP
        SUB     BEXP        ;Acc = AEXP-BEXP
        SACL    DIFFEXP     ;Save the difference
        BCND    AEQB,EQ     ;If |A| == |B|
        BCND    ALTB,LT     ;If |A| < |B|

AGTB    LACC    DIFFEXP     ;If |A| > |B|
        SAMM    TREG1       ;Load TREG1 with # of right shifts reqd.
        SUB     #32
        BCND    AGRT32,GEQ  ;If difference > 32
        LACB                ;Acc = BHIBLO
        SATL
        SATH                ;Right justify BHIBLO
        SACB                ;Store the result back in AccB

AEQB    LACC    ASIGN       ;Copy sign and exponent values of
        SACL    CSIGN       ;A in C (i.e. the result)
        LACC    AEXP
        SACL    CEXP

CHKSGN  LACC    ASIGN       ;Acc=ASIGN-BSIGN
        SUB     BSIGN
        CLRC    TC          ;Clear TC flag
        XC      1,LT        ;If A<0 and B>0
         SETC   TC          ;Set TC flag
        BCNDD   ADNOW,EQ    ;If both A and B have same sign
        LACL    ALO
```

```
        ADD     AHI,16          ;Acc = AHIALO
        SBB                     ;Acc=A-B
        XC      1,TC            ;If A<0 and B>0
         NEG                    ;then Acc=B-A
        BCND    CZERO,EQ        ;If A-B == 0
        XC      2,LT            ;If A-B < 0
         SPLK   #0FFFFH,CSIGN   ; then CSIGN=-1
        XC      2,GT            ;If A-B > 0
         SPLK   #0,CSIGN        ; then CSIGN=0
        XC      1,LT            ;If A-B<0
         ABS                    ; then Acc=|A-B|
        BD      NORMAL          ;delayed branch
         SACH   CHI             ;Save the result
         SACL   CLO

CZERO   LACL    #0              ;If A-B == 0
        SACL    CEXP            ;then result is zero
        SACL    CSIGN           ;Make sign positive
        RETD                    ;Return delayed
        SACL    CHI
        SACL    CLO             ;Clear CHICLO

ADNOW   ADDB                    ;If signs are same
        BCNDD   OVFLOW,OV       ;then add two numbers
         SACH   CHI
         SACL   CLO             ;Save it in CHICLO
        BCND    CZERO,EQ        ;If CHICLO is zero, goto CZERO

NORMAL  CPL     #0,CHI          ;Compare CHI with 0
        NOP                     ;Dead cycle for XC
        XC      2,TC            ;If CHI is 0
         LACC   CLO,16          ;then normalize only the CLO part
        LAR     AR0,#16         ;AR0 has exponent value
        XC      2,NTC           ;If CHI != 0
         LACC   CHI,16          ;Acc=CHICLO
         ADDS   CLO
        CLRC    SXM             ;Disable sign extension mode
        XC      2,LT            ;If MSB of CLO is 1
         SBRK   1               ;then shift right once
         SFR                    ;and decrement exponent.
        SETC    SXM             ;Enable sign extension mode
        RPT     #13             ;Repeat 14 times
         NORM   *+              ;Normalize

OUTPUT  SACH    CHI             ;Store high part
        SACL    CLO             ;Store low part of the result
        LACC    CEXP
        SAR     AR0,CEXP        ;Save exponent
        RETD                    ;Return delayed
         SUB    CEXP
         SACL   CEXP            ;CEXP=CEXP-AR0

OVFLOW  CLRC    SXM             ;Disable sign extension mode
        SFR                     ;Shift Acc right
        SACH    CHI
        SACL    CLO             ;Save the result
        LACC    CEXP
        ADD     #1              ;Increment exponent by one
        SACL    CEXP            ;Save it

ALTB    LACC    BSIGN           ;Copy sign of B in C
        SACL    CSIGN
        LACC    BEXP            ;Copy exponent of B in C
        SACL    CEXP
        LACC    DIFFEXP
        NEG                     ;since A-B < 0 here
        SAMM    TREG1           ;No. of shifts reqd. for right-justification
        SUB     #32
        BCND    BGRT32,GEQ      ;difference in exponent >= 32
        LACL    ALO
        ADD     AHI,16          ;Acc=AHIALO
        SATL
```

```
        SATH                    ;Right-justify ALOAHI
        BD      CHKSGN          ;Jump back after next two instructions
         SACL   ALO             ;Save normalized value
         SACH   AHI             ;in ALO and AHI

BGRT32  LACC    BHI             ;If exponent of B > 32
        SACL    CHI             ;then C <- B.
        RETD                    ;Return after
         LACC   BLO             ;saving CHI and CLO
         SACL   CLO

AGRT32  LACC    AHI             ;If exponent of A > 32
        SACL    CHI             ;then C <- A.
        LACC    ALO
        SACL    CLO             ;Copy ALO to CLO
        LACC    ASIGN
        SACL    CSIGN           ;Copy ASIGN to CSIGN
        RETD                    ;Return after
         LACC   AEXP            ;copying AEXP to CEXP
         SACL   CEXP
```

Example 7–19. Floating-Point Multiplication Using BSAR

```
        .title  'Floating Point Multiplication Routine'
*
*   THIS SUBROUTINE MULTIPLIES TWO FLOATING-POINT NUMBERS PRODUCING
*   A NORMALIZED FLOATING-POINT PRODUCT. THE FORMAT OF FLOATING-
*   POINT NUMBERS IS SPECIFIED BELOW.
*
*   INPUT / OUTPUT FORMAT
*   =====================
*   ------------------
*   |  ALL 0 OR 1    |    SIGN WORD
*   ------------------
*
*   ------------------
*   |    16 BITS     |    EXPONENT
*   ------------------
*
*   ------------------
*   |0|    15 BITS   |    HIGH PART OF MANTISSA
*   ------------------
*
*   ------------------
*   |    16 BITS     |    LOW PART OF MANTISSA
*   ------------------
*
*   NOTE THAT EVEN IF THE PRODUCT IS ZERO, SIGN OF THE PRODUCT MAY
*   EITHER BE POSITIVE OR NEGATIVE DEPENDING ON THE INPUTS.
*
*   Key C5x Instructions:
*   BSAR    1-16 bit right barrel arithmetic shift in one cycle
*   CLRC    reset control bit
*   SETC    set control bit
*   BD  branch after executing next two one-word instructions
*       or one two-word instruction
*
ASIGN   .set    60h         ;Sign, exponent, high and low parts of mantissa
AEXP    .set    61h         ;of input number A
AHI     .set    62h
ALO     .set    63h

BSIGN   .set    64h         ;Sign, exponent, high and low parts of mantissa
BEXP    .set    65h         ;of input number B
BHI     .set    66h
BLO     .set    67h

CSIGN   .set    68h         ;Sign, exponent, high and low parts of mantissa
CEXP    .set    69h         ;of the resulting floating point number C
```

```
CHI     .set    6ah
CLO     .set    6bh

    .text

MULT    LDP     #0
        MAR     *,AR0           ;ARP <- AR0
        LAR     AR0,#0          ;Reset exponent counter
        SPM     0               ;No left shift of P register
        LACC    AEXP
        ADD     BEXP
        SACL    CEXP            ;CEXP = AEXP + BEXP
        CLRC    SXM             ;for barrel shift, disable sign extension
        LT      ALO             ;T = ALO
        MPYU    BHI             ;P = ALO*BHI
        LTP     AHI             ;Acc=ALO*BHI, T=AHI
        MPYU    BLO             ;P=AHI*BLO
        MPYA    BHI             ;Acc=ALO*BHI + AHI*BLO, P=AHI*BHI
        BSAR    16              ;Retain upper 16 bits plus 1 additional
        APAC                    ;bit due to zero MSBs of BLO & ALO
        BCND    NZERO,NEQ       ;If the product is not zero
        SACH    CHI             ;If the product is zero
        BD      SIGN            ;then clear CHI,CLO and CEXP
         SACL   CLO             ;and jump to SIGN
         SACL   CEXP
NZERO   SFL                     ;Discard additional sign bit (Q63)
        NORM    *+              ;Remove leading zero if any
        SACH    CHI             ;Save product
        SACL    CLO
        SETC    SXM             ;Enable sign extension mode
        LACC    CEXP
        SAR     AR0,CEXP        ;CEXP<-AR0
        SUB     CEXP
        SACL    CEXP            ;CEXP=CEXP-AR0
SIGN    LACL    ASIGN           ;If signs are same then product is +ve
        RETD                    ;Return after next two instructions
         XOR    BSIGN           ;otherwise it is -ve.
         SACL   CSIGN
```

7.10 Application-Oriented Operations

7.10.1 Modem Application

Digital signal processors are especially appropriate for modem applications. The 'C5x devices with their enhanced instruction set and reduced instruction cycle time are particularly effective in implementing encoding and decoding algorithms. Features like circular addressing, repeat block, and single-cycle barrel shift reduce the execution time of such routines.

Example 7–20 implements a differential and convolutional encoder for a 9600-bit/s V.32 modem. This encoder uses trellis coding with 32 carrier states. The data stream to be transmitted is divided into groups of four consecutive data bits. The first two bits in time $Q1_n$ and $Q2_n$ in each group are differentially encoded into $Y1_n$ and $Y2_n$ according to the following equations:

$$Y1_n = Q1_n \oplus Y1_{n-1}$$

$$Y2_n = (Q1_n \bullet Y1_{n-1}) \oplus Y2_{n-1} \oplus Q2_n$$

This is done by a subroutine called DIFF. The two differentially encoded bits Y1n and Y2n are used as inputs to a convolutional encoder subroutine ENCODE, which generates a redundant bit Y0n. These five bits are packed into a single word by the PACK subroutine.

Example 7–20. V.32 Encoder Using Accumulator Buffer

```
        .title  'Convolutional Encoding for a V.32 Modem'
        .mmregs
STATMEM .set    60h             ;(60h - 62h) Delay States S1,S2,S3
INPUT   .set    64h             ;(64h - 67h) Four input bits
YPAST   .set    68h             ;(68h - 69h) Past values of Y1 and Y2
OUTPUT  .set    63h             ;Y0, the redundant bit
LOCATE  .set    6ah             ;Temporary storage for current input word
PCKD_IP .set    1000h           ;Input buffer (4 bits packed per word)
PCKD_OP .set    2000h           ;Output buffer (5 bits packed per word)
COUNT   .set    50              ;# of input data words
        .text
INIT    LAR     AR1,#PCKD_IP
        LAR     AR2,#PCKD_OP
        LAR     AR3,#COUNT-1    ;COUNT contains # of input words
        LDP     #0
START   MAR     *,AR1
        LACC    *+,0,AR0
        SACL    LOCATE          ;Temporary storage for current input word
        LAR     AR0,#INPUT+3
        LACL    #3              ;Loop 4 times
        SAMM    BRCR
        LACL    #1
        SAMM    DBMR            ;Load DBMR with the mask for LSB
UNPACK  LACC    LOCATE          ;Acc = packed input bits
        RPTB    LOOP1-1         ;for I=0,I<=3,I++
```

```
            SACL   *              ;Save it
            APL    *-             ;Mask off all bits except LSB
            SFR                   ;Shift right to get next bit
LOOP1
            CALL   DIFF           ;Call differential encoder
            CALL   ENCODE         ;Call convolutional encoder

PACK        LAR    AR0,#INPUT
            LACL   #3             ;Loop 4 times only
            SAMM   BRCR
            LACC   *+             ;Get first bit (MSB)

            RPTB   LOOP2-1        ;for I=0,I<=2,I++
            SFL                   ;make space by left-shifting once
            ADD    *+             ;Pack next bit by left-shifting other
            NOP
LOOP2
            MAR    *,AR2          ;ARP <- AR2
            SACL   *+,0,AR3       ;Save it in packed form
            BANZ   START          ;Loop if COUNT is not zero
            RET                   ;Return

;  This subroutine differentially encodes Q1n and Q2n (INPUT
;  buffer) according to previous output values Y1n-1 and
;  Y2n-1 (YPAST buffer). The resulting values Y1n and Y2n overwrite
;  previous Q1n and Q2n.

DIFF        LACC   YPAST          ;Acc=Y1n-1
            AND    INPUT          ;Q1n & Y1n-1
            XOR    INPUT+1        ;(Q1n & Y1n-1) xor Q2n
            XOR    YPAST+1        ;(Q1n & Y1n-1) xor Q2n xor Y2n-1
            SACL   INPUT+1
            SACL   YPAST+1        ;Save Y2n
            LACC   YPAST
            XOR    INPUT          ;Q1n xor Y1n-1
            RETD                  ;Delayed return
            SACL   INPUT          ;Save Y1n
            SACL   YPAST          ;save Y1n-1

;  This subroutine generates a redundant bit Y0n by convolutional encoding,
;  taking Y1n and Y2n as input. Three delay states S1, S2 and S3 are
;  located in STATMEM buffer.

   ENCODE   LACC   STATMEM
            SACL   OUTPUT         ;Y0 <- S1
            LACC   INPUT+1
            XOR    STATMEM+1      ;Y2 xor S2
            SACB                  ;Save in AccB
            LACC   OUTPUT
            AND    INPUT          ;Y0 & Y1
            XORB                  ;(Y0 & Y1) xor (Y2 xor S2)
            SACL   STATMEM        ;Save it in S1
            LACC   OUTPUT
            ANDB                  ;Y0 & (Y2 xor S2)
            SACB
            LACC   INPUT
            XOR    INPUT+1        ;Y1 xor Y2
            XOR    STATMEM+2      ;(Y1 xor Y2) xor S3
            XORB                  ;((Y1 xor Y2) xor S3) xor (Y0 & (Y2 xor S2))
            SACL   STATMEM+1      ;Update S2
            RETD                  ;Delayed return
            LACC   OUTPUT
            SACL   STATMEM+2      ;Update S3
```

7.10.2 Adaptive Filtering

There are many practical applications of adaptive FIR/IIR filtering; one example is in the adapting or updating of coefficients. This can become computationally expensive and time-consuming. The MPYA, ZALR, and RPTB instructions on the 'C5x can reduce execution time.

A means of adapting the coefficients on the 'C5x is the least-mean-square algorithm given by the following equation:

$$b_k\ (i + 1) = b_k\ (i) + 2Be(i)x(i - k)$$

where $e\ (i) = x\ (i) - y\ (i)$
and

$$y(i) = \sum_{k=0}^{N-1} b_k\ x(i - k)$$

Quantization errors in the updated coefficients can be minimized if the result is obtained by rounding rather than truncating. For each coefficient in the filter at a given point in time, the factor 2*B*e(i) is a constant. This factor can then be computed once and stored in the T register for each of the updates.

MPYA and ZALR instructions help in reducing the number of instructions in the main adaptation loop. Furthermore, the RPTB (repeat block) instruction allows the block of instructions to be repeated without any penalty for looping.

Example 7–21 shows a routine that implements a 128-tap FIR filter and an LMS adaptation of its coefficients. The single-access internal RAM of the 'C50/C51 can be mapped in both the program and data spaces at the same time by setting OVLY and RAM control flags to 1. This feature can be used to advantage by locating the coefficients table in single-access internal RAM so that it can be accessed by MACD and MPY instructions without modifying RAM configuration. Note that the MACD instruction requires one of its operands to be in program space.

If the address of the coefficient table is to be determined in runtime, load the BMAR (block move address register) with the address computed dynamically and replace the instruction

```
MACD    COEFFP,*–
by
MADD    *–
```

Example 7–21. Adaptive FIR Filter Using RPT and RPTB

```
        .title 'Adaptive Filter'
        .def   ADPFIR
        .def   X,Y
        .mmregs
*
* This 128-tap adaptive FIR filter uses on-chip memory block B0 for
* coefficients and block B1 for data samples.  The newest input should
* be in memory location X when called. The output will be in memory location Y
* when returned.
* OVLY =1 , RAM =1 when this routine is called.
*
COEFFP  .set    02000h          ;Program memory address of the coeff. in S/A RAM
COEFFD  .set     02000h         ;Data memory address of the coeff. in S/A RAM
*   For TMS320C51, COEFFD is 0800h instead of 02000h
ONE     .set    7Ah             ;Constant one.              (DP=0).
BETA    .set    7Bh             ;Adaptation constant.       (DP=0).
ERR     .set    7Ch             ;Signal error.              (DP=0).
ERRF    .set    7Dh             ;Error function.            (DP=0).
Y       .set    7Eh             ;Filter output.             (DP=0).
X       .set    037Fh           ;Newest data sample.
FRSTAP  .set    0380h           ;Next newest data sample.
LASTAP  .set    03FFh           ;Oldest data sample.
*
* Finite impulse response (FIR) filter.
*
ADPFIR  ZPR                     ;Clear P register.
        LACC    #1,14           ;Load output rounding bit.
        MAR     *,AR3
        LAR     AR3,#LASTAP     ;Point to oldest sample.
FIR     RPT     #127
         MACD   COEFFP,*-       ;128-tap FIR filter.
        APAC
        SACH    Y,1             ;Store the filter output.
        NEG                     ;Acc = -y(n)
        LAR     AR3,#X
        ADD     *,15            ;Add the newest input sample.
        SACH    ERR,1           ;err(n) = x(n) - y(n)
        DMOV    *               ;Include newest sample
*
* LMS Adaption of Filter Coefficients.
*
        LT      ERR             ;T = err
        MPY     BETA            ;P = beta*err(i)
        PAC                     ;errf(i) = beta * err(i)
        ADD     ONE,14          ;Round the results.
        SACH    ERRF,1          ;Save errf(i)
*
        LACC    #126
        SAMM    BRCR            ;127 coefficients to update
*                                in the loop.
        LAR     AR2,#COEFFD     ;Point to the coefficients.
        LAR     AR3,#LASTAP     ;Point to the data samples.
        LT      ERRF
        MPY     *-,AR2          ;P = 2*beta*err(i)*x(i-255)
*
        RPTB    LOOP-1          ;For I=0,I<=126,I++
ADAPT    ZALR   *,AR3           ;Load ACCH with ak(i).
         MPYA   *-,AR2          ;P = 2*beta*err(i)*x(i-k-1)
*                Acc = ak(i) + 2*beta*err(i)*x(i-k)
         SACH   *+              ;Store ak(i+1)
*
LOOP    ZALR    *,AR3           ;Finally update last coeff. a0(i)
        RETD                    ;Delayed return
         APAC                   ;Acc = a0(i) + 2*beta*err(i)*x(i)
         SACH   *+              ;Save a0(i+1)
```

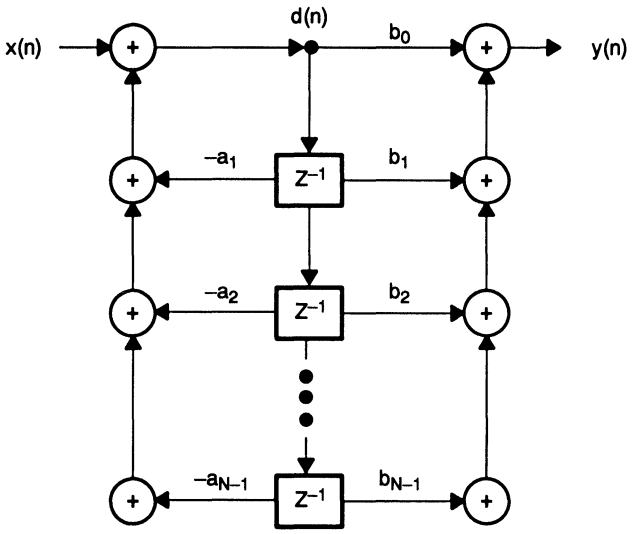
7.10.3 IIR Filters

Infinite impulse response (IIR) filters are widely used in digital signal processing applications. The transfer function of an IIR filter is given by:

$$H(z) = \frac{b_0 + b_1z^{-1} + ... + b_Mz^{-M}}{1 + a_1z^{-1} + ... + a_Nz^{-N}} = \frac{Y(z)}{X(z)}$$

Figure 7–5 shows a block diagram of an Nth order direct-form II IIR filter:

Figure 7–5. Nth Order Direct-Form Type II IIR Filter



In the time domain, an Nth order IIR filter is represented by the following two difference equations:

at time interval n:

x(n) is the current input sample

y(n) is the output of the IIR filter

$d(n) = x(n) - d(n-1)a_1 - ... - d(n-N+1)a_{N-1}$

$y(n) = d(n)b_0 + d(n-1)b_1 + ... + d(n-N+1)b_{N-1}$

The two equations above can easily be implemented on the 'C5x by using multiply-accumulate instructions (MAC, MACD, MADS, MADD). Note that the second equation would also require a data-move operation to update the state variable sequence d(n). Example 7–22 implements an Nth order IIR filter using single-instruction repeat (RPT) and multiply-accumulate (MAC, MACD) instructions.

Example 7–22. Using RPT and MACD

```
.title "Nth Order IIR Type II Filter"
.mmregs
```

```
*
* This routine implements an N-th order type II IIR filter.
*    d(n) = x(n) - d(n-1)a1 - d(n-2)a2 +...- d(n-N+1)aN-1
*    y(n) = d(n)b0 + (dn-1)b1 +...+ d(n-N+1)bN-1
* Memory Requirement:
*   State variables (low to high data memory):
*    d(n) d(n-1) ... d(n-N+1)
*   Coefficient (low to high program memory):
*    b(N-1) b(N-2) ... b(1) -a(N-1) -a(N-2) ... -a(1) -a(0)
* Entry Conditions:
*    AR0 -> Input
*    AR1 -> d(n-N+1)
*    AR2 -> Output
*    COEFFA -> -a(N-1)
*    COEFFB -> b(N-1)
*    ARP = AR0
*

IIR_N:
       ZPR                 ;Clear P register
       LACC   *,15,AR1     ;Get Q15 input
       RPT    #(N-2)       ;for i=1,i<=N-1,++i
        AC    COEFFB,*-    ;Acc+=-a(N-i))*d(n-N+i)
       APAC                ;Final accumulation
       SACH   *,1          ;Save d(n)
       ADRK   N-1          ;AR1 -> d(n-N+1)
       RPTZ   #(N-1)       ;for i=1,i<=N,++i
        MACD  COEFFA,*-    ;Acc+=b(N-i)*d(n-N+i)
       LTA    *,AR2        ;Final accumulation
       SACH   *,1          ;Save Yn
```

Due to the recursive nature of an IIR filter, quantization of filter coefficients may cause significant variation from the desired frequency response. To avoid this problem, the desired filter transfer function can be broken up into lower order sections that are cascaded with each other. The following example shows an implementation of N cascaded second-order IIR sections (also called biquad sections). The filter coefficients and the state variables are stored in data memory. Note the use of LTD and MPYA instructions to perform multiply-accumulate and data-move operations.

Example 7–23. Using LTD and MPYA

```
      .title "N Cascaded BiQuad IIR Filters"
      .mmregs

*
* This routine implements N cascaded blocks of biquad IIR
* canonic type II filters. Each biquad requires 3 data
* memory locations d(n),d(n–1),d(n–2), and 5 coefficients
* –a1,–a2,b0,b1,b2.
* For each block:  d(n) = x(n)–d(n–1)a1–d(n–2)a2
*                  y(n) = d(n)b0+d(n–1)b1+d(n–2)b2
* Coefficients Storage: (low to high data memory)
*    –a2,–a1,b2,b1,b0, ... ,–a2,–a1,b2,b1,b0
*       1st biquad            Nth biquad
*
* State Variables: (low to high data memory)
*    d(n),d(n–1),d(n–2), ... ,d(n),d(n–1),d(n–2)
*        Nth biquad             1st biquad
*
* Entry Conditions:
*    AR1 –> d(n–2) of 1st biquad
*    AR2 –> –a2 of 1st biquad
*    AR3 –> input sample (Q15 number)
*    AR4 –> output sample (Q15 number)
*    DP = 0, PM = 0, ARP = 3
*

BIQUAD:              ; Setup variables
      ZPR            ; Clear P register
      LACC *,15,AR1  ; Get Q15 input
      SPLK #2,INDX   ; Setup index register
      SPLK #N–1,BRCR ; Setup count

                     ; Begin computation;
      RPTB ELOOP–1   ; repeat for N biquads
LOOP:
      LT   *–,AR2    ; T = d(n–2)
      MPYA *+,AR1    ; Acc = x(n), P = –d(n–2)a2
      LTA  *–,AR2    ; Acc += –d(n–2)a2, T = d(n–1)
      MPY  *+        ; P = –d(n–1)a1
      LTA  *+,AR1    ; Acc += –d(n–1)a1, T = b2
      SACH *0+,1     ; Save d(n)
      MPY  *–        ; P = d(n–2)b2
      LACL #0        ; Acc = 0
      LTD  *–,AR2    ; T = d(n–1), d(n–2) = d(n–1)
      MPY  *+,AR1    ; Acc += d(n–2)b2, P = d(n–1)b1
      LTD  *–,AR2    ; T = d(n), d(n–1) = d(n)
      MPY  *+,AR1    ; Acc += d(n–1)b1, P = d(n)b0
ELOOP:
      LTA  *,AR4     ; Final accumulation
      SACH *,1       ; Save output in Q15 format
```

7.10.4 Dynamic Programming

Dynamic programming techniques are widely used in optimal search algorithms. Applications such as speech recognition, telecommunications, and robotics use dynamic programming algorithms. The 'C5x digital signal processors have an enhanced instruction set for efficient implementation of dynamic programming methods.

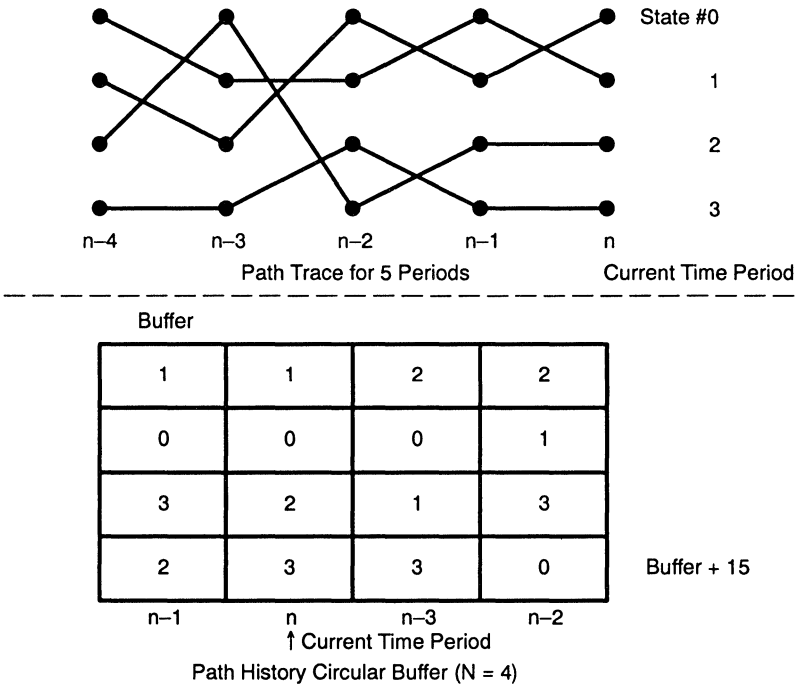
Most real-time search algorithms use the basic dynamic programming principle that the final optimal path from the start state to the goal state always passes through an optimal path from the start state to an intermediate state.

Identifying intermediate paths reduces a long, time-consuming search to the final goal. An integral part of any optimal search scheme based on the dynamic programming principle is the backtracking operation. The backtracking is necessary to retrace the optimal path when the goal state is reached.

Example 7–24 shows an implementation of the backtracking algorithm in which the path history consists of four independent path traces for *N* time periods. This path history is stored in a circular buffer. After each back-tracking operation, the path history is updated by a search algorithm (not shown) for the next time period. The path history buffer is shown in Figure 7–6 for *N* equal to 4. Each group of four consecutive memory locations in the buffer corresponds to the expansion of the four paths by one node (or by one time period). Each element of a group corresponds to one of the four states in that time period. In addition, each element of a group points to an element in the previous time period that belongs to that path.

As an illustration of backtracking using the path history buffer shown in Figure 7–4, the element corresponding to state #0 at the current time period contains a 1. This points to the second element of the previous time period that contains a 0. In this way, beginning from the current time period and using pointers to step back in time, this path is traced back as 1–0–2–1. Note that this simplified backtracking approach is taken here to illustrate 'C5x programming techniques. Most real applications would require more complex backtracking algorithms.

Figure 7–6. Backtracking With Path History



Example 7–24. Backtracking Algorithm Using Circular Addressing

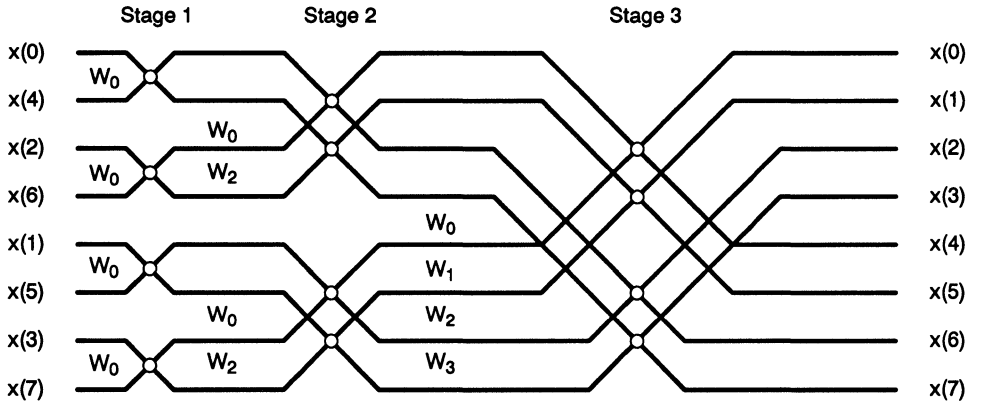
```
*
* Backtracking Example
* This program back-tracks the optimal path expanded by
* a dynamic programming algorithm. The path history
* consists of four paths expanded N times. It is set up
* as a circular buffer of length N*4.
* Note that decrement type circular buffer is used.
* The start and end address of the circular buffer are
* initialized this way because of two reasons:
* 1- to avoid skipping the end-address of circ buffer
* 2- to ensure that wrap-around is complete before next
* iteration.
*

    LAR    AR0,#BUFFER ;get buffer address
    LMMR   INDX,PATH   ;get the selected path [0..3]
    SPLK   #N-1,BRCR   ;trace back N time periods
* init. AR0 as pointer to circular buffer#1; length=N*4 words
    SPLK   #BUFFER+(N-1)*4,CBSR1
    SPLK   #BUFFER-3,CBER1
    SPLK   #08h,CBCR
*
    RPTB   TLOOP-1     ;for i=0,i<N,i++
    MAR    *0+         ;offset by state#
    LACC   *0-         ;get next pointer & reset to state#0
    SAMM   INDX        ;save next state#
    SBRK   3           ;decrement AR0 to avoid skipping CBER1
    SBRK   1           ;now AR0 is correctly positioned 1 time
TLOOP:                 ;period back (circular addressing)
```

7.11 Fast Fourier Transforms

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. Computationally efficient implementations of the Fourier transforms are known as fast Fourier transforms (FFT).

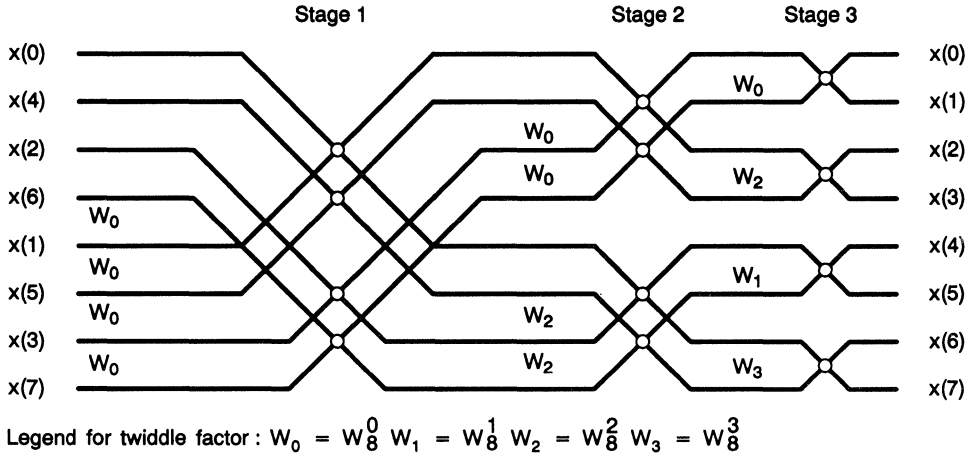
The 'C5x reduces the execution time of all FFTs by virtue of its 50-ns instruction cycle time. Also, the bit-reversed addressing mode helps reduce execution time for radix-2 FFTs. As demonstrated in Figure 7–7 and Figure 7–8, the inputs or outputs of an FFT are not in sequential order. This scrambling of data locations is a direct result of the radix-2 FFT derivation. Observation of the figures and the relationship of the input and output addressing reveal that the address indexing is in bit-reversed order, as shown in Table 7–1. As a result, either the input data sequence or the output data sequence must be scrambled in association with the execution of the FFT. In Example 7–27, the input data is scrambled before the execution of FFT algorithm so that the output is in order.

Figure 7–7. An In-Place DIT FFT With In-Order Outputs and Bit-Reversed Inputs



Legend for twiddle factor : $W_0 = W_8^0$ $W_1 = W_8^1$ $W_2 = W_8^2$ $W_3 = W_8^3$

Figure 7–8. An In-Place DIT FFT With In-Order Inputs but Bit-Reversed Outputs



Legend for twiddle factor : $W_0 = W_8^0$ $W_1 = W_8^1$ $W_2 = W_8^2$ $W_3 = W_8^3$

Table 7–1. Bit-Reversal Algorithm for an 8-Point Radix-2 DIT FFT

Index	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

The bit-reversed addressing mode is part of the indirect addressing implemented with the auxiliary registers and the associated arithmetic unit. In this mode, a value (index) contained in INDX is either added to or subtracted from the auxiliary register being pointed to by the ARP. However, the carry bit is not propagated in the forward direction; instead, it is propagated in the reverse direction. The result is a scrambling in the address access.

The procedure for generating the bit-reversed address sequence is to load INDX with a value corresponding to one-half the length of the FFT and to load another auxiliary register—for example, AR1—with the base address of the data array. However, implementations of FFTs involve complex arithmetic; as a result, two data memory locations (one real and one imaginary) are associated with each data sample. For ease of addressing, the samples are stored in workspace memory in pairs with the real part in the even address locations and the imaginary part in the odd address locations. This means that the offset from the base address for any given sample is twice the sample index. If the incoming data is in the following form:

```
XR(0),XR(1),...,XR(7),XI(0),XI(1),...,XI(7)
WHERE
   XR – real component of input sample
   XI – imaginary component of input sample
```

then it is easily transferred into the data memory and stored in the scrambled order:

```
XR(0),XI(0),XR(4),XI(4),XR(2),XI(2),...XR(7),XI(7)
```

by loading INDX register with the size of FFT and by using bit-reversed addressing to save each input word.

The following list shows the contents of auxiliary register AR1 when INDX is initialized with a value of 8 and when the data is being transferred by the code that follows.

```
      MSB                                       LSB
INDX  0 0 0 0  0 0 0 0  0 0 0 0  1 0 0 0 FOR 8-POINT FFT
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 0 0 0 BASE ADDRESS

      RPT   15
      BLDD  #INPUT,*BR0+

AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 0 0 0 XR(0)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 0 0 0 XR(4)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 1 0 0 XR(2)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 1 0 0 XR(6)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 0 1 0 XR(1)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 0 1 0 XR(5)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 1 1 0 XR(3)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 1 1 0 XR(7)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 0 0 1 XI(0)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 0 0 1 XI(4)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 1 0 1 XI(2)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 1 0 1 XI(6)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 0 1 1 XI(1)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 0 1 1 XI(5)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  0 1 1 1 XI(3)
AR1   0 0 0 0  0 0 1 0  0 0 0 0  1 1 1 1 XI(7)
```

This is shown in the FFT subroutine for 16 input samples.

Example 7–25. Macros for 16-Point DIT FFT

```
*************************************************************************
* FILE: c5cxrad2.mac   —> macro file for radix 2 fft's based on 320c5x  *
*                                                                       *
* COPYRIGHT TEXAS INSTRUMENTS INC. 1990                                 *
*************************************************************************
*                                                                       *
* MACRO 'COMBO2X' FOR THE COMPLEX, RADIX-2 DIT FFT                      *
*                                                                       *
* ORGANIZATION OF THE INPUT DATA MEMORY: R1,I1,R2,I2,R3,I3,R4,I4        *
*                                                                       *
*************************************************************************
*                                                                       *
* THE MACRO 'COMBO2x' PERFORMS FOLLOWING CALCULATIONS:                  *
*                                                                       *
* R1   := [(R1+R2)+(R3+R4)]/4       INPUT              OUTPUT           *
* R2   := [(R1-R2)+(I3-I4)]/4       ----------------   ----------------- *
* R3   := [(R1+R2)-(R3+R4)]/4       AR0 =  7                            *
* R4   := [(R1-R2)-(I3-I4)]/4       AR1 -> R1,I1       AR1 - > R5,I5    *
* I1   := [(I1+I2)+(I3+I4)]/4       AR2 -> R2,I2       AR2 - > R6,I6    *
* I2   := [(I1-I2)-(R3-R4)]/4       ARP-> AR3 -> R3,I3 ARP - > AR3 - > R7,I7 *
* I3   := [(I1+I2)-(I3+I4)]/4       AR4 -> R4,I4       AR4 - > R8,I8    *
* I4   := [(I1-I2)+(R3-R4)]/4                                           *
*                                                                       *
* For a 16-point Radix 2 complex FFT the Macro 'COMBO2x' has to be      *
* repeated N/4 times (e.g. 4  times for a  16 point FFT).               *
*                                                                       *
*************************************************************************
*
COMBO5x $MACRO  num             ; REPEAT MACRO 'COMBO5x': N/4 times
        SPLK    #:num:-1,BRCR   ; execute 'num' times 'COMBO5x'
*
        RPTB    comboend  ;                                  ARP AR1 AR2 AR3 AR4 AR5
*                                                            --- --- --- --- --- ---
        LACC    *,14,AR4  ; ACC := (R3)/4                     4  R1  R2  R3  R4  T1
        SUB     *,14,AR5  ; ACC := (R3-R4)/4                  5  R1  R2  R3  R4  T1
        SACH    *+,1,AR4  ; T1   = (R3-R4)/2                  4  R1  R2  I3  R4  T2
*
        ADD     *+,15,AR5 ; ACC := (R3+R4)/4                  5  R1  R2  R3  I4  T2
        SACH    *,1,AR2   ; T2   = (R3+R4)/2                  2  R1  R2  R3  I4  T2
*
        ADD     *,14,AR1  ; ACC := (R2+R3+R4)/4               1  R1  R2  R3  I4  T2
        ADD     *,14      ; ACC := (R1+R2+R3+R4)/4            1  R1  R2  R3  I4  T2
        SACH    *+,0,AR5  ; R1  := (R1+R2+R3+R4)/4            5  I1  R2  R3  I4  T2
        SUB     *,16,AR3  ; ACC := (R1+R2-(R3+R4))/4          3  I1  R2  R3  I4  T2
        SACH    *+,0,AR5  ; R3  := (R1+R2-(R3+R4))/4          5  I1  R2  I3  I4  T2
*
        ADD     *,15,AR2  ; ACC := (R1+R2)/4                  2  I1  R2  I3  I4  T2
        SUB     *,15,AR3  ; ACC := (R1-R2)/4                  3  I1  R2  I3  I4  T2
        ADD     *,14,AR4  ; ACC := ((R1-R2)+(I3))/4           4  I1  R2  I3  I4  T2
        SUB     *,14,AR2  ; ACC := ((R1-R2)+(I3-I4))/4        2  I1  R2  I3  I4  T2
        SACH    *+,0,AR4  ; R2  := ((R1-R2)+(I3-I4))/4        4  I1  I2  I3  I4  T2
        ADD     *-,15,AR3 ; ACC := ((R1-R2)+ I3+I4 )/4        3  I1  I2  I3  R4  T2
        SUB     *,15,AR4  ; ACC := ((R1-R2)-(I3-I4))/4        4  I1  I2  I3  R4  T2
        SACH    *+,0,AR1  ; R4  := ((R1-R2)-(I3-I4))/4        1  I1  I2  I3  I4  T2
*
        LACC    *,14,AR2  ; ACC := (I1)/4                     2  I1  I2  I3  I4  T2
        SUB     *,14,AR5  ; ACC := (I1-I2)/4                  5  I1  I2  I3  I4  T2
        SACH    *,1,AR2   ; T2  := (I1-I2)/2                  2  I1  I2  I3  I4  T2
        ADD     *,15,AR3  ; ACC := ((I1+I2))/4                4  I1  I2  I3  I4  T2
        ADD     *,14,AR4  ; ACC := ((I1+I2)+(I3))/4           4  I1  I2  I3  I4  T2
        ADD     *,14,AR1  ; ACC := ((I1+I2)+(I3+I4))/4        1  I1  I2  I3  I4  T2
        SACH    *0+,0,AR3 ; I1  := ((I1+I2)+(I3+I4))/4        3  R5  I2  I3  I4  T2
        SUB     *,15,AR4  ; ACC := ((I1+I2)-(I3+I4))/4        4  R5  I2  I3  I4  T2
        SUB     *,15,AR3  ; ACC := ((I1+I2)-(I3+I4))/4        3  R5  I2  I3  I4  T2
        SACH    *0+,0,AR5 ; I3  := ((I1+I2)-(I3+I4))/4        5  R5  I2  R7  I4  T2
*
        LACC    *-,15     ; ACC := (I1-I2)/4                  5  R5  I2  R7  I4  T1
```

```
        SUB     *,15,AR2  ; ACC := ((I1-I2)-(R3-R4))/4  2  R5  I2  R7  I4  T1
        SACH    *0+,0,AR5 ; I2  := ((I1-I2)-(R3-R4))/4  5  R5  R6  R7  I4  T1
        ADD     *,16,AR4  ; ACC := ((I1-I2)+(R3-R4))/4  4  R5  R6  R7  I4  T1
comboend:
        SACH    *0+,0,AR3 ; I4  := ((I1-I2)+(R3-R4))/4  3  R5  R6  R7  R8  T1
*
        MAR     *,AR2     ; ARP=AR2
        $ENDM
*
*******************************************************************************
*                                                                             *
*     MACRO 'ZEROI'    number of words : 10                                    *
*                                                                             *
*        ARP=2 FOR INPUT AND OUTPUT                                           *
*        AR2 -> QR,QI,QR+1,...                                                *
*        AR3 -> PR,PI,PR+1,...                                                *
*                                                                             *
*        CALCULATE Re[P+Q] AND Re[P-Q]                                        *
*        QR'=(PR-QR)/2                                                        *
*        PR'=(PR+QR)/2                                                        *
*        PI'=(PI+QI)/2                                                        *
*        PI'=(PI-QI)/2                                                        *
*                                                                             *
*******************************************************************************
ZEROI   $MACRO            ;                                 AR1  AR2  ARP
        LACC    *,15,AR1  ; ACC := (1/2)(QR)                PR   QR    1
        ADD     *,15      ; ACC := (1/2)(PR+QR)             PR   QR    1
        SACH    *+,0,AR2  ; PR  := (1/2)(PR+QR)             PI   QR    2
        SUB     *,16      ; ACC := (1/2)(PR+QR)-(QR)        PI   QR    2
        SACH    *+        ; QR  := (1/2)(PR-QR)             PI   QI    2
*
        LACC    *,15,AR1  ; ACC := (1/2)(QI)                PI   QI    1
        ADD     *,15      ; ACC := (1/2)(PI+QI)             PI   QI    1
        SACH    *+,0,AR2  ; PI  := (1/2)(PI+QI)             PR+1 QI    2
        SUB     *,16      ; ACC := (1/2)(PI+QI)-(QI)        PR+1 QI    2
        SACH    *+        ; QI  := (1/2)(PI-QI)             PR+1 QR+1  2
        $ENDM
*******************************************************************************
*                                                                             *
*     MACRO 'PBY2I'    number of words: 12                                    *
*                                                                             *
*        PR'=(PR+QI)/2      PI'=(PI-QR)/2                                     *
*        QR'=(PR-QI)/2      QI'=(PI+QR)/2                                     *
*                                                                             *
*******************************************************************************
PBY2I   $MACRO            ;                                 AR1  AR2  ARP
        LACC    *+,15,AR5 ;                                 PR   QI    5
        SACH    *,1,AR2   ; TMP=QR                          PR   QI    2
*
        LACC    *,15,AR1  ; ACC := QI/2                     PR   QI    1
        ADD     *,15      ; ACC := (PR+QI)/2                PR   QI    1
        SACH    *+,0,AR2  ; PR  := (PR+QI)/2                PI   QI    2
        SUB     *-,16     ; ACC := (PR-QI)/2                PI   QR    2
        SACH    *+,0,AR1  ; QR  := (PR-QI)/2                PI   QI    1
*
        LACC    *,15,AR5  ; ACC := (PI)/2                   PI   QI    5
        SUB     *,15,AR1  ; ACC := (PI-QR)/2                PI   QI    1
        SACH    *+,0,AR5  ; PI  := (PI-QR)/2                PR+1 QI    5
        ADD     *,16,AR2  ; ACC := (PI+QR)/2                PR+1 QI    2
        SACH    *+        ; QI  := (PI+QR)/2                PR+1 QI+1  2
        $ENDM
*
*******************************************************************************
*                                                                             *
*     MACRO 'PBY4J'    number of words: 16                                    *
*                                                                             *
*     T=SIN(45)=COS(45)=W45                                                   *
*                                                                             *
*        PR'= PR + (W*QI + W*QR) = PR + W * QI + W * QR     (<- AR1)          *
```

```
*       QR'= PR - (W*QI + W*QR) = PR - W * QI - W * QR    (<- AR2)          *
*       PI'= PI + (W*QI - W*QR) = PI + W * QI - W * QR    (<- AR1+1)        *
*       QI'= PI - (W*QI - W*QR) = PI - W * QI + W * QR    (<- AR1+2)        *
*                                                                           *
*****************************************************************************
;
PBY4J   $MACRO              ; TREG= W                    AR5    PREG    AR1  AR2 ARP
        MPY     *+,AR5      ; PREG= W*QR/2                 -    W*QR/2  PR   QI   5
        SPH     *,AR1       ; TMP = W*QR/2             W*QR/2 W*QR/2  PR   QI   1
        LACC    *,15,AR2    ; ACC = PR/2               W*QR/2 W*QR/2  PR   QI   2
        MPYS    *-          ; ACC = (PR-W*QR)/2        W*QR/2 W*QI/2  PR   QR   2
        SPAC                ; ACC = (PR-W*QI-W*QR)/2   W*QR/2 W*QI/2  PR   QR   2
        SACH    *+,0,AR1    ; QR  = (PR-W*QI-W*QR)/2   W*QR/2 W*QI/2  PR   QI   1
        SUB     *,16        ; ACC = (-PR-W*QI-W*QR)/2  W*QR/2 W*QI/2  PR   QI   1
        NEG                 ; ACC =  (PR+W*QI+W*QR)/2  W*QR/2 W*QI/2  PR   QI   1
        SACH    *+          ; QR  =  (PR+W*QI+W*QR)/2  W*QR/2 W*QI/2  PI   QI   1
;
        LACC    *,15,AR5    ; ACC = (PI)/2             W*QR/2 W*QI/2  PI   QI   5
        SPAC                ; ACC = (PI-W*QI)/2        W*QR/2   -     PI   QI   5
        ADD     *,16,AR2    ; ACC = (PI-W*QI+W*QR)/2      -     -     PI   QI   2
        SACH    *+,0,AR1    ; QI  = (PI-W*QI+W*QR)/2      -     -     PI   QR1  1
        SUB     *,16        ; ACCU= (-PI-W*QI+W*QR)/2     -     -     PI   QR1  1
        NEG                 ; ACCU=  (PI+W*QI-W*QR)/2     -     -     PI   QR1  1
        SACH    *+,0,AR2    ; PI  =  (PI+W*QI-W*QR)/2     -     -     PR1  QR1  2
        $ENDM
*
*****************************************************************************
*                                                                           *
*     MACRO 'P3BY4J'   number of words: 16                                  *
*                                                                           *
*   ENTRANCE IN THE MACRO: ARP=AR2                                          *
*                          AR1->PR,PI                                       *
*                          AR2->QR,QI                                       *
*                          TREG=W=COS(45)=SIN(45)                           *
*                                                                           *
*       PR'= PR + (W*QI - W*QR) = PR + W * QI - W * QR    (<- AR1)          *
*       QR'= PR - (W*QI - W*QR) = PR - W * QI + W * QR    (<- AR2)          *
*       PI'= PI - (W*QI + W*QR) = PI - W * QI - W * QR    (<- AR1+1)        *
*       QI'= PI + (W*QI + W*QR) = PI + W * QI + W * QR    (<- AR1+2)        *
*                                                                           *
*      EXIT OF THE MACRO: ARP=AR2                                           *
*                         AR1->PR+1,PI+1                                    *
*                         AR2->QR+1,QI+1                                    *
*                                                                           *
*****************************************************************************
P3BY4J  $MACRO              ; TREG= W                    AR5    PREG    AR1  AR2 ARP
        MPY     *+,AR5      ; PREG= W*QR/2                 -    W*QR/2  PR   QI   5
        SPH     *,AR1       ; TMP = W*QR/2             W*QR/2 W*QR/2  PR   QI   1
        LACC    *,15,AR2    ; ACC = PR/2               W*QR/2 W*QR/2  PR   QI   2
        MPYA    *-          ; ACC = (PR+W*QR)/2        W*QR/2 W*QI/2  PR   QR   2
        SPAC                ; ACC = (PR-W*QI+W*QR)/2   W*QR/2 W*QI/2  PR   QR   2
        SACH    *+,0,AR1    ; QR' = (PR-W*QI+W*QR)/2   W*QR/2 W*QI/2  PR   QI   1
        SUB     *,16        ; ACC = (-PR-W*QI+W*QR)/2  W*QR/2 W*QI/2  PR   QI   1
        NEG                 ; ACC =  (PR+W*QI-W*QR)/2  W*QR/2 W*QI/2  PR   QI   1
        SACH    *+          ; PR' =  (PR+W*QI-W*QR)/2  W*QR/2 W*QI/2  PI   QI   1
;
        LACC    *,15,AR5    ; ACC = (PI)/2             W*QR/2 W*QI/2  PI   QI   5
        APAC                ; ACC = (PI+W*QI)/2        W*QR/2   -     PI   QI   5
        ADD     *,16,AR2    ; ACC = (PI+W*QI+W*QR)/2      -     -     PI   QI   2
        SACH    *0+,0,AR1   ; QI' = (PI+W*QI+W*QR)/2      -     -     PI   QR5  1
        SUB     *,16        ; ACCU= (-PI+W*QI+W*QR)/2     -     -     PI   QR5  1
        NEG                 ; ACCU=  (PI-W*QI-W*QR)/2     -     -     PI   QR5  1
        SACH    *0+,0,AR2   ; PI' =  (PI-W*QI-W*QR)/2     -     -     PR5  QR5  2
        $ENDM
;
*****************************************************************************
*                                                                           *
*     MACRO 'stage3'   number of words: 54                                  *
*                                                                           *
```

```
*****************************************************************************
stage3    $macro num
          SPLK    #:num:-1,BRCR ; execute 'num'-1 times 'stage3'
          LT      cos45
          RPTB    stage3e
          ZEROI
          PBY4J
          PBY2I
          P3BY4j
stage3e:  .set    $-1
          $ENDM
*
*****************************************************************************
*                                                                           *
*  MACRO:  'BUTTFLYI'       general butterfly radix 2 for 320C5x            *
*                                                                           *
*  THE MACRO 'BUTTFLYI' REQUIRES 18 WORDS                                   *
*                                                                           *
*  Definition: ARP -> AR2  (input)  ARP -> AR2       (output)               *
*                                                                           *
*  Definition: AR1 -> QR   (input)  AR1 -> QR+1      (output)               *
*  Definition: AR2 -> PR   (input)  AR2 -> PR+1      (output)               *
*  Definition: AR3 -> Cxxx (input)  AR3 -> Cxxx+1    (output)  --> WR=cosine *
*  Definition: AR4 -> Sxxx (input)  AR4 -> Sxxx+1    (output)  --> WI=sine   *
*  Definition: AR5 -> temporary variable (unchanged)                        *
*                                                                           *
*  uses index register                                                      *
*                                                                           *
*        PR' = (PR+(QR*WR+QI*WI))/2        WR=COS(W)   WI=SIN(W)          *
*        PI' = (PI+(QI*WR-QR*WI))/2                                       *
*        QR' = (PR-(QR*WR+QI*WI))/2                                       *
*        QI' = (PI-(QI*WR-QR*WI))/2                                       *
*                                                                           *
*****************************************************************************
BUTTFLYI $MACRO
;                                                  (contents of register after exec.)
;                                                  TREG AR1  AR2   AR3 AR4 ARP
                                                   ---- ---  ---   --- --- ---
  RPTB  btflyend  ;                                
  LT    *+,AR3    ;TREG:= QR                        QR PR   QI    C   S   3
  MPY   *,AR2     ;PREG:= QR*WR/2                   QR PR   QI    C   S   2
  LTP   *-,AR4    ;ACC := QR*WR/2                   QI PR   QR    C   S   4
  MPY   *,AR3     ;PREG:= QI*WI/2                   QI PR   QR    C   S   3
  MPYA  *+,AR2    ;ACC := (QR*WR+QI*WI)/2           QR PR   QR    C+1 S   2
;                  PREG:= QI*WR
  LT    *,AR5     ;TREG = QR                        QR PR   QR    C+1 S   5
  SACH  *,1,AR1   ;H0  := (QR*WR+QI*WI)             QR PR   QR    C+1 S   1
;
  ADD   *,15      ;ACC := (PR+(QR*WR+QI*WI))/2      QR PR   QR    C+1 S   1
  SACH  *+,0,AR5  ;PR  := (PR+(QR*WR+QI*WI))/2      QR PI   QR    C+1 S   5
  SUB   *,16,AR2  ;ACC := (PR-(QR*WR+QI*WI))/2      QR PI   QR    C+1 S   2
  SACH  *+,0,AR1  ;QR  := (PR-(QR*WR+QI*WI))/2      QR PI   QI    C+1 S   1
;
  LACC  *,15,AR4  ;ACC := PI   /PREG=QI*WR          QI PI   QI    C+1 S   4
  MPYS  *+,AR2    ;PREG:= QR*WI/2                   QI PI   QI    C+1 S+1 2
;                 ;ACC := (PI-QI*WR)/2
  APAC            ;ACC := (PI-(QI*WR-QR*WI))/2      QI PI   QI    C+1 S+1 2
  SACH  *+,0,AR1  ;QI  := (PI-(QI*WR-QR*WI))/2      QI PI   QR+1  C+1 S+1 1
  NEG             ;ACC :=(-PI+(QI*WR-QR*WI))/2      QI PI   QR+1  C+1 S+1 1
  ADD   *,16      ;ACC := (PI+(QI*WR-QR*WI))/2      QI PI   QR+1  C+1 S+1 1
btflyend:
  SACH  *+,0,AR2  ;PI  := (PI+(QI*WR-QR*WI))/2      QI PR+1 QR+1  C+1 S+1 2
  $ENDM

; end of file
```

Example 7–26. Initialization Routine

```
*
*  file: INIT-FFT.ASM
*
*  Initialized  variables
*
            .bss    NN,1            ;number of fft-points
            .bss    NN2,1           ;2*N-1
            .bss    DATAADD,1       ;START ADDRESS OF DATA
            .bss    cos45,1
            .bss    sin4,1          ;start of sine in stage    4
            .bss    cos4,1          ;start of cosine in stage  4

*  Temp variables
*
            .bss    TEMP,2          ;used for temporary numbers
*
            .sect   "vectors"
            B       INIT,*,AR0

            .sect   "init"
TABINIT:    .word   N,N-1,2*N-1,DATA
            .word   5A82h           ;cos(45)=sin(45)
            .word   TWID,TWID+4
TABEND:     .set    $
*
INIT:       LDP     #0              ;use only B2 and mmregs for direct addressing
            SPM     0               ;no shift from PREG to ALU
            CLRC    OVM             ;disable overflowmode
            SETC    SXM             ;enable sign extension mode
            SPLK    #pmstmask,PMST :ndx=trm=1
*
* INIT Block B2
*
            LAR     AR0,#NN         ;arp is already pointing to ar0
            LACC    #TABINIT
            RPT     #TABEND-TABINIT
            TBLR    *+
*
* INIT TWIDDLE FACTORS
*
            LAR     AR0,#TWID       ;arp is already pointing to ar0
            LACC    #TWIDSTRT
            RPT     #TWIDLEN
            TBLR    *+
*
* EXECUTE THE FFT
*
            LAR     AR5,#TEMP       ;pointer to 2 temp register
            CALL    FFT,*,AR3       ;ARP=AR3 FOR MACRO COMBO
*
WAIT        RET                     ;Return
*
```

Example 7–27. 16-Point Radix-2 Complex FFT

```
          .file      "c5cx0016.asm"
          .title     "0016 point DIT Radix-2, Complex FFT"
          .width     120
N         .set       16   ; NUMBER OF POINTS FOR FFT
          .mmregs
pmstmask  .set       0110b ; ndx=trm=1
*******************************************************************************
*                                                                             *
*      16 - POINT COMPLEX, RADIX-2 DIF FFT WITH THE TMS320C5x / LOOPED CODE   *
*      ------------------------------------------------------------------     *
*******************************************************************************
* THE PROGRAM IS BASED ON THE BOOK 'DIGITAL SIGNAL PROCESSING APPLICATIONS'   *
* FROM TEXAS INSTRUMENTS P. 69. IT IS OPTIMIZED FOR THE TMS320C5x INCLUDING   *
* BIT REVERSAL ADDRESSING MODE.                                               *
*                                                                             *
*******************************************************************************
*                                                                             *
*     USED REGISTERS: INDX,AR1,AR2,AR3,AR4,AR5,ACCU,PREG,TREG0, PMST, BRCR    *
*                     2 Stacklevel, Block B2 for temp variables               *
*                                                                             *
*     PROGRAM MEMORY:  164 WORDS ('END' - 'FFT') WITHOUT INITIALIZATION       *
*                                                                             *
*     COEFFICIENTS   :   16 BITS  (Q15 Format)  SCALING:   1/2^4              *
*                                                                             *
*     PROGRAM SEQUENCE:0.   INITIALIZATION FOR FFT/COEFF     ADD: 240H -  20BH *
*                      1.   INPUT NEW DATA INTO 'INPUT'      ADD: 220H -  23FH *
*                      2.   CALL SUBROUTINE FFT              ADD: 600H -  6A3H *
*                      2.1. BITREVERSAL FROM INPUT TO DATA   ADD: 200H -  21FH *
*                      2.2. FFT WITH WORK SPACE DATA         ADD: 200H -  21FH *
*                      3.   OUTPUT THE RESULTS FROM DATA     ADD: 200H -  21FH *
*                                                                             *
*     INPUT DATA AT ADDRESS 0220h-023fh:                                      *
*     ----------------------------------                                      *
*     THE DATA IS STORED IN 'INPUT' AS THE SEQUENCE: X(0),X(1),...,X(15)      *
*                                                    Y(0),Y(1),...,Y(15)      *
*                                                                             *
*     OUTPUT DATA AT ADDRESS 0200h-021fh:                                     *
*     -----------------------------------                                     *
*     THE DATA IS STORED IN 'DATA' AS THE SEQUENCE:                           *
*     X(0),Y(0),X(1),Y(1),... ... ,X(15),Y(15)                                *
*******************************************************************************
*                                                                             *
*     THIS PROGRAM INCLUDES FOLLOWING FILE:                                   *
*     -------------------------------------                                   *
*     THE FILE 'TWIDDLES.Q15' CONSISTS OF TWIDDLE FACTORS IN Q15 FORMAT       *
*     THE FILE 'C5CXRAD2.MAC' macro files                                     *
*     THE FILE 'INIT-FFT.ASM' for initialization                              *
*******************************************************************************
*
            .include   C5CXRAD2.MAC
            .def       TWIDLEN,FFTLEN,TEMP,WAIT,cos45
            .def       INIT,FFT,TWIDSTRT,TWIDEND
            .def       STAGE1,STAGE3,STAGE4,INPUT,DATA,TWID
;
            .sect      "twiddles"
; table of twiddle factors for the FFT
TWIDSTRT    .set    $
            .include   twiddles.q15
TWIDEND     .set    $
TWIDLEN     .set       TWIDEND-TWIDSTRT
*
INPUT       .usect     "input",N*2     ;input data array
DATA        .usect     "data",N*2      ;working data array
TWID        .usect     "twid",N*2      ;reserve space for twiddles
*
*           .include   init-fft.asm
*
```

```
        .sect       "fftprogram"
*
* FFT CODE WITH BIT-REVERSED INPUT SAMPLES / ARP=AR3
*
FFT:        LAR         AR3,DATAADD     ;TRANSFER  32 WORDS FROM 'input' to 'data'
            LACC        NN
            SAMM        INDX            ;indexregister=7
            RPT         NN2             ;N TIMES
            BLDD        #INPUT,*BR0+
*
*           FFT CODE for STAGES 1 and 2
*
STAGE1:     SPLK        #7,INDX         ;indexregister = 7
            LAR         AR1,DATAADD     ;pointer to DATA r1,i1
            LAR         AR2,#DATA+2     ;pointer to DATA + 2 r2,i2
            LAR         AR3,#DATA+4     ;pointer to DATA + 4 r3,i3
            LAR         AR4,#DATA+6     ;pointer to DATA + 6 r4,i4
            COMBO5X 4                   ;repeat 4 times
*
* FFT CODE FOR STAGE 3  /  ARP=AR2
*
STAGE3:     SPLK        #9,INDX         ;index register = 9
            LAR         AR1,DATAADD     ;ar1 -> DATA
            LAR         AR2,#DATA+8     ;ar2 -> DATA+8
            stage3      2               ;repeat 2 times
*
*  FFT CODE FOR STAGE 4  / ARP=ARP
*
STAGE4:     SPLK        #1,INDX         ;index register = 1
            LAR         AR1,DATAADD
            LAR         AR2,#DATA+16
            LAR         AR3,cos4        ;start of cosine in stage 4
            LAR         AR4,sin4        ;start of sine in stage   4
            SPLK        #6,BRCR
            ZEROI                       ;execute ZEROI
            BUTTFLYI                    ;execute 7 times BUTTFLYI
            RET
END:        .set        $
FFTLEN      .set        END-FFT+1
            .end
```

Appendix A

Electrical Specifications

This appendix contains data sheet information on the TMS320C5x digital signal processors family, including the following devices:

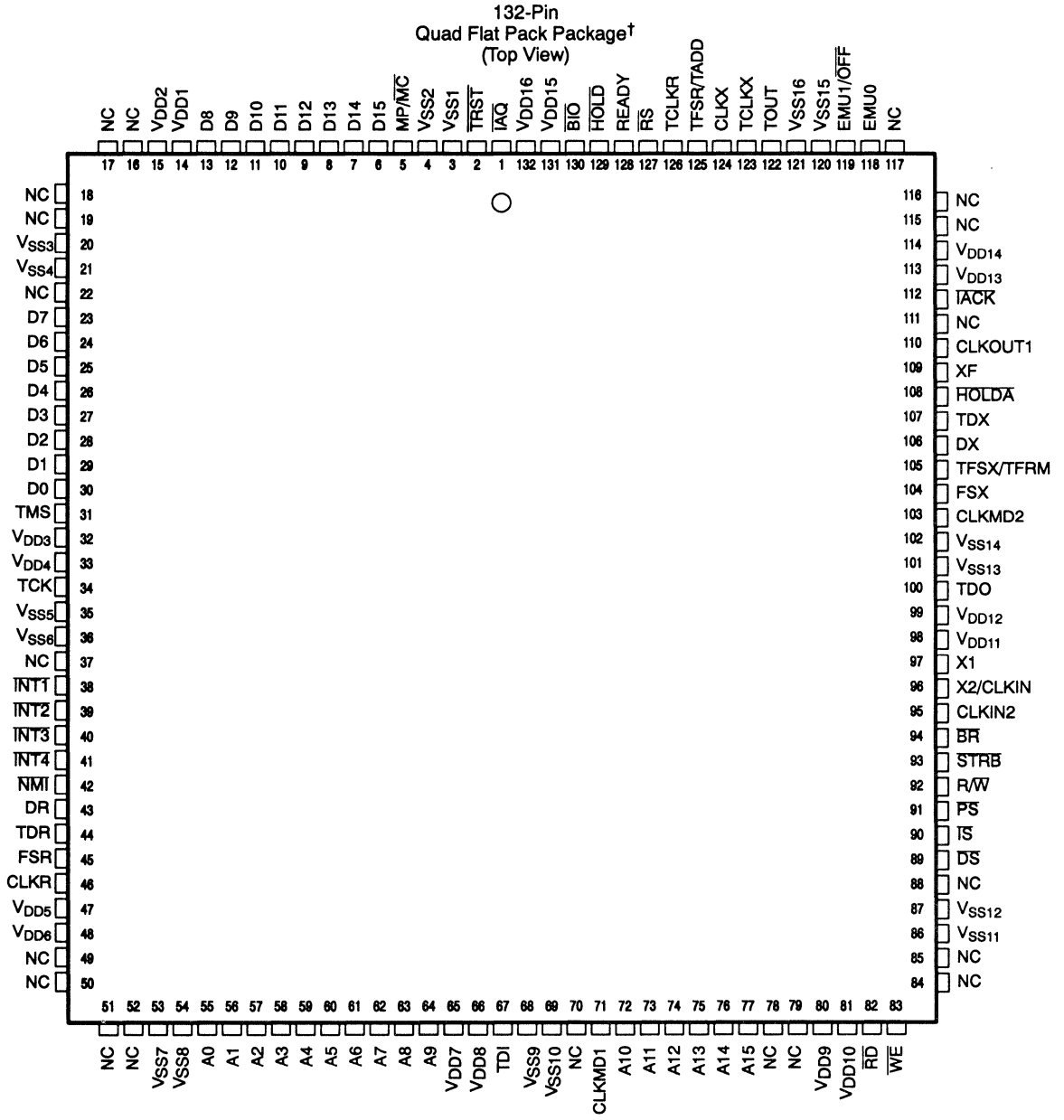
- TMS320C50
- TMS320C51
- TMS320C53

Figure A–1 shows the pinout of the 'C5x devices in a 132-pin quad flat pack; the pin assignments are given in Table A–1. This appendix also contains the electrical characteristics of the 'C5x devices and the mechanical data of the 132-pin quad flat pack.

Topic **Page**

A.1 Pinout and Signal Descriptions

Figure A–1. TMS320C5x Pinout



† See Pin Assignments, Table A–1 (page A-3) for location and description of all pins. The 'C50, 'C51, and 'C53 are packaged in 132-pin plastic QFP in production. See Figure A–20 for mechanical data.

Note: NC = No connect. (These pins are reserved.)

Table A–1. TMS320C5x Pin Assignments

Pin	Name	Type	Description
1	$\overline{IAQ}$	O/Z	Instruction Acquisition
2	$\overline{TRST}$	I	JTAG Test Reset
3	V_{SS}	Supply	Ground
4	V_{SS}	Supply	Ground
5	MP/$\overline{MC}$	I	Microprocessor/Microcomputer
6	D15 (MSB)	I/O/Z	Parallel Data Port, High-Byte (8 pins)
7	D14	I/O/Z	
8	D13	I/O/Z	
9	D12	I/O/Z	
10	D11	I/O/Z	
11	D10	I/O/Z	
12	D9	I/O/Z	
13	D8	I/O/Z	
14	V_{DD}	Supply	+5 V
15	V_{DD}	Supply	+5 V
16	NC†		Reserved
17	NC†		Reserved
18	NC†		Reserved
19	NC†		Reserved
20	V_{SS}	Supply	Ground
21	V_{SS}	Supply	Ground
22	NC†		Reserved
23	D7	I/O/Z	Parallel Data Port, Low-Byte (8 pins)
24	D6	I/O/Z	
25	D5	I/O/Z	
26	D4	I/O/Z	
27	D3	I/O/Z	
28	D2	I/O/Z	
29	D1	I/O/Z	
30	D0 (LSB)	I/O/Z	
31	TMS	I	JTAG Test Mode
32	V_{DD}	Supply	+5 V
33	V_{DD}	Supply	+5 V
34	TCK	I	JTAG Test Clock

† NC = No connect

Table A–1. TMS320C5x Pin Assignments (Continued)

Pin	Name	Type	Description
35	V_{SS}	Supply	Ground
36	V_{SS}	Supply	Ground
37	NC†		Reserved
38	$\overline{INT1}$	I	Interrupt #1
39	$\overline{INT2}$	I	Interrupt #2
40	$\overline{INT3}$	I	Interrupt #3
41	$\overline{INT4}$	I	Interrupt #4
42	$\overline{NMI}$	I	Nonmaskable Interrupt
43	DR	I	Serial Port 1 Data Receive
44	TDR	I	Serial Port 2 Data Receive
45	FSR	I	Serial Port 1 Receiver Frame Sync
46	CLKR	I	Serial Port 1 Receiver Clock
47	V_{DD}	Supply	+5 V
48	V_{DD}	Supply	+5 V
49	NC†		Reserved
50	NC†		Reserved
51	NC†		Reserved
52	NC†		Reserved
53	V_{SS}	Supply	Ground
54	V_{SS}	Supply	Ground
55	A0 (LSB)	I/O/Z	Parallel Port Address Bus
56	A1	I/O/Z	(10 pins)
57	A2	I/O/Z	
58	A3	I/O/Z	
59	A4	I/O/Z	
60	A5	I/O/Z	
61	A6	I/O/Z	
62	A7	I/O/Z	
63	A8	I/O/Z	
64	A9	I/O/Z	
65	V_{DD}	Supply	+5 V
66	V_{DD}	Supply	+5 V
67	TDI	I	JTAG Scan Input

† NC = No connect

Table A–1. TMS320C5x Pin Assignments (Continued)

Pin	Name	Type	Description
68	V_{SS}	Supply	Ground
69	V_{SS}	Supply	Ground
70	NC†		Reserved
71	CLKMD1	I	Clock Mode Pin 1
72	A10	I/O/Z	Parallel Port Address Bus
73	A11	I/O/Z	(6 pins)
74	A12	I/O/Z	
75	A13	I/O/Z	
76	A14	I/O/Z	
77	A15	I/O/Z	
78	NC†		Reserved
79	NC†		Reserved
80	V_{DD}	Supply	+5 V
81	V_{DD}	Supply	+5 V
82	$\overline{RD}$	O/Z	Read Enable
83	$\overline{WE}$	O/Z	Write Enable
84	NC†		Reserved
85	NC†		Reserved
86	V_{SS}	Supply	Ground
87	V_{SS}	Supply	Ground
88	NC†		Reserved
89	$\overline{DS}$	O/Z	Data Space Select
90	$\overline{IS}$	O/Z	I/O Space Select
91	$\overline{PS}$	O/Z	Program Space Select
92	R/$\overline{W}$	I/O/Z	Read/Write
93	$\overline{STRB}$	I/O/Z	External Parallel Access Active
94	$\overline{BR}$	I/O/Z	Bus Request
95	CLKIN2	I	Divide-by-One Clock Input
96	X2/CLKIN	I	Divide-by-Two Clock Input
97	X1	O	Oscillator Output
98	V_{DD}	Supply	+5 V
99	V_{DD}	Supply	+5 V
100	TDO	O/Z	JTAG Scan Output

† NC = No connect

Table A–1. TMS320C5x Pins (Concluded)

Pin	Name	Type	Description
101	V_{SS}	Supply	Ground
102	V_{SS}	Supply	Ground
103	CLKMD2	I	Clock Mode Pin 2
104	FSX	I/O/Z	Serial Port 1 Transmitter Frame Sync
105	TFSX/TFRM	I/O/Z	Serial Port 2 Transmitter Frame Sync
106	DX	O/Z	Serial Port 1 Transmitter Output
107	TDX	O/Z	Serial Port 2 Transmitter Output
108	$\overline{\text{HOLDA}}$	O/Z	Hold Acknowledge
109	XF	O/Z	External Flag
110	CLKOUT1	O/Z	Machine Clock Output
111	NC†		Reserved
112	$\overline{\text{IACK}}$	O/Z	Interrupt Acknowledge
113	V_{DD}	Supply	+5 V
114	V_{DD}	Supply	+5 V
115	NC†		Reserved
116	NC†		Reserved
117	NC†		Reserved
118	EMU0	I/O/Z	Emulator Interrupt 0
119	EMU1/$\overline{\text{OFF}}$	I/O/Z	Emulator Interrupt 1
120	V_{SS}	Supply	Ground
121	V_{SS}	Supply	Ground
122	TOUT	O/Z	Timer Output
123	TCLKX	I/O/Z	Serial Port 2 Transmitter Clock
124	CLKX	I/O/Z	Serial Port 1 Transmitter Clock
125	TFSR/TADD	I/O/Z	Serial Port 2 Receive Frame/Address
126	TCLKR	I	Serial Port 2 Receiver Clock
127	$\overline{\text{RS}}$	I	Device Reset
128	READY	I	External Access Ready to Complete
129	$\overline{\text{HOLD}}$	I	Request Access of Local Memory
130	$\overline{\text{BIO}}$	I	Bit I/O Pin
131	V_{DD}	Supply	+5 V
132	V_{DD}	Supply	+5 V

† NC = No connect

A.2 Electrical Characteristics and Operating Conditions

Table A–2. Absolute Maximum Ratings Over Specified Temperature Range (Unless Otherwise Noted)†

Supply voltage range, V_{DD} ‡ . . . . . –0.3 V to 7 V
Input voltage range . . . . . –0.3 V to 7 V
Output voltage range . . . . . –0.3 V to 7 V
Operating case temperature range . . . . . 0° to 85°C
Storage temperature range . . . . . –55° to 150°C

† Stresses beyond those listed under "Absolute Maximum Ratings" may cause damage to the device. This is a stress rating only, and functional operation of the device at these or any other conditions beyond those indicated in the "Recommended Operating Conditions" sections of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect reliability.

‡ All voltage values are with respect to V_{SS}.

Table A–3. Recommended Operating Conditions

Parameter			Min	Nom	Max	Unit
V_{DD}	Supply voltage		4.75	5	5.25	V
V_{SS}	Supply voltage			0		V
V_{IH}	High-level input voltage	CLKIN, CLKIN2	3.0		V_{DD}+0.3	V
		CLKX,CLKR, TCLKX, TCLKR	2.5		V_{DD}+0.3	
		All others	2.0		V_{DD}+0.3	
V_{IL}	Low-level input voltage		–0.3		0.8	V
I_{OH}	High-level output current				–300†	μA
I_{OL}	Low-level output current				2	mA
T	Operating case temperature		0		85	°C

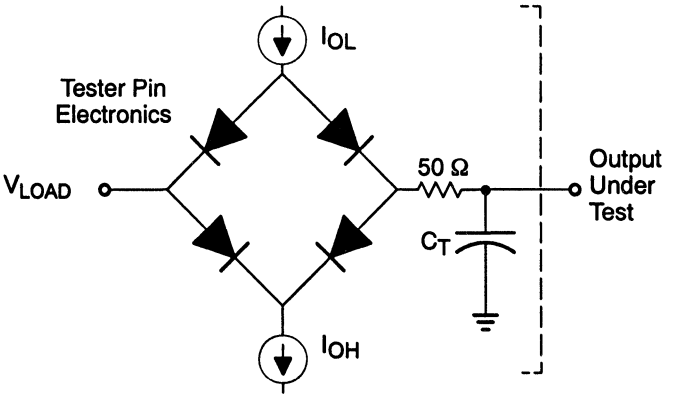
† This I_{OH} may be exceeded when using a 1-kΩ pull-down resistor on the TDM serial port TADD output, however, this output still meets V_{OH} specifications under these conditions.

Table A–4. Electrical Characteristics Over Specified Free-Air Temperature Range (Unless Otherwise Noted)

Parameter		Test Conditions	Min	Typ†	Max	Unit
V_{OH}	High-level output voltage §	I_{OH}=Max	2.4	3		V
V_{OL}	Low-level output voltage §	I_{OL}=Max		0.3	0.6	V
I_Z	Three-state current (V_{DD} = Max)	$\overline{BR}$	–400	‡	20	µA
		All other three-state	–20	‡	20	
I_I	Input current (V_I=V_{SS} to V_{DD})	$\overline{TRST}$ pin (with internal pulldown)	–10	‡	800	µA
		TMS, TCK, TDI pins (with internal pullups)	–400	‡	10	
		X2/CLKIN pin	–50	‡	+50	µA
		All other input-only pins	–10	‡	10	
I_{DDC}	Supply current, core CPU	Operating T_A=25°C, V_{DD}=5.25 V, f_x=40.96 MHz		60		mA
I_{DDP}	Supply current, pins	Operating T_A=25°C, V_{DD}=5.25 V, f_x=40.96 MHz		40		mA
I_{DD}	Supply current, standby	IDLE2, clocks shut off		5		µA
C_i	Input capacitance			15		pF
C_o	Output capacitance			15		pF

† All typical nominal values are at V_{DD}=5 V, T_A=25°C.
‡ These values are not specified, pending detailed characterization.
§ All input and output voltage levels are TTL-compatible. Figure A-2 shows the test load circuit and Figure A-3 shows the voltage reference levels.

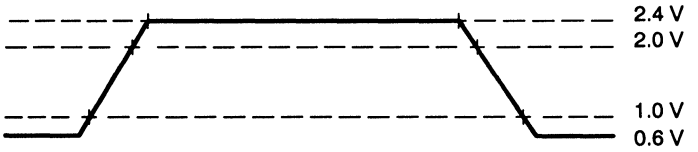
Figure A–2. Test Load Circuit



Where: I_{OL} = 2.0 mA (all outputs)
I_{OH} = 300 µA (all outputs)
V_{LOAD} = 1.5 V
C_T = 80 pF typical load circuit capacitance.

TTL 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 A–3 shows the TTL-level outputs.

Figure A–3. TTL-Level Outputs

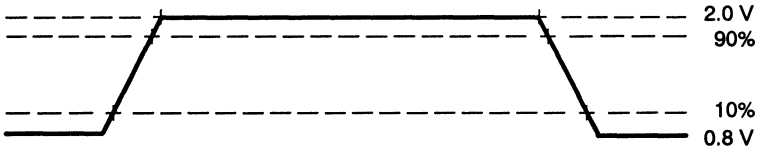


TTL-output transition times are specified as follows:

- 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.

Figure A–4 shows the TTL-level inputs.

Figure A–4. 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.

A.3 Clock Characteristics and Timing

The 'C5x can use either its internal oscillator or an external frequency source for a clock. The clock mode is determined by the CLKMD1 (pin 71) and CLKMD2 (pin 103) clock mode pins. The following table outlines the selection of the clock mode by these pins.

CLKMD1	CLKMD2	Clock Source
1	0	External divide-by-one clock option.
0	1	Reserved for test purposes.
1	1	External divide-by-two option or internal divide-by-two clock option with an external crystal.
0	0	External divide-by-two option with the internal oscillator disabled.

A.3.1 Internal Divide-by-Two Clock Option With External Crystal

The internal oscillator is enabled by connecting a crystal across X1 and X2/CLKIN. The frequency of CLKOUT1 is one-half the crystal's oscillating frequency. The crystal should be in either fundamental or overtone operation and parallel resonant, with an effective series resistance of 30 ohms and a power dissipation of 1 mW; it should be specified at a load capacitance of 20 pF. Note that overtone crystals require an additional tuned-LC circuit. Figure A-4 shows an external crystal (fundamental frequency) connected to the on-chip oscillator.

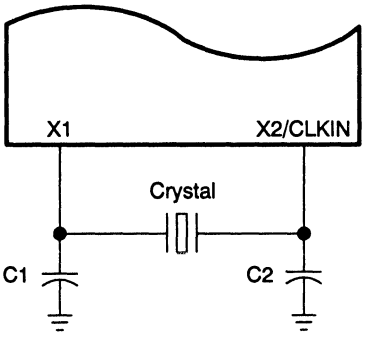
Table A–5. Recommended Operating Conditions

Parameter		Min	Nom	Max	Unit
f_x Input clock frequency	TMS320C5x-40	0†		40.96	MHz
	TMS320C5x-57‡	**0†**		**57.14**	MHz
C1, C2			10		pF

† This device utilizes a fully static design and therefore can operate with $t_{c(CI)}$ approaching ∞. The device is characterized at frequencies approaching 0 Hz but is tested at a minimum of 3.3 MHz to meet device test time requirements.

‡ Other timings for the 57-MHz CLKIN devices are the same as those for the 40-MHz CLKIN devices, except where otherwise indicated.

Figure A–5. Internal Clock Option



A.3.2 External Divide-by-Two Clock Option

An external frequency source can be used by injecting the frequency directly into X2/CLKIN, with X1 left unconnected, CLKMD1 set high, and CLKMD2 set high. This external frequency is divided by two to generate the internal machine cycle.

The external frequency injected must conform to specifications listed in the timing requirements table.

Table A–6. Switching Characteristics Over Recommended Operating Conditions (H = 0.5 $t_{c(CO)}$)

Parameter			Min	Typ	Max	Unit
$t_{c(CO)}$	CLKOUT1 cycle time	TMS320C5x-40	48.8	$2t_{c(CI)}$	†	ns
		TMS320C5x-57‡	**35**	**$2t_{c(CI)}$**	†	ns
$t_{d(CIH-CO)}$	CLKIN high to CLKOUT1 high/low		3	11	20	ns
$t_{f(CO)}$	CLKOUT1 fall time			5		ns
$t_{r(CO)}$	CLKOUT1 rise time			5		ns
$t_{w(COL)}$	CLKOUT1 low pulse duration		H – 2	H	H + 2	ns
$t_{w(COH)}$	CLKOUT1 high pulse duration		H – 2	H	H + 2	ns

† This device utilizes a fully static design and therefore can operate with $t_{c(CI)}$ approaching ∞. The device is characterized at frequencies approaching 0 Hz but is tested at a minimum of 3.35 MHz to meet device test time requirements.

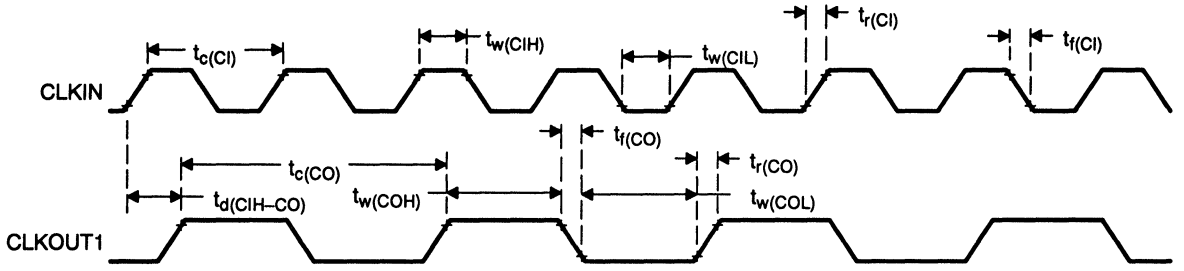
‡ Other timings for the 57-MHz CLKIN devices are the same as those for the 40-MHz CLKIN devices, except where otherwise indicated.

Table A–7. Timing Requirements Over Recommended Operating Conditions (H = 0.5 $t_{c(CO)}$)

Parameter			Min	Max	Unit
$t_{c(CI)}$	CLKIN cycle time	TMS320C5x-40	24.4	§	ns
		TMS320C5x-57‡	**17.5**	§	ns
$t_{f(CI)}$	CLKIN fall time †			5	ns
$t_{r(CI)}$	CLKIN rise time †			5	ns
$t_{w(CIL)}$	CLKIN low pulse duration	TMS320C5x-40	11	§	ns
		TMS320C5x-57‡	**8**	§	ns
$t_{w(CIH)}$	CLKIN high pulse duration	TMS320C5x-40	11	§	ns
		TMS320C5x-57‡	**8**	§	ns

† Values derived from characterization data and not tested.
‡ Other timings for the 57-MHz CLKIN devices are the same as those for the 40-MHz CLKIN devices, except where otherwise indicated.
§ This device utilizes a fully static design and therefore can operate with $t_{c(CI)}$ approaching ∞. The device is characterized at frequencies approaching 0 Hz, but is tested at a minimum of 6.7 MHz to meet device test time requirements.

Figure A–6. External Divide-by-Two Clock Timing



A.3.3 External Divide-by-One Clock Option

An external frequency source can be used by injecting the frequency directly into CLKIN2, with X1 left unconnected and X2 connected to V_{DD}. This external frequency is divided by one to generate the internal machine cycle. The divide-by-one option is used when the CLKMD1 pin is strapped high and CLKMD2 is strapped low.

The external frequency injected must conform to specifications listed in the timing requirements table.

Table A–8. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5\ t_{c(CO)}$)

Parameter			Min	Typ	Max	Unit
$t_{c(CO)}$	CLKOUT1 cycle time	TMS320C5x-40	48.8	$t_{c(CI)}$	75§	ns
		TMS320C5x-57‡	**35**	$t_{c(CI)}$	**75§**	ns
$t_{d(CIH-CO)}$	CLKIN2 high to CLKOUT1 high		2	9	16	ns
$t_{f(CO)}$	CLKOUT1 fall time			5		ns
$t_{r(CO)}$	CLKOUT1 rise time			5		ns
$t_{w(COL)}$	CLKOUT1 low pulse duration		H – 2	H	H + 2	ns
$t_{w(COH)}$	CLKOUT1 high pulse duration		H – 2	H	H + 2	ns
t_p	Transitory phase—PLL synchronized after CLKIN2 supplied		256¶		1000†	cycles

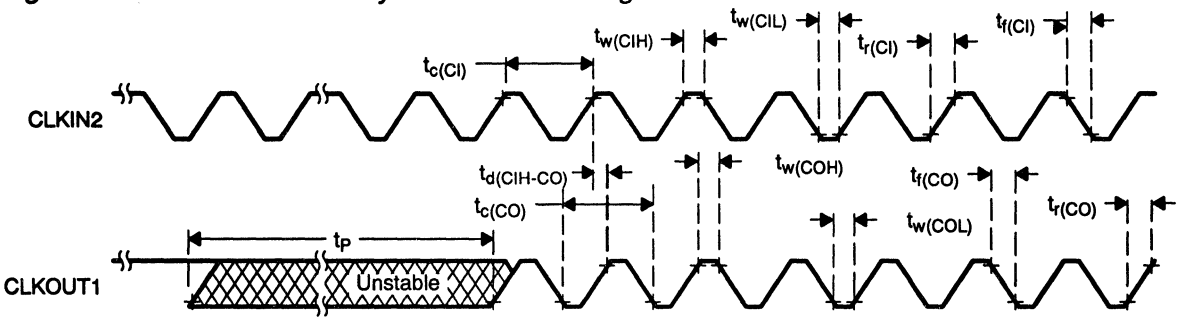
† Values derived from characterization data and not tested.
‡ Other timings for the 57-MHz CLKIN devices are the same as those for the 40-MHz CLKIN devices, except where otherwise indicated.
§ Clocks can be stopped only while the device executes IDLE2 when using the external divide-by-one clock option.
¶ Values guaranteed by design and not tested.

Table A–9. Timing Requirements Over Recommended Operating Conditions ($H = 0.5\ t_{c(CO)}$)

Parameter			Min	Max	Unit
$t_{c(CI)}$	CLKIN2 cycle time	TMS320C5x-40	48.8	75§	ns
		TMS320C5x-57‡	**35**	**75§**	ns
$t_{f(CI)}$	CLKIN2 fall time †			5	ns
$t_{r(CI)}$	CLKIN2 rise time †			5	ns
$t_{w(CIL)}$	CLKIN2 low pulse duration	TMS320C5x-40	15	60	ns
		TMS320C5x-57‡	**11**	**64**	ns
$t_{w(CIH)}$	CLKIN2 high pulse duration	TMS320C5x-40	15	60	ns
		TMS320C5x-57‡	**11**	**64**	ns

† Values derived from characterization data and not tested.
‡ Other timings for the 57-MHz CLKIN devices are the same as those for the 40-MHz CLKIN devices, except where indicated otherwise.
§ Clocks can be stopped only while the device executes IDLE2 when using the external divide-by-one clock option. Note that t_p (the transitory phase) will occur when restarting clock from IDLE2 in this mode.

Figure A–7. External Divide-by-One Clock Timing



A.3.4 Memory and Parallel I/O Interface Read Timing

Table A–10. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{su(A)R}$	Setup time, address valid before $\overline{RD}$ low †	H – 10¶		ns
$t_{h(A)R}$	Hold time, address valid after $\overline{RD}$ high †	0¶		ns
$t_{w(RL)}$	$\overline{RD}$ low pulse duration ‡#	H – 2	H + 2	ns
$t_{w(RH)}$	$\overline{RD}$ high pulse duration ‡#	H – 2		ns
$t_{d(RW)}$	Delay time, $\overline{RD}$ high to $\overline{WE}$ low	2H – 5		ns

† A15–A0, $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, and $\overline{BR}$ timings are all included in timings referenced as address.
‡ $\overline{STRB}$ and $\overline{RD}$ rising and falling edges track and are 0–4 and ± 2 ns, respectively, from CLKOUT1 edges on reads, following the cycle after reset, which is always 7 wait states; thus, tolerance of resulting pulsewidths is ± 2 ns, not ± 4 ns. See Appendix B.
# Values derived from characterization data and are not tested.
¶ See Figure A–9 for address bus timing variation with load capacitance.

Table A–11. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter			Min	Max	Unit
$t_{a(A)}$	Read data access from address valid	TMS320C5x-40		2H – 18†	ns
		TMS320C5x-57‡		**2H – 15†**	ns
$t_{su(D)R}$	Read data setup time before $\overline{RD}$ high		10		ns
$t_{h(D)R}$	Read data hold time after $\overline{RD}$ high		0		ns
$t_{a(R)}$	Read data access time after $\overline{RD}$ low			H – 10	ns

† See Figure A–9 for address bus timing variation with load capacitance.
‡ Other timings for 57-MHz CLKIN devices are the same as for the 40-MHz devices, except where indicated otherwise.

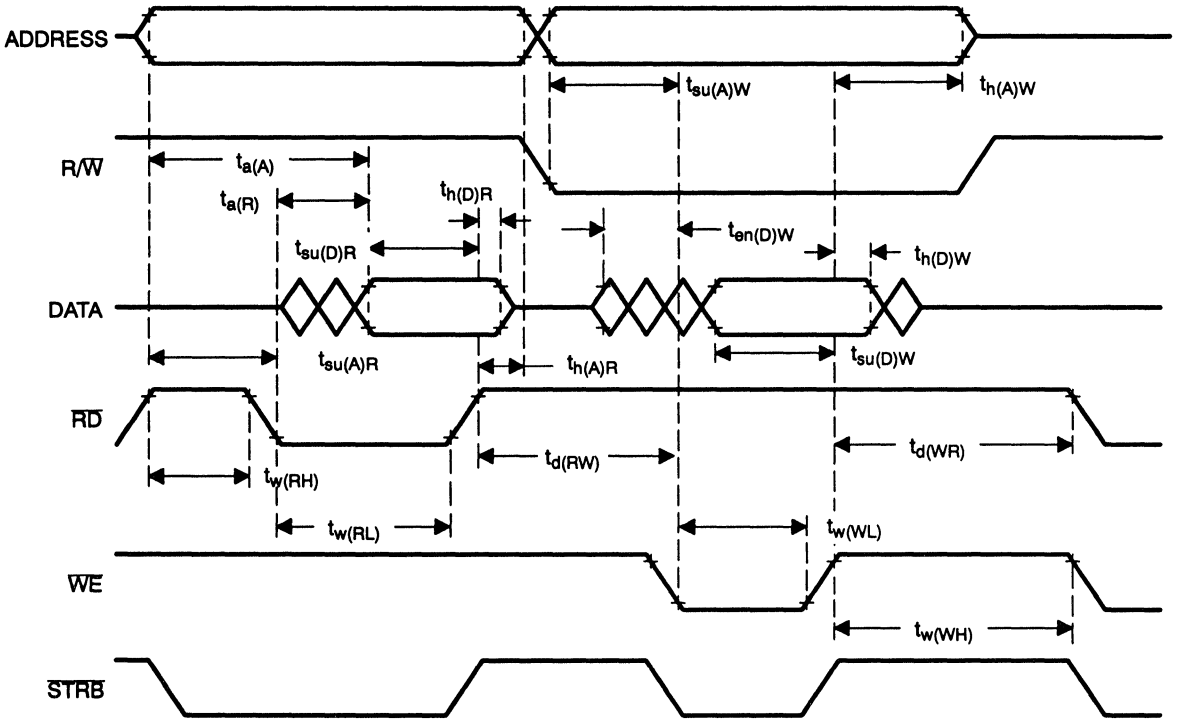
A.3.5 Memory and Parallel I/O Interface Write Timing

Table A–12. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{su(A)W}$	Setup time, address valid before $\overline{WE}$ low †	H – 5#		ns
$t_{h(A)W}$	Hold time, address valid after $\overline{WE}$ high †	H – 10#		ns
$t_{w(WL)}$	$\overline{WE}$ low pulse duration ‡¶	2H – 2	2H + 2	ns
$t_{w(WH)}$	$\overline{WE}$ high pulse duration ‡¶	2H – 2		ns
$t_{d(WR)}$	Delay time, $\overline{WE}$ high to $\overline{RD}$ low	2H – 10		ns
$t_{su(D)W}$	Setup time, write data valid before $\overline{WE}$ high ‡	2H – 20	2H¶§	ns
$t_{h(D)W}$	Hold time, write data valid after $\overline{WE}$ high ‡	H – 5	H+10¶	ns
$t_{en(D)W}$	Enable time, $\overline{WE}$ to data bus driven	–5¶		ns

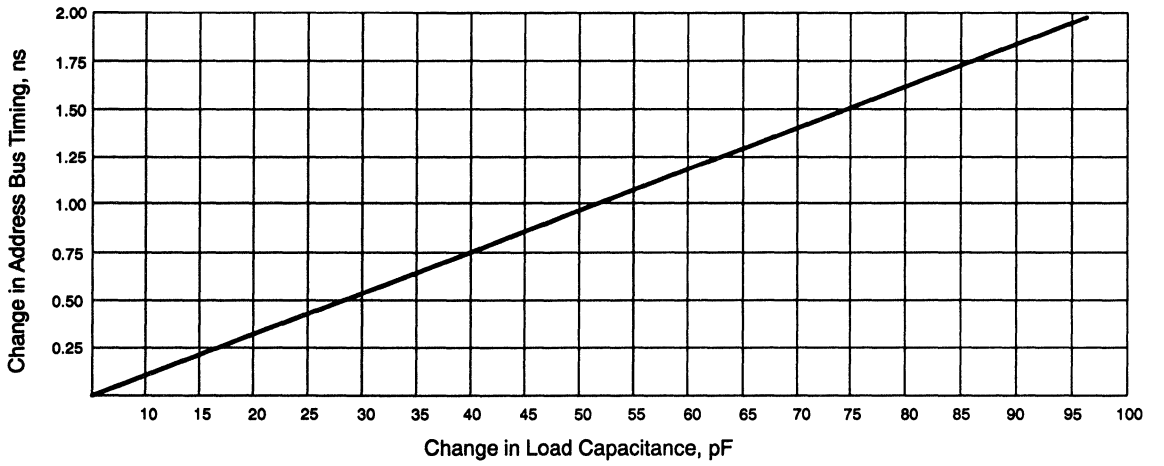
† A15–A0, $\overline{PS}$, $\overline{DS}$, $\overline{IS}$, R/$\overline{W}$, and $\overline{BR}$ timings are all included in timings referenced as address.
‡ $\overline{STRB}$ and $\overline{WE}$ edges are 0–4 ns from CLKOUT1 edges on writes. Rising and falling edges of these signals track each other; tolerance of resulting pulsewidths is ± 2 ns, not ± 4 ns. See Appendix B for logical device interface timings.
¶ Values derived from characterization data and are not tested.
§ This value holds true for zero or one wait state only.
# See Figure A–9 for address bus timing variation with load capacitance.

Figure A–8. Memory and Parallel I/O Interface Read and Write Timing



Note: All timings are for 0 wait states. However, external writes always require two cycles to prevent external bus conflicts. The above diagram illustrates a one-cycle read and a two-cycle write and is not drawn to scale. All external writes immediately preceded by an external read or immediately followed by an external read require three machine cycles.

Figure A–9. Address Bus Timing Variation With Load Capacitance



A.3.6 Ready Timing for Externally Generated Wait States

Table A–13. Timing Requirements Over Recommended Operating Conditions

Parameter		Min	Max	Unit
$t_{su(R-CO)}$	READY setup time before CLKOUT1 rises	10		ns
$t_{h(CO-R)}$	READY hold time after CLKOUT1 rises	0		ns
$t_{su(R)R}$	READY setup time before $\overline{RD}$ falls	10		ns
$t_{h(R)R}$	READY hold time after $\overline{RD}$ falls	5		ns
$t_{v(R)W}$	READY valid after $\overline{WE}$ falls	H – 15		ns
$t_{h(R)W}$	READY hold after $\overline{WE}$ falls	H + 5		ns

Note: The external READY input is sampled only after the internal software wait states are completed.

Figure A–10. Ready Timing for Externally Generated Wait States During an External Read Cycle

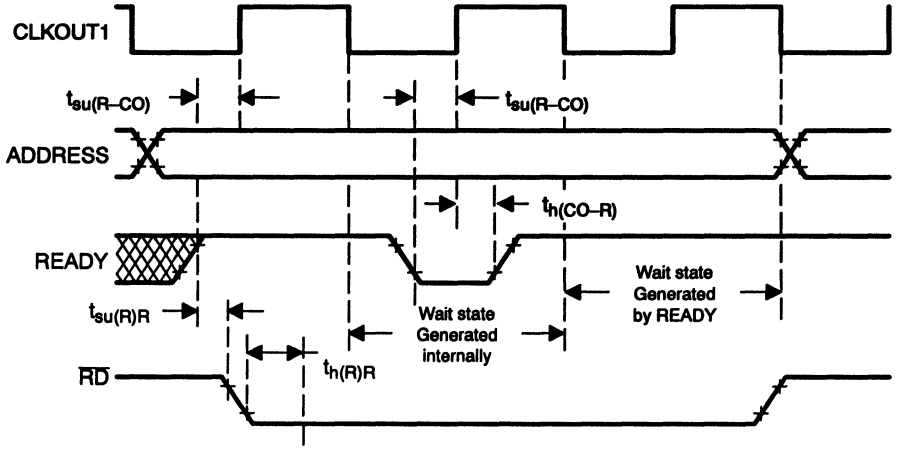
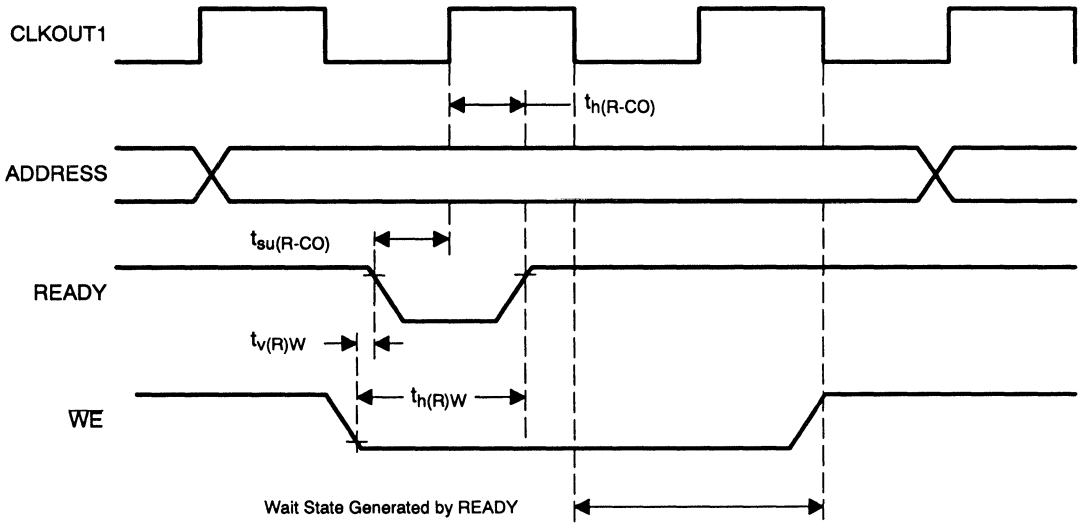


Figure A–11. Ready Timing for Externally Generated Wait States During an External Write Cycle



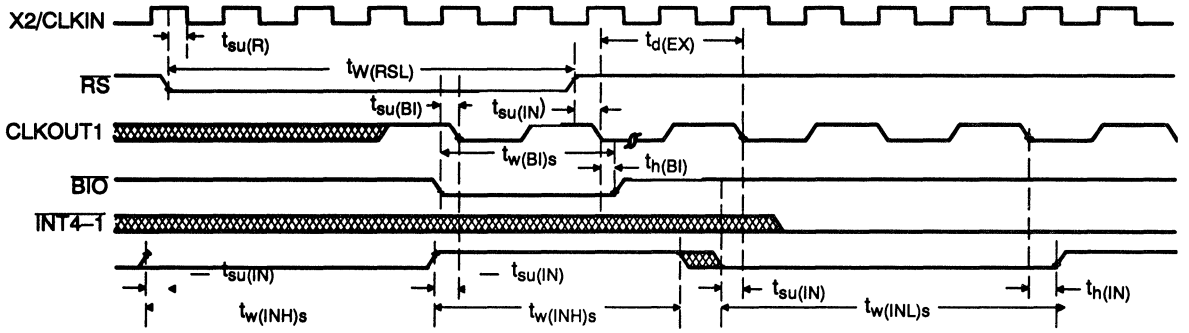
A.3.7 Reset, Interrupt, and $\overline{BIO}$ Timings

Table A–14. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{su(IN)}$	$\overline{INT1}$–$\overline{INT4}$, $\overline{NMI}$, $\overline{RS}$ setup time before CLKOUT1 low †	15		ns
$t_{h(IN)}$	$\overline{INT1}$–$\overline{INT4}$, $\overline{NMI}$, $\overline{RS}$ hold time after CLKOUT1 low †	0		ns
$t_{w(INL)s}$	$\overline{INT1}$–$\overline{INT4}$, $\overline{NMI}$ low pulse duration, synchronous	4H+15‡		ns
$t_{w(INH)s}$	$\overline{INT1}$–$\overline{INT4}$, $\overline{NMI}$ high pulse duration, synchronous	2H+15‡		ns
$t_{w(INL)a}$	$\overline{INT1}$–$\overline{INT4}$, $\overline{NMI}$ low pulse duration, asynchronous #	6H+15‡		ns
$t_{w(INH)a}$	$\overline{INT1}$–$\overline{INT4}$, $\overline{NMI}$ high pulse duration, asynchronous #	4H+15‡		ns
$t_{su(R)}$	$\overline{RS}$ set up time before X2/CLKIN low	10		ns
$t_{w(RSL)}$	$\overline{RS}$ low pulse duration	12H		ns
$t_{d(EX)}$	$\overline{RS}$ high to reset vector fetch	34H		ns
$t_{w(BI)s}$	$\overline{BIO}$ low pulse duration, synchronous	15		ns
$t_{w(BI)a}$	$\overline{BIO}$ low pulse duration, asynchronous #	H+15		ns
$t_{su(BI)}$	$\overline{BIO}$ setup before CLKOUT1 low	15		ns
$t_{h(BI)}$	$\overline{BIO}$ hold time after CLKOUT1 low	0		ns

† These parameters must be met to use the synchronous timings. Both reset and the interrupts can operate asynchronously. The pulse widths require an extra half-cycle to guarantee internal synchronization.
‡ If in IDLE2, add 4H to these timings.
# Values derived from characterization data and are not tested.

Figure A–12. Reset, Interrupt, and $\overline{BIO}$ Timings



A.3.8 Instruction Acquisition ($\overline{IAQ}$), Interrupt Acknowledge ($\overline{IACK}$), External Flag (XF), and TOUT Timings

Table A–15. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

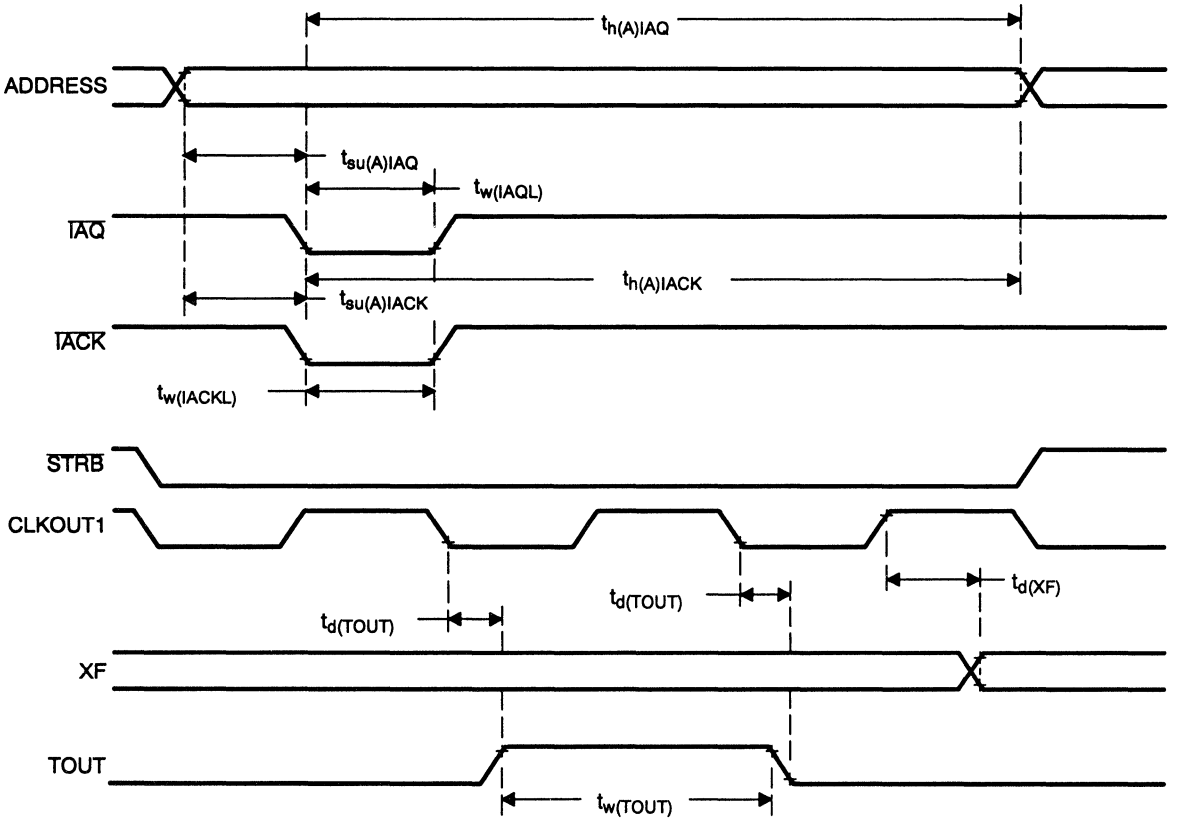
Parameter		Min	Max	Unit
$t_{su(A)IAQ}$	Setup time, address valid before $\overline{IAQ}$ low †	H – 12¶		ns
$t_{h(A)IAQ}$	Hold time, address valid after $\overline{IAQ}$ low	H –10¶		ns
$t_{w(IAQL)}$	$\overline{IAQ}$ low pulse duration	H – 10¶		ns
$t_{d(TOUT)}$	Delay time, CLKOUT1 falling to TOUT	–6	6	ns
$t_{su(A)IACK}$	Setup time, address valid before $\overline{IACK}$ low ‡	H – 12¶		ns
$t_{h(A)IACK}$	Hold time, address valid after $\overline{IACK}$ high ‡	H – 10¶		ns
$t_{w(IACKL)}$	$\overline{IACK}$ low pulse duration	H – 10¶		ns
$t_{w(TOUT)}$	TOUT pulse width	2H – 12		ns
$t_{d(XF)}$	Delay time, XF valid after CLKOUT1	0	12	ns

† $\overline{IAQ}$ goes low during an instruction acquisition. It goes low only on the first cycle of the read when wait states are used. The falling edge should be used to latch the valid address. The AVIS bit in the PMST register must be set to zero for the address to be valid when the instruction being addressed resides in on-chip memory.

‡ $\overline{IACK}$ goes low during the fetch of the first word of the interrupt vector. It goes low only on the first cycle of the read when wait states are used. Address pins A1 – A4 can be decoded at the falling edge to identify the interrupt being acknowledged. The AVIS bit in the PMST register must be set to zero for the address to be valid when the vectors reside in on-chip memory.

¶ Valid only if the external address reflects the current instruction activity (that is, code is executing on chip with no external bus cycles and AVIS is on or code is executing off-chip).

Figure A–13. $\overline{IAQ}$, $\overline{IACK}$, and XF Timings Example With Two External Wait States



Note: $\overline{IAQ}$ and $\overline{IACK}$ are not affected by wait states.

A.3.9 External DMA Timing

Table A–16. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{d(H-HA)}$	Delay time, $\overline{HOLD}$ low to $\overline{HOLDA}$ low	4H	§	ns
$t_{d(HH-HA)}$	Delay time, $\overline{HOLD}$ high before $\overline{HOLDA}$ high	2H		ns
$t_{z(M-HA)}$	Address three-state before $\overline{HOLDA}$ low †	H – 15¶		ns
$t_{en(HA-M)}$	Enable time, $\overline{HOLDA}$ high to address driven	H – 5¶		ns
$t_{d(B-I)}$	Delay time, $\overline{XBR}$ low to $\overline{IAQ}$ low	4H¶	6H¶	ns
$t_{d(BH-I)}$	Delay time, $\overline{XBR}$ high to $\overline{IAQ}$ high	2H¶	4H¶	ns
$t_{d(D)XR}$	Delay time, read data valid after $\overline{XSTRB}$ low		40	ns
$t_{h(D)XR}$	Read data hold time after $\overline{XSTRB}$ high	0		ns
$t_{en(I-D)}$	Enable time, $\overline{IAQ}$ low to read data driven ‡	0¶	2H¶	ns
$t_{z(\overline{W})}$	XR/$\overline{W}$ low to data three-state	0¶	15¶	ns
$t_{z(I-D)}$	$\overline{IAQ}$ high to data three-state		H	ns
$t_{en(D)RW}$	Enable time, data from XR/$\overline{W}$ going high		4	ns

† This parameter includes all memory control lines.
‡ This parameter refers to the delay between the time the condition ($\overline{IAQ}$ = 0 and XR/$\overline{W}$ = 1) is satisfied and the time that the 'C5x data lines become valid.
§ $\overline{HOLD}$ is not acknowledged until current external access request is complete.
¶ Values derived from characterization data and are not tested.

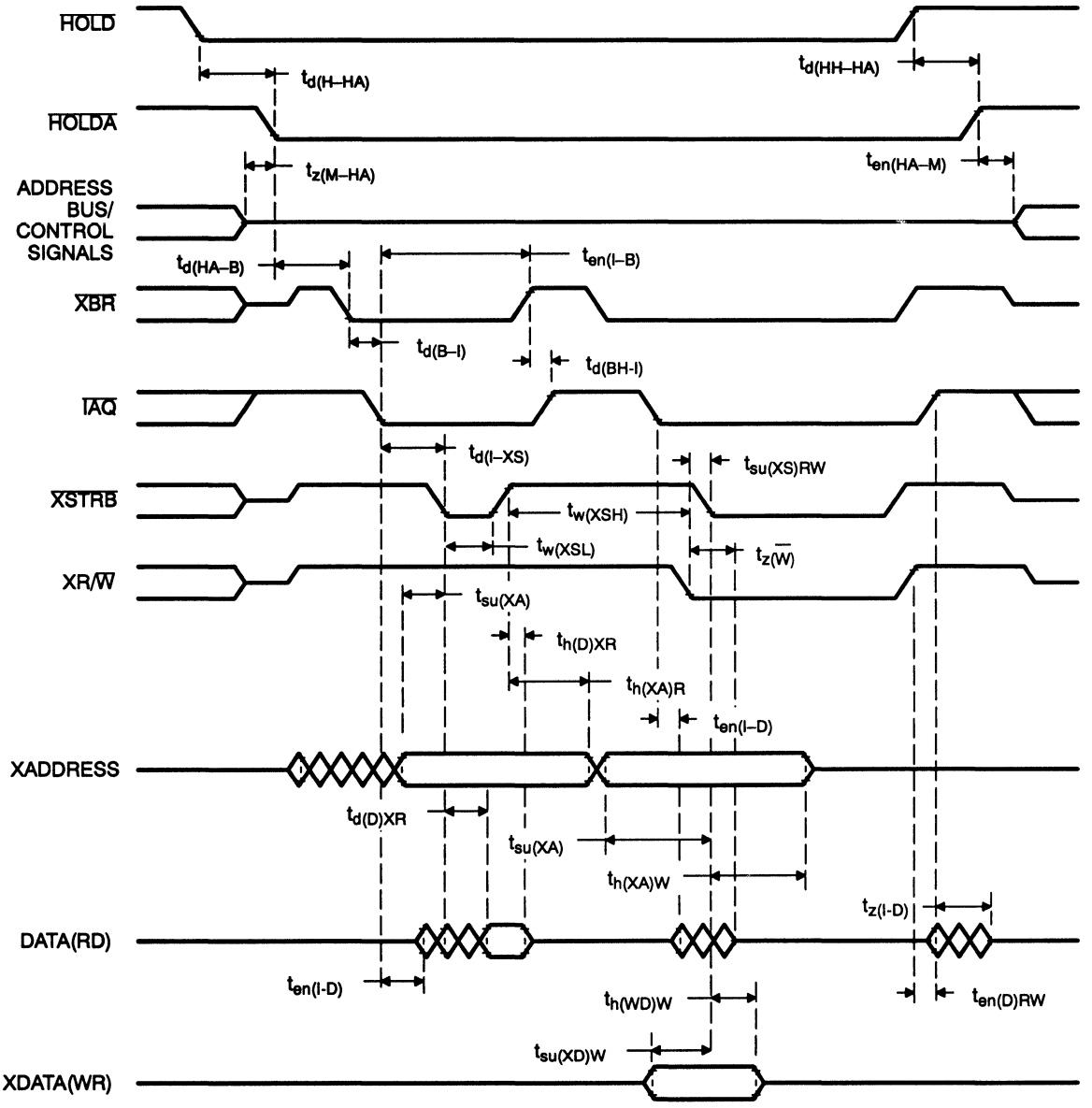
Note: X preceding a name refers to external drive of the signal.

Table A–17. Timing Requirements Over Recommended Operating Conditions

Parameter		Min	Max	Unit
$t_{d(HA-B)}$	Delay time, $\overline{HOLDA}$ low to $\overline{XBR}$ low †	0¶		ns
$t_{d(I-XS)}$	Delay time, $\overline{IAQ}$ low to $\overline{XSTRB}$ low †	0¶		ns
$t_{su(XA)}$	Setup time, Xaddress valid before $\overline{XSTRB}$ low	15		ns
$t_{su(XD)W}$	Setup time, Xdata valid before $\overline{XSTRB}$ low	15		ns
$t_{h(WD)W}$	Hold time, Xdata hold after $\overline{XSTRB}$ low	15		ns
$t_{h(XA)W}$	Hold time, Write Xaddress hold after $\overline{XSTRB}$ low	15		ns
$t_{w(XSL)}$	Width $\overline{XSTRB}$ low pulse	45		ns
$t_{w(XSH)}$	Width $\overline{XSTRB}$ high pulse	45		ns
$t_{su(XS)RW}$	Setup time, R/$\overline{W}$ valid before $\overline{XSTRB}$ low	20		ns
$t_{h(XA)R}$	Hold time, read Xaddress after $\overline{XSTRB}$ high	0		ns

† $\overline{XBR}$, XR/$\overline{W}$, and $\overline{XSTRB}$ lines should be pulled up with a 10-kΩ resistor to assure that they are in an inactive high state during the transition period between the TMS320C5x driving them and the external circuit driving them.
¶ Values derived from characterization data and are not tested.
Note: X preceding a name refers to external drive of the signal.

Figure A–14. External DMA Timing



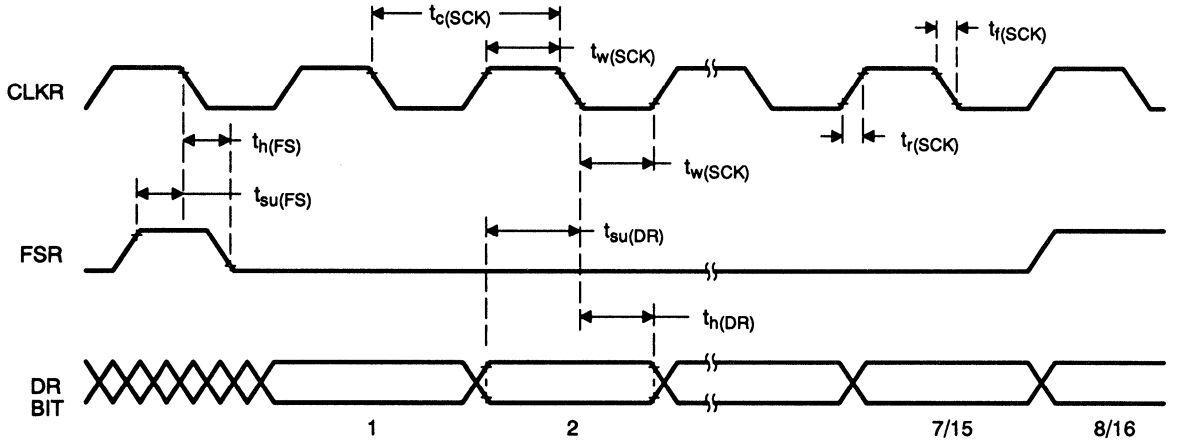
A.3.10 Serial Port Receive Timing

Table A–18. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{c(SCK)}$	Serial port clock cycle time	5.2H	‡	ns
$t_{f(SCK)}$	Serial port clock fall time		8¶	ns
$t_{r(SCK)}$	Serial port clock rise time		8¶	ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	2.1H		ns
$t_{su(FS)}$	FSR setup time before CLKR falling edge	10		ns
$t_{h(FS)}$	FSR hold time after CLKR falling edge	10		ns
$t_{su(DR)}$	DR setup time before CLKR falling edge	10		ns
$t_{h(DR)}$	DR hold time after CLKR falling edge	10		ns

‡ The serial port design is fully static and therefore can operate with $t_{c(SCK)}$ approaching ∞. It is characterized approaching an input frequency of 0 Hz but tested at a much higher frequency to minimize test time.
¶ Values derived from characterization data and are not tested.

Figure A–15. Serial Port Receive Timing



A.3.11 Serial Port Transmit Timing of External Clocks and External Frames (see Note)

Table A–19. Switching Characteristics Over Recommended Operating Conditions ($S = 0.5t_{c(SCK)}$)

Parameter		Min	Max	Unit
$t_{d(DX)}$	Delay time, DX valid after CLKX rising		25	ns
$t_{dis(DX)}$	Disable time, DX after CLKX rising		40	ns
$t_{h(DX)}$	Hold time, DX valid after CLKX rising	–5		

Table A–20. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{c(SCK)}$	Serial port clock cycle time	5.2H	‡	ns
$t_{f(SCK)}$	Serial port clock fall time		8 ¶	ns
$t_{r(SCK)}$	Serial port clock rise time		8 ¶	ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	2.1H		ns
$t_{d(FS)}$	FSX delay time after CLKX rising edge		2H–8	ns
$t_{h(FS)}$	FSX hold time after CLKX falling edge	10		ns
$t_{h(FS)H}$	FSX hold time after CLKX rising edge		2H–8 †	ns

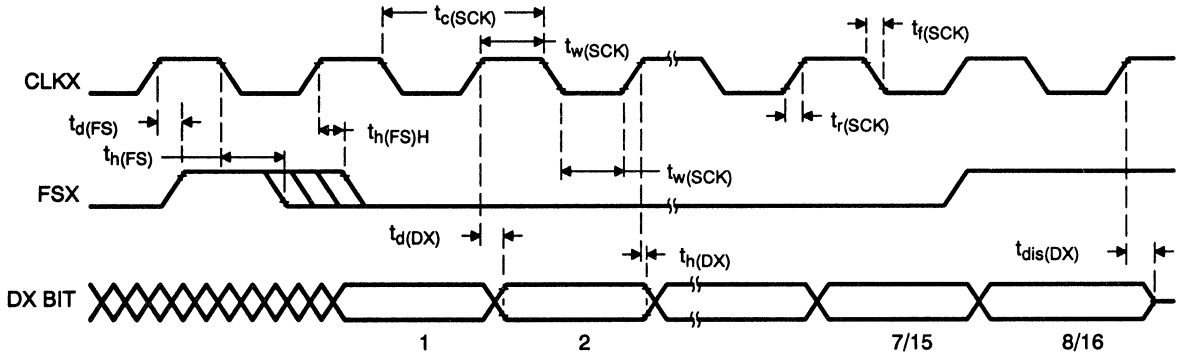
† If the FSX pulse does not meet this specification, the first bit of serial data will be driven on the DX pin until the falling edge of FSX. After the falling edge of FSX, data will be shifted out on the DX pin. The transmit buffer empty interrupt will be generated when the $t_{h(FS)}$ and $t_{h(FS)H}$ specification is met.

‡ The serial port design is fully static and therefore can operate with $t_{c(SCK)}$ approaching ∞. It is characterized approaching an input frequency of 0 Hz but tested at a much higher frequency to minimize test time.

¶ Values derived from characterization data and are not tested.

Note: Internal clock with external FSX and vice versa are also allowable. However, FSX timings to CLKX are always defined depending on the source of FSX, and CLKX timings are always dependent upon the source of CLKX. Specifically, the relationship of FSX to CLKX is independent of the source of CLKX. Table A–20 shows external FSX and external CLKX timings; Table A–21 shows internal FSX and internal CLKX timings.

Figure A–16. Serial Port Transmit Timing of External Clocks and External Frames



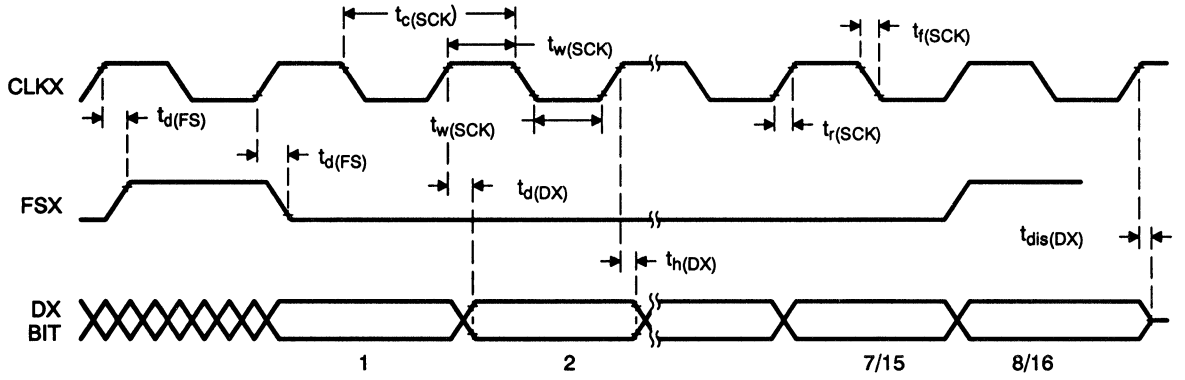
A.3.12 Serial Port Transmit Timing of Internal Clocks and Internal Frames (see Note)

Table A–21. Switching Characteristics Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$, $S = 0.5t_{c(SCK)}$)

Parameter		Min	Typ	Max	Unit
$t_{d(FS)}$	Delay time, CLKX rising to FSX			25	ns
$t_{d(DX)}$	Delay time, CLKX rising to DX			25	ns
$t_{dis(DX)}$	Disable time, CLKX rising to DX			40	ns
$t_{c(SCK)}$	Serial port clock cycle time		8H		ns
$t_{f(SCK)}$	Serial port clock fall time		5		ns
$t_{r(SCK)}$	Serial port clock rise time		5		ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	4H – 20			ns
$t_{h(DX)}$	Hold time, DX valid after CLKX rising	–5			ns

Note: Internal clock with external FSX and vice versa are also allowable. However, FSX timings to CLKX are always defined depending on the source of FSX, and CLKX timings are always dependent upon the source of CLKX. Specifically, the relationship of FSX to CLKX is independent of the source of CLKX. Table A–20 shows external FSX and external CLKX timings; Table A–21 shows internal FSX and internal CLKX timings.

Figure A–17. Serial Port Transmit Timing of Internal Clocks and Internal Frames



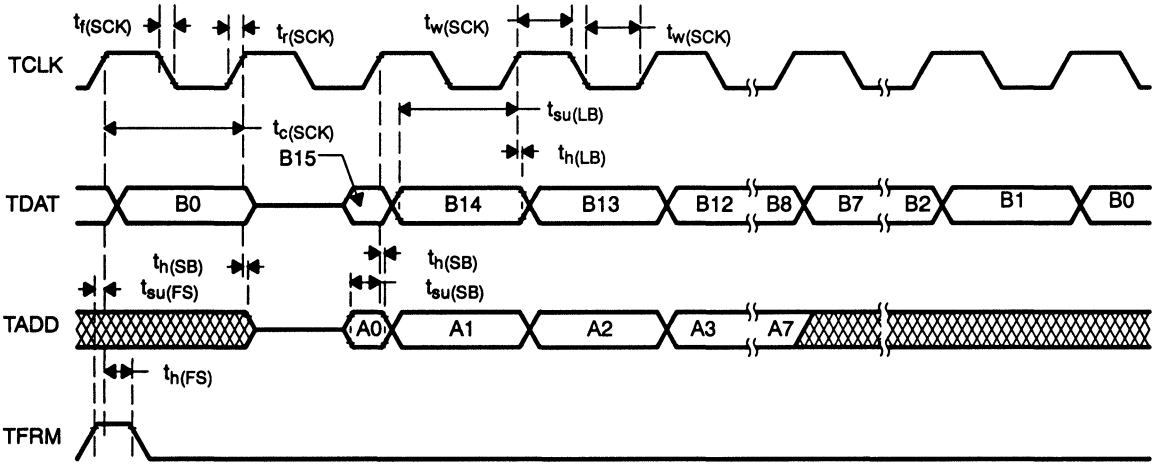
A.3.13 Serial Port Receive Timing in TDM Mode

Table A–22. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Max	Unit
$t_{c(SCK)}$	Serial port clock cycle time	5.2H	§	ns
$t_{f(SCK)}$	Serial port clock fall time		8#	ns
$t_{r(SCK)}$	Serial port clock rise time		8#	ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	2.1H		ns
$t_{su(LB)}$	TDAT/TADD setup time before TCLK rising	30		ns
$t_{h(LB)}$	TDAT/TADD hold time after TCLK rising	–5		ns
$t_{su(SB)}$	TDAT/TADD setup time before TCLK rising †	25		ns
$t_{h(SB)}$	TDAT/TADD hold time after TCLK rising †	0		ns
$t_{su(FS)}$	TRFM setup time before TCLK rising edge ‡	10		ns
$t_{h(FS)}$	TRFM hold time after TCLK rising edge ‡	10		ns

† These parameters apply only to the first bits in the serial bit string.
‡ TFRM timing and waveforms shown in Figure A–18 are for external TFRM. TFRM can also be configured as internal. The TFRM internal case is illustrated in the transmit timing diagram in Figure A–19.
§ The serial port design is fully static and therefore can operate with $t_{c(SCK)}$ approaching ∞. It is characterized approaching an input frequency of 0 Hz but tested at a much higher frequency to minimize test time.
# Values derived from characterization data and are not tested.

Figure A–18. Serial Port Receive Timing in TDM Mode



A.3.14 Serial Port Transmit Timing in TDM Mode

Table A–23. Switching Characteristics Over Recommended Operating Conditions ($S = 0.5t_{c(SCK)}$)

Parameter		Min	Typ	Max	Unit
$t_{h(AD)}$	Hold time, TDAT/TADD valid after TCLK rising	–2			ns
$t_{d(FS)}$	Delay time, TFRM valid after TCLK rising ‡	H		3H+10	ns
$t_{d(AD)}$	Delay time, TCLK to valid TDAT/TADD			25	ns

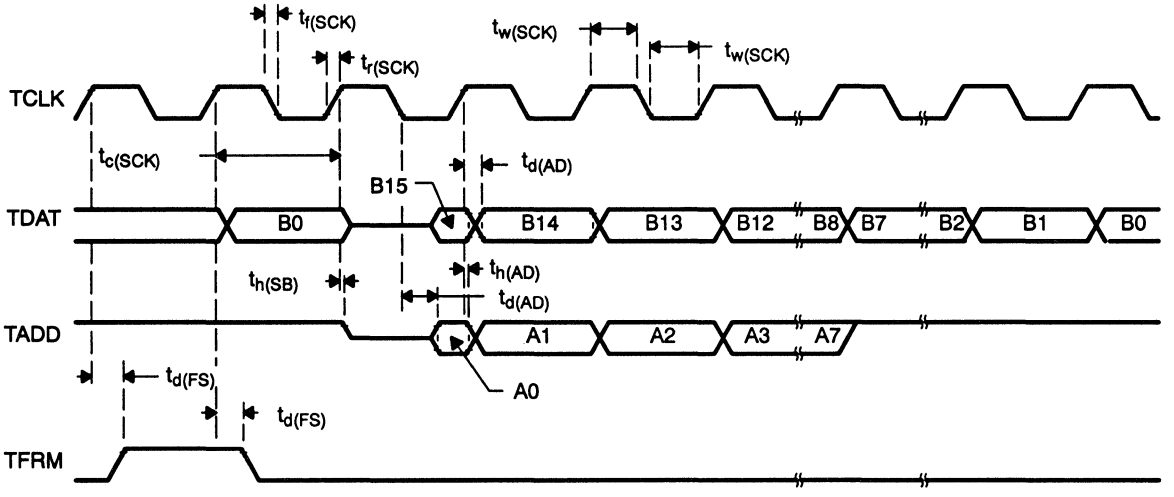
† These parameters apply only to the first bits in the serial bit string.
‡ TFRM timing and waveforms shown in Figure A–19 are for internal TFRM. TFRM can also be configured as external, and the TFRM external case is illustrated in the receive timing diagram in Figure A–18.

Table A–24. Timing Requirements Over Recommended Operating Conditions ($H = 0.5t_{c(CO)}$)

Parameter		Min	Typ	Max	Unit
$t_{c(SCK)}$	Serial port clock cycle time	5.2H	8H†	‡	ns
$t_{f(SCK)}$	Serial port clock fall time			8#	ns
$t_{r(SCK)}$	Serial port clock rise time			8#	ns
$t_{w(SCK)}$	Serial port clock low/high pulse duration	2.1H			ns

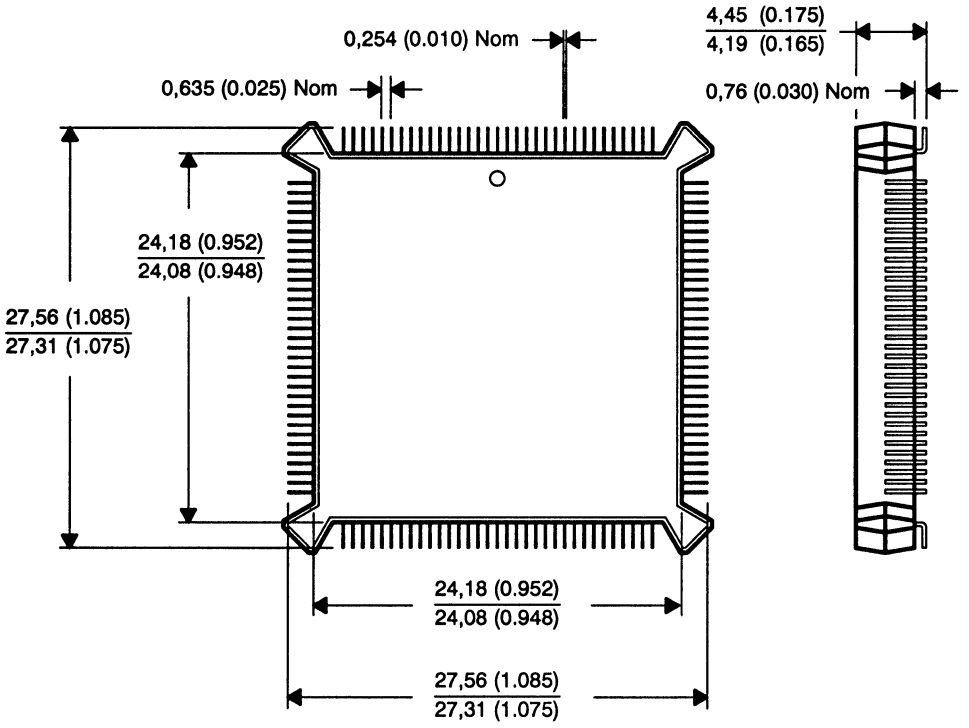
† When SCK is generated internally.
‡ The serial port design is fully static and therefore can operate with $t_{c(SCK)}$ approaching ∞. It is characterized approaching an input frequency of 0 Hz but tested at a much higher frequency to minimize test time.
# Values derived from characterization data and are not tested.

Figure A–19. Serial Port Transmit Timing in TDM Mode



A.4 Mechanical Data

Figure A–20. 132-Pin Quad Flat Pack Plastic Package



Thermal Resistance Characteristics

Parameter	°C/W	Air Flow LFPM
$R_{\theta JC}$	10	N/A
$R_{\theta JA}$	50	0
$R_{\theta JA}$	36	200
$R_{\theta JA}$	28	400
$R_{\theta JA}$	24	600
$R_{\theta JA}$	22	800
$R_{\theta JA}$	20	1000

ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

Note: The contact points are within 0,15 (0.006) of being planar.

Appendix B

External Interface Timings

This appendix discusses functional timing operations on the external memory interface bus. Detailed timing specifications for all 'C5x signals are contained in Appendix A, *Electrical Specifications*.

The 'C5x memory is organized into four selectable spaces: program, local data, global data, and I/O space. These spaces are multiplexed through a 16-bit data bus and a 16-bit address bus. Each space is selected by its corresponding select signal: data select ($\overline{DS}$), program select ($\overline{PS}$), and I/O space select ($\overline{IS}$). Global data memory accesses are distinguished by the bus request ($\overline{BR}$) pin. The read and write diagrams shown apply to accesses to all spaces.

B.1 Read/Write Timings

All bus cycles comprise integral numbers of CLKOUT1 cycles. One CLKOUT1 cycle is defined to be from one falling edge of CLKOUT1 to the next falling edge of CLKOUT1. For full-speed, zero-wait state operation, reads require one cycle and writes require two cycles. A write immediately preceded by a read or immediately followed by a read requires three bus cycles.

For read cycles, $\overline{\text{STRB}}$ goes low and ADDRESS becomes valid with the falling edge of CLKOUT1. The $\overline{\text{RD}}$ signal then goes low with the rising edge of CLKOUT1 and goes high again at the next falling edge of CLKOUT1 (for zero wait-states read cycles). For one more wait state (multicycle) read, $\overline{\text{RD}}$ stays low but goes high again with the falling edge of CLKOUT1 before the next cycle, even if the cycles are contiguous. Read data is sampled at the rising edge of $\overline{\text{RD}}$.

The R/$\overline{\text{W}}$ signal goes high at least one half CLKOUT1 cycle before any read cycle; for contiguous read cycles, $\overline{\text{STRB}}$ stays low. At the end of a read cycle or sequence of reads, $\overline{\text{STRB}}$ goes high along with $\overline{\text{RD}}$ on the falling edge of CLKOUT1.

Write cycles always have at least one inactive (pad) cycle of CLKOUT1 before and after the actual write operation, including contiguous writes. This allows a smooth transition between the write and any adjacent bus operations as well as other writes. For this pad cycle, $\overline{\text{STRB}}$ and $\overline{\text{WE}}$ are always high. The R/$\overline{\text{W}}$ signal always changes state on the rising edge of CLKOUT1 during the pad cycle before and after a write or sequence of writes. This prevents bus contention when making the transition between read and write operations. Note that for a sequence of contiguous writes, R/$\overline{\text{W}}$ stays low.

Timing of valid addresses for writes differs, depending on what activities occur before and after the write; between writes, and for the first and last write in a series, valid ADDRESS occurs on the rising edge of CLKOUT1. If a read immediately follows a write or series of writes, valid ADDRESS for that read cycle occurs one half CLKOUT1 cycle early — that is, on the rising edge, rather than on the falling edge, of CLKOUT1. Note that this is an exception to the usual read cycle address timing.

For the actual write operation, $\overline{\text{STRB}}$ and $\overline{\text{WE}}$ both go low on the falling edge of CLKOUT1 and stay low until the next falling edge of CLKOUT1 (for zero wait-state write cycles). For one or more wait-state (multicycle) writes, $\overline{\text{STRB}}$ and $\overline{\text{WE}}$ remain low but go high again on the falling edge of CLKOUT1 at the beginning of the pad cycle. *Write data* is driven approximately at the falling edge of $\overline{\text{STRB}}$ and $\overline{\text{WE}}$ and is held for approximately one half cycle of CLKOUT1 after $\overline{\text{STRB}}$ and $\overline{\text{WE}}$ go high (see Appendix A for actual timing specifications).

Note that transitions on the external parallel interface control outputs (CLKOUT1, $\overline{\text{STRB}}$, $\overline{\text{WE}}$, and $\overline{\text{RD}}$) are all initiated by the same two internal

clocks. Since these signals also use the same output buffer circuitry, they all switch within close tolerances of each other, as specified in Appendix A.

Transitions on the address bus and other related outputs ($\overline{IS}$, $\overline{PS}$, $\overline{DS}$, R/$\overline{W}$, and $\overline{BR}$) are initiated by the same internal signals that cause transitions on the control outputs; however, the internal device logic used to generate these outputs differs somewhat from the circuitry used for the control outputs. Because of this, transitions on the address bus and related outputs typically occur somewhat later than control-line transitions.

Timings of control outputs with respect to CLKOUT1 are specified in Appendix A; address timing with respect to CLKOUT1 can be derived from timings provided for address with respect to control signals and control signal timing with respect to CLKOUT1. Therefore, for example, the delay from CLKOUT1 falling to address bus valid at the begininng of a read cycle is calculated as [H – $t_{su(A)R}$] + *maximum positive* $\overline{RD}$ *to CLKOUT1 skew* (refer to Appendix A for specific timing values). Other interface timings with respect to CLKOUT1 can be calculated in the same manner.

The following timing diagrams illustrate the varieties of logical timings for both read and write cycles in various orders.

Figure B–1. Memory Interface Operation for Read-Read-Write (0 Wait States)

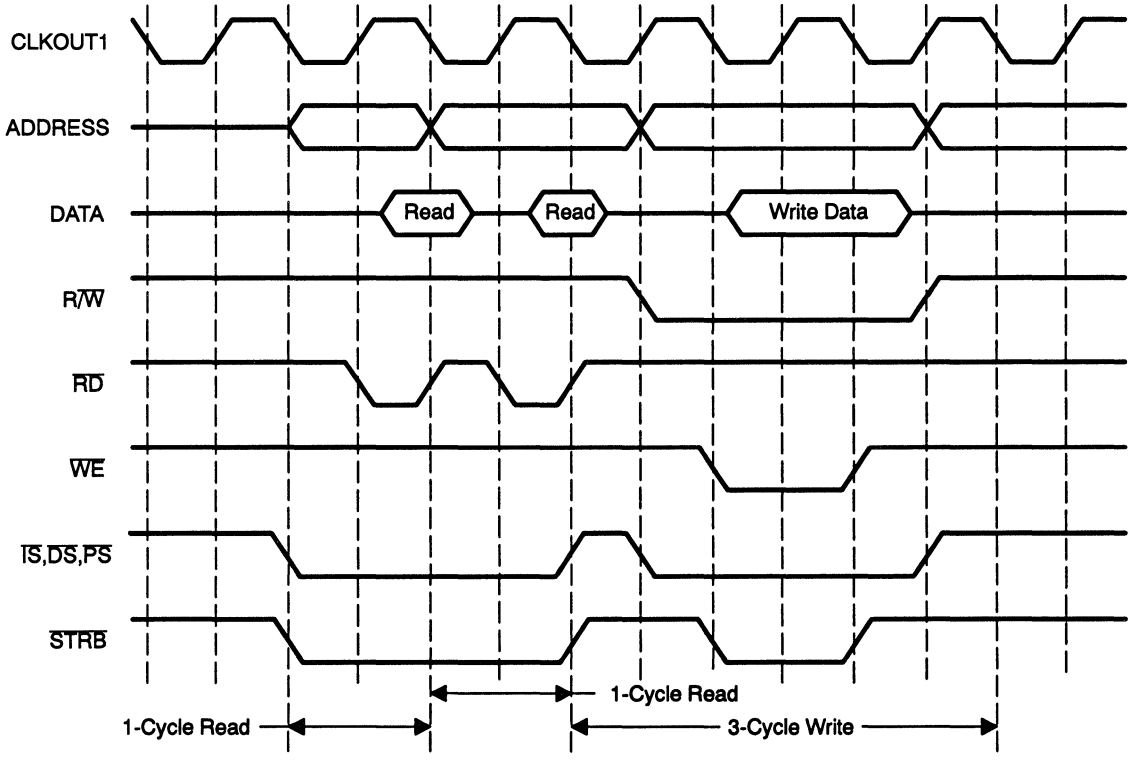


Figure B–2. Memory Interface Operation for Write-Write-Read (0 Wait States)

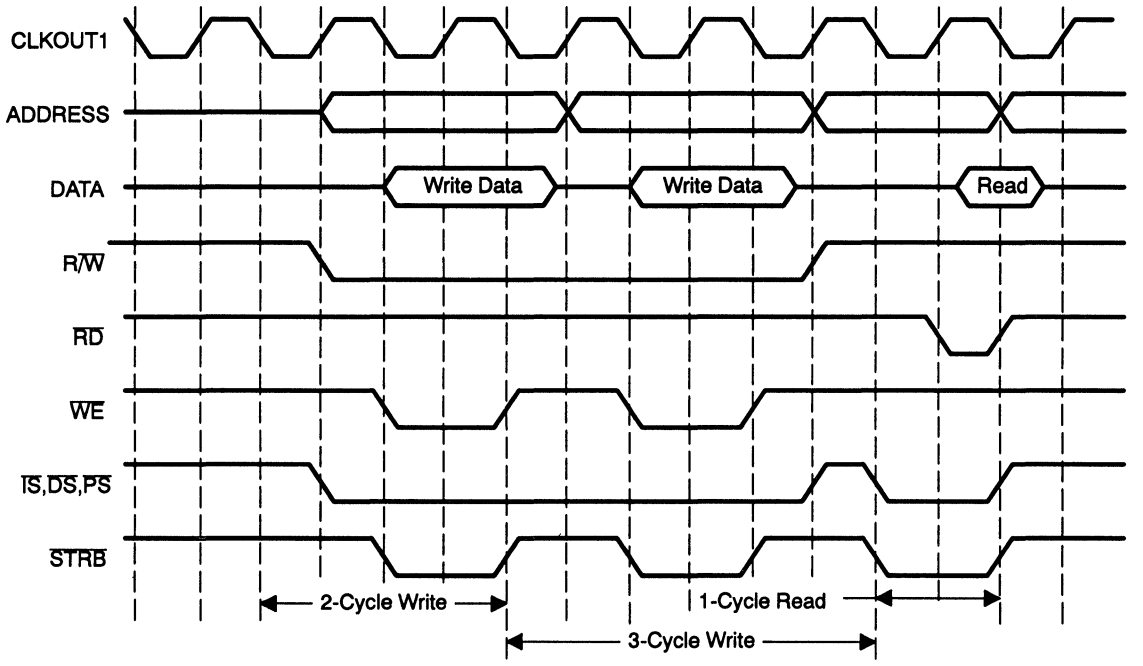
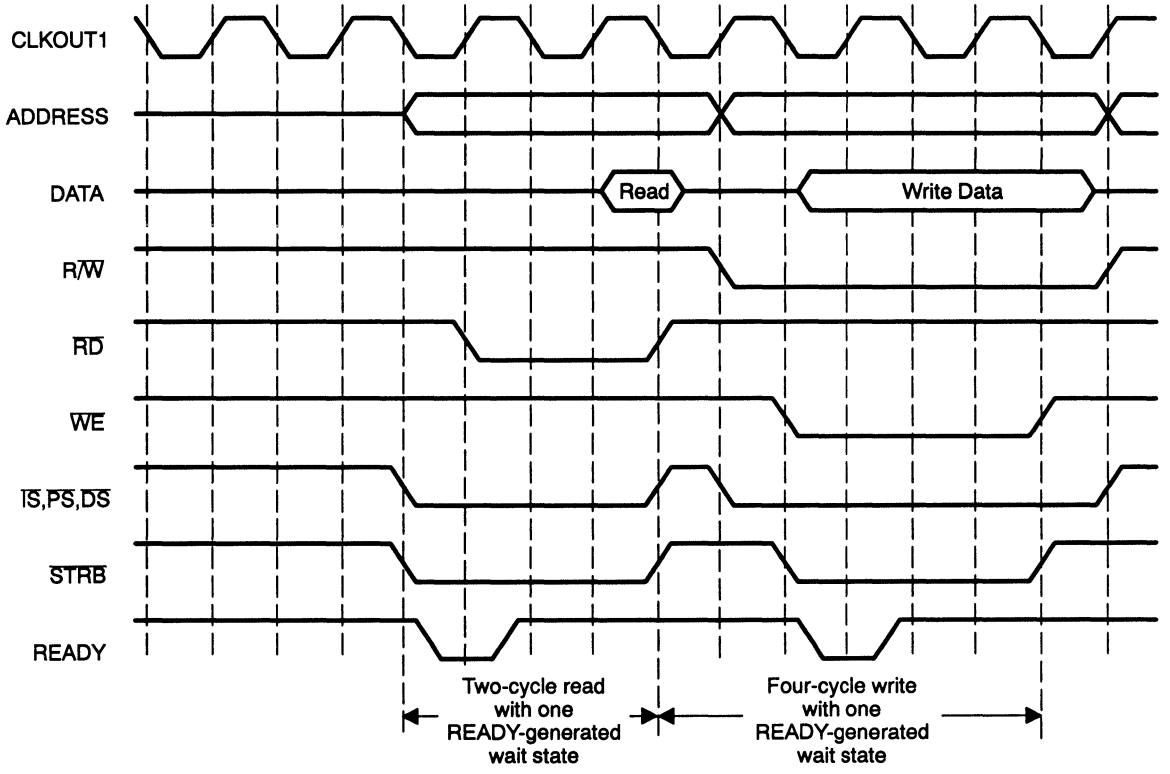


Figure B–3. Memory Interface Operation for Read-Write (1 Wait State)



Appendix C

Instruction Cycle Timings

This appendix details the instruction cycle timings for the 'C5x processors. Instructions are classified into several categories according to their cycle timings.

C.1 Instruction Cycle Summary

Each class of instructions is listed in a separate table showing the number of cycles required for a 'C5x instruction to execute in a given memory configuration singly or in repeat mode. The column headings in the table indicate the program source location (PR, PDA, PSA, PE), defined as follows:

PR The instruction executes from internal program ROM.
PDA The instruction executes from internal dual-access program RAM.
PSA The instruction executes from internal single-access program RAM.
PE The instruction executes from external program memory.

If a class of instructions requires memory operand(s), row divisions in table indicate the location(s) of the operand(s), as defined below:

DARAM The operand is in internal dual-access RAM.
SARAM The operand is in internal single-access RAM.
Ext The operand is in external memory.
ROM The operand is in internal program ROM.
MMR The operand is a memory-mapped register.
MMPORT The operand is a memory-mapped io port.

Note that the internal single-access memory on each 'C5x processor is divided into 2K-word blocks that are contiguous in address space:

'C50		
Four 2K-word block	0800h–0FFFh 1000h–17FFh 1800h–1FFFh 2000h–27FFh	Data address range
One 1K-word block	2800h–2BFFh	Data address range
'C51		
One 1K-word block	0800h–0BFFh	Data address range
'C53		
One 2K-word block	0800h–0FFFh	Data address range
One 1K-word block	1000h–13FFh	Data address range

All 'C5x processors support parallel accesses to these internal single-access blocks. However, one single-access block allows only one access per cycle. In other words, the processor can read/write on one single-access memory block while accessing another single-access block.

The number of cycles required for each instruction is given in terms of the processor machine cycles (CLKOUT1 period). The additional wait states for program/data memory and I/O accesses are defined below:

p Program memory wait states. Represents the number of additional clock cycles the device waits for external program memory to respond to an access.

d Data memory wait states. Represents the number of additional clock cycles the device waits for external data memory to respond to an access.

Io I/O wait states. Represents the number of additional clock cycles the device waits for an external I/O to respond to an access.

n Repetitions (where n>2 to fill the pipeline). Represents the number of times a repeated instruction is executed.

The above variables can also use the subscripts *src, dst,* and *code* to indicate source, destination, and code, respectively.

Note that all external reads require at least one machine cycle, while all external writes require at least two machine cycles. However, if an external write is immediately followed or preceded by an external read cycle, the external write requires three cycles. See Appendix B for details. If an on-chip wait-state generator is used to add *m* (*m*>0) wait states to an external access, both the external reads and the external writes require *m*+1 cycles, assuming that the external READY line is pulled high. If the READY input line is used to add *m* additional cycles to an external access, external reads require *m*+1 cycles and external write accesses require *m*+2 cycles. Refer to software wait state generation in Section 5.3 and to Appendix A for READY electrical specs.

The instruction cycle timings are based on following assumptions:

- ❑ At least the next four instructions are fetched from the same memory section (internal or external) that was used to fetch the current instruction (except in case of PC discontinuity instructions like B, CALL, etc.).
- ❑ In the single execution mode, there is no pipeline conflict between the current instruction and the instructions immediately preceding or following that instruction. The only exception is the conflict between the fetch phase of the pipeline and the memory read/write (if any) access of the instruction under consideration. See Chapter 3 for pipeline operation.
- ❑ In the repeat execution mode, all conflicts caused by the pipelined execution of that instruction are considered.

Class I

1-word, 1-cycle, no memory operands

ABS, ADCB, ADD, ADDB, ADRK, ANDB, APAC, BSAR, CLRC, SETC, CMPL, CMPR, CRGT, CRLT, EXAR, IDLE, IDLE2, LACB, LACL #k, MAR, MPY #k, NEG, NOP, NORM, ORB, PAC, POP, PUSH, RPT #k, ROL, ROLB, ROR, RORB, SACB, SATH, SATL, SBB, SBBB, SBRK, SFL, SFLB, SFR, SFRB, SPAC, SPM, SUB #k, XC, XORB, ZAP, ZPR

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
1	1	1	1+p
Cycle Timings for a Repeat (RPT) Execution†			
n	n	n	n+p

† ADD, ADRK, LACL, MPY, SBRK, SPM, SUB, XC, and RPT are nonrepeatable instructions.

Class IIA

1-word, 1-cycle, memory read operand

ADD, ADDC, ADDS, ADDT, AND, BIT, BITT, CPL, LACC, LACL, LACT, LPH, LT, LTA, LTP, LTS, MPY, MPYA, MPYS, MPYU, OR, PSHD, RPT, SQRA, SQRS, SUB, SUBB, SUBC, SUBS, SUBT, XOR, ZALR

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	1+d	1+d	1+d	2+d+p
Cycle Timings for a Repeat (RPT) Execution‡				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+1†	n+p
Operand Ext	n+nd	n+nd	n+nd	n+1+p+nd

† If the operand and the code are in the same SARAM block.
‡ RPT is a nonrepeatable instruction.

Class IIB

1-word, 1-cycle, memory-mapped register read

LAMM

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand MMR†	1	1	1	1+p
Operand MMPORT	$1+io_{src}$	$1+io_{src}$	$1+iod_{src}$	$1+2+p+iod_{src}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand MMR‡	n	n	n	n+p
Operand MMPORT	$n+mio_{src}$	$n+mio_{src}$	$n+mio_{src}$	$n+p+mio_{src}$

† Add one more cycle for peripheral memory-mapped access.
‡ Add *n* more cycles for peripheral memory-mapped access.

Class III

2-word, 2-cycle, long-immediate operand, no memory access

ADD, AND, LACC, LAR, MPY, OR, SUB, XOR, RPT, RPTB, RPTZ

Cycle Timings for a Single Instruction			
PR	**PDA**	**PSA**	**PE**
2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution			
Not Repeatable			

Class IVA

1-word, 1-cycle, memory write operand

SACH, SACL, SAR, SPH, SPL, SST #0, SST #1, POPD

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 2†	1+p
Operand Ext	2+d	2+d	2+d	4+d+p

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	n	n	n n+2†	n+p
Operand Ext	2n+nd	2n+nd	2n+nd	2n+2+nd+p

† If the operand and the code are in the same SARAM block.

Class IVB

1-word, 1-cycle, memory-mapped register write

SAMM

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand MMR†	1	1	1	1+p
Operand MMPORT	2+io_{dst}	2+io_{dst}	2+io_{dst}	4+io_{dst}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand MMR‡	n	n	n	n+p
Operand MMPORT	2+nio_{dst}	2+nio_{dst}	2+nio_{dst}	2n+2+p+p nio_{dst}

† Add one more cycle if source is a peripheral memory-mapped register.
‡ Add *n* more cycles if source is a peripheral memory-mapped register.

Class V

1-word, 1-cycle, read and write memory

APL, OPL, XPL, DMOV, LTD

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	1	1	1	1+p
Operand SARAM	1	1	1 3†	1+p
Operand Ext	2+2d	2+2d	2+2d	5+2d+p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n	n	n	n+p
Operand SARAM	2n–2	2n–2	2n–2 2n+1†	2n–2+p
Operand Ext	4n–2+2nd	4n–2+2nd	4n–2+2nd	4n+1+2nd+p

† If the operand and the code are in the same SARAM block.

Class VI

2-word, 2-cycle, memory read and write

APL, OPL, XPL

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2	2+2p
Operand Ext	3+2d	3+2d	3+2d	6+2d+2p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n+1	n+1	n+1	n+1+2p
Operand SARAM	2n–1	2n–1	2n–1 2n+2†	2n–1+2p
Operand Ext	4n–1+2nd	4n–1+2nd	4n–1+2nd	4n+2+2nd+2p

† If the operand and the code reside in same SARAM block.

Class VIIa

2-word, 2-cycle, memory read operand

CPL #lk,dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2 3†	2+2p
Operand Ext	2+d	2+d	2+d	3+d+2p
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand DARAM	n+1	n+1	n+1	n+1+2p
Operand SARAM	n+1	n+1	n+1 n+2†	n+1+2p
Operand Ext	n+1	n+1	n+1	n+2+2p

† If the operand and the code are in the same SARAM block.

Class VIIb

2-word, 2-cycle, memory write operand

SPLK #lk

Cycle Timings for a Single Instruction				
	PR	PDA	PSA	PE
Operand DARAM	2	2	2	2+2p
Operand SARAM	2	2	2 3†	2+2p
Operand Ext	3+d	3+d	3+d	5+d+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† If the operand and the code are in the same SARAM block.

Class VIII

2-word, 4-cycle, PC discontinuity, no delay slot

B, BANZ, BCND, CALL, CC

Cycle Timings for a Single Instruction				
	PR	PDA	PSA	PE
Conditions True	4	4	4	4+4p‡
Condition False†	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† Applicable only to conditional instructions.

‡ The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.

Class IX

2-word, 2-cycle, PC discontinuity, 2 delayed slots

BD, BANZD, BCNDD, CALLD, CCD

Cycle Timings for a Single Instruction				
	PR	PDA	PSA	PE
Conditions True	2	2	2	2+2p
Condition False†	2	2	2	2+2p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† Applicable only to conditional instructions.

Class X

1-word, 4-cycle, PC discontinuity, no delayed slots

BACC, CALA, RETC, RET, NMI, INTR, RETE, RETI, TRAP

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	4	4	4	4+3p†
Condition False‡	2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† The 'C5x performs speculative fetching by reading two additional instruction words. If PC discontinuity is taken, these two instruction words are discarded.
‡ Applicable only to conditional instructions.

Class XI

1-word, 2-cycle, PC discontinuity, 2 delayed slots

BACCD, CALAD, RETCD, RETD, TRAPD

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Conditions True	2	2	2	2+p
Condition False†	2	2	2	2+p
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† Only applicable to conditional instructions.

Class XII

2-word, 3-cycle, block data transfer, data to data space

BLDD #lk,dma; BLDD dma,#lk

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	3	3	3	3+2p
Source SARAM Destination DARAM	3	3	3	3+2p
Source Ext Destination DARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$	$3+d_{src}+2p$

Source DARAM Destination SARAM	3	3	3 4†	3+2p
Source SARAM Destination SARAM	3	3	3 4†	3+2p
Source Ext Destination SARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$ $4+d_{src}$	$3+d_{src}+2p$
Source DARAM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	
Source SARAM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	
Source Ext Destination Ext	$4+d_{src}+d_{dst}$	$4+d_{src}+d_{dst}$	$4+d_{src}+d_{dst}$	$6+d_{src}+d_{dst}+2p$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+2	n+2	n+2	n+2+2p
Source SARAM Destination DARAM	n+2	n+2	n+2	n+2+2p
Source Ext Destination DARAM	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}$
Source DARAM Destination SARAM	n+2	n+2	n+2 n+4†	n+2+2p
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ n+4† 2n+2§	n+2+2p 2n+2p‡
Source Ext Destination SARAM	$n+2nd_{src}$	$n+2nd_{src}$	$n+2nd_{src}$ $n+4+nd_{src}$†	$n+2+nd_{src}+2p$
Source DARAM Destination Ext	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}+2p$
Source SARAM Destination Ext	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}+2p$
Source Ext Destination Ext	$4n+nd_{src}+nd_{dst}$‡	$4n+nd_{src}+nd_{dst}$	$4n+nd_{src}+nd_{dst}$	$4n+2+nd_{src}+nd_{dst}+2p$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Class XIII

1-word, 2-cycle, block data transfer, data to data space

BLDD BMAR,dma; BLDD dma,BMAR

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	2	2	2	2+p

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	2	2	2	2+p
Source SARAM Destination DARAM	2	2	2	2+p
Source Ext Destination DARAM	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$2+d_{src}+p$
Source DARAM Destination SARAM	2	2	2 3†	2+p
Source SARAM Destination SARAM	2	2	2 3†	2+p
Source Ext Destination SARAM	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$ $3+d_{src}$†	$2+d_{src}+p$
Source DARAM Destination Ext	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+p$
Source SARAM Destination Ext	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+p$
Source Ext Destination Ext	$3+d_{src}+d_{dst}$	$3+d_{src}+d_{dst}$	$3+d_{src}+d_{dst}$	$5+d_{src}+d_{dst}+p$
Cycle Timings for a Repeat (RPT) Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+1	n+1	n+1	n+1+p
Source SARAM Destination DARAM	n+1	n+1	n+1	n+1+p
Source Ext Destination DARAM	$n+1+nd_{src}$	$n+1+nd_{src}$	$n+1+nd_{src}$	$n+1+nd_{src}+p$
Source DARAM Destination SARAM	n+1	n+1	n+1 n+3†	n+1+p
Source SARAM Destination SARAM	n+1 2n–1‡	n+1 2n–1‡	n+1 2n–1‡ n+3§ 2n+1§	n+1+p 2n–1+p‡
Source Ext Destination SARAM	$n+1+nd_{src}$†	$n+1+nd_{src}$	$n+1+nd_{src}$ $n+3+nd_{src}$†	$n+1+nd_{src}+p$
Source DARAM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p$

Cycle Timings for a Repeat (RPT) Instruction (Concluded)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p$
Source Ext Destination Ext	$4n-1+nd_{src}+nd_{dst}$	$4n-1+nd_{src}+nd_{dst}$	$4n-1+nd_{src}+nd_{dst}$	$4n+1+nd_{src}+nd_{dst}+p$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Class XIV

2-word, 3-cycle, block data transfer, program to data space

BLPD #lk,dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	3	3	3	$3+2p_{code}$
Source SARAM Destination DARAM	3	3	3	$3+2p_{code}$
Source Ext Destination DARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$	$3+p_{src}+2p_{code}$
Source DARAM/ROM Destination SARAM	3	3	3 4†	$3+2p_{code}$
Source SARAM Destination SARAM	3	3	3 4†	$3+2p_{code}$
Source Ext Destination SARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$ $4+p_{src}$†	$3+p_{src}+2p_{code}$
Source DARAM/ROM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p_{code}$
Destination Ext Source SARAM	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p_{code}$
Source Ext Destination Ext	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$6+p_{src}+d_{dst}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Source SARAM Destination DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Source Ext Destination DARAM	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}+2p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination SARAM	n+2	n+2	n+2 n+4†	n+2+2p_{code}
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ n+4† 2n+2§	n+2+2p_{code} 2n+2p_{code}‡
Source Ext Destination SARAM	n+2+np_{src}†	n+2+np_{src}	n+2+np_{src} n+4+np_{src}†	n+2+np_{src}+2p_{code}
Source DARAM/ROM Destination Ext	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}+2p_{code}
Source SARAM Destination Ext	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}+2p_{code}
Source Ext Destination Ext	4n+np_{src}+nd_{dst}†	4n+np_{src}+nd_{dst}	4n+np_{src}+nd_{dst}	4n+2+np_{src}+nd_{dst}+2p_{code}

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Class XV

1-word, 2-cycle, block data transfer, program to data space

BLPD BMAR,dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	2	2	2	2+p_{code}
Source SARAM Destination DARAM	2	2	2	2+p_{code}
Source Ext Destination DARAM	2+p_{src}	2+p_{src}	2+p_{src}	2+p_{src}+p_{code}
Source DARAM/ROM Destination SARAM	2	2	2 3†	2+p_{code}
Source SARAM Destination SARAM	2	2	2 3†	2+p_{code}
Source Ext Destination SARAM	2+p_{src}	2+p_{src}	2+p_{src} 3+p_{src}†	2+p_{src}+2p_{code}
Source DARAM/ROM Destination Ext	3+d_{dst}	3+d_{dst}	3+d_{dst}	5+d_{dst}+p_{code}

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination Ext	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+p_{code}$
Source Ext Destination Ext	$3+p_{src}+d_{dst}$	$3+p_{src}+d_{dst}$	$3+p_{src}+d_{dst}$	$5+p_{src}+d_{dst}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Source SARAM Destination DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Source Ext Destination DARAM	$n+1+np_{src}$	$n+1+np_{src}$	$n+1+np_{src}$	$n+1+np_{src}+p_{code}$
Source DARAM/ROM Destination SARAM	n+1	n+1	n+1 n+3†	$n+1+p_{code}$
Source SARAM Destination SARAM	n+1 2n–1‡	n+1 2n–1‡	n+1 2n–1‡ n+3† 2n+1§	$n+1+p_{code}$ $2n-1+p_{code}$‡
Source Ext Destination SARAM	$n+1+np_{src}$	$n+1+np_{src}$	$n+1+np_{src}$ $n+3+np_{src}$†	$n+1+np_{src}+p_{code}$
Source DARAM/ROM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p_{code}$
Source SARAM Destination Ext	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}$	$2n+1+nd_{dst}+p_{code}$
Source Ext Destination Ext	$4n-1+np_{src}+nd_{dst}$	$4n-1+np_{src}+nd_{dst}$	$4n-1+np_{src}+nd_{dst}$	$4n+1+np_{src}+nd_{dst}+p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Class XVI

1-word, 2-cycle, block data transfer, data to program space

BLDP dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	2	2	2	2+p
Source SARAM Destination DARAM	2	2 3¶	2	2+p
Source Ext Destination DARAM	2+d_{src}	2+d_{src}	2+d_{src}	3+d_{src}+p_{code}
Source DARAM Destination SARAM	2	2	2 3†	2+p
Source SARAM Destination SARAM	2	2	2 3† or ¶ 4§	2+p
Source Ext Destination SARAM	2+d_{src}	2+d_{src}	2+d_{src} 3+d_{src}†	3+d_{src}+p_{code}
Source DARAM Destination Ext	3+p_{dst}	3+p_{dst}	3+p_{dst}	4+p_{dst}+p_{code}
Source SARAM Destination Ext	3+p_{dst}	3+p_{dst}	3+p_{dst} 4+p_{dst}¶	4+p_{dst}+p_{code}
Source Ext Destination Ext	3+d_{src}+p_{dst}	3+d_{src}+p_{dst}	3+d_{src}+p_{dst}	5+d_{src}+p_{dst}+p_{code}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+1	n+1	n+1	n+1+p_{code}
Source SARAM Destination DARAM	n+1	n+1	n+1 n+2¶	n+1+p_{code}
Source Ext Destination DARAM	n+1+nd_{src}	n+1+nd_{src}	n+1+nd_{src}	n+2+nd_{src}+p_{code}
Source DARAM Destination SARAM	n+1	n+1	n+1 n+2†	n+1+p_{code}

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination SARAM	n+1 2n–1‡	n+1 2n–1‡	n+1 2n–1‡ n+2† or ¶ 2n+1§	n+1+p_{code} 2n–1+p_{code}2
Source Ext Destination SARAM	n+1+nd_{src}	n+1+nd_{src}	n+1+nd_{src} n+2+np_{src}†	n+2+nd_{src}+p_{code}
Source DARAM Destination Ext	2n+1+np_{dst}	2n+1+np_{dst}	2n+1+np_{dst}	2n+2+np_{dst}+p_{code}
Source SARAM Destination Ext	2n+1+np_{dst}	2n+1+np_{dst}	2n+1+np_{dst} 2n+2+np_{dst}¶	2n+2+np_{dst}+p_{code}
Source Ext Destination Ext	4n–1+nd_{src}+np_{dst}	4n–1+nd_{sr}+np_{dst}	4n–1+nd_{src}+np_{dst}	4n+1+nd_{src}+np_{dst}+ p_{code}

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.
¶ If the source operand and the code are in the same SARAM block.

Class XVII

1-word, 3-cycle, table read

TBLR

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	3	3	3	3+p_{code}
Source SARAM Destination DARAM	3	3	3	3+p_{code}
Source Ext Destination DARAM	3+p_{src}	3+p_{src}	3+p_{src}	3+p_{src}+p_{code}
Source DARAM/ROM Destination SARAM	3	3	3 4†	3+p_{code}
Source SARAM Destination SARAM	3	3	3 4†	3+p_{code}
Source Ext Destination SARAM	3+p_{src}	3+p_{src}	3+p_{src} 4+p_{src}†	3+p_{src}+p_{code}
Source DARAM/ROM Destination Ext	4+d_{dst}	4+d_{dst}	4+d_{dst}	6+d_{dst}+p_{code}

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination Ext	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+p_{code}$
Source Ext Destination Ext	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$6+p_{src}+d_{dst}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM/ROM Destination DARAM	n+2	n+2	n+2	$n+2+p_{code}$
Source SARAM Destination DARAM	n+2	n+2	n+2	$n+2+p_{code}$
Source Ext Destination DARAM	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}+p_{code}$
Source DARAM/ROM Destination SARAM	n+2	n+2	n+2 n+4†	$n+2+p_{code}$
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ 2n+2§	$n+2+p_{code}$ 2n‡
Source Ext Destination SARAM	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}$ $n+4+np_{src}$†	$n+2+np_{src}+p_{code}$
Source DARAM/ROM Destination Ext	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+4+nd_{dst}+p_{code}$
Source SARAM Destination Ext	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+2+nd_{dst}$	$2n+4+nd_{dst}+p_{code}$
Source Ext Destination Ext	$4n+np_{src}+nd_{dst}$	$4n+np_{src}+nd_{dst}$	$4n+np_{src}+nd_{dst}$	$4n+2+np_{src}+nd_{dst}+p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Class XVIII

1-word, 3-cycle, table write

TBLW

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	3	3	3	$3+p_{code}$
Source SARAM Destination DARAM	3	3	3	$3+p_{code}$
Source Ext Destination DARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$	$3+d_{src}+p_{code}$
Source DARAM Destination SARAM	3	3	3 4†	$3+p_{code}$
Source SARAM Destination SARAM	3	3	3 4†	$3+p_{code}$
Source Ext Destination SARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$ $4+d_{src}$†	$3+d_{src}+p_{code}$
Source DARAM Destination Ext	$4+p_{dst}$	$4+p_{dst}$	$4+p_{dst}$	$5+p_{dst}+p_{code}$
Source SARAM Destination Ext	$4+p_{dst}$	$4+p_{dst}$	$4+p_{dst}$	$5+p_{dst}+p_{code}$
Source Ext Destination Ext	$4+d_{src}+p_{dst}$	$4+d_{src}+p_{dst}$	$4+d_{src}+p_{dst}$	$5+d_{src}+p_{dst}+p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination DARAM	n+2	n+2	n+2	$n+2+p_{code}$
Source SARAM Destination DARAM	n+2	n+2	n+2	$n+2+p_{code}$
Source Ext Destination DARAM	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}+p_{code}$
Source DARAM Destination SARAM	n+2	n+2	n+2 n+3†	$n+2+p_{code}$
Source SARAM Destination SARAM	n+2 2n‡	n+2 2n‡	n+2 2n‡ 2n+1§	$n+2+p_{code}$ 2n‡
Source Ext Destination SARAM	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}$ $n+3+nd_{src}$†	$n+2+nd_{src}+p_{code}$
Source DARAM Destination Ext	$2n+2+np_{dst}$	$2n+2+np_{dst}$	$2n+2+np_{dst}$	$2n+3+np_{dst}+p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source SARAM Destination Ext	$2n+2+np_{dst}$	$2n+2+np_{dst}$	$2n+2+np_{dst}$	$2n+3+np_{dst}+p_{code}$
Source Ext Destination Ext	$4n+nd_{src}+np_{dst}$	$4n+nd_{src}+np_{dst}$	$4n+nd_{src}+np_{dst}$	$4n+1+nd_{src}+np_{dst}+p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ If both the source and the destination operands are in the same SARAM block.
§ If both operands and the code are in the same SARAM block.

Class XIX

2-word, 3-cycle, multiply accumulate

MAC #lk,dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	3	3	3	$3+2p_{code}$
Operand1 SARAM Operand2 DARAM	3	3	3	$3+2p_{code}$
Operand1 Ext Operand2 DARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	3	3	3	$3+2p_{code}$
Operand1 SARAM Operand2 SARAM	3 4†	3 4†	3 4†	$3+2p_{code}$ $4+2p_{code}$†
Operand1 Ext Operand2 SARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand1 DARAM/ROM Operand2 Ext	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+dop2+2p_{code}$
Operand1 SARAM Operand2 Ext	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand1 Ext Operand2 Ext	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand1 SARAM Operand2 DARAM	n+2	n+2	n+2	$n+2+2p_{code}$

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 Ext Operand2 DARAM	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 SARAM	n+2	n+2	n+2	n+2+2p_{code}
Operand1 SARAM Operand2 SARAM	n+2 2n+2†	n+2 2n+2†	n+2 2n+2†	n+2+2p_{code} 2n+2†
Operand1 Ext Operand2 SARAM	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 Ext	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}
Operand1 SARAM Operand2 Ext	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}
Operand1 Ext Operand2 Ext	2n+2+np_{op1}+nd_{op2}	2n+2+np_{op1}+nd_{op2}	2n+2+np_{op1}+nd_{op2}	2n+2+np_{op1}+nd_{op2}+2p_{code}

† If both operands are in the same SARAM block.

Class XX

1-word, 2-cycle, multiply-accumulate

MADS dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	2	2	2	2+p_{code}
Operand1 SARAM Operand2 DARAM	2	2	2	2+p_{code}
Operand1 Ext Operand2 DARAM	2+p_{op1}	2+p_{op1}	2+p_{op1}	
Operand1 DARAM/ROM Operand2 SARAM	2	2	2	2+p_{code}
Operand 1 SARAM Operand2 SARAM	2 3†	2 3†	2 3†	2+p_{code} 3+p_{code}†
Operand1 Ext Operand2 SARAM	2+p_{op1}	2+p_{op1}	2+p_{op1}	2+p$op1$+p_{code}
Operand1 DARAM/ROM Operand2 Ext	2+d_{op2}	2+d_{op2}	2+d_{op2}	2+d_{op2}+p_{code}

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 Ext	2+d_{op2}	2+d_{op2}	2+d_{op2}	2+d_{op2}+p_{code}
Operand1 Ext Operand2 Ext	3+p_{op1}+d_{op2}	3+p_{op1}+d_{op2}	3+p_{op1}+d_{op2}	3+p_{op1}+d_{op2}+p_{code}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+1	n+1	n+1	n+1+p_{code}
Operand1 SARAM Operand2 DARAM	n+1	n+1	n+1	n+1+p_{code}
Operand1 Ext Operand2 DARAM	n+1+np_{op1}	n+1+np_{op1}	n+1+np_{op1}	n+1+np_{op1}+p_{code}
Operand1 DARAM/ROM Operand2 SARAM	n+1	n+1	n+1	n+1+p_{code}
Operand1 SARAM Operand2 SARAM	n+1 2n+1†	n+1 2n+1†	n+1 2n+1†	n+1+p_{code} 2n+1†
Operand1 Ext Operand2 SARAM	n+1+np_{op1}	n+1+np_{op1}	n+1+np_{op1}	n+1+np_{op1}+p_{code}
Operand1 DARAM/ROM Operand2 Ext	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}+p_{code}
Operand1 SARAM Operand2 Ext	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}	n+1+nd_{op2}+p_{code}
Operand1 Ext Operand2 Ext	2n+1+np_{op1}+ nd_{op2}	2n+1+np_{op1}+ nd_{op2}	2n+1+np_{op1}+ nd_{op2}	2n+1+np_{op1}+nd_{op2}+ p_{code}

† If both operands are in the same SARAM block.

Class XXI

2-word, 3-cycle, multiply accumulate with data move

MACD #lk,dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 DARAM	3	3	3	3+2p_{code}
Operand1 DARAM/ROM Operand2 DARAM	3	3	3	3+2p_{code}

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 Ext Operand2 DARAM	3+p_{op1}	3+p_{op1}	3+p_{op1}	3+p_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 SARAM	3	3	3	3+2p_{code}
Operand1 SARAM Operand2 SARAM	3	3	3 4‡ 5§	3+2p_{code} 4+2p_{code}‡
Operand1 Ext Operand2 SARAM	3+p_{op1}	3+p_{op1}	3+p_{op1}	3+p_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 Ext§	3+d_{op2}	3+d_{op2}	3+d_{op2}	3+d_{op2}+2p_{code}
Operand1 SARAM Operand2 Ext§	3+d_{op2}	3+d_{op2}	3+d_{op2}	3+d_{op2}+2p_{code}
Operand1 Ext Operand2 Ext¶	4+p_{op1}+d_{op2}	4+p_{op1}+d_{op2}	4+p_{op1}+d_{op2}	4+p_{op1}+d_{op2}+2p_{code}
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+2	n+2	n+2	n+2+2p_{code}
Operand1 SARAM Operand2 DARAM	n+2	n+2	n+2	n+2+2p_{code}
Operand1 Ext Operand2 DARAM	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 SARAM	2n	2n	2n 2n+2†	2n+2p_{code}
Operand1 SARAM Operand2 SARAM	2n 3n‡	2n 3n‡	2n 2n+2† 3n2 3n+2§	2n+2p_{code} 3n‡
Operand1 Ext Operand2 SARAM	2n+np_{op1}	2n+np_{op1}	2n+np_{op1} 2n+2+np_{op1}†	2n+np_{op1}+2p_{code}
Operand1 DARAM/ROM Operand2 Ext¶	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}

Cycle Timings for a Repeat (RPT) Execution (Continued)				
	PR	**PDA**	**PSA**	**PE**
Operand1 SARAM Operand2 Ext¶	$n+2+nd_{op2}$	$n+2+nd_{op2}$	$n+2+nd_{op2}$	$n+2+nd_{op2}+2p_{code}$
Operand1 Ext Operand2 Ext¶	$2n+2+np_{op1}+nd_{op2}$	$2n+2+np_{op1}+nd_{op2}$	$2n+2+np_{op1}+nd_{op2}$	$2n+2+np_{op1}+nd_{op2}+2p_{code}$

† If operand2 and code are in the same SARAM block.
‡ If both operands are in the same SARAM block.
§ If both operands and code are in the same SARAM block.
¶ Data move operation is not performed when operand 2 is in external data memory.

Class XXII

1-word, 2-cycle, multiply accumulate with data move

MADD dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	2	2	2	$2+p_{code}$
Operand1 SARAM Operand2 DARAM	2	2	2	$2+p_{code}$
Operand1 Ext Operand2 DARAM	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	2	2	2	$2+p_{code}$
Operand1 SARAM Operand2 SARAM	2	2	2 3‡ 4§	$2+p_{code}$ $3+p_{code}$‡
Operand1 Ext Operand2 SARAM	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}$	$2+p_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 Ext¶	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}+p_{code}$
Operand1 SARAM Operand2 Ext¶	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}$	$2+d_{op2}+p_{code}$
Operand1 Ext Operand2 Ext¶	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}$	$3+p_{op1}+d_{op2}+p_{code}$

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Operand1 DARAM/ROM Operand2 DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Operand1 SARAM Operand2 DARAM	n+1	n+1	n+1	$n+1+p_{code}$
Operand1 Ext Operand2 DARAM	$n+1+np_{op1}$	$n+1+np_{op1}$	$n+1+np_{op1}$	$n+1+np_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 SARAM	2n–1	2n–1	2n–1 2n+1†	$2n-1+p_{code}$
Operand1 SARAM Operand2 SARAM	2n–1 3n–1‡	2n–1 3n–1‡	2n–1 2n+1† 3n–1‡ 3n+1§	$2n-1+p_{code}$ 3n–1‡
Operand1 Ext Operand2 SARAM	$2n-1+np_{op1}$	$2n-1+np_{op1}$	$2n-1+np_{op1}$ $2n+1+np_{op1}$†	$2n-1+np_{op1}+p_{code}$
Operand1 DARAM/ROM Operand2 Ext¶	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}+p_{code}$
Operand1 SARAM Operand2 Ext¶	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}$	$n+1+nd_{op2}+p_{code}$
Operand1 Ext Operand2 Ext¶	$2n+1+np_{op1}+nd_{op2}$	$2n+1+np_{op1}+nd_{op2}$	$2n+1+np_{op1}+nd_{op2}$	$2n+1+np_{op1}+nd_{op2}+p_{code}$

† If operand 2 and code reside in same SARAM block.
‡ If both operands reside in same SARAM block.
§ If both operands and code reside in same SARAM block.
¶ Data move operation is not performed when operand2 is in external data memory.

Class XXIII

2-word, 2-cycle, memory map register load

LMMR dma,#lk

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination MMR‡	2	2	2	$2+2p_{code}$
Source SARAM Destination MMR‡	2	2	2 3†	$2+2p_{code}$
Source Ext Destination MMR‡	$2+p_{src}$	$2+p_{src}$	$2+p_{src}$	$3+p_{src}+2p_{code}$

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination MMPORT	$3+io_{dst}$	$3+io_{dst}$	$3+io_{dst}$	$5+2p_{code}+io_{dst}$
Source SARAM Destination MMPORT	$3+io_{dst}$	$3+io_{dst}$	$3+io_{dst}$ 4†	$5+2p_{code}+io_{dst}$
Source Ext Destination MMPORT	$3+p_{src}+io_{dst}$	$3+p_{src}+io_{dst}$	$3+p_{src}+io_{dst}$	$6+p_{src}+2p_{code}+io_{dst}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM Destination MMR§	2n	2n	2n	$2n+2p_{code}$
Source SARAM Destination MMR§	2n	2n	2n 2n+1†	$2n+2p_{code}$
Source Ext Destination MMR§	$2n+nd_{src}$	$2n+nd_{src}$	$2n+nd_{src}$	$2n+1+nd_{src}+2p_{code}$
Source DARAM Destination MMPORT	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+3+nio_{dst}+2p_{code}$
Source SARAM Destination MMPORT	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$ $3n+1+nio_{dst}$†	$3n+3+nio_{dst}+2p_{code}$
Source Ext Destination MMPORT	$4n-1+nd_{src}+nio_{dst}$	$4n-1+nd_{src}+nio_{dst}$	$4n-1+nd_{src}+nio_{dst}$	$4n+2+nd_{src}+nio_{dst}+2p_{code}$

† If the source operand and the code are in the same SARAM block.
‡ Add one more cycle for peripheral memory-mapped register access.
§ Add *n* more cycles for peripheral memory-mapped register access.

Class XXIV

2-word, 2-cycle, memory map register store

SMMR dma,#lk

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Destination DARAM Source MMR‡	2	2	2	$2+2p_{code}$
Destination SARAM Source MMR‡	2	2	2 3†	$2+2p_{code}$
Destination Ext Source MMR‡	$3+d_{dst}$	$3+d_{dst}$	$3+d_{dst}$	$5+d_{dst}+2p_{code}$
Destination DARAM Source MMPORT	$3+io_{src}$	$3+io_{src}$	$3+io_{src}$	$4+io_{src}+2p_{code}$

Cycle Timings for a Single Instruction (Continued)				
	PR	**PDA**	**PSA**	**PE**
Destination SARAM Source MMPORT	$3+io_{src}$	$3+io_{src}$	$3+io_{src}$ $4+io_{src}$†	$3+io_{src}+2p_{code}$
Destination Ext Source MMPORT	$4+io_{src}+d_{dst}$	$4+io_{src}+d_{dst}$	$4+io_{src}+d_{dst}$	$6+io_{src}+d_{dst}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Destination DARAM Source MMR§	2n	2n	2n	$2n+2p_{code}$
Destination SARAM Source MMR§	2n	2n	2n 2n+2†	$2n+2p_{code}$
Destination Ext Source MMR§	$3n+nd_{dst}$	$3n+nd_{dst}$	$3n+nd_{dst}$	$3n+3+nd_{dst}+2p_{code}$
Destination DARAM Source MMPORT	$2n+nio_{src}$	$2n+nio_{src}$	$2n+nio_{src}$	$2n+1+nio_{src}+2p_{code}$
Destination SARAM Source MMPORT	$2n+nio_{src}$	$2n+nio_{src}$	$2n+nio_{src}$ $2n+2+nio_{src}$†	$2n+1+nio_{src}+2p_{code}$
Destination Ext Source MMPORT	$5n-2+nd_{dst}+nio_{src}$	$5n-2+nd_{dst}+nio_{src}$	$5n-2+nd_{dst}+nio_{src}$	$5n+1+nd_{dst}+nio_{src}+2p_{code}$

† If the destination operand and the code are in the same SARAM block.
‡ Add one more cycle for peripheral memory-mapped register.
§ Add *n* more cycles for peripheral memory-mapped register access.

Class XXV

2-word, 3-cycle, output port

OUT dma,port

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	$3+io_{dst}$	$3+io_{dst}$	$3+io_{dst}$	$5+io_{dst}+2p_{code}$
Source SARAM	$3+io_{dst}$	$3+io_{dst}$	$3+io_{dst}$ $4+io_{dst}$†	$5+io_{dst}+2p_{code}$
Source Ext	$3+d_{src}+io_{dst}$	$3+d_{src}+io_{dst}$	$3+d_{src}+io_{dst}$	$6+d_{src}+io_{dst}+2p_{code}$

Cycle Timings for a Repeat (RPT) Execution				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+3+nio_{dst}+2p_{code}$
Source SARAM	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$ $3n+1+nio_{dst}$†	$3n+3+nio_{dst}+2p_{code}$
Source Ext	$5n-2+nd_{src}+nio_{dst}$	$5n-2+nd_{src}+nio_{dst}$	$5n-2+nd_{src}+nio_{dst}$	$5n+1+nd_{src}+nio_{dst}+2p_{code}$

† If the source operand and the code are in the same SARAM block.

Class XXVI

2-word, 2-cycle, input port

IN dma,port

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Destination DARAM	$2+io_{src}$	$2+io_{src}$	$2+io_{src}$	$3+io_{src}+2p_{code}$
Destination SARAM	$2+io_{src}$	$2+io_{src}$	$2+io_{src}$ $3+io_{src}$†	$3+io_{src}+2p_{code}$
Destination Ext	$3+d_{dst}+io_{src}$	$3+d_{dst}+io_{src}$	$3+d_{dst}+io_{src}$	$6+d_{dst}+io_{src}+2p_{code}$
Cycle Timings for a Repeat (RPT) Execution				
Destination DARAM	$2n+nio_{src}$	$2n+nio_{src}$	$2n+nio_{src}$	$2n+1+nio_{src}+2p_{code}$
Destination SARAM	$2n+nio_{src}$	$2n+nio_{src}$	$2n+nio_{src}$ $2n+2+nio_{src}$†	$2n+1+nio_{src}+2p_{code}$
Destination Ext	$4n-1+nd_{dst}+nio_{src}$	$4n-1+nd_{dst}+nio_{src}$	$4n-1+nd_{dst}+nio_{src}$	$4n+2+nd_{dst}+nio_{src}+2p_{code}$

† If the destination operand and the code are in the same SARAM block.

Class XXVII

1-word, 2-cycle, pipeline-protected, memory read

LDP dma; LST #0,dma; LST #1,dma, LAR ARn,dma

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	2	2	2	$2+p_{code}$
Source SARAM	2	2	2 3†	$2+p_{code}$
Source Ext	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$

Cycle Timings for a Repeat (RPT) Execution				
Source DARAM	2n	2n	2n	2n+p_{code}
Source SARAM	2n	2n	2n 2n+1†	2n+p_{code}
Source Ext	2n+nd_{src}	2n+nd_{src}	2n+nd_{src}	2n+1+nd_{src}+p_{code}

† If the source operand and the code are in the same SARAM block.

Class XXVIII

1-word, 2-cycle, pipeline-protected, nonrepeatable

LDP #k; LAR ARN,#k

Cycle Timings for a Single Instruction				
	PR	**PDA**	**PSA**	**PE**
Source DARAM	2	2	2	2+p_{code}
Source SARAM	2	2	2 3†	2+p_{code}
Source Ext	2+d_{src}	2+d_{src}	2+d_{src}	3+d_{src}+p_{code}
Cycle Timings for a Repeat (RPT) Execution				
Not Repeatable				

† If the source operand and the code are in the same SARAM block.

Appendix D

System Migration

This appendix contains information that is necessary to upgrade a 'C25 system into a 'C5x system. The information consists of a detailed list of the programming differences and hardware and timing differences between the two generations of TMS320 DSPs. Note that the 'C50, C51, and 'C53 have the same features with the exception of memory map; so within this appendix, any reference to 'C5x applies to 'C50, 'C51, and 'C53, unless otherwise stated. This appendix contains the following:

Topic **Page**

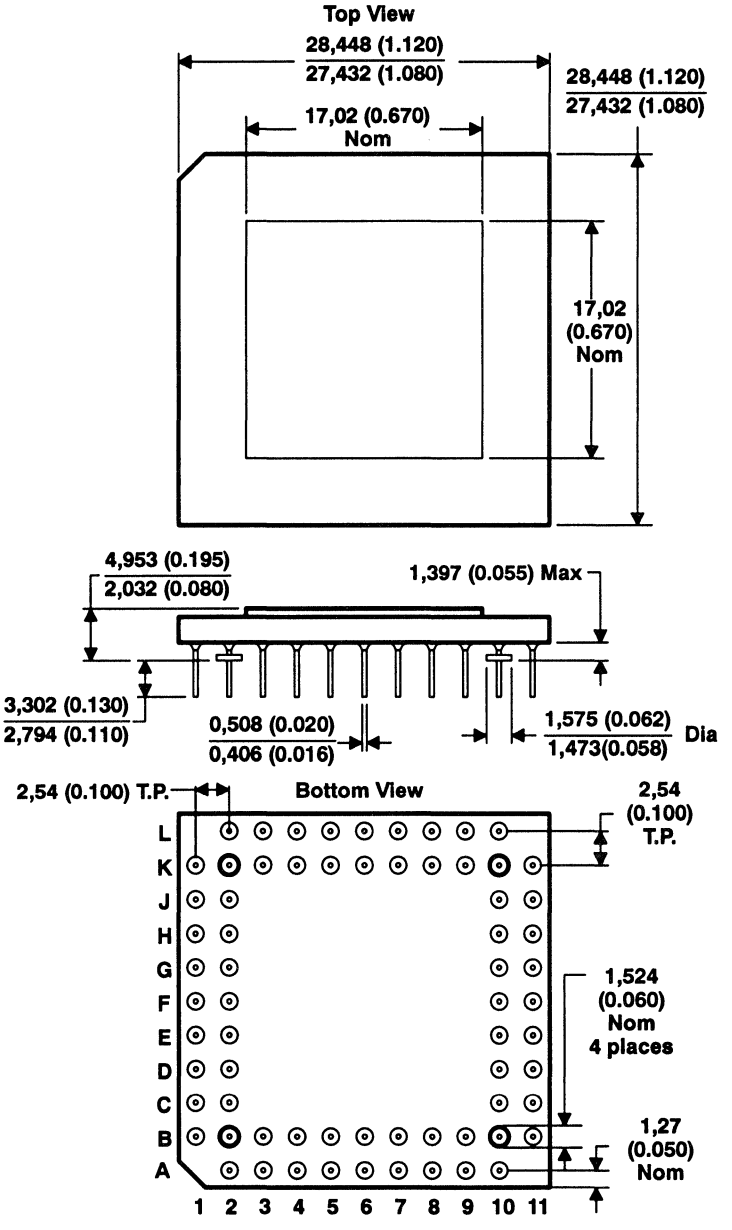
D.1 Package and Pin Layout

The 'C25 is available in both a 68-pin CPGA and a 68-pin PLCC as shown in Figure D–1 and Figure D–2, respectively. The 'C5x devices are packaged in a 132-pin Quad Flat Pack package (QFP), as shown in Appendix A.

Figure D–1.'C25 68-Pin Ceramic Pin Grid Array

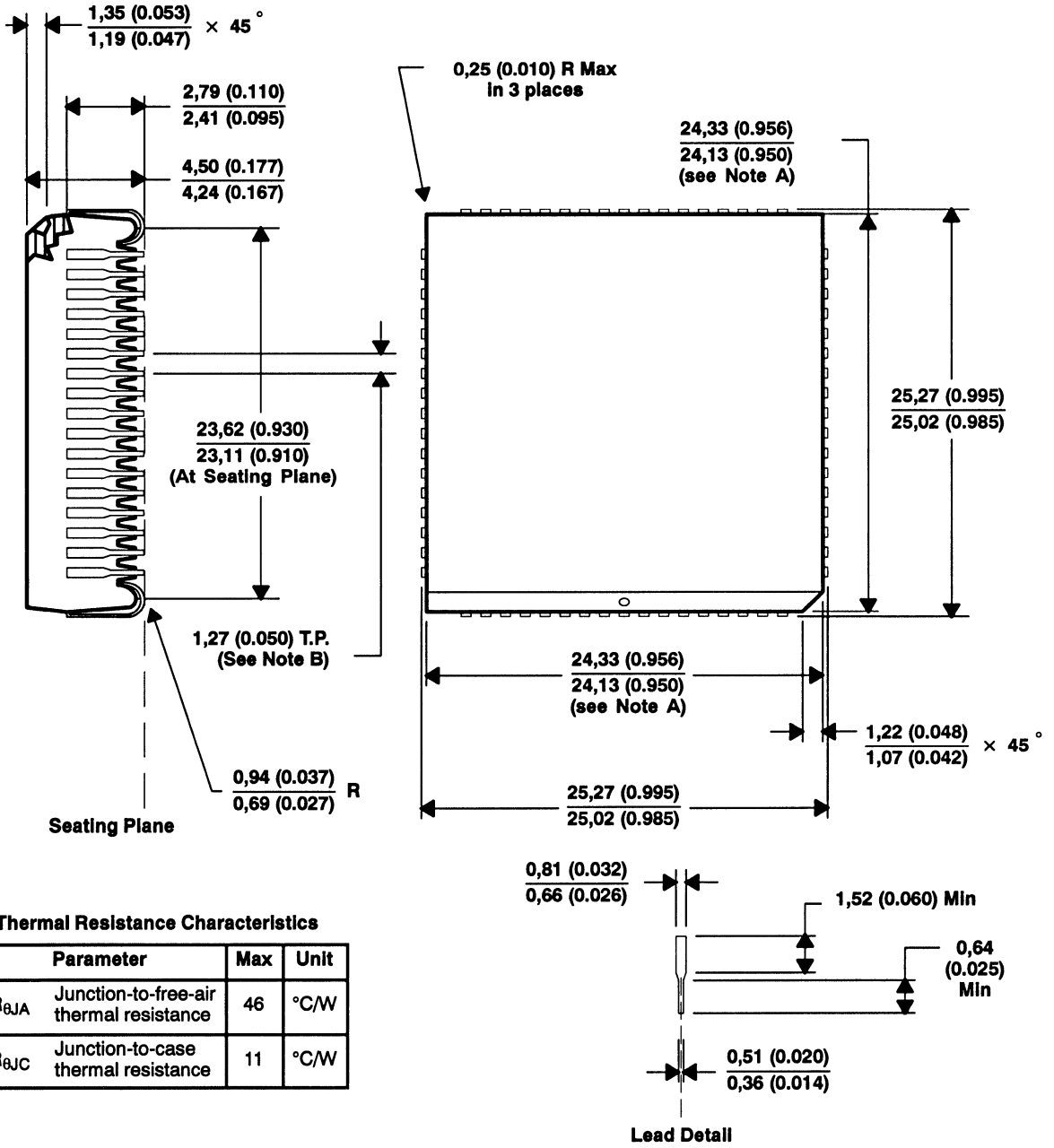
Thermal Resistance Characteristics

Parameter		Max	Unit
$R_{\theta JA}$	Junction-to-free-air thermal resistance	36	°C/W
$R_{\theta JC}$	Junction-to-case thermal resistance	6	°C/W



ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES.

Figure D–2. 'C25 68-Pin Plastic Leaded Chip Carrier



Thermal Resistance Characteristics

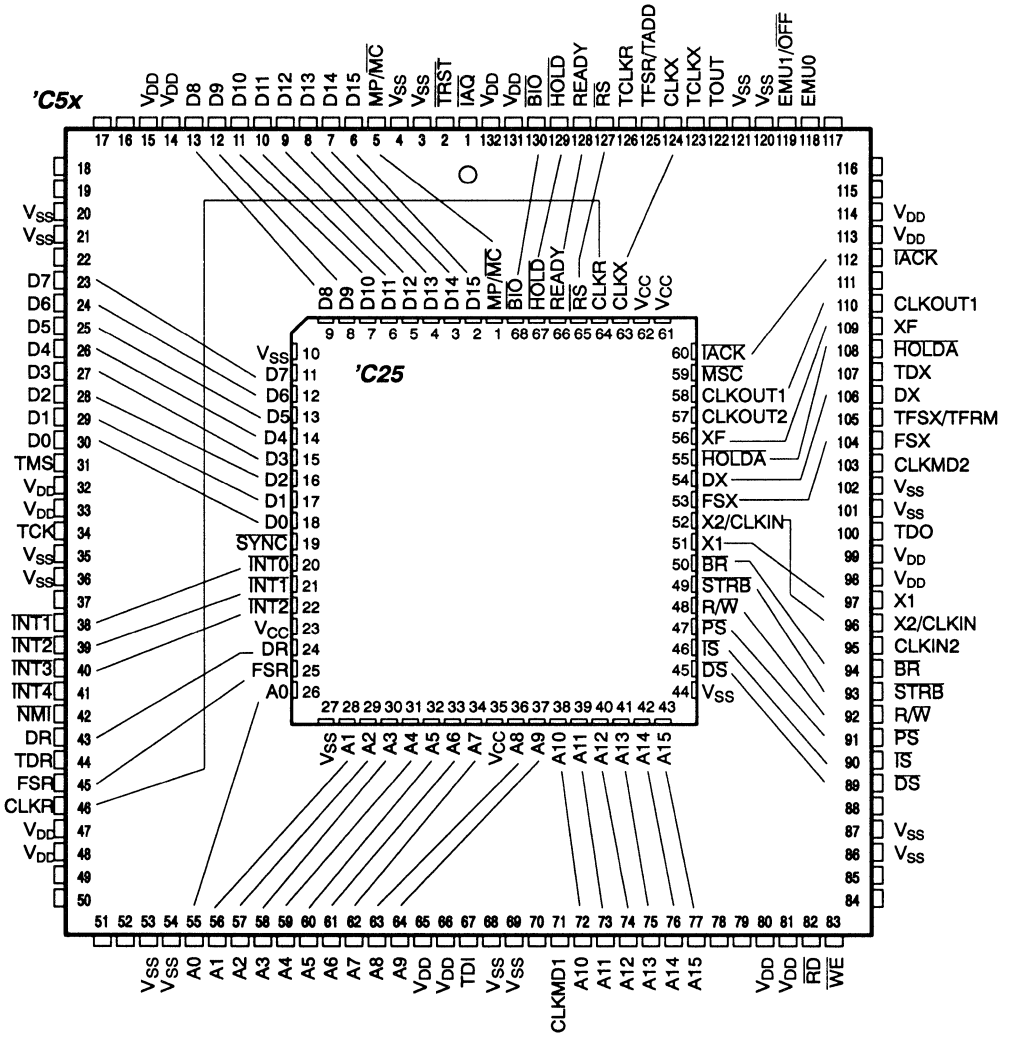
Parameter		Max	Unit
$R_{\theta JA}$	Junction-to-free-air thermal resistance	46	°C/W
$R_{\theta JC}$	Junction-to-case thermal resistance	11	°C/W

Notes: A. Centerline of center pin, each side, is within 0,10 (0.004) of package centerline as determined by this dimension.
B. Location of each pin is within 0,127 (0.005) of true position with respect to center pin on each side.

ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES.

When a 'C25 is upgraded to a 'C5x, there is minimal layout modification. The 'C5x signals are on the same side (except the CLKR and A0 pins) and in the same order (except the X1 and X2/CLKIN pins) as those of the 'C25. Figure D–3 shows the pin-to-pin relationship between the 'C25 and the 'C5x devices in J-leaded chip carrier packages. Note that the two devices are not drawn to scale. The power (V_{DD}) and ground (V_{SS}) signals are symmetrically positioned on the 'C5x so that, in conjunction with the $\overline{OFF}$ signal, the device is not damaged by inserting it in the wrong orientation. The 'C5x has more power and ground pins to provide higher performance and more noise immunity than the 'C25.

Figure D–3. 'C25-to-'C5x Pin/Signal Relationship



Note: Pins without callouts are unassigned (reserved).

Three 'C25 signals (CLKOUT2, $\overline{\text{MSC}}$ and $\overline{\text{SYNC}}$) are not present on the 'C5x. Because the 'C5x operates with a divide-by-two clock, it can be synchronized with reset. Therefore, there is no need for the $\overline{\text{SYNC}}$ signal. With only two phases, there are no external timings that tie to the CLKOUT2 of the 'C25.

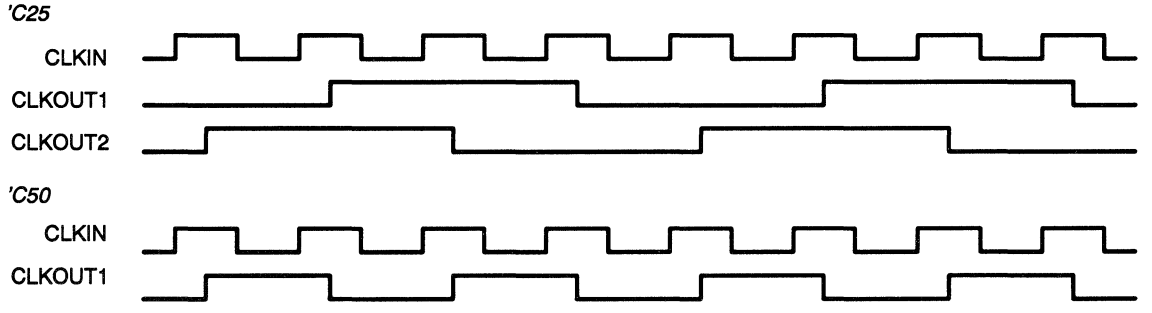
Some of the 'C25-equivalent pins have additional capabilities on the 'C5x. The 'C5x supports external direct memory access of the on-chip single-access RAM block. For this reason, the following signals are now bidirectional:

A0–A15 = address lines
$\overline{\text{STRB}}$ = memory access strobe
R/$\overline{\text{W}}$ = read/write
$\overline{\text{BR}}$ = bus request

The 'C5x serial port transmit clock (CLKX) can now be configured as an output that operates at one-fourth the machine clock rate. CLKX is configured as an input by reset. The 'C25 CLKX pin is always an input.

The 'C25 operates with a four-phase clock. This device's machine rate is one-fourth the CLKIN rate. CLKOUT1 and CLKOUT2 operate at the machine rate and are 90° out of phase. The 'C5x operates with a two-phase clock. The device's machine rate is one-half the CLKIN rate. In addition, the 'C5x offers a divide-by-one clock input feature so that the device's machine rate equals the CLKIN rate. CLKOUT1 operates at the machine rate. Figure D–4 shows both the 'C25 and the 'C5x clocking schemes.

Figure D–4. 'C25 and 'C5x Clocking Schemes



The 'C5x MP/$\overline{MC}$ (microprocessor/microcomputer) pin is sampled only while $\overline{RS}$ is low. Changes on this pin are ignored while $\overline{RS}$ is high. The mode can be changed during execution by changing the MP/$\overline{MC}$ bit in the PMST register. On the 'C25, any change on the MP/$\overline{MC}$ pin affects the operation of the device, regardless of the state of $\overline{RS}$.

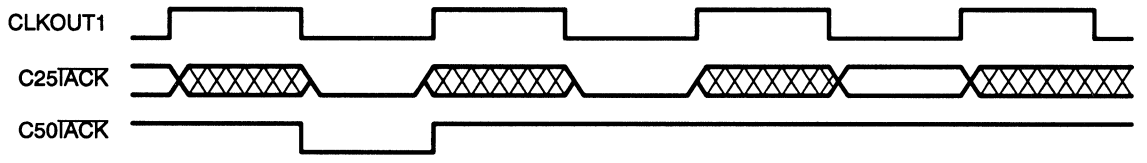
The 'C5x $\overline{IACK}$ signal goes low only on the first machine cycle of the fetch of the first word of the interrupt vector. The 'C25 $\overline{IACK}$ goes low on each wait-state cycle, as well as on the first machine cycle, but it is valid only during CLKOUT1 low (during CLKOUT1 high, it has a specific meaning for emulator/test operations). Figure D–5 illustrates this difference.

The 'C5x device includes some additional functions not included with the 'C25. These functions and associated pins are as follows:

- TDM serial port = TCLKR, TCLKX, TDR, TDX, TADD, TFRM
- Emulation interface = EMU0, EMU1/$\overline{OFF}$, $\overline{IAQ}$, TCK, TDI, TDO, TMS, TRST
- Timer borrow = TOUT
- Divide-by-one clock = CLKIN2, CLKMD1, and CLKMD2
- Fourth external interrupt = $\overline{INT4}$
- Nonmaskable interrupt = $\overline{NMI}$
- Read enable = $\overline{RD}$
- Write enable = $\overline{WE}$

The 'C5x package also includes 12 additional power and 13 additional ground pins. These additional power and ground pins enable the device to operate at much faster speeds. Twenty pins are reserved for future 'C5x spinoff devices.

Figure D–5. 'C25 $\overline{IACK}$ Versus 'C5x $\overline{IACK}$



D.2 Timing

The 'C25 and the 'C5x operate with some timing differences. These timing differences include aspects of the on-chip operation as well as aspects of the external memory interfacing. One key difference is that the 'C5x is capable of operating at two to three times the speed of a 'C25. Another key difference is that the 'C25 operates with a three-deep pipeline, while the 'C5x operates with a four-deep pipeline. Key differences in the external memory interface encompass the faster 'C5x and include certain external interface enhancements. The final key difference is that some compatible operations execute in a different number of machine cycles. This section describes these differences.

D.2.1 Device Clock Speed

The 'C25 operates its machine cycles with a divide-by-four clocking scheme. The 'C5x uses a divide-by-two clocking scheme. This means that a 'C25, operating with a 40-MHz CLKIN, executes its machine cycles within 100 ns, while the 'C5x, which is operating with the same CLKIN, executes its machine cycles in 50 ns. This clocking arrangement changes the way that the signals of the devices are specified. Many of the 'C25 timing values, given in the *TMS320 Second-Generation Digital Signal Processor Data Sheet*, are specified as quarter-phase **(Q)** ± **N** ns. The timing values of the 'C5x are defined in half-phases **(H)**.

D.2.2 Pipeline

The 'C25 operates with a three-deep pipeline, while the 'C5x operates with a four-deep pipeline. This means that anytime there is a program counter (PC) discontinuity (for example, branch, call, return, interrupt, etc.), it takes four cycles to complete with the 'C5x, whereas it takes three cycles on the 'C25. The 'C5x, however, also has delayed instructions that take only two cycles to complete.

D.2.3 External Memory Interfacing

The 'C5x is designed to execute external memory operations with the same signals as the 'C25. As mentioned above, the 'C5x operates at twice the instruction rate of the 'C25 when both operate with the same input clock. The 'C5x uses its software wait-state generators to compensate for this interface difference. The 'C5x device, operating with one software wait state, has similar memory timing to the 'C25 operating with no wait states. However, external writes require two cycles on the 'C5x devices. The exact timing of the signals differ because of the more advanced process used with the 'C5x.

The 'C5x has two additional memory interface signals to reduce the amount of external interfacing circuitries. The $\overline{RD}$ signal can be used to interface direct-

ly to the output enable pin of another device, while the $\overline{WE}$ signal can be directly connected to the write enable pin of another device. This alleviates the need of gating $\overline{STRB}$ and R/$\overline{W}$ to generate the equivalent signals.

D.2.4 Execution Cycle Times

Some of the 'C25 instructions require additional cycles or program words to execute on the 'C5x. The function of these instructions is the same, but the format and pipeline execution are enhanced to operate with the 'C5x architecture.

The IN and OUT instructions are now two-word instructions.They execute on the 'C5x in the same number of cycles as with the 'C25, but the assembler generates a two-word instruction for the 'C5x. Note that the 'C5x IN and OUT instructions behave differently in RPT mode. See Chapter 4 for details. Two words are used because the 'C5x can address 65,536 I/O ports; the 'C25 addresses 16. The 'C5x can address sixteen of its I/O ports in data memory space. This allows any instruction with data-memory-addressing capability to also read or write directly to an I/O port instead of having to pass it through a temporary on-chip data memory location. For example, a value can be read directly from an external analog-to-digital converter into the ALU via an I/O port.

The modification of the three mode bits of the serial port are executed in two-cycle/two-word instructions with the 'C5x. However, any or all of three bits can be modified with one instruction without affecting other bits in the register. This is done with the PLU instructions.

The NORM instruction modifies the auxiliary register (AR) on the execute (fourth) phase of the pipeline, while the ARAU operations occur on the decode (second) phase. The two instructions following a NORM instruction should not use the same auxiliary register for an address. If the two instructions following NORM change the auxiliary register pointer (ARP), then the NORM update of the AR is executed on the new ARP, not the old one. See Chapter 4 for NORM instruction description. The assembler supports an optional way to test for this condition and automatically compensate by adding NOP instructions to the code. This modification is made to the listing and object files and does not affect your source code.

Unlike the 'C25, the auxiliary registers are also accessible in the data address space on the 'C5x. This allows these registers to be loaded with the CALU instructions for advanced-addressing modes. However, take care when using this feature because the CALU operations write to the auxiliary registers on the execute phase of the pipeline and, therefore, are subject to the same characteristics of the NORM instruction. The assembler supports the option to flag these conflicts for resolution.

D.3 Instruction Set

The 'C5x instruction set is a superset of the 'C25 instruction set. The instruction set of the 'C25 is upward source-code compatible. This means that all of the instruction features of the 'C25, implemented and code written for the 'C25, can be reassembled to run on the 'C5x.

The serial port mode control bits have been moved from the status registers to the serial port control register. Because they are no longer part of the CPU registers, they no longer have direct instructions to set or clear them. The bits of the SPC can be manipulated easily with the PLU instructions. The following table shows the instructions used to replace the serial port instructions (note that the data page pointer must be set to zero to execute these new instructions):

'C25	'C5x	
RFSM	APL	#0FFF7h,SPC
SFSM	OPL	#8,SPC
RTXM	APL	#0FFDFh,SPC
STXM	OPL	#020h,SPC
FORT0	APL	#0FFFBh,SPC
FORT1	OPL	#4,SPC

Note that any or all three bits can be set in one execution of the OPL instruction, while any or all three bits can be cleared using the APL. The bits can be toggled with the XPL instruction. The I/O ports of the device are addressable in data memory space on the 'C5x devices. This means any instruction that can address data memory can also address the I/O ports.

There are a number of new instructions on the 'C5x devices. These instructions provide a more orthogonal addressing scheme and exercise the new CPU enhancements. To simplify the description of the instruction set, a number of different instructions are combined into single new instructions with additional operand formats, as in this example:

'C25		'C5x	
ADD	*+	ADD	*+
ADDK	0FFh	ADD	#0FFh
ADLK	0FFFFh	ADD	#0FFFFh
ADDH	*+	ADD	*+,16

Refer to Chapter 4 for the detailed discussion of the instruction set.

The IDLE instruction, when executed, stops the CPU from fetching and executing instructions until an unmasked interrupt occurs. The 'C25 automatically enables the interrupts globally with the execution of the IDLE instruction; this saves the extra instruction word/cycle required to execute the EINT (enable interrupts globally) instruction. Upon receipt of the interrupt, the 'C25 executes the interrupt vector and resumes operations. The 'C5x does not automatically enable the interrupts globally with its IDLE instruction. If the interrupts are not

globally enabled, then the CPU resumes execution with the instructions following the IDLE instruction, without taking the interrupt trap. If the interrupts are globally enabled, the 'C5x operates like the 'C25. In addition, a second low-power mode is available with IDLE2 instruction. This mode operates the same as IDLE except that the CPU will resume only after an external interrupt. See Chapter 4 for IDLE/IDLE2 instruction details.

The 'C5x repeat counter is 16 bits wide (the 'C25 repeat counter is 8 bits wide). This means that, when loading from RAM, the RPT instruction supports repeat counts up to 65,536. The assembler allows the RPT to support a16-bit immediate repeat count also. Note that RPT with long immediate addressing is, however, a two-word instruction.

D.4 On-Chip Peripheral Interfacing

The 'C5x has more peripherals than the 'C25; many 'C5x peripherals are enhancements of the 'C25 peripherals. The 'C25 has three peripheral circuits: serial port, timer, and 16 I/O ports. In addition to these peripherals, the 'C5x has software wait states and a divide-by-one clock.

The serial port of the 'C5x has been enhanced in that the CLKX pin can be configured as either an input or an output (CLKX is always an input on the 'C25). CLKX is configured as an input upon a device reset to maintain compatibility with the 'C25. The new serial port status bits are now mapped to a memory-mapped register that is used exclusively for the serial port. The serial port modes are no longer controlled via status register 1. Therefore, serial port modes that are changed by using LST1 instruction will no longer work. The mode bits must be set/reset via the serial port control register (SPC). The data transmit (DXR) and data receive (DRR) registers have been moved in the memory map from locations 1 and 0 to 33 and 32, respectively.

The timer has been enhanced on the 'C5x to include a divide-down factor of 1 to 17 and can be stopped or reset via software. These additional features are controlled via the timer control register (TCR). Upon reset, the divide-down factor is set to 1, and the timer is enabled to maintain compatibility with the 'C25. The timer (TIM) and period (PRD) registers have been moved in the memory map from locations 2 and 3 to locations 36 and 37, respectively.

The 16 input/output ports of the 'C5x are addressable in the data memory space. This allows direct access of the I/O space by the core CPU and supports bit operation in the I/O space via the PLU. The I/O space is increased from 16 ports to 65,536 ports. However, no additional decode circuitry is necessary if only 16 ports are used.

The 'C5x includes software wait-state generators that are mapped on 16K-word page sizes in the program and data memory spaces. There are also wait-state generators for the I/O ports. The I/O space wait-state generators can be mapped on two-word or on 8K-word boundaries. These wait-state generators allow the system to be programmed for 0, 1, 2, 3, 4, or 7 wait states, eliminating the need of an off-chip interfacing circuitry. External access wait states can be extended further via the READY signal.

Appendix E

XDS510 Design Considerations

The 'C5x DSPs support emulation through a dedicated emulation port. The emulation port is a superset of the IEEE 1149.1 (JTAG) standard and can be accessed by the XDS510 emulator. For details on the JTAG protocol, refer to the IEEE 1149.1 specification. The information in this appendix supports XDS510 Cable #2563988-001 Rev B.

This appendix contains the following sections

Topic **Page**

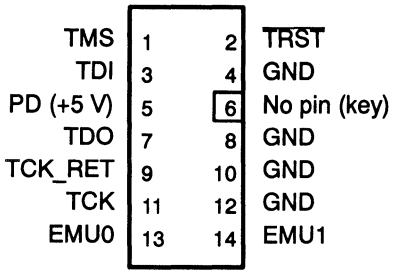
E.1 Cable Header and Signals

To perform emulation with the XDS510, your target system must have a 14-pin header (two 7-pin rows) with connections as shown in Figure E–1. Table E–1 describes the emulation signals.

Although you can use other headers, recommended parts include:

Straight header, unshrouded	DuPont Electronics® part number 67996–114
Right-angle header, unshrouded	DuPont Electronics® part number 68405–114

Figure E–1. 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

Table E–1. XDS510 Header Signal Description

Signal	State	Target State	Description
TMS	O	I	JTAG test mode select.
TDI	O	I	JTAG test data input.
TDO	I	O	JTAG test data output.
TCK	O	I	JTAG test clock. TCK is a 10-MHz clock source from the emulation cable pod. This signal can be used to drive the system test clock.
TRST	O	I	JTAG test reset.
EMU0	I	I/O	Emulation pin 0.
EMU1	I	I/O	Emulation pin 1.
PD	I	O	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to +5 volts in the target system.
TCK_RET	I	O	JTAG test clock return. Test clock input to the XDS510 emulator. May be a buffered or unbuffered version of TCK.

E.2 Bus Protocol

The IEEE 1149.1 specification covers the requirements for JTAG bus slave devices ('C5x) and provides certain rules. Those rules are summarized as follows:

- The TMS/TDI inputs are sampled on the rising edge of the TCK signal of the device.
- The TDO output is clocked from the falling edge of the TCK signal of the device.

When JTAG devices are daisy-chained together, the TDO of one device has approximately a half TCK cycle set up to the next device's TDI signal. This type of timing scheme minimizes race conditions that would occur if both TDO and TDI were timed from the same TCK edge. The penalty for this timing scheme is a reduced TCK frequency.

The IEEE 1149.1 specification does not provide rules for JTAG bus master (XDS510) devices. Instead, it states that it expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules and also provides an optional timing mode that allows you to run the emulation at a much higher frequency for improved performance.

E.3 Cable Pod

Figure E–2 shows a portion of the XDS510 emulator cable pod. These are the functional features of the emulator pod:

- Signals TDO and TCK_RET can be parallel-terminated inside the pod if required by the application. The default is that these signals are not terminated.
- Signal TCK is driven with a 74AS1034 device. Because of the high current drive (48 mA I_{OL}/I_{OH}), this signal can be parallel-terminated. If TCK is tied to TCK_RET, then you can use the parallel terminator in the pod.
- Signals TMS and TDI can be generated from the falling edge of TCK_RET, according to the IEEE 1149.1 bus slave device timing rules.They can also be driven from the rising edge of TCK_RET, which allows a higher TCK_RET frequency. The default is to match the IEEE 1149.1 slave device timing rules. This is an emulator software option that can be selected when the emulator is invoked. In general, single-processor applications can benefit from the higher clock frequency. However, in multiprocessing applications, you may wish to use the IEEE 1149.1 bus slave timing mode to minimize emulation system timing constraints.
- Signals TMS and TDI are series-terminated to reduce signal reflections.
- A 10-MHz test clock source is provided. You may also provide your own test clock for greater flexibility.

Figure E–2. Emulator Pod Interface

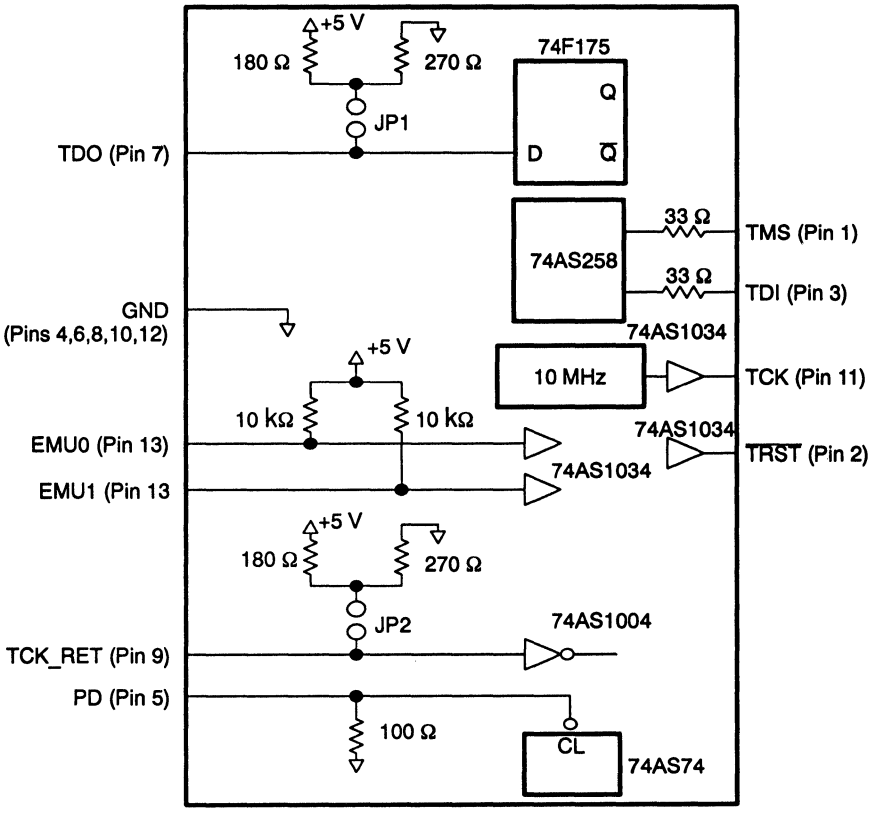


Figure E–3 and Table E–2 show the signal timings for the XDS510. Timing parameters are calculated from standard data sheet parts used in the cable pod. These timings are for reference only. Texas Instruments does not test or guarantee these timings.

The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

Figure E–3. Emulator Pod Timings

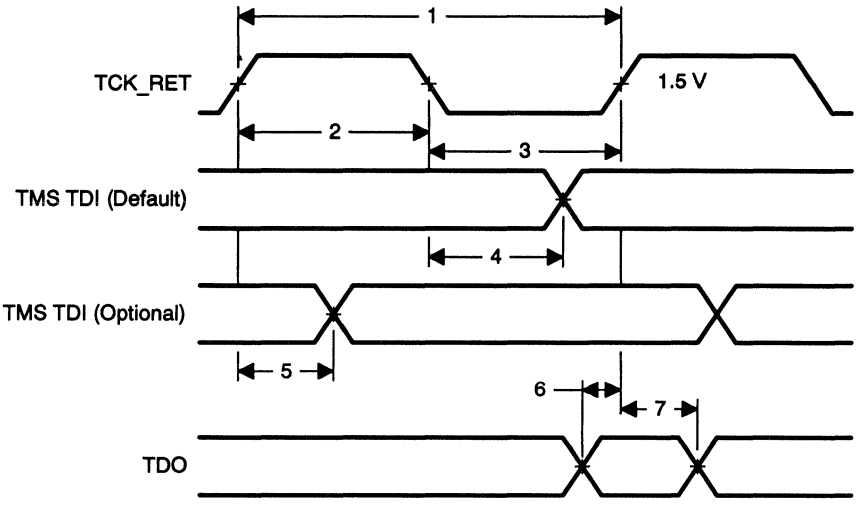


Table E–2. Emulator Pod Timing Parameters

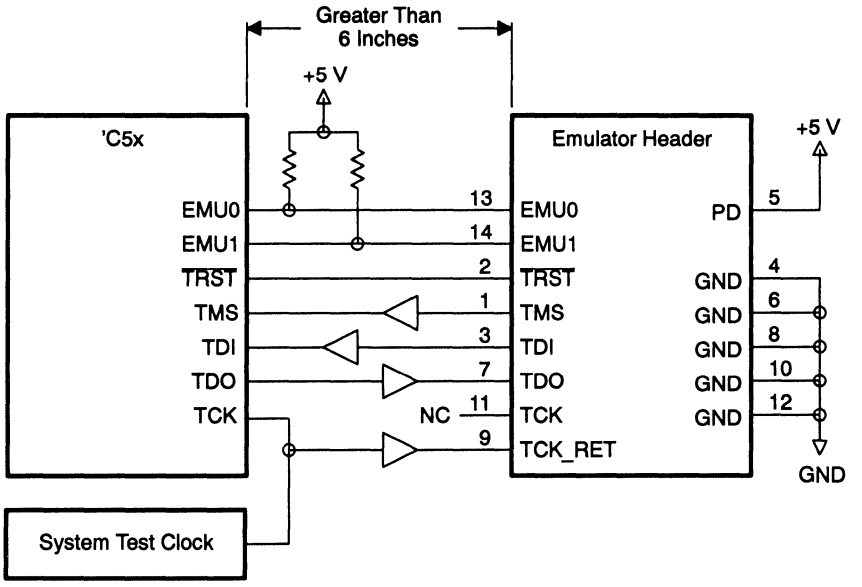
No.	Reference	Description	Min	Max	Unit
1	t_{TCKmin} t_{TCKmax}	TCK_RET period	35	200	ns
2	$t_{TCKhighmin}$	TCK_RET high pulse duration	15		ns
3	$t_{TCKlowmin}$	TCK_RET low pulse duration	15		ns
4	$t_{d(XTMXmin)}$ $t_{d(XTMXmax)}$	TMS/TDI valid from TCK_RET low (default timing)	6	20	ns
5	$t_{d(XTMSmin)}$ $t_{d)XTMSmax)}$	TMS/TDI valid from TCK_RET high (optional timing)	7	24	ns
6	$t_{su(XTDOmin)}$	TDO setup time to TCK_RET high	3		ns
7	$t_{hd(XTDOmin)}$	TDO hold time from TCK_RET high	12		ns

It is extremely important to provide high-quality signals between the emulator and the target processor. If the distance between the emulation header and the processor is greater than 6 inches, the emulation signals should be buffered. Sections E.4 and E.5 illustrate typical connections between the target processor and the emulation header.

E.4 Target System Test Clock

Figure E–4 shows an application with the system test clock generated in the target system. In this application the TCK signal is left unconnected.

Figure E–4. Target-System Generated Test Clock



There are two benefits to having the target system generate the test clock:

1) You can set the test clock frequency to match your system requirements. The emulator provides only a single 10-MHz test clock.

2) You may have other devices in your system that require a test clock when the emulator is not connected.

E.5 Multiprocessor Configuration

Figure E–5. Multiprocessor Connections

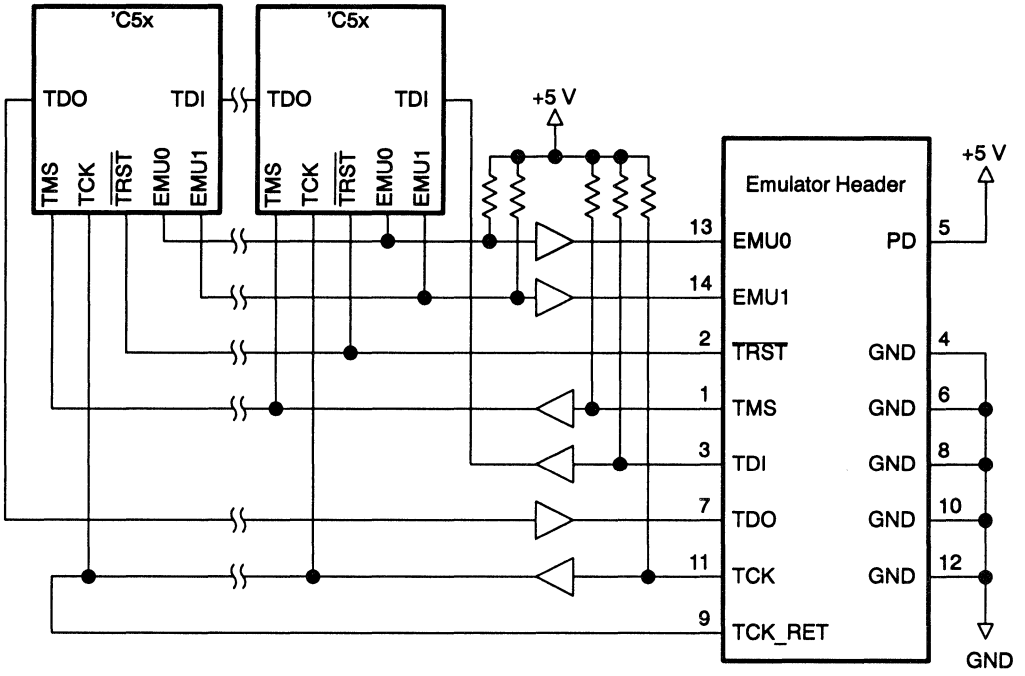
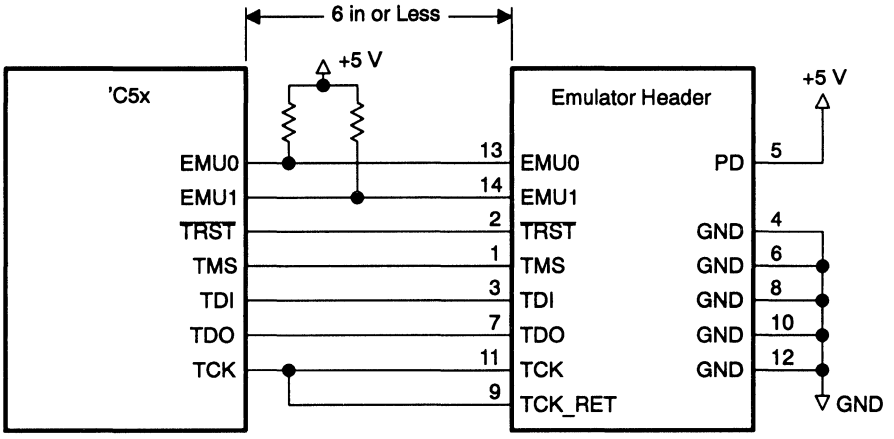


Figure E–5 shows a typical multiprocessor configuration. This is a daisy-chained configuration (TDO-TDI daisy-chained), which meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals in this example are buffered to isolate the processors from the emulator and provide adequate signal drive for the target system. One of the benefits of a JTAG test interface is that you can generally slow down the test clock to eliminate timing problems. Several key points to multiprocessor support are as follows:

- The processor TMS, TDI, TDO, and TCK should be buffered through the same physical package to control timing skew better.
- The input buffers for TMS, TDI, and TCK should have pullups to 5 volts. This will hold these signals at a known value when the emulator is not connected. A pullup of 4.7 kΩ or greater is suggested.
- Buffering EMU0 and EMU1 is optional, but highly recommended to provide isolation. These are not critical signals and do not need to be buffered through the same physical package as TMS, TCK, TDI, and TDO. Buffered and unbuffered signals are shown in Figure E–6 and Figure E–7.

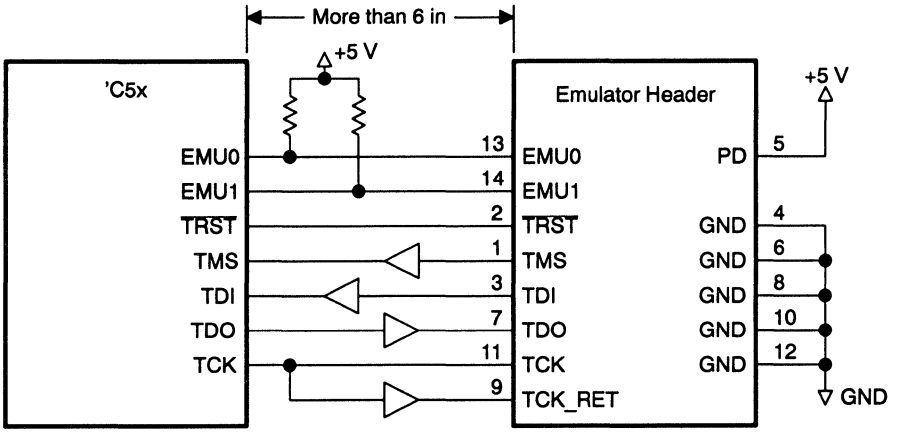
No signal buffering. In this situation, the distance between the header and the processor should be no more than 6 inches.

Figure E–6. Unbuffered Signals



Emulation signals buffered. The distance between the emulation header and the processor is greater than 6 inches. The emulation signals —TMS, TDI, TDO, and TCK_RET— are buffered through the same package.

Figure E–7. Buffered Signals



- The EMU0 and EMU1 signals must have pullups to 5 volts. The pullup resistor value should be chosen to provide a signal rise time less than 10 μs. A 4.7-kΩ resistor is suggested for most applications. EMU0 – 1 are I/O pins on the 'C4X and 'C5X; however, they are only inputs to the XDS510. In general, these pins are used in multiprocessor systems to provide global run/stop operations.

- It is extremely important to provide high quality signals, especially on the processor TCK and the emulator TCK_RET signal. In some cases, this may require you to provide special PWB trace routing and to use termination resistors to match the trace impedance. The emulator pod does provide optional internal parallel terminators on the TCK_RET and TDO. TMS and TDI provide fixed series termination.

E.6 Emulation Timing Calculations

The following are a few examples on how to calculate the emulation timings in your system. For actual target timing parameters, see the appropriate device data sheets.

Assumptions:

$t_{su(TTMS)}$	Target TMS/TDI setup to TCK high	10 ns
$t_{h(TTMS)}$	Target TMS/TDI hold from TCK high	5 ns
$t_{d(TTDO)}$	Target TDO delay from TCK low	15 ns
$t_{d(bufmax)}$	Target buffer delay maximum	10 ns
$t_{d(bufmin)}$	Target buffer delay minimum	1 ns
$t_{(bufskew)}$	Target buffer skew between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{tckfactor}$	Assume a 40/60 duty cycle clock	0.4

Given in Table E–2 (page E-6):

$t_{d(XTMSmax)}$	XDS510 TMS/TDI delay from TCK_RET low, maximum	20 ns
$t_{d(XTMX)}$	min XDS510 TMS/TDI delay from TCK_RET low, minimum	6 ns
$t_{d(XTMSmax)}$	XDS510 TMS/TDI delay from TCK_RET high, max	24 ns
$t_{d(XTMXmin)}$	XDS510 TMS/TDI delay from TCK_RET high, minimum	7 ns
$t_{su(XTDOmin)}$	TDO setup time to XDS510 TCK_RET high	3 ns

There are two key timing paths to consider in the emulation design:

(1) the TCK_RET/TMS/TDI (t_{prdtck_TMS}) path, and

(2) the TCK_RET/TDO (t_{prdtck_TDO}) path.

In each case, the worst case path delay is calculated to determine the maximum system test clock frequency.

Case 1: Single processor, direct connection, TMS/TDI timed from TCK_RET low (default timing).

$t_{prdtck\_TMS} = [t_{d(XTMSmax)} + t_{su(TTMS)}] / t_{tckfactor}$
$= (20 \text{ ns} + 10 \text{ ns}) / 0.4$
$= 75 \text{ ns (13.3 MHz)}$

$t_{prdtck\_TDO} = [t_{d(TTDO)} + t_{su(XTDOmin)}] / t_{tckfactor}$
$= (15 \text{ ns} + 3 \text{ ns}) / 0.4$
$= 45 \text{ ns (22.2 MHz)}$

In this case, the TCK/TMS path is the limiting factor.

Case 2: Single processor, direct connection, TMS/TDI timed from TCK_RET high (optional timing).

$t_{prdtck\_TMS} = t_{d(XTMSmax)} + t_{su(TTMS)}$
$= (24 \text{ ns} + 10 \text{ ns})$
$= 34 \text{ ns (29.4 MHz)}$

$t_{prdtck\_TDO} = [t_{d(TTDO)} + t_{su(XTDOmin)}] / t_{tckfactor}$
$= (15 + 3) / 0.4$
$= 45 \text{ ns (22.2 MHz)}$

In this case, the TCK/TDO path is the limiting factor. One other thing to consider in this case is the TMS/TDI hold time. The minimum hold time for the XDS510 cable pod is 7 ns, which meets the 5-ns hold time of the target device.

Case 3: Single/multiple processor, TMS/TDI buffered input; TCK_RET/TDO buffered output, TMS/TDI timed from TCK_RET high (optional timing).

$t_{prdtck\_TMS} = t_{d(XTMSmax)} + t_{su(TTMS)} + 2t_{d(bufmax)}$
$= 24 \text{ ns} + 10 \text{ ns} + 2 (10)$
$= 54 \text{ ns (18.5 MHz)}$

$t_{prdtck\_TDO} = [t_{d(TTDO)} + t_{su(XTDOmin)} + t_{(bufskew)}] / t_{tckfactor}$
$= (15 \text{ ns} + 3 \text{ ns} + 1.35 \text{ ns}) / 0.4$
$= 58.4 \text{ ns} \quad (20.7 \text{ MHz})$

In this case, the TCK/TMS path is the limiting factor. The hold time on TMS/TDI is also reduced by the buffer skew (1.35 ns) but still meets the minimum device hold time.

Case 4: Single/multiprocessor, TMS/TDI/TCK buffered input; TDO buffered output, TMS/TDI timed from TCK_RET low (default timing).

t_{prdtck_TMS} = $[t_{d(XTMSmax)} + t_{su(TTMS)} + t_{bufskew}] / t_{ckfactor}$
= (24 ns + 10 ns + 1.35 ns) / 0.4
= 88.4 ns (11.3 MHz)

t_{prdtck_TDO} = $[t_{d(TTDO)} + t_{su(XTDOmin)} + t_{d(bufmax)}] / t_{ckfactor}$
= (15 ns + 3 ns + 10 ns) / 0.4
= 70 ns (14.3 MHz)

In this case, the TCK/TMS path is the limiting factor.

In a multiprocessor application, it is necessary to ensure that the EUM0–1 lines can go from a logic low level to a logic high level in less than 10 μs. This can be calculated as follows (remember that t = 5 RC):

t_{rise} = $5(R_{pullup} \times N_{devices} \times C_{load\_per\_device})$
= $5(4.7k\Omega \times 16 \times 15pF)$
= 5.64 μs

Appendix F

Analog Interface Peripherals and Applications

Texas Instruments 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.

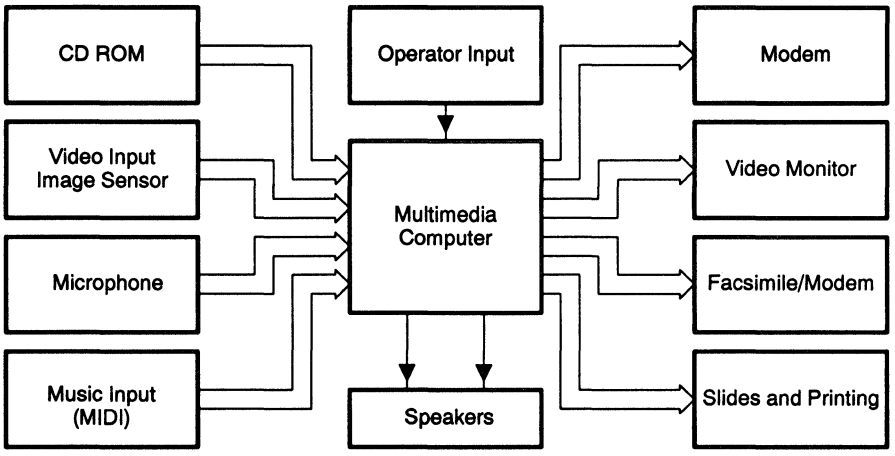
Topic **Page**

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 enhances 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 x 480 pixels with 1, 2, 4, and 8-bit memory-mapped color. Also, 24-bit true color is supported, and 1024 x 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 3-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

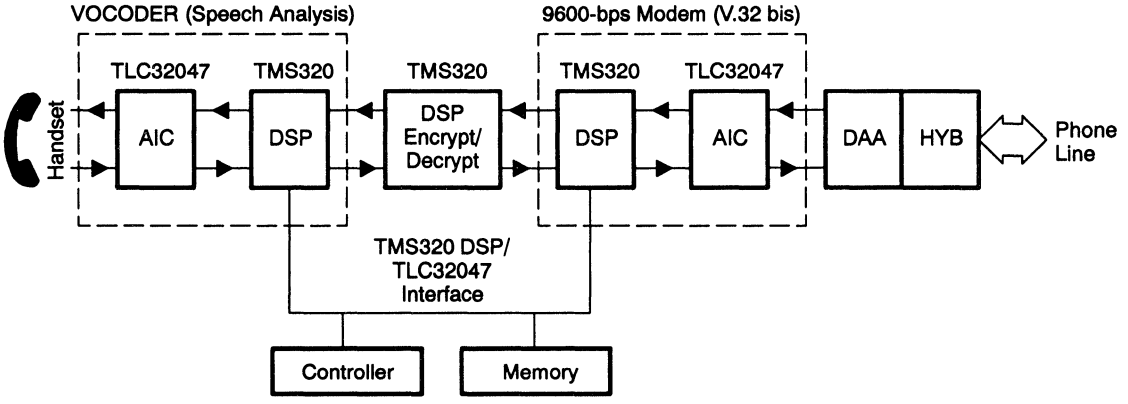
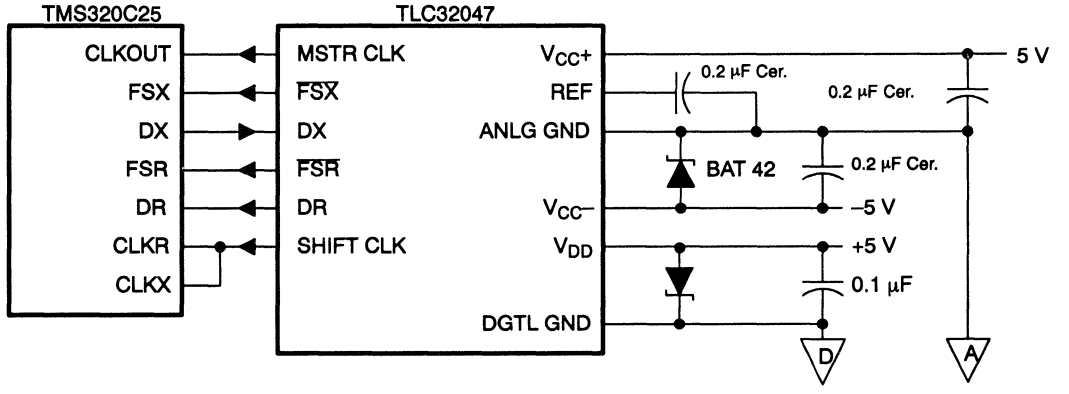


Figure F–3 shows the interfacing of the 'C25 DSP to the TLC32047 AIC that constitutes the building blocks of the 9600-bps V.32 bis modem shown in Figure F–2.

Figure F–3. TMS320C25 to TLC32047 Interface



F.1.2 Multimedia-Related Devices

As shown in Table F–1, 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.

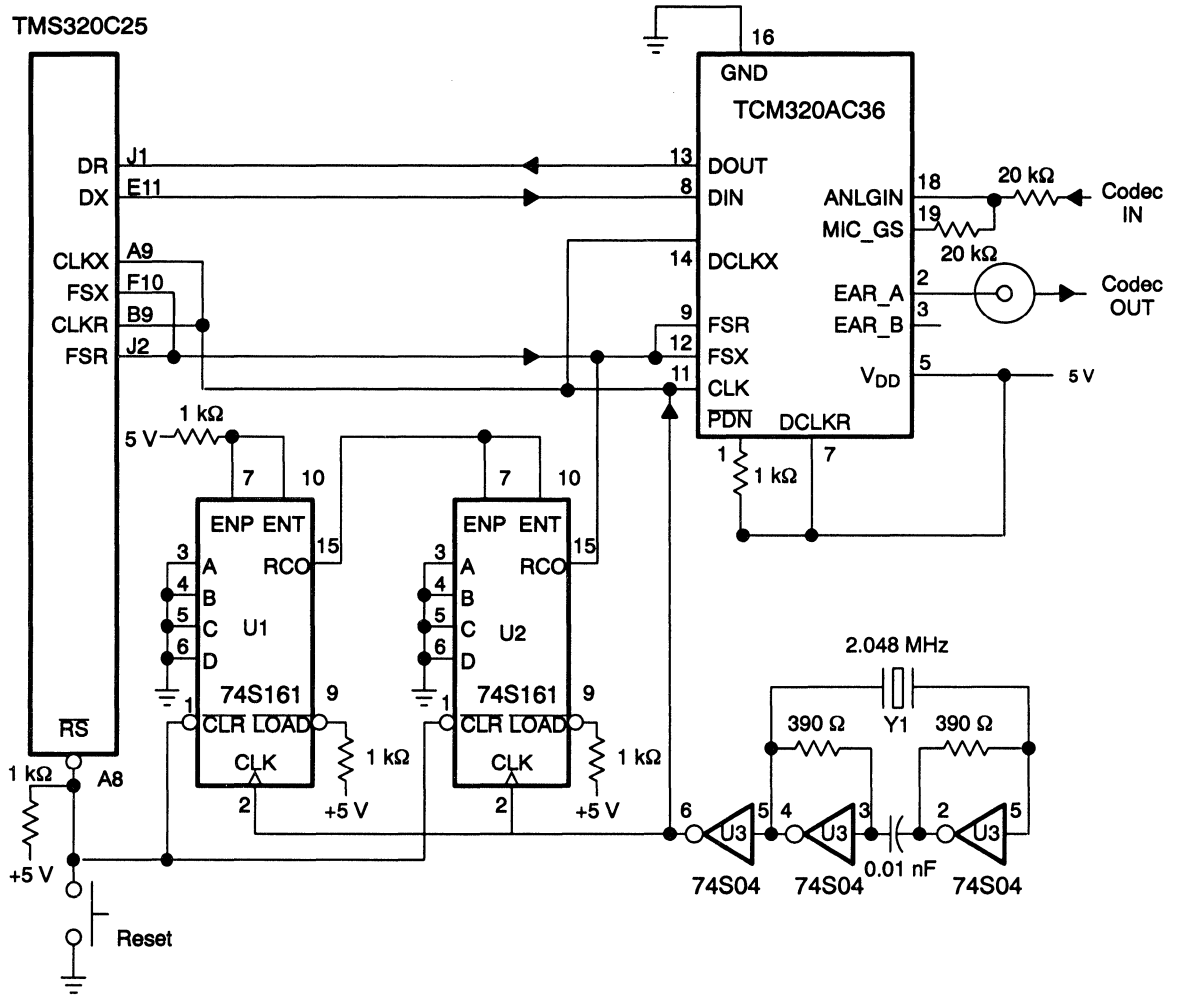
F.2 Telecommunications Applications

The TI linear product line focuses on three primary telecommunications application areas: subscriber instruments (telephones, modems, etc.), central office line card products, and personal communications. Subscriber instruments include 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 include 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 communication (PCN) and cellular products include the TCM320AC3x family of 5-volt voice-band audio processors (VBAP).

TI continues to develop new telecom integrated circuits, such as a high-performance 3-volt combo family for personal communications applications, and an RF power amplifier family for hand-held and mobile cellular phones.

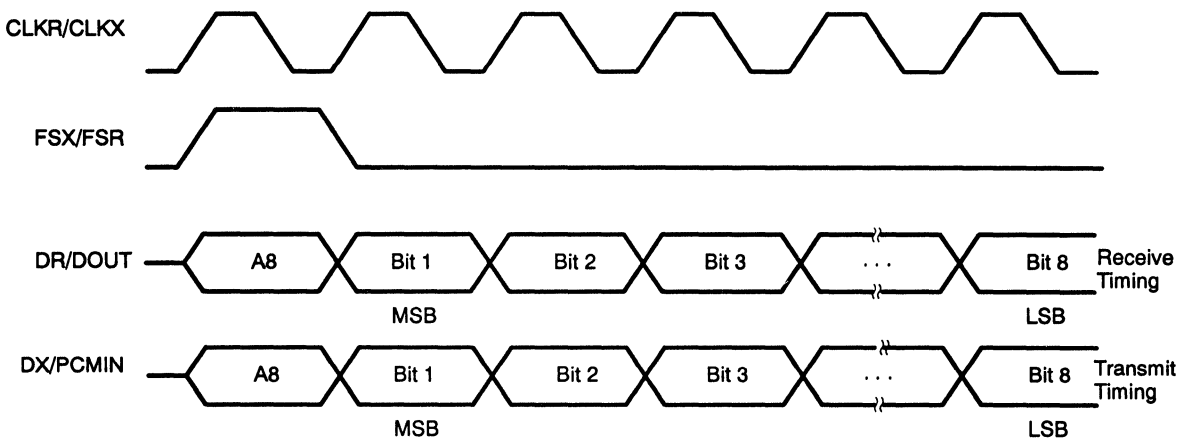
System Design Considerations. The size, network complexity, and compatibility requirements of telecommunications central office systems create demanding performance requirements. Combo voice-band filter performance is typically ± 0.15 dB in the passband. Idle channel noise must be on the order of 15 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 'C25 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 4-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 are contained in the *1991 Telecommunications Circuits Databook*, (literature number SCTD001). To request your copy, contact your nearest Texas Instruments field sales office.

For further information on these telecommunications products, please call TI Linear Applications at (214) 997–3772.

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. 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

Figure F–6. General Telecom Applications

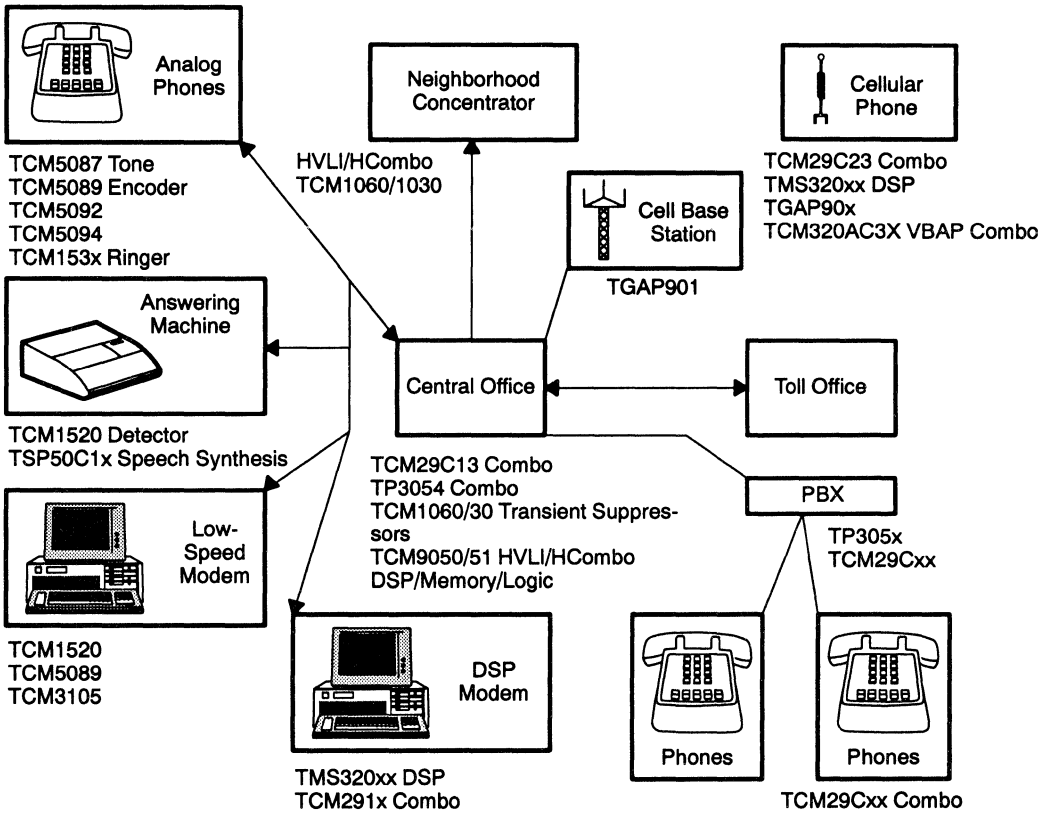
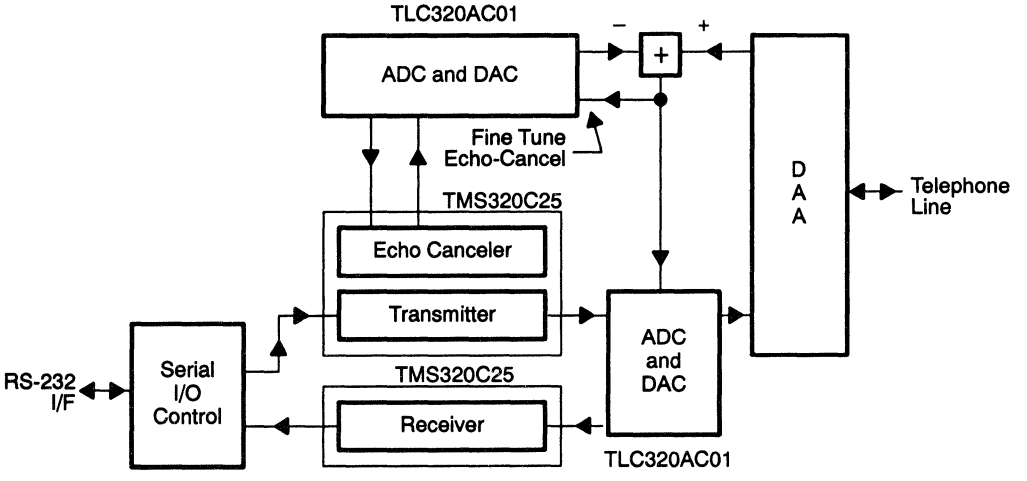


Figure F–7. Generic Telecom Application



F.3 Dedicated Speech Synthesis Applications

For dedicated speech synthesis applications, Texas Instruments 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 effective in low-cost applications. The speech synthesis technology provided by the dedicated chips is either LPC (linear-predictive coding) or CVSD (continuously variable slope delta modulation). Table F–5 shows the characteristics of the TI voice synthesizers.

Table F–5. Voice Synthesizers

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

TI has low-cost memories that are ideal for use with speech synthesizers chips. Texas Instruments 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					
	TSP60C18	TSP60C19	TSP60C20	TSP60C80	TSP60C81
Size	256K	256K	256K	1M	1M
No. of Pins	16	16	28	28	28
Interface	Parallel 4-bit	Serial	Parallel/serial 8-bit	Serial	Parallel 4-bit
For use with:	TSP50C1x	TSP50C4x	TSP50C4x	TSP50C4x	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

Speech Synthesis Development Tools

Software:

EVM	Code development tool

Speech:

SAB	Speech audition board
SD85000	PC-based speech analysis system

System:

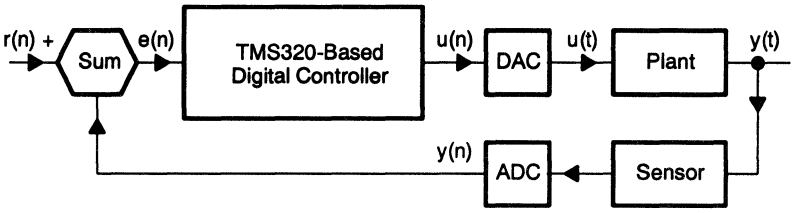
SEB	System emulator board
SEB60Cxx	System emulator boards for speech memories

For further information on these speech synthesis products, please call TI Linear Applications at (214) 997–3772.

F.4 Servo Control/Disk Drive Applications

Several years ago, most servo control systems used only analog circuitry. However, the growth of digital signal processing has made digital control theory a reality. Figure F–8 shows a block diagram of a generic digital control system using a DSP, along with an ADC and 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. Disk drives have increased storage capacity from 5 megabytes to over 1 gigabyte in the past decade, which 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 shows a block diagram of a disk drive control system.

Figure F–9. Disk Drive Control System Block Diagram

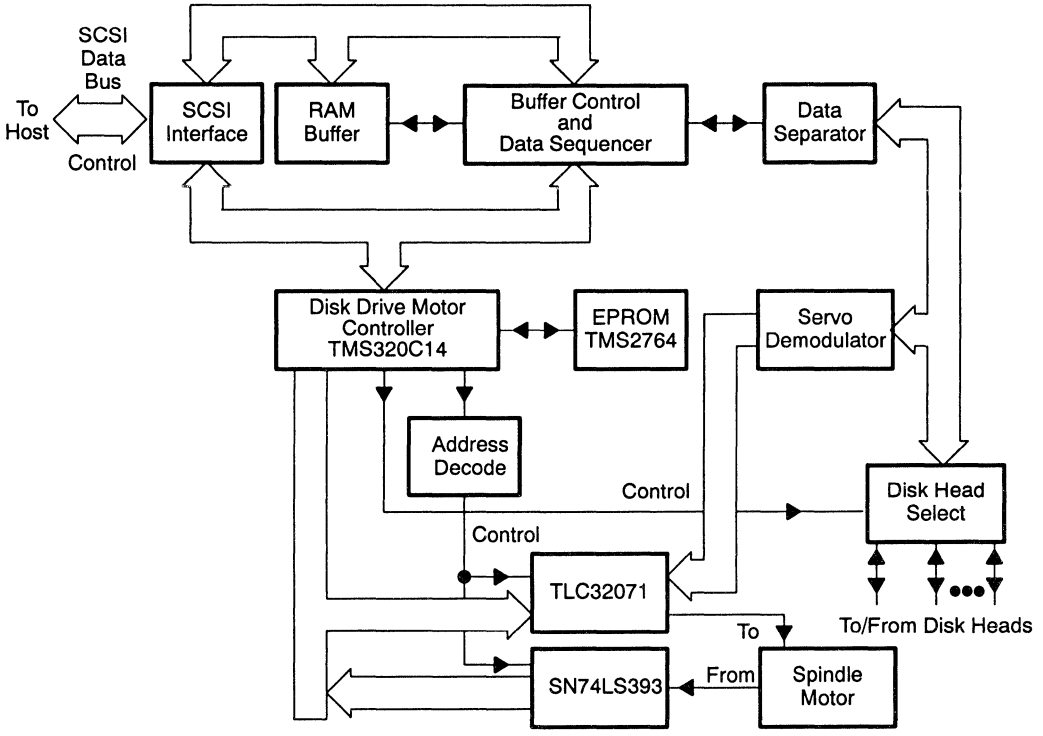


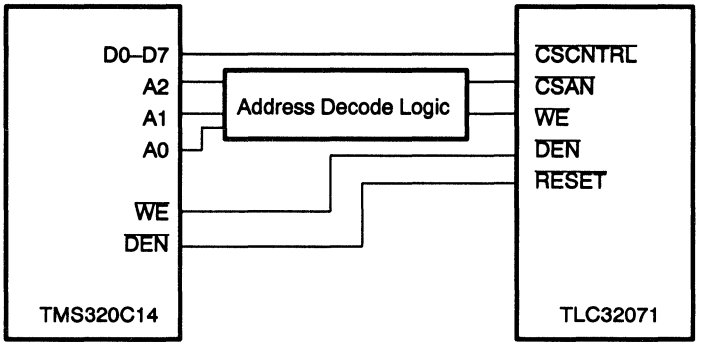
Table F–8 lists analog/digital interface devices used for servo control.

Table F–8. 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 'C14 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 analog interface circuits (AIC) 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 these parameters are software programmable, making the AIC suitable for a variety of applications. Table F–9 has the description and characteristics of these devices.

Table F–9. 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 TLC3204x family performs level-shifting, filtering, and A/D and D/A data conversion. The DSP's many 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 fall-back 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. High-Speed V.32 Bis and Multistandard Modem With the TLC320AC01 AIC

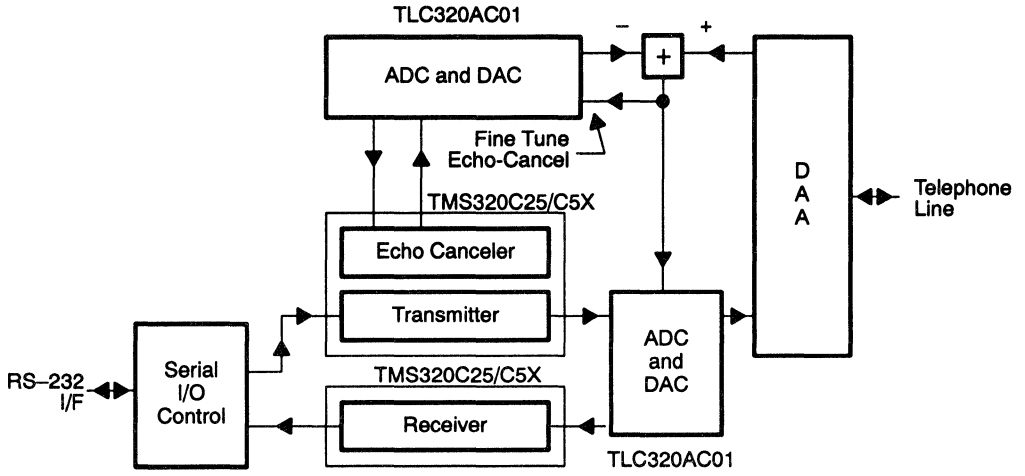


Figure F–11 shows a V.32 bis modem implementation using the'C25 and a TLC320AC01. The upper 'C25 performs echo cancellation and transmit data functions, while the lower 'C25 performs receive data and timing recovery functions. The echo canceler simulates the telephone channel and generates an estimated echo of the transmit data signal. The TLC320AC01 performs the following functions:

Upper TLC320AC01 D/A Path: Converts the estimated echo, as computed by the upper 'C25, 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 'C25.

Lower TLC320AC01 D/A Path: Converts the upper 'C25 transmit output to an analog signal, performs a smoothing filter function, and drives the DAC.

Lower TLC320AC01 D/A Path: Converts the echo-free receive signal to a digital signal, which is sent to the lower 'C25 to be decoded.

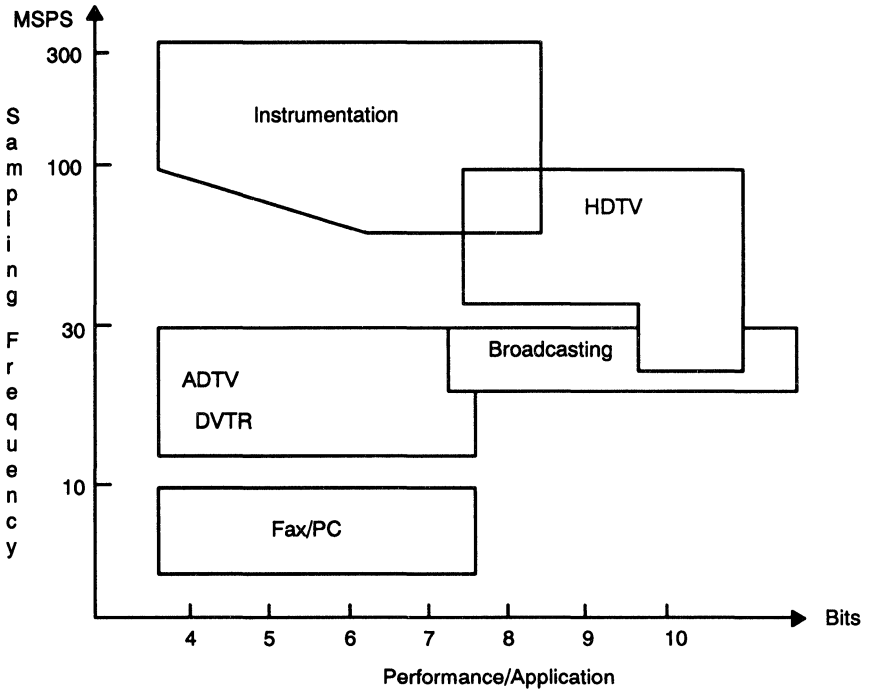
Note:

The example in Figure F–11 is for illustration only. In reality, one single 'C5x DSP can implement high-speed modem functions.

F.6 Advanced Digital Electronics Applications for Consumers

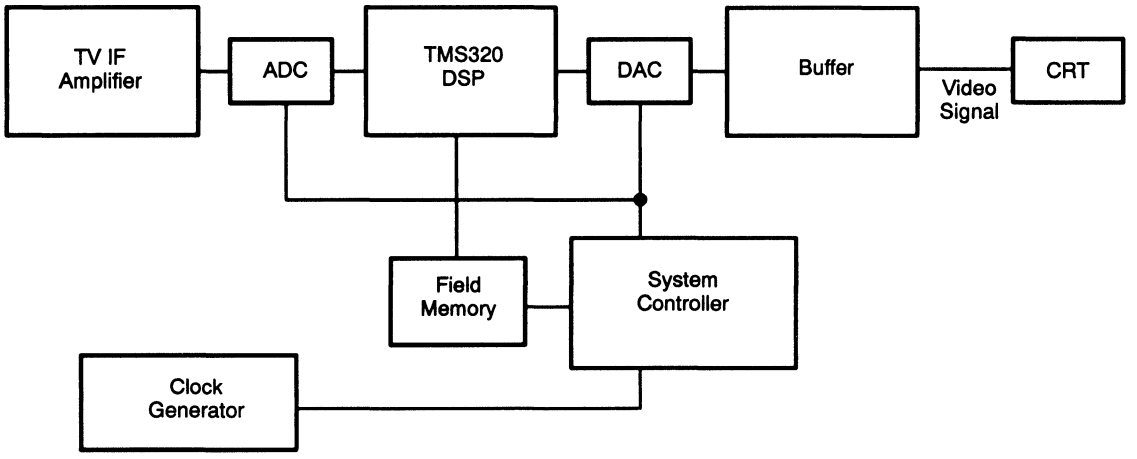
With the extensive use of the TMS320 DSPs in consumer electronics, much electromechanical control and signal processing can be done in the digital domain. Digital systems generally require some form of analog interface, usually in the form of high-performance ADCs and DACs. Figure F–12 shows the general performance requirements for a variety of applications.

Figure F–12. Applications Performance Requirements



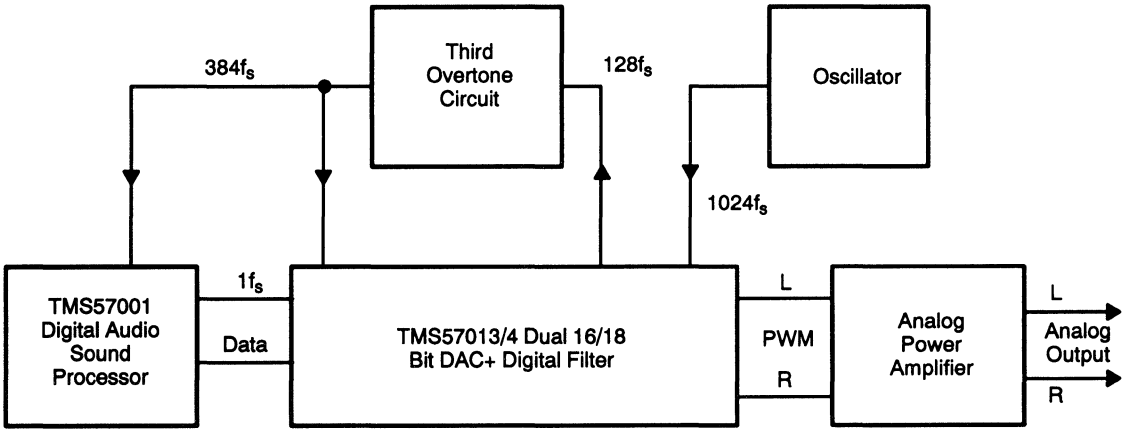
Advanced Television System Design Considerations. Advanced Digital Television (ADTV) is a technology that uses digital signal processing to enhance video and audio presentations and to reduce noise and ghosting. Because of these DSP techniques, a variety of features can be implemented, including frame store, picture-in-picture, improved sound quality, and zoom. The bandwidth requirements remain at the existing 6-MHz television allocation. From the IF(intermediate frequency) output, the video signal is converted by an 8-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



VCRs, compact disc and DAT players, and PCs are a few of the products that have taken a major position in the marketplace in the last ten 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 disc 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 1-bit digital-to-analog converters (DAC) include an 8 times over sampling digital filter designed for digital audio systems, such as CDPs, DATs, CDIs, LDPs, digital amplifiers, car stereos, and BS tuners. They are also suitable for all systems that include digital sound processing like TVs, VCRs, musical instruments, NICAM systems, 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 FIR filter and third-order $\Delta\Sigma$ modulation to get –75-dB stop band attenuation and 96-dB SNR. The output is PWM wave, which facilitates analog signal through a low-pass filter.

Table F–10 lists TI products for analog interfacing to digital systems.

Table F–10.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

Memories, Sockets, and Crystals

This appendix provides product information regarding memories and sockets that are manufactured by Texas Instruments and are compatible with the 'C5x. Information is also given regarding crystal frequencies, specifications, and vendors.

The contents of the major areas in this appendix are listed below.

Topic **Page**

G.1 Memories

This section provides product information on EPROM memories that can be interfaced with 'C5x processors. Refer to *Digital Signal Processing Applications with the TMS320* Family for additional information on interfaces using memories and analog conversion devices.

Data sheets for EPROM memories are located in the *MOS Memory Data Book* (literature number SMYD008).

TMS27C64
TMS27C128
TMS27C256
TMS27C512

Another EPROM memory, TMS27C291/292, is described in a data sheet (literature number SMLS291A).

G.2 Sockets

AMP manufactures a 132-pin quad flat pack socket for the 'C5x devices. There are two pieces — a base (the socket itself) and a lid. The part numbers are

Base	AMP part number 821942-1
Lid	AMP part number 821949-5

For additional information about TI sockets, contact the nearest TI sales office or:

Texas Instruments Incorporated
Connector Systems Dept, M/S 14–3
Attleboro, MA 02703
(617) 699–5242/5269
Telex: 92–7708

G.3 Crystals

This section lists the commonly used crystal frequencies, crystal specification requirements, and the names of suitable vendors.

Table G–1 lists the commonly used crystal frequencies and the devices with which they can be used.

Table G–1. Commonly Used Crystal Frequencies

Device	Frequency
TMS320C25	40.96 MHz
TMS320C5x	20.48 MHz 40.96 MHz

When connected across X1 and X2/CLKIN of the TMS320 processor, a crystal enables the internal oscillator. Crystal specification requirements are listed below.

Load capacitance = 20 pF
Series resistance = 30 ohm
Power dissipation = 1 mW

Vendors of crystals suitable for use with TMS320 devices are listed below.

RXD, Inc.
Norfolk, NB
(800) 228–8108

N.E.L. Frequency Controls, Inc.
Burlington, WI
(414) 763–3591

CTS Knight, Inc.
Contact the local distributor.

Appendix H

ROM Codes

The size of a printed circuit board must be considered in many DSP applications. To fully utilize the board space, Texas Instruments offers an option that reduces the chip count and provides a single-chip solution to its customers. On the 'C51, this option incorporates 8K words of on-chip program from a mask programmable ROM. This allows you to use a code-customized processor for a specific application while taking advantage of the following:

- ❑ Greater memory expansion
- ❑ Lower system cost
- ❑ Less hardware and wiring
- ❑ Smaller PCB

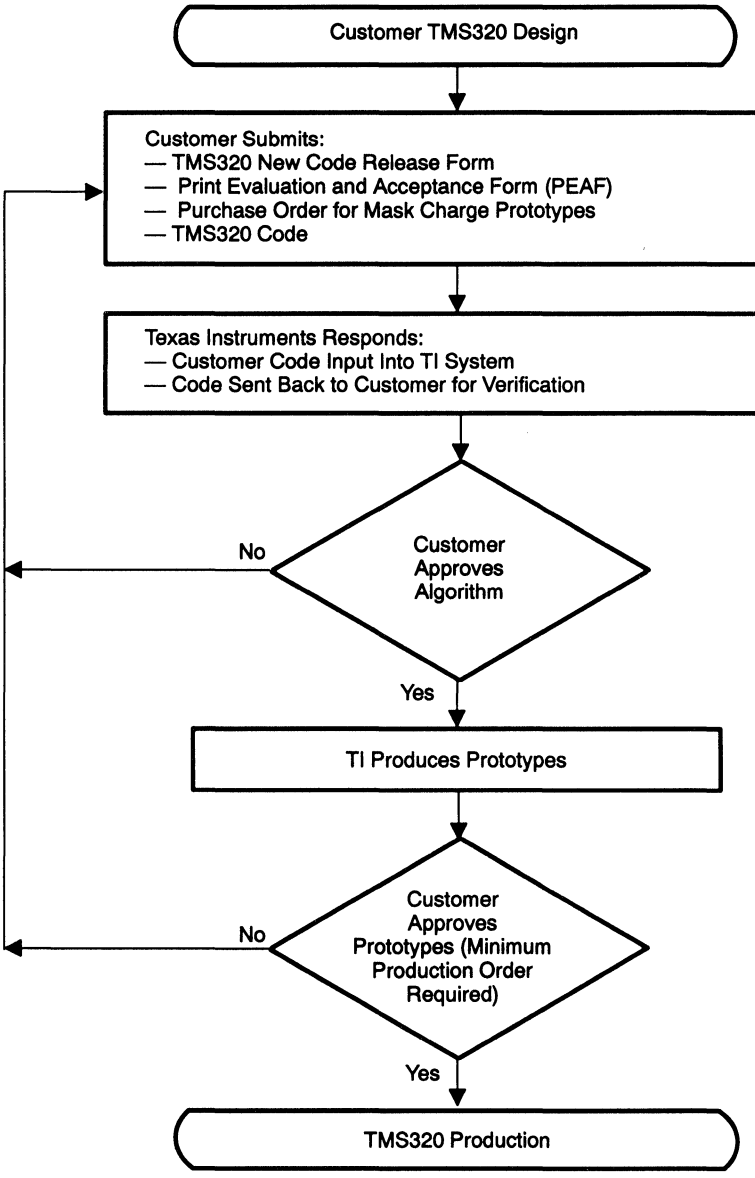
If used often, the routine or entire algorithm can be programmed into the on-chip ROM of a TMS320 DSP. TMS320 programs can also be expanded by using external memory; this reduces chip count and allows for a more flexible program memory. Multiple functions are easily implemented by a single device, thus enhancing system capabilities.

TMS320 development tools are used to develop, test, refine, and finalize the algorithms. The microprocessor/microcomputer (MP/$\overline{\text{MC}}$) mode is available on all ROM-coded TMS320 DSP devices when accessing either on-chip or off-chip memory is required. The microprocessor mode is used to develop, test, and refine a system application. In this mode of operation, the TMS320 acts as a standard microprocessor by using external program memory. When the algorithm has been finalized, the designer may submit the code to Texas Instruments for masking into the on-chip program ROM. At that time, the TMS320 becomes a microcomputer that executes customized programs from the on-chip ROM. Should the code need changing or upgrading, the TMS320 may once again be used in the microprocessor mode. This shortens the field upgrade time and avoids the possibility of inventory obsolescence.

H.1 ROM Code Flow

Figure H–1 illustrates the procedural flow for developing and ordering TMS320 masked parts. When ordering, there is a one-time/nonrefundable charge for mask tooling. A minimum production order per year is required for any masked-ROM device. ROM codes will be deleted from the TI system one year after the last delivery.

Figure H–1.TMS320 ROM Code Flowchart



A TMS320 ROM code may be submitted in one of the following formats (the preferred media is 5-1/4-in floppies):

5-1/4-in Floppy:	COFF format from macro-assembler/linker (preferred)
Modem (BBS):	COFF format from macro-assembler/linker
EPROM (others):	TMS27C64
PROM:	TBP28S166, TBP28S86

When a code is submitted to Texas Instruments for masking, the code is reformatted to accommodate the TI mask generation system. System-level verification by the customer is therefore necessary. Although the code has been reformatted, it is important that the changes remain transparent to the user and do not affect the execution of the algorithm. The formatting changes involve the removal of address relocation information (the code address begins at the base address of the ROM in the TMS320 device and progresses without gaps to the last address of the ROM on the TMS320 device) and the addition of data in the reserved locations of the ROM for device ROM test. Note that because these changes have been made, a checksum comparison is not a valid means of verification.

With each masked device order, the customer must sign a disclaimer stating:

"The units to be shipped against this order were assembled, for expediency purposes, on a prototype (that is, nonproduction qualified) manufacturing line, the reliability of which is not fully characterized. Therefore, the anticipated inherent reliability of these prototype units cannot be expressly defined."

and a release stating:

"Any masked ROM device may be resymbolized as TI standard product and resold as though it were an unprogrammed version of the device, at the convenience of Texas Instruments."

The use of the ROM-protect feature does not hold for this release statement. Additional risk and charges are involved when the ROM-protect feature is selected. Contact the nearest TI Field Sales Office for more information on procedures, leadtimes, and cost associated with the ROM-protect feature.

Appendix I

Development Support

Texas Instruments offers an extensive line of development tools for the 'C5x 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 'C5x-based applications:

Software Development Tools:
- Assembler/Linker
- Simulator
- Optimizing ANSI C compiler
- Application Algorithms
- C/Assembly Debugger and Code Profiler

Hardware Development Tools:
- Emulator XDS510
- 'C5x EVM (Evaluation Module)

Each 'C5x support product is described in the *TMS320 Family Development Support Reference Guide* (literature number SPRU011). In addition, more than 100 TMS320 third-party developers provide support products to complement TI's offering. For more information on third-party support 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 contains the following:

Topic **Page**

I.1 Device and Development Support Tool Nomenclature

To designate the stages in the product development cycle, Texas Instruments assigns prefixes to the part numbers of all TMS320 devices and support tools. Each TMS320 member has one of three prefixes: TMX, TMP, and TMS. Texas Instruments recommends two of three possible prefix designators for its support tools: TMDX and TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices/tools (TMS/TMDS). This development flow is defined below.

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 Texas Instruments internal qualification testing.

TMDS Fully qualified development support product.

TMX and TMP devices and TMDX development support tools are shipped against the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

TMS devices and TMDS development support tools have been fully characterized, and the quality and reliability of the device has been fully demonstrated. Texas Instruments standard warranty applies.

Note:

Predictions show that prototype devices (TMX or TMP) will have a greater failure rate than the standard production devices. Texas Instruments recommends that these devices *not* be used in any production system because their expected end-use failure rate is still undefined. Only qualified production devices are to be used.

TI device nomenclature also includes a suffix with the device family name. This suffix indicates the package type (for example, N, FN, or GB) and temperature range (for example, L). Figure I–1 provides a legend for reading the complete device name for any TMS320 family member.

Figure I–1. TMS320 Device Nomenclature

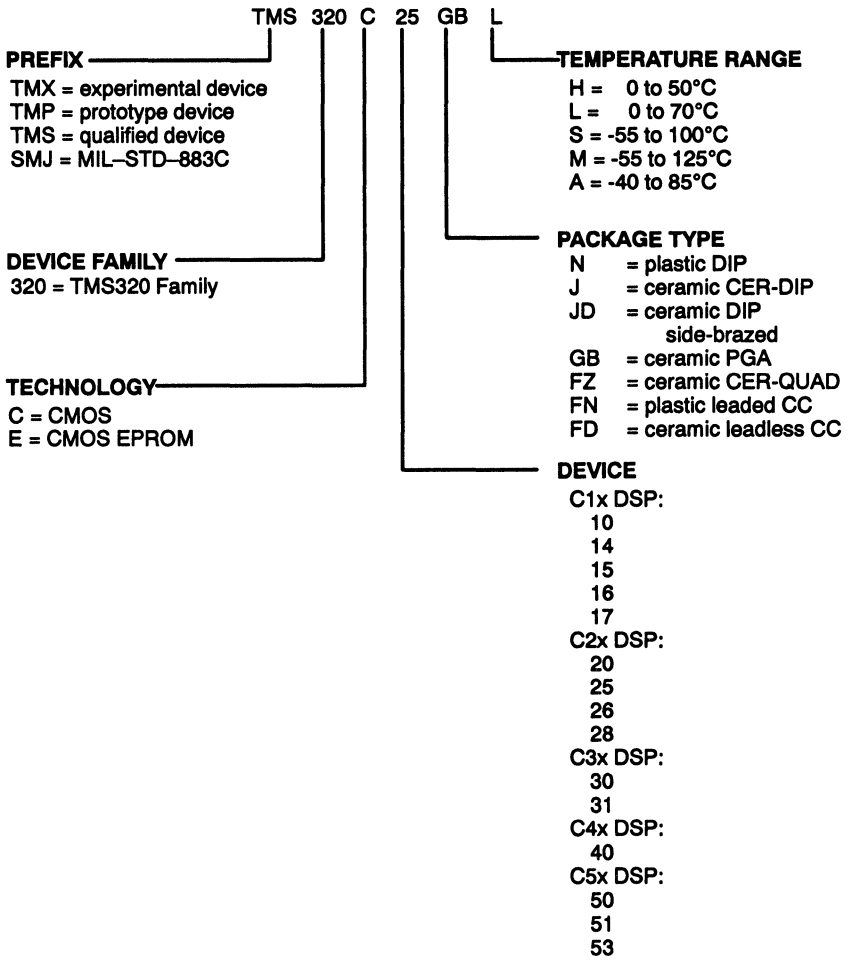
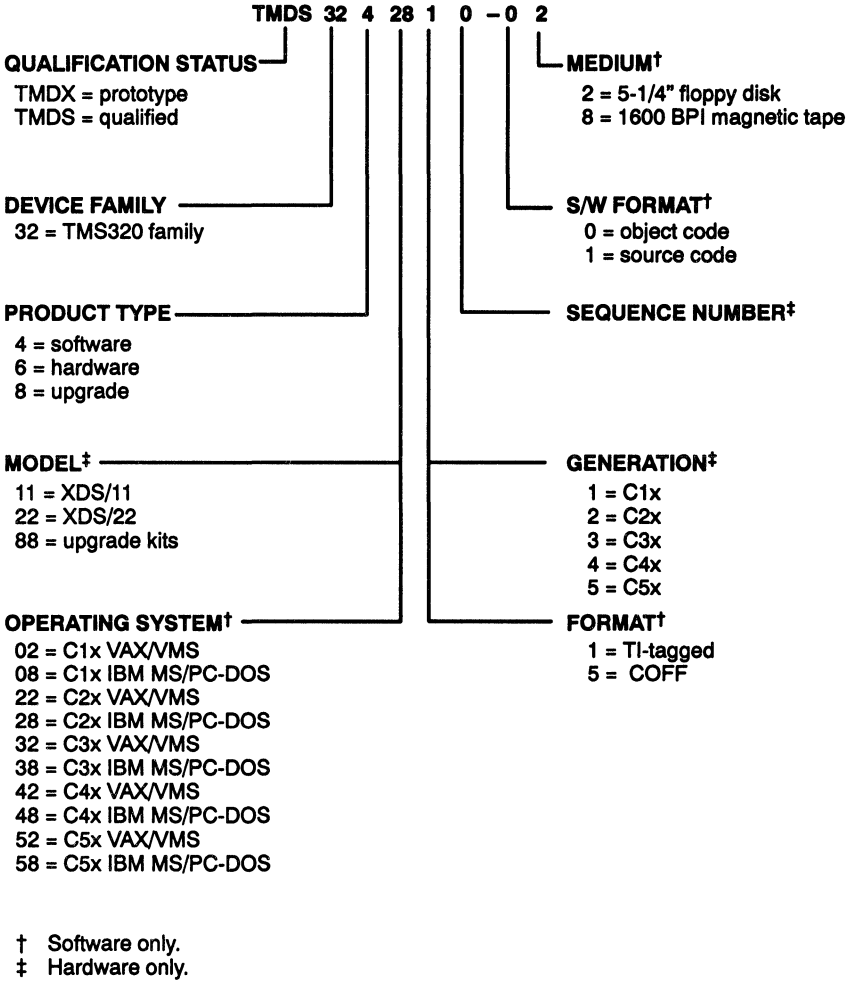


Figure I–2 provides a legend for reading the part number for any TMS320 hardware or software development tool.

Figure I–2. TMS320 Development Tool Nomenclature



† Software only.
‡ Hardware only.

I.2 Hewlett-Packard E2442A Preprocessor 'C5x Interface

The Hewlett-Packard E2442A 'C5x preprocessor interface provides a mechanical and electrical connection between your target system and an HP logic analizer. Preprocessor hardware captures processor signals and passes them to the logic analyzer at the appropriate time, depending on the type of measurement you are making. With the preprocessor plugged in, both state and timing analysis is available. Two connectors are loaded onto the preprocessor to facilitate communications with other debugging tools. You can use a BNC connector, when used with the sequencer of the logic analyzer to halt the processor on a condition. Then the 'C5x HLL debugger can be used to examine the state of the system (for example, microprocessor registers). Likewise, a 14-pin connector is available to receive signals from the XDS510 development system. These signals can be used when defining a trigger condition for the analyzer.

The HP E2442A includes software which automatically labels address, data and status lines. Additionally, a disassembler is included. The disassembler processes state traces and displays the information on TMS320 mnemonics.

I.2.1 'C5x Devices Supported

The Hewlett-Packard E2442A preprocessor 'C5x interface supports the 'C50, 'C51, and 'C53 devices.

I.2.2 Capabilities

The preprocessor supports three modes of operation: in the first mode, *State per Transfer,* the preprocessor clocks the logic analizer only when a bus transfer is complete. In this mode, wait and halt states are filtered out. In the second mode, CLKOUT1 clocks the analyzer every time the microprocessor is clocked. This mode captures all bus states. An example application would be to locate memory locations that do not respond to requests for data. In the third mode, you can use the HP E2442A to make timing measurements.

The JTAG TAP (test access port) controller can be monitored in realtime. TAP state can be viewed under the predefined label *TAP.*

I.2.3 Logic Analyzers Supported

- ❑ HP 1650A/B
- ❑ HP 16510B
- ❑ HP 16511B
- ❑ HP16540/41(A/D)
- ❑ HP16550A
- ❑ HP 1660A/61A/62A

I.2.4 Pods Required

There are eight pod-connectors on the preprocessor. Three are terminated and best used for state analysis as all signals needed for disassembly are available. The other five connectors are not terminated and contain all processor signals, including a second set of the signals needed for disassembly. This allows you to double probe these signals, making simnultaneous state and timing measurements.

I.2.5 Termination Adapters (TAs)

Of the eight pods, three are terminated. You may need to order up to five termination adapters, depending on how many pods are connected at the same time.

I.2.6 Availability

For more information and availabilty of the Hewlett-Packard E2442A contact:

Hewlett-Packard Company
2000 South Park Place
Atlanta, GA 30339
(404) 980–7351

Index

N

O

P

Q

R

S

T

U

V

W

X

Z

NOTES

NOTES

NOTES

NOTES

NOTES

TI Worldwide Sales and Representative Offices

AUSTRALIA / NEW ZEALAND: Texas Instruments Australia Ltd.: Sydney [61] 2-910-3100, Fax 2-805-1186; **Melbourne** 3-696-1211, Fax 3-696-4446.
BELGIUM: Texas Instruments Belgium S.A./N.V.: Brussels [32] (02) 242 75 80, Fax (02) 726 72 76.
BRAZIL: Texas Instrumentos Electronicos do Brasil Ltda.: Sao Paulo [55] 11-535-5133.
CANADA: Texas Instruments Canada Ltd.: Montreal (514) 335-8392; **Ottawa** (613) 726-3201; **Toronto** (416) 884-9181.
DENMARK: Texas Instruments A/S: Ballerup [45] (44) 68 74 00.
FINLAND: Texas Instruments/OY: Espoo [358] (0) 43 54 20 33, Fax (0) 46 73 23.
FRANCE: Texas Instruments France: Velizy-Villacoublay Cedex [33] (1) 30 70 10 01, Fax (1) 30 70 10 54.
GERMANY: Texas Instruments Deutschland GmbH.: Freising [49] (08161) 80-0, Fax (08161) 80 45 16; **Hannover** (0511) 90 49 60, Fax (0511) 64 90 331; **Ostfildern** (0711) 34 03 0, Fax (0711) 34 032 57.
HONG KONG: Texas Instruments Hong Kong Ltd.: Kowloon [852] 956-7288, Fax 956-2200.
HUNGARY: Texas Instruments Representation: Budapest [36] (1) 269 8310, Fax (1) 267 1357.
INDIA: Texas Instruments India Private Ltd.: Bangalore [91] 80 226-9007.
IRELAND: Texas Instruments Ireland Ltd.: Dublin [353] (01) 475 52 33, Fax (01) 478 14 63.
ITALY: Texas Instruments Italia S.p.A.: Agrate Brianza [39] (039) 68 42.1, Fax (039) 68 42.912; **Rome** (06) 657 26 51.
JAPAN: Texas Instruments Japan Ltd.: Tokyo [81] 03-769-8700, Fax 03-3457-6777; **Osaka** 06-204-1881, Fax 06-204-1895; **Nagoya** 052-583-8691, Fax 052-583-8696; **Ishikawa** 0762-23-5471, Fax 0762-23-1583; **Nagano** 0263-33-1060, Fax 0263-35-1025; **Kanagawa** 045-338-1220, Fax 045-338-1255; **Kyoto** 075-341-7713, Fax 075-341-7724; **Saitama** 0485-22-2440, Fax 0425-23-5787; **Oita** 0977-73-1557, Fax 0977-73-1583.
KOREA: Texas Instruments Korea Ltd.: Seoul [82] 2-551-2800, Fax 2-551-2828.
MALAYSIA: Texas Instruments Malaysia: Kuala Lumpur [60] 3-230-6001, Fax 3-230-6605.
MEXICO: Texas Instruments de Mexico S.A. de C.V.: Colina del Valle [52] 5-639-9740.
NORWAY: Texas Instruments Norge A/S: Oslo [47] (02) 264 75 70.
PEOPLE'S REPUBLIC OF CHINA: Texas Instruments China Inc.: Beijing [86] 1-500-2255, Ext. 3750, Fax 1-500-2705.
PHILIPPINES: Texas Instruments Asia Ltd.: Metro Manila [63] 2-817-6031, Fax 2-817-6096.
PORTUGAL: Texas Instruments Equipamento Electronico (Portugal) LDA.: Mala [351] (2) 948 10 03, Fax (2) 948 19 29.
SINGAPORE / INDONESIA / THAILAND: Texas Instruments Singapore (PTE) Ltd.: Singapore [65] 390-7100, Fax 390-7062.
SPAIN: Texas Instruments España S.A.: Madrid [34] (1) 372 80 51, Fax (1) 372 82 66; **Barcelona** (3) 31 791 80.
SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): Kista [46] (08) 752 58 00, Fax (08) 751 97 15.
SWITZERLAND: Texas Instruments Switzerland AG: Dietikon [41] 886-2-3771450.
TAIWAN: Texas Instruments Taiwan Limited: Taipei [886] (2) 378-6800, Fax 2-377-2718.
UNITED KINGDOM: Texas Instruments Ltd.: Bedford [44] (0234) 270 111, Fax (0234) 223 459.

UNITED STATES: Texas Instruments Incorporated: ALABAMA: Huntsville (205) 430-0114; **ARIZONA: Phoenix** (602) 244-7800; **CALIFORNIA: Irvine** (714) 660-1200; **San Diego** (619) 278-9600; **San Jose** (408) 894-9000; **Woodland Hills** (818) 704-8100; **COLORADO: Aurora** (303) 368-8000; **CONNECTICUT: Wallingford** (203) 265-3807; **FLORIDA: Orlando** (407) 260-2116; **Fort Lauderdale** (305) 425-7820; **Tampa** (813) 882-0017; **GEORGIA: Atlanta** (404) 662-7967; **ILLINOIS: Arlington Heights** (708) 640-2925; **INDIANA: Indianapolis** (317) 573-6400; **KANSAS: Kansas City** (913) 451-4511; **MARYLAND: Columbia** (410) 312-7900; **MASSACHUSETTS: Boston** (617) 895-9100; **MICHIGAN: Detroit** (313) 553-1500; **MINNESOTA: Minneapolis** (612) 828-9300; **NEW JERSEY: Edison** (908) 906-0033; **NEW MEXICO: Albuquerque** (505) 345-2555; **NEW YORK: Poughkeepsie** (914) 897-2900; **Long Island** (516) 454-6601; **Rochester** (716) 385-6770; **NORTH CAROLINA: Charlotte** (704) 522-5487; **Raleigh** (919) 876-2725; **OHIO: Cleveland** (216) 765-7258; **Dayton** (513) 427-6200; **OREGON: Portland** (503) 643-6758; **PENNSYLVANIA: Philadelphia** (215) 825-9500; **PUERTO RICO: Hato Rey** (809) 753-8700; **TEXAS: Austin** (512) 250-6769; **Dallas** (214) 917-1264; **Houston** (713) 778-6592; **WISCONSIN: Milwaukee** (414) 798-1001.

North American Authorized Distributors

COMMERCIAL
Almac / Arrow
Anthem Electronics
Arrow / Schweber
Future Electronics (Canada)
Hamilton Hallmark
Marshall Industries
Wyle

MILITARY
Alliance Electronics Inc
Future Electronics (Canada)
Hamilton Hallmark
Zeus, An Arrow Company

CATALOG
Allied Electronics
Arrow Advantage
Newark Electronics

OBSOLETE PRODUCTS
Rochester Electronics 508/462-9332

For Distributors outside North America, contact your local Sales Office.

A1194

TEXAS INSTRUMENTS

TEXAS INSTRUMENTS

Printed in U.S.A., January 1993
2547301-9761 revision D

SPRU056B